;login:
SPRING 2016 VOL. 41, NO. 1

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

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

March 16–18, 2016, Santa Clara, CA, USA
www.usenix.org/nsdi16

Co-located with NSDI '16

CoolDC '16: USENIX Workshop on Cool Topics on Sustainable Data Centers
March 19, 2016
www.usenix.org/cooldc16

SREcon16

April 7–8, 2016, Santa Clara, CA, USA
www.usenix.org/srecon16

USENIX ATC '16: 2016 USENIX Annual Technical Conference

June 22–24, 2016, Denver, CO, USA
www.usenix.org/atc16

Co-located with USENIX ATC '16:

HotCloud '16: 8th USENIX Workshop on Hot Topics in Cloud Computing
June 20–21, 2016
www.usenix.org/hotcloud16

HotStorage '16: 8th USENIX Workshop on Hot Topics in Storage and File Systems
June 20–21, 2016
www.usenix.org/hotstorage16

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

SREcon16 Europe

July 11–13, 2016, Dublin, Ireland
www.usenix.org/srecon16europe

Do you know about the USENIX Open Access Policy?

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

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

www.usenix.org/membership

USENIX Security '16: 25th USENIX Security Symposium

August 10–12, 2016, Austin, TX, USA
www.usenix.org/sec16

Co-located with USENIX Security '16

WOOT '16: 10th USENIX Workshop on Offensive Technologies
August 8–9, 2016
Submissions due May 17, 2016
www.usenix.org/woot16

CSET '16: 9th Workshop on Cyber Security Experimentation and Test
August 8, 2016
Submissions due May 3, 2016
www.usenix.org/cset16

FOCI '16: 6th USENIX Workshop on Free and Open Communications on the Internet
August 8, 2016
Submissions due May 19, 2016
www.usenix.org/foci16

ASE '16: 2016 USENIX Workshop on Advances in Security Education
August 9, 2016
Submissions due May 3, 2016
www.usenix.org/ase16

HotSec '16: 2016 USENIX Summit on Hot Topics in Security
August 9, 2016
www.usenix.org/hotsec16

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

November 2–4, 2016, Savannah, GA, USA
Abstracts due May 3, 2016
www.usenix.org/osdi16

Co-located with OSDI '16

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

LISA16

December 4–9, 2016, Boston, MA, USA
Submissions due April 25, 2016
www.usenix.org/lisa16

Co-located with LISA16

SESA '16: 2016 USENIX Summit for Educators in System Administration
December 6, 2016

USENIX Journal of Education in System Administration (JESA)
Published in conjunction with SESA
Submissions due August 26, 2016
www.usenix.org/jesa

;login:

SPRING 2016 VOL. 41, NO. 1

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Michele Nelson
michele@usenix.org

COPY EDITORS
Steve Gilmartin
Amber Ankerholz

PRODUCTION
Arnold Gatilao
Jasmine Murcia

TYPESETTER
Star Type
startype@comcast.net

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

www.usenix.org

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

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

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

EDITORIAL

Musings

RIK FARROW

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

This time around I thought I would write about the future of the Internet. Please note the capital "I," as that's what you will find in the Future Internet Design (FIND) final report [1], where the authors suggest strategies to the NSF for funding research into networking.

That initial conference took place in 2009 and looked at 49 projects. One outcome of the NSF NeTS FIND Initiative [2] was to continue funding several of the projects. I was vaguely aware of this work, but I also wondered how in the world anyone could hope to change the Internet, the system of networks we've all grown to rely upon—really, to depend upon—at this point in time.

On the economic side, there is the issue of sunk costs: companies have spent billions creating the network we have today. Then there is conservatism: people have learned (at least enough) to work with TCP/IP, with all its quirks. And, finally, any new protocols will require hardware support, and that's the issue I found worried the people whom I talked to about the NSF project I chose to focus on.

Named Data Networking

I didn't pick Named Data Networking (NDN) out of a hat. kc claffy had just 45 minutes to introduce some of the concepts behind this protocol during LISA15 [3], and I had heard something about NDN earlier. I think it was kc's mention of the importance of security that got me interested. If you read the FIND report [1], you will also see that security often gets mentioned *first* in lists of desirable new features in future protocols. But NDN is about a lot more than just supporting security over network traffic.

NDN comes out of research done by Van Jacobson and others at Palo Alto Research Center (PARC) [4] in 2009. The authors of that paper created a protocol called Content-Centric Networking (CCN), largely because of the realization that then-current Internet traffic was mostly about shared content. Today, streaming video (content) makes up close to two-thirds of all Internet traffic, making the notion of a network focused on content even more relevant.

The NDN researchers started with many of the ideas expressed in the CCN research to create a new protocol with similar goals. The very name, Named Data Networking, hints at the key ideas.

Today's Internet, based on the Internet Protocol (IP), relies on binary addressing for point-to-point communication. We start with DNS names, DNS provides the binary addresses (although we generally think of them as four decimal bytes separated by dots), and communication is between a pair of endpoints. Point-to-point communication made a lot of sense in the 1970s, when computers were rare and just connecting a computer to a shared network required the use of a mini-computer, called Interface Message Processors, IMPs [5], a 16-bit computer the size of a refrigerator, not including its console. The computers that connected to the ARPANET were multi-million-dollar machines themselves. You could say that the world then (just 40 years ago) was very different. Researchers really wanted ways to share data and remotely log in in those days, and those two goals were the focus for designing TCP/IP.

Today, over a quarter of the world's population uses the Internet, and what they want from it is content. *Named data* refers to the requests for data in NDN, called *Interests,* which look a lot like URLs in a RESTful interface. Naming is hierarchical, something that IP addressing has never managed to have, although IPv6 is better in this regard.

The responses to Interests are called *Content*, and the data in Content packets are signed by the source. Having signed data means you can trust that the data came from the source you are interested in, even if that data had been cached by a cooperating router.

Of course, signing relies on there being a secure method for sharing public keys, and secure sharing of certificates is also an important part of NDN. NDN plans on using a Web of Trust, where you have local roots for your own organization, but must trust other certificate signers for trusting certs from the greater Internet. The details of this must still be worked out.

The Hard Part

Well, I jest, because there are lots of hard parts. But one of the things that really caught my attention about this design is how much more involved routers will be in a network where NDN is the underlying protocol. In TCP/IP, IP is what network designers call the "thin waist." What they mean is that one relatively simple protocol, IP, is what is used to get packets delivered across the Internet.

NDN's thin waist are Interests and Content. Routers need to be able to interpret the names in Interests, decide how to forward those Interests, keep track of which port Interests arrived on (so they can return Content via that port), as well as cache Content. Compared to IP routers, that's a huge departure from the way things are currently done.

Since routers replaced gateways (like the IMP, and later Sun and DEC servers), routers started having special hardware that supported the *fastpath*. The fastpath represented the port pair for a particular route and avoided having to use the much, much slower router CPU to make routing decisions for each packet. The fastpath allowed parallel lookups, using Ternary Content Addressable Memory (TCAM [6]) to route packets. TCAM solved what was becoming the problem that would "kill" the Internet in the late '90s, when the number of routes was doubling every several years, requiring four times longer to look up routing information for each packet for each doubling in routing table size.

There aren't any TCAMs for names. In fact, parsing names using current hardware for routing seems like an impossible task today. But then, we faced a similar problem just 20 years ago with IP routing.

There are the other issues that would need to be solved, ones that we have not been able to solve so far, like a trustworthy means for distributing public key certificates. X.509 is itself a terrible protocol—just consider how often libraries for parsing X.509 have resulted in exploits, because X.509 is too ambiguous. We also have certificate authorities, like Symantec, having its root certificates banned by Google [7] because of abuse. And that's not the only case of CAs behaving as paper mills—producers of nice certificates for a fee—instead of identity authorities.

NDN runs over a UDP overlay today, but plans are for NDN to run natively some day. If we ever expect to replace cable with the Internet, we really need a way to stream popular entertainment, like sports events, in an efficient manner. And TCP/IP is not designed for streaming, while NDN would do streaming well, as its design easily and naturally handles multicasting.

The Lineup

We begin the features in this issue with Filebench, a project started within Sun Microsystems many years ago for benchmarking NFS. Vasily Tarasov, Erez Zadok, and Spencer Shepler explain how to use Filebench for benchmarking file systems. Filebench does include templates for several common uses, but the real power in Filebench is your ability to tune the benchmarks to your particular use cases.

Amandeep Khurana and Jayant Shekhar tell us about different systems for processing streaming data. They cover Kafka, Spark, Storm, and Flink, describing the strengths and weakness of each system, all of which add streaming over Hadoop-related architectures. Kafka handles data ingestion, where Spark, Storm, and Flink provide different approaches to analysis.

Jonathan Mace, Rodrigo Fonseca, and Ryan Roelke reprise their SOSP '15 award-winning paper about Pivot Tracing. Pivot Tracing adds metadata to requests in distributed systems on-the-fly, allowing you to monitor and debug these applications, much the way you would use DTrace or Systemtap on local applications.

I interview Doug McIlroy, who was a manager at Bell Labs when the UNIX system was being created. Doug is best known for his work in adding pipes to the UNIX system, but also wrote code from some tools that we still use today.

Arnaud Tomeï takes a comprehensive look at his experiences with creating portable shell scripts. While POSIX was all about creating a standard for UNIX-like features, Tomeï discovered many places where using features found in the most common shells and popular commands will get you in trouble when you try to write one script for multiple *nix systems.

Barclay Osborn, Justin McWilliams, Betsy Beyer, and Max Saltonstall provide another look at BeyondCorp, Google's project

to replace VPNs into sensitive networks with gateways over encrypted connection to services. Rory Ward and Beyer provided a view into this project in a December 2014 *;login:* article [8], and the authors update us on how the project has evolved, and what challenges have been overcome over the intervening year.

Mark Gondree, Zachary N J Peterson, and Portia Pusey share the work being done surrounding the issues of naming in the area of gaming for security education. Terms like Capture the Flag (CTF) have wound up being applied to games that have little to do with the original notion, and not having a standard terminology for styles of games hurts attempts at using gaming for any form of computer education that might take advantage of it.

I interviewed Lixia Zhang and kc claffy about NDN, the subject of this column. I recommend reading this interview and checking out the resources at the end of it so you can learn more about NDN. You might even want to try out some of the sample applications.

Dave Beazley tells us about a problem when using Python 3.5's new asyncio functions: you don't know what other functions will fail when you start using asyncio functions. Dave deftly describes this as the red/blue problem and provides some interesting Python function decorators as possible solutions.

David Blank-Edelman wants us to use Swagger, not an exaggerated way of walking but a Java-based tool that makes writing the code for APIs between client and servers a stroll in the park. Swagger includes code generators for many languages, although only for the client-side of Perl.

Dave Josephsen doesn't want you to be a hero. Dave refers specifically to Brent in the novel *The Phoenix Project,* the one person who can solve any problem, and thus the bottleneck to getting any IT project completed. Dave uses an example to demonstrate how things should work.

Kelsey Hightower introduces his column on Go for sysadmins, where he describes how to use RPCs to build a distributed tool that could be the basis for a monitoring system. Kelsey will be writing Go columns designed to help system administrators, and anyone new to Go, take advantage of one of the best-designed languages.

Dan Geer bets on growth over magnitude. When looking at the problems you will need to solve, do you choose the ones with the most current problems or the ones with the fastest growing list of issues? Dan explains his reasoning behind picking growth.

Robert G. Ferrell, inspired by my look at NDN, considers how he helped with organizing RFCs in the '90s, then ponders NDN, without naming it.

Mark Lamourine has just one book review in this issue. Mark writes about *The Logician and the Engineer: How George Boole and Claude Shannon Created the Information Age*. Like the author, Paul Nahin, Mark considers Boole and Shannon unsung heroes (the good kind) in the creation of computers.

In USENIX Notes, Dan Klein tells us why he has worked with USENIX—as education director, paper author, and now Board member—for over 25 years.

It has been said that pornography was the driving force behind the incredible growth of the Internet. During the 1990s, I would meet with UUnet employees at USENIX conferences and hear that since the last time we had seen each other, the size of the Internet had doubled. While I don't really have any idea whether this was because of pornography, attempts at streaming football games might have a similar effect on the introduction of new protocols in the Internet.

Fortuitously, while I was pondering this column, Bloomberg published a magazine article about how, if the NFL were to get serious about live streaming football games [9], they would need a different Internet. TCP/IP was designed for point-to-point transfer, not the one-to-many streaming that huge events require. And entertainment providers like Netflix now dominate Internet traffic. These uses, and more, could really benefit from new protocols like NDN.

Resources

[1] National Science Foundation, "FIND Observer Panel Report," April 9, 2009: http://www.nets-find.net/FIND_report_final.pdf.

[2] NSF NeTS FIND Initiative: http://www.nets-find.net/.

[3] kc claffy, "Named Data Networking": https://www.usenix.org/conference/lisa15/conference-program/presentation/claffy.

[4] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, Rebecca L. Braynard, "Networking Named Content," ACM CoNEXT 2009: http://conferences.sigcomm.org/co-next/2009/papers/Jacobson.pdf.

[5] Interface Message Processor: https://en.wikipedia.org/wiki/Interface_Message_Processor.

[6] Content Addressable Memory: https://en.wikipedia.org/wiki/Content-addressable_memory.

[7] Lucian Armasu, "Google to Remove a Symantec Root Certificate from Chrome and Android," December 11, 2015, Tom's Hardware: http://www.tomshardware.com/news/google-removes-symantec-root-certificate,30742.html.

[8] Rory Ward and Betsy Beyer, "A New Approach to Enterprise Security," *;login:*, vol. 39, no. 6, December 2014: https://www.usenix.org/publications/login/dec14/ward.

[9] Joshua Brustein, "How NFL Thursdays Could Break the Internet," Bloomberg Business, December 15, 2015: http://www.bloomberg.com/news/articles/2015-12-15/how-nfl-thursdays-could-break-the-internet?cmpid=BBD121515.

Filebench

A Flexible Framework for File System Benchmarking

VASILY TARASOV, EREZ ZADOK, AND SPENCER SHEPLER

Vasily Tarasov is a Researcher at IBM Almaden Research Center. He started to use and contribute to Filebench extensively while working on his PhD at Stony Brook University. Vasily's interests include system performance analysis, design and implementation of distributed systems, and efficient I/O stacks for ultra-fast storage devices. vtarasov@us.ibm.com

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@cs.stonybrook.edu

Spencer Shepler is a Private Cloud Architect at Microsoft. Prior to Microsoft, Spencer worked at a failed startup and before that at Sun Microsystems. While at Sun, Spencer worked on Filebench along with many others and at his full-time job of bringing NFSv4 to market. sshepler@microsoft.com

File system benchmarks constitute a crucial part of a storage evaluator's toolbox. Due to the wide variety of modern workloads and ever-growing list of storage stack features, modern benchmarks have become fairly complex tools. This article describes Filebench, one of the most popular modern file system benchmark tool suites. Using several practical examples, we demonstrate Filebench's versatility, expressiveness, and ease of use. It is our hope that this article will encourage people to use Filebench to describe their real-life workloads as well as publicly contribute new workloads.

Filebench is a highly flexible framework for file system and storage benchmarking. The project started around 2002 inside Sun Microsystems and was open-sourced around 2005. It is now hosted at sourceforge.net [2] and maintained by the community, centered around the File systems and Storage Lab (FSL) at Stony Brook University. According to Google Scholar, Filebench was used in over 500 publications and remains one of the most popular file system benchmarks both in academia and industry. The popularity of the Filebench framework comes mainly from the fact that it is shipped with several predefined macro workloads, e.g., Web-server, Mail-server, and File-server. This allows users to easily benchmark their file systems against several sufficiently different workloads with a single tool.

The intrinsic power of Filebench originates, however, not from the included workloads but rather from its expressive Workload Model Language (WML), which allows users to encode a wide variety of workloads. We therefore find ourselves disappointed that most users do not go beyond the predefined workloads and consequently do not utilize the full power of Filebench. The goal of this article is to educate the community on Filebench's WML and demonstrate both its long-standing and recently added features. In addition, we describe best practices for using Filebench to avoid common beginners' mistakes.

Basic Functionality

Many existing storage benchmarks (e.g., fio, mdtest, and SPECsfs) hard code the workloads they generate quite rigidly. A user can specify some basic workload parameters (e.g., I/O size, number of threads, read/write ratio) but cannot really control the execution flow in detail. Expressing a workload with a general-purpose programming language (e.g., C/C++ or Python) is another extreme that offers the utmost flexibility but is time-consuming. The Filebench framework provides a much needed middle ground: high flexibility combined with the ease of describing a workload.

In Filebench, users define workloads using a Workload Model Language (WML). There are four main entities in WML: *fileset*, *process*, *thread*, and *flowop*. Every defined entity must have a user-assigned name that is mainly used to print per-process and per-thread statistics. A fileset is a named collection of files. To define a fileset, a user specifies its name, path, number of files, and a few other optional attributes. Listing 1 shows two filesets with 1,000 files of 128 KB size that will be located in the `/tmp` directory.

```
define fileset name="test1",path="/tmp",
                        entries=1000,filesize=128k
define fileset name="test2",path="/tmp",
            entries=1000,filesize=128k,prealloc=80
```

Listing 1: Examples of fileset definitions

A Filebench run proceeds in two stages: fileset preallocation and an actual workload execution. By default, Filebench does not create any files in the file system and only allocates enough internal file entries to accommodate all defined files. To actually create files, one should specify the percentage of files to precreate with the `prealloc` attribute. Listing 1 shows how Filebench precreates 800 files in the fileset `test2`—80% of 1,000.

The reason for Filebench not to precreate all (or any) files is that certain workloads include file creates. When a workload-defined file create operation should be executed, Filebench picks a non-existent file entry in a fileset and creates the file. The total number of simultaneously existing files in a fileset can never exceed the fileset size at any point during a Filebench run. If a workload tries to create a file but there are no more non-existent file entries, then an internal `Out-of-Resources` event is triggered, which can be interpreted either as an end of the run or an error, depending on the user's objective. Consider a WML snippet in Listing 2 that can be used to measure peak file create rate. At first, the fileset is empty and the workload starts to create files in a loop. When the workload tries to create the 10,001st file, Filebench graciously exits and reports the measurements. This happens because quit mode is set to `firstdone`; more information on this and other quit modes is described later in this article. Note that delete operations reduce the number of existing files and can balance out the file create operations.

```
set mode quit firstdone

define fileset name="fcrset",path="/tmp",
            entries=10000,filesize=16k

define process name="filecreate",instances=1 {
 thread name="filecreatethread",instances=2 {
  flowop createfile name="crfile",filesetname="fcrset"
  flowop closefile name="clfile"
 }
}

run
```

Listing 2: WML snippet to measure file create performance

WML processes represent real UNIX processes that are created by Filebench during the run. Every process consists of one or more threads representing actual POSIX threads. The attribute `instances=`*N* instructs Filebench to replicate the corresponding processes and threads *N* times. Listing 2 defines one process named `filecreate` with two identical threads. WML allows users to define any number of identical or different processes containing any number of identical or different threads. Listing 3 demonstrates a more complex workload description with five processes in total. Three processes contain one reader thread and two writer threads; two other processes contain four identical threads that create and delete files. All processes and threads run simultaneously.

```
define process name="testprocA",instances=3 {
 thread name="reader",instances=1 {
  flowop openfile name="readop",filesetname="testset"
  flowop readwholefile name="readop",iosize=4k
  flowop closefile name="closeop1"
 }
 thread name="writer",instances=2 {
  flowop openfile name="readop",filesetname="testset"
  flowop writewholefile name="writeop",iosize=4k
  flowop closefile name="closeop2"
 }
}

define process name="testprocB",instances=2 {
 thread name="crdelthread",instances=4 {
  flowop createfile name="createop",filesetname="testset"
  flowop closefile name="closeop3"
  flowop deletefile name="deleteop",filesetname="testset"
 }
}
```

Listing 3: Example of defining multiple different processes and threads

Every thread executes a loop of flowops. Flowop is a representation of a file system operation and is translated to a system call by Filebench: e.g., the `createfile` flowop creates a file and the `write` flowop writes to a file. Table 1 lists the most common WML flowops, which cover the majority of operations that real applications execute against a file system. When Filebench reaches the last flowop defined in a thread, it jumps to the beginning of the thread definition and executes flowops repeatedly until a quit condition is met (e.g., requested runtime elapsed).

Flowop	Description
openfile	Opens a file. One can specify a virtual file descriptor to use in the following flowops.
closefile	Closes a file referenced by a virtual file descriptor
createfile	Creates a file. One can specify a virtual file descriptor to use in the following flowops.
deletefile	Deletes a file
read	Reads from a file
readwholefile	Reads whole file even if it requires multiple system calls
write	Writes to a file
writewholefile	Writes whole file even if it requires multiple system calls
appendfile	Appends to the end of a file
statfile	Invokes stat() on a file
fsync	Calls fsync() on a file

Table 1: List of most frequently used flowops. In addition, Filebench supports a number of directory, asynchronous I/O, synchronization, operation limiting, and CPU consuming and idling operations.

Filebench uses *Virtual File Descriptors* (VFDs) to refer to files in flowops. VFDs are not actual file descriptors returned by `open()`; instead, users assign VFDs explicitly in `openfile` and `createfile` flowops. Later, these VFDs can be used in flowops that require a file to operate on. Listing 4 provides an example where the attribute `fd` is used to specify two different VFDs. First, the thread opens one file, assigning it VFD 1 and creates another file with VFD 2. Then the thread reads from one file and writes to another, keeping both files open and referring to them by their VFDs. Finally, both files are closed. This represents a simple copy workload in WML. Note that VFDs are per-thread entities in Filebench: a VFD in one thread does not impact an identically numbered VFD in another thread.

VFDs specified in `openfile` and `createfile` must not be opened prior to the flowops execution. Therefore, in most of the cases it is necessary to explicitly close VFDs with a `closefile` flowop. Other flowops that require a VFD (e.g., `read`) will open a file automatically if the corresponding VFD is not opened yet. If the `fd` attribute is not specified in a flowop then Filebench assumes that it is equal to zero. This is a useful convention for a large class of workloads that keep only one file open at a time (see Listings 2 and 3). Describing such workloads in WML does not require specifying the `fd` attribute, which streamlines the workload description further.

```
set mode quite firstdone

define fileset name="testfset",path="/tmp",
           entries=10000,filesize=4k,prealloc=50

define process name="filecopy",instances=2 {
 thread name="filecopythread",instances=2 {
   flowop openfile name=opfile",
               filesetname="testfset",fd=1
   flowop createfile name="crfile",
               filesetname="testfset",fd=2
   flowop readwholefile name="rdfile",
               filesetname="testfset",fd=1
   flowop writewholefile name="wrfile",
               filesetname="testfset",fd=2
   flowop closefile name="clfile1",
               filesetname="testfset",fd=2
   flowop closefile name="clfile2",
               filesetname="testfset",fd=1
 }
}
```

Listing 4: Simple file copying expressed in WML

When opening a file, Filebench first needs to pick a file from a fileset. By default this is done by iterating over all file entries in a fileset. To change this behavior one can use the `index` attribute that allows one to refer to a specific file in a fileset using a unique index. In most real cases, instead of using a constant number for the index, one should use custom variables described in the following section.

Filebench supports a number of attributes to describe access patterns. First, one can specify an I/O size with the `iosize` attribute. Second, one can pick between sequential (default) and random accesses. Sequential patterns usually make sense only if a file is kept open between the flowop executions so that the operating system can maintain the current position in a file. When the end of a file is reached, sequential flowops start accessing the file from the beginning. Third, for random workloads, one can specify the working set size in a file using the `wss` attribute. Finally, direct and synchronous I/Os are supported as well.

The very last line of a WML file usually contains a `run` or `psrun` command. These commands tell Filebench to allocate the defined filesets, spawn the required number of UNIX processes and threads, and, finally, start a cycled flowops execution. Both commands take the duration of the run in seconds as an argument; the `psrun` command in addition takes a period with which to print performance numbers.

To generate a workload described, e.g., in a `workload.f` WML file (`.f` is a traditional extension used by Filebench), one executes the `filebench -f workload.f` command. A non-abortive run terminates under two conditions. First, the run can be time-

based; this is the default mode and if the `run` command does not have any arguments, then the workload will run for one minute only. Second, a Filebench run might finish if one or all threads completed their job. To specify Filebench's quit mode, a `set mode quit` command can be used. In Listing 2, we change the quit mode to `firstdone`, which means that whenever one of the threads runs out of resources (e.g., there are no more non-existent files to create), Filebench stops. Another scenario is when a thread explicitly declares that it completed its job using the `finishoncount` or `finishonbytes` flowops. These flowops allow one to terminate a thread after a specific number of operations completed (e.g., writes or reads) or a specific number of bytes were read or written by a thread.

In the end of the run, Filebench prints a number of different metrics. The most important one is operations per second. This is the total number of executed flowop instances (in all processes and threads) divided by the runtime. For flowops that read and write data, Filebench also prints the throughput in bytes per second. Finally, one can measure the average and distribution of latencies of individual flowops. In addition, Filebench can maintain and print statistics per process, per thread, or per flowop.

Long-time Filebench framework users might be surprised that we described Filebench's run as a non-interactive experience. In fact, before version 1.5, Filebench supported interactive runs: a console in which one could type workloads and execute various commands. However, one of the big changes in v1.5 is the elimination of interactive mode. The majority of experienced users did not use non-interactive runs. Beginners, on the other hand, made a lot of systematic mistakes in interactive mode (e.g., did not drop caches or remove existing filesets between runs). In v1.5, therefore, we made a strategic decision not to support interactive mode. This further helped reduce the total amount of code to maintain.

Advanced Features

In this section, we highlight some advanced Filebench features. They were either less known before or were just recently added in version 1.5. Listing 5 demonstrates most of these features.

Variables

Filebench supports two types of variables: regular and custom. Variable names, irrespective of their type, are prefixed with a dollar sign. With a few exceptions, variables can be used instead of constants in any process, thread, or flowop attribute. Regular variables hold constant values, are defined with the `set` keyword, and are mainly used for convenience. It is considered a good style to define all parameters of the workload (e.g., I/O sizes or file numbers) in the beginning of a WML file and then use variables in the actual workload definition; it also facilitates easier changes to the workload. Listing 5 demonstrates how the `$iosize` regular variable is used to set I/O size.

```
set $iosize=4k
set $findex=cvar(type=cvar-normal,min=0,max=999,
                parameters=mean:500;sigma:100)
set $off=cvar(type=cvar-triangular,min=0,max=28k,
                parameters=lower:0;upper:28k;mode:16k)

enable lathist

define fileset name="test",path="/tmp",entries=1000,
                filesize=32k,prealloc=100

eventgen rate=100

define process name="testproc1" {
 thread name="reader",memsize=10m {
  flowop read name="rdfile",filesetname="test",
                indexed=$findex,offset=$off,iosize=$iosize
  flowop closefile name="clsfile1"
  flowop block name="blk"
 }
 thread name="writer",memsize=20m {
  flowop write name="wrfile",
                filesetname="test",iosize=$iosize
  flowop closefile name="clsfile2"
  flowop opslimit name="limit"
 }
 thread name="noio",memsize=40m {
  flowop hog name="eatcpu",value=1000
  flowop delay name="idle",value=1
  flowop wakeup name="wk",target="blk"
 }
}
```

Listing 5: Demonstration of some advanced Filebench features

The use of custom variables (`cvar`) powerfully enables any Filebench attribute to follow some statistical distribution. Distributions are implemented through dynamically loadable libraries with a simple and well-defined interface that allows users to add new distributions easily. When Filebench starts, it looks for the libraries in a certain directory and loads all supported distributions. We ported the Mtwist package [4] to the custom variables subsystem; this immediately made Filebench support eight distributions, and this number is growing.

In Listing 5 the `indexed` attribute of the `rdfile` flowop follows the distribution described by the `$findex` custom variable. The `$findex` variable uses a normal distribution with values bounded to the 0–999 range. The minimum and maximum bounds are in sync with the number of files in the fileset here—1,000. Distribution-specific parameters—mean and standard deviation (sigma) in case of a normal distribution—are specified with the `parameters` keyword. As we mentioned in the Basic Functionality section, Filebench by default picks files from a fileset in a rotating

manner and the `indexed` attribute can pick specific files. Assigning `findex` makes Filebench access some files more frequently than others using a normal distribution from a custom variable. This simulates a real-world scenario in which some files are more popular than others.

Earlier Filebench versions actively used the so-called random variables, which are essentially similar to custom variables. But we found random variables limiting because the number of supported distributions was small, and adding more distributions required significant knowledge of Filebench's code base. In version 1.5 we replaced random variables with custom variables (random variables are still supported for backward compatibility but will be phased out in the future).

Synchronization Primitives

When a workload is multithreaded, it sometimes makes sense to emulate the process by which requests from one thread depend on requests from other threads. For this, Filebench provides the `block`, `wakeup`, `semblock`, and `sempost` flowops. They allow Filebench to block certain threads until other threads complete the required steps. Listing 5 shows how a reader thread blocks in every loop until the `noio` thread wakes it up.

The ability to quickly define multiple processes and synchronization between them was one of the main requirements during Filebench framework conception. The task for Sun Microsystems engineers at the time was to improve file system performance for a big commercial database. Setting up TPC-C [7], database, and all of the required hardware was expensive and time-consuming for an uninvolved file system engineer. The key for simulating database load on a file system was how log writes cause generic table updates to block. With this use case in mind, Filebench's WML was designed, and a corresponding `oltp.f` workload personality was created and then validated against the real database. Having the Filebench framework and a workload description in WML gave engineers the time to focus just on the file system tuning task.

CPU and Memory Consumption

Filebench provides a `hog` flowop that consumes CPU cycles and memory bandwidth. Conversely, the flowop `delay` simulates idle time between requests. Also, when defining a thread, one must specify its memory usage with the `memsize` attribute. Every thread consumes this amount of memory and performs reads and writes from it. In Listing 5 the `noio` thread burns CPU by copying memory 1,000 times and then sleeps for one second per loop.

Speed Limiting

In many cases one wants to evaluate system behavior under moderate or low loads (which are quite common in real systems) instead of measuring peak performance. Filebench supports this with the flowops `iopslimit` (limits the rate of data operations only) and `bwlimit` (limits the bandwidth). In Listing 5, the reader thread issues only 100 reads per second (or fewer if the system cannot fulfill this rate). The command `eventgen` sets the rate, which is global for all processes and threads.

Complex Access Patterns

Originally Filebench supported only simple access patterns: uniformly random and sequential. We added the `offset` attribute which, in combination with custom variables, allows one to emulate any distribution of accesses within a file. For example, for virtualized workloads with big VMDK files, we observed that some offsets are more popular than others [6]. In Listing 5, the writer thread accesses file's offsets following a triangular distribution.

Latency Distribution

Measuring only the average latency often does not provide enough information to understand a system's behavior in detail. We added latency distribution profiling with the `enable lathist` command to Filebench [3].

Composite Flowops

In WML one can define a flowop that is a combination of other flowops. This is especially useful in cases when one wants to execute certain group of flowops more frequently than other flowops. The attribute `iters` can be used to repeat regular or composite flowops. In addition, Filebench's internal design allows users to easily implement new flowops in C. We do not provide examples of composite or user-defined flowops in this article but offer documentation online [2].

File System Importing

Another upcoming feature in Filebench v1.5 is importing existing file system trees. Older versions of Filebench could only work with trees that it generated itself. This new feature allows one to generate a file system with a third-party tool (e.g., Impressions [1]), or use a real file system image and run a Filebench workload against this file system.

Data Generation

Earlier, Filebench versions generated all zeros or some arbitrary content for writes. In v1.5, we are introducing the notion of a *datasource*, which can be attached to any flowop. Different datasources can generate different types of data: one controlled by some entropy, duplicates distribution, file types, etc. This new feature is especially important for benchmarking modern storage systems that integrate sophisticated data reduction techniques (e.g., deduplication, compression).

Predefined Workloads

It is important to understand that Filebench is merely a framework, and only its combination with a workload description defines a specific benchmark. The framework comes with a set of predefined useful workloads that are especially popular among users. We are often asked about the details of those workloads. In this section, we describe the three most frequently used Filebench workloads: Web-server, File-server, and Mail-server.

What does a simple real Web-server do from the perspective of a file system? For every HTTP request, it opens one or more HTML files, reads them completely, and returns their content to the client. At times it also flushes client-access records to a log file. Filebench's Web-server workload description was created with exactly these assumptions. Every thread opens a file, reads it in one call, then closes the file. Every 10th read, Filebench's Web-server appends a small amount of data to a log file. File sizes follow a gamma distribution, with an average file size of 16 KB. By default, the Web-server workload is configured with 100 threads and only 1,000 files. As described later in the section, it is almost always necessary to increase the number of files to a more appropriate number.

Filebench's File-server workload was also designed by envisioning a workload that a simple but real File-server produces on a file system. Fifty processes represent 50 users. Every user creates and writes to a file; opens an existing file and appends to it; then opens another file and reads it completely. Finally, the user also deletes a file and invokes a stat operation on a file. Such operation mix represents the most common operations that one expects from a real File-server. There are 10,000 files of 128 KB size defined in this workload by default.

The Mail-server workload (called `varmail.f`) represents a workload experienced by a `/var/mail` directory in a traditional UNIX system that uses Maildir format (one message per file). When a user receives an email, a file is created, written, and fsynced. When the user reads an email, another file is opened, read completely, marked as read, and fsynced. Sometimes, users also reread previously read emails. Average email size is defined as 16 KB, and only 16 threads are operating by default.

In addition to the workloads described above, Filebench comes with OLTP, Video-server, Web-proxy, and NFS-server macro-workloads and over 40 micro-workloads. It is important to recognize that workloads observed in specific environments can be significantly different from what is defined in the included WML files. This is an intrinsic problem of any benchmark. The aforementioned workloads are merely an attempt to define workloads that are logically close to reality and provide common ground for evaluating different storage systems. We encourage the community to analyze their specific workloads, define them in Filebench's WML [5], validate the resulting synthetic workloads against the original workloads, and contribute WML descriptions to Filebench.

Best Practices

In this section, we share several important considerations when using the Filebench framework. These considerations originated from many conversations that we had with Filebench users over the past seven years.

File system behavior depends heavily on the data-set size. Using Filebench terminology, performance results depend on the number and size of files in defined filesets. It is almost always necessary to adjust fileset size in accordance with the system's cache size. For example, the default data-set size for Filebench's Web-server workload is set to only 16 MB (1,000 files of 16 KB size). Such a data set often fits entirely in the memory of the majority of modern servers; therefore, without adjustments, the Web-server workload measures the file system's in-memory performance. If in-memory performance is not the real goal, then the number of files should be increased so that the total fileset size is several times larger than the available file system cache. Specific data set-to-cache ratio varies a lot from one environment to another.

Similarly, it is important to pick an appropriate duration of an experiment. By default, timed Filebench workloads run for only 60 seconds, which is not enough time to warm the cache up and cover multiple cyclic events in the system (e.g., bdflush runs every 30 seconds in Linux). Our recommendation is to monitor file system performance and other system metrics (e.g., block I/O and memory usage) during the run and ensure that the readings remain stationary for at least ten minutes. We added a `psrun` command to Filebench 1.5 that prints performance numbers periodically. Using these readings, one can plot how performance depends on time and identify when the system reaches stable state. Anecdotally, we found that such plots often allow one to detect and fix mistakes in experimental methodology early in the evaluation cycle.

As with any empirical tool, every Filebench-based experiment should be conducted several times, and some measure of the results' stability needs to be calculated (e.g., confidence interval, standard deviations). To get reproducible results it is important to bring the system to an identical state before every run. Specifically, in a majority of the cases, one needs to warm the cache up to the same state as it would be after a long run of the workload. In other words, the frequently accessed part of the data set (as identified by the storage system) should reside in the cache. Therefore it is preferable to start the workload's execution with a cold cache, wait until the cache warms up under the workload, and then, if appropriate, report performance for warm

cache only. Note, however, that regardless of whether the cache is cold or warm, in order to ensure sufficient I/O activity for a *file system* benchmark, the workload size should exceed the size of the system memory (historically it was considered at least 2×).

Furthermore, before executing an actual workload, Filebench first creates filesets, so parts of the filesets might be in memory before the actual workload runs. This might either benefit or hurt further workload operations. We recommend to drop caches between the fileset preallocation and the workload run stages. To achieve that for standard Linux file systems add

```
create fileset
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"
```

before the `run` or `psrun` commands. The `system` command allows one to execute arbitrary shell commands from WML.

Users often want to measure file system performance while varying some workload parameter. A typical example is benchmarking write or read throughput for different I/O sizes. We found it convenient to write shell scripts that generate WML files for different values of the same parameter (I/O size, in this example). It is also helpful to save any generated `.f` files along with the results so that later on one can correlate the results to the exact workload that was executed.

Future

Filebench is a powerful and very flexible tool for generating file system workloads. We encourage storage scientists, engineers, and evaluators to explore the functionalities that Filebench offers to their fullest. We plan to improve Filebench further to accommodate changing realities and user requests. Here, in conclusion, we only mention major directions of future work.

First, Filebench provides a unique platform for both quick development of new workloads and (formal or informal) standardization of workloads that are universally accepted as reflecting reality. Standardization only makes sense if a broad storage community is adequately involved. Moreover, we believe the involvement should be continuous rather than one-time because the set of widespread workloads changes over time. To that end, we plan to make further efforts to build stronger community and conduct BoF and similar meetings at storage conferences. We invite everyone interested in this direction to communicate with us [2].

Second, from the technical side, Filebench currently translates flowops to POSIX system calls only. However, the internal design of Filebench is based on flowop engines that map flowops to specific low-level interfaces. Specifically, we consider adding NFS and Object interfaces to Filebench. With the advent of very fast storage devices, overheads caused by the benchmark itself become more visible. In fact, we fixed several performance issues in Filebench over the last few years. More generally, we plan to work on the overhead control system that is integrated into Filebench itself.

Although Filebench already has rudimentary support for distributed storage systems benchmarking, it is not enough from both functionality and convenience points of view. We plan to design and implement features that will make Filebench practical for distributed system users.

Acknowledgments

We would like to acknowledge several people from the Filebench community who contributed and continue to contribute to Filebench: George Amvrosiadis is a devoted Filebench user and developer who added multiple improvements, with importing the external file system tree feature being just one of them; Sonam Mandal and Bill Jannen contributed to duplicated and entropy-based data generation; Santhosh Kumar is the individual behind custom variables; and, finally, we borrowed both expertise and code on statistical distributions from Geoff Kuenning.

References

[1] N. Agrawal, A. C. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Generating Realistic Impressions for File-System Benchmarking," in *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, 2009.

[2] Filebench: http://filebench.sf.net.

[3] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok, "Operating System Profiling via Latency Analysis," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006.

[4] G. Kuenning, Mersenne Twist Pseudorandom Number Generator Package, 2010: http://lasr.cs.ucla.edu/geoff/mtwist.html.

[5] V. Tarasov, K. S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok, "Extracting Flexible, Replayable Models from Large Block Traces," in *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, 2012.

[6] V. Tarasov, D. Hildebrand, G. Kuenning, and E. Zadok, "Virtual Machine Workloads: The Case for New Benchmarks for NAS," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '13)*, 2013.

[7] Transaction Processing Performance Council, TPC Benchmark C, Standard Specification: www.tpc.org/tpcc, 2004.

Calling SREs and Sysadmins

Register Today!

SRECon16

April 7–8, 2016 | Santa Clara, CA, USA

www.usenix.org/srecon16

Call for Participation Now Open!

SRECon16 EUROPE

July 11–13, 2016 | Dublin, Ireland

www.usenix.org/srecon16europe

Call for Participation Now Open!

LISA16

Dec. 4–9, 2016 | Boston, MA, USA

www.usenix.org/lisa16

Streaming Systems and Architectures

JAYANT SHEKHAR AND AMANDEEP KHURANA

Jayant is Principal Solutions Architect at Cloudera working with various large and small companies in various Verticals on their big data and data science use cases, architecture, algorithms, and deployments. For the past 18 months, his focus has been streaming systems and predictive analytics. Prior to Cloudera, Jayant worked at Yahoo and at eBay building big data and search platforms. jayant@cloudera.com

Amandeep is a Principal Solutions Architect at Cloudera, where he works with customers on strategizing on, architecting, and developing solutions using the Hadoop ecosystem. Amandeep has been involved with several large-scale, complex deployments and has helped customers design applications from the ground up as well as scale and operationalize existing solutions. Prior to Cloudera, Amandeep worked at Amazon Web Services. Amandeep is also the co-author of *HBase in Action*, a book on designing applications on HBase. amansk@gmail.com

Over the last few years, we have seen a disruption in the data management space. It started with innovation in the data warehousing and large-scale computing platform world. Now we are seeing a similar trend in real-time streaming systems. In this article, we survey a few open source stream processing systems and cover a sample architecture that consists of one or more of these systems, depending on the access patterns.

Data Management Systems

Data management systems have existed for decades and form a very mature industry that we all know about. Notable vendors playing in this market include Oracle, Microsoft, Teradata, IBM—some of the most valuable companies on the planet. Data is at the core of a lot of businesses, and they spend millions of dollars for systems that make it possible to ingest, store, analyze, and use data relevant to their customers, channels, and the market. Although mature, the data management industry is going through a disruption right now. This is being caused by the explosion of data being created by humans and machines owing to cheaper and more widespread connectivity. This has given rise to the entire big data movement and a plethora of open source data management frameworks that allow companies to manage data more cheaply and in a more scalable and flexible manner.

Data management systems can be broken down into different categories, depending on the criteria you pick. Databases, file systems, message queues, business intelligence tools are all part of this ecosystem and serve different purposes inside of a larger architecture that solves the business problem. One way to categorize these systems is based on whether they handle data at rest or in motion.

Data at Rest

Systems for data at rest include databases, file systems, processing engines, and grid computing systems. Most architectures for data at rest have a separate storage tier to store raw data, a compute tier to process or clean up data, and a separate database tier to store and analyze structured data sets. In some cases a single system might be performing multiple such functions. That's not necessarily an ideal architecture from a cost, scale, and performance perspective, but they do exist out there in the wild.

Data in Motion

Systems for managing data in motion include things like message queues and stream processing systems. Architectures for data in motion consist of multiple such systems wired and working together toward a desired end state. Some solutions are simply to ingest data from sources that are creating events. Others have a stream processing aspect that writes back into the same ingestion layer, creating multiple data sets that get ingested into the system managing data at rest. Others have the stream processing system as part of the ingestion pipeline so that output is written straight to the system managing data at rest. The stream processing systems could also have different characteristics and design principles.

In this article, we'll survey a few open source systems that deal with streaming data and conclude with a section on architectures that consist of one or more of these systems, depending on the access patterns that the solution is trying to address.

Streaming Systems

There are two types of streaming systems: stream ingestion systems and stream analytics systems. Stream ingestion systems are meant to capture and ingest streaming data as it gets produced, or shortly thereafter, from sources that spew out data. Stream ingestion systems capture individual or small batches of payloads at the source and transport them to the destination. Stream analytics systems, on the other hand, process data as it streams into the system. Work is done on the payloads as they become available. It does not necessarily wait for entire batches, files, or databases to get populated before processing starts. Stream ingestion systems are typically the source for the stream analytics systems. After the stream is analyzed, the output could either be put back into the ingestion system or written to a system that handles data at rest. We'll dive deeper into the following systems:

1. Kafka, a messaging system that falls under the category of stream ingestion systems per the criteria above [1].
2. Spark Streaming, a stream processing system that works with small batches of data as they come in [2].
3. Storm, a stream processing system that works with individual events as they come in [3].
4. Flink, a distributed stream processing system that builds batch processing on top of the streaming engine [4].

Kafka

Apache Kafka [1] is a publish-subscribe messaging system; it is also a distributed, partitioned, replicated commit log service. It has been designed to handle high-throughput for writes and reads of events, handle low-latency delivery of events, and handle machine failures.

Kafka is usually deployed in a cluster. Each node in the cluster is called a *broker*. A single Kafka broker can handle hundreds of megabytes of reads and writes per second from thousands of clients. The cluster can be elastically expanded without downtime.

Kafka has a core abstraction called *topics*, and each message coming in belongs to a topic. Clients sending messages to Kafka topics are called *producers*. Clients that consume data from the Kafka topics are called *consumers*. Clients can be implemented in a programming language of your choice.

Communication between the clients and the Kafka brokers is done in a language-agnostic binary TCP protocol. There are six core client request APIs.

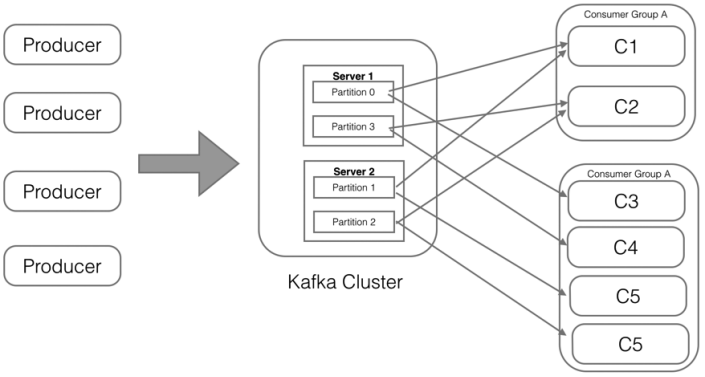


Figure 1: Kafka producers, cluster, partitions, and consumer groups

The topics are split into pre-defined *partitions*. Each partition is an ordered sequence of events that is continuously appended to a commit log. Each message in a partition is assigned a sequential event ID. In Figure 1, we have four partitions for the topic. Partitions can reside on different servers, and hence a topic can scale horizontally. Each partition can be replicated across the brokers for high availability. Messages are assigned to specific partitions by the clients and not the Kafka brokers.

Producers can round-robin between the partitions of the topic when writing to them. If there are too many producers, each producer can just write to one randomly chosen partition, resulting in far fewer connections to each broker.

Partitioning also allows different consumers to process different parts of data from the topic. For simple load balancing, the client can round-robin between the different brokers. Consumers can belong to a consumer group as shown in Figure 1, and each message is delivered to one subscribing consumer in the group.

You can batch events when writing to Kafka. This helps to increase the overall throughput of the system. Batching can also take place across topics and partitions.

Kafka stores the messages it receives to disk and also replicates them for fault-tolerance.

Apache Kafka includes Java clients and Scala clients for communicating with a Kafka cluster. It ships with a library that can be used to implement custom consumers and producers.

There are many tools that integrate with Kafka, including Spark Streaming, Storm, Flume, and Samza.

Spark Streaming

Spark Streaming [2] runs on top of the Spark [5] cluster computing framework. Spark is a batch processing system that can run in standalone mode or on top of resource management frameworks like YARN [7] or Mesos [8]. Spark Streaming is

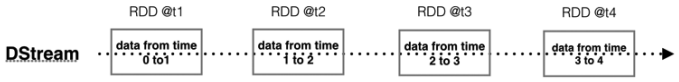


Figure 2: DStreams consists of multiple RDDs based on the time interval.

a subcomponent of the Spark project that supports processing microbatches of streams of events as they come in. Spark Streaming also supports windowing, joining streams with historical data.

Spark Streaming can ingest data from many sources, including Kafka, Flume, Kinesis, Twitter, and TCP sockets. It has inherent parallelism built in for ingesting data. The core abstraction of Spark Streaming is Discretized Streams (DStreams), which represents a continuous stream of events, created either from the incoming source or as a result of processing a source stream. Internally, DStreams consists of multiple Resilient Distributed Datasets (RDDs) [9], which are a core abstraction of the Spark project. These RDDs are created based on the time interval configured in the Spark Streaming application that defines the frequency with which the data from DStreams will be consumed by the application. A visual representation of this is shown in Figure 2.

Spark Streaming processes the data with high-level functions like `map`, `reduce`, `join`, and `window`. After processing, the resulting data can be saved on stores like HDFS, HBase, Solr, and be pushed out to be displayed in a dashboard or written back into a new Kafka topic for consumption later.

When it receives streaming data, Spark Streaming divides the data into small batches (mini batches). Each batch is stored in an RDD, and the RDDs are then processed by Spark to generate new RDDs.

Spark Streaming supports Window Operations, and it allows us to perform transformations over a sliding window of data. It takes in the window duration and the sliding interval in which the window operations are performed.

For Complex Event Processing (CEP), Spark Streaming supports stream-stream joins. Apart from inner-joins, left, right, and full outer-joins are supported. Joins over windows of streams are also supported as are stream-data set joins.

Storm

Apache Storm [3] is an open source project designed for distributed processing of streaming data at an individual event level. A Storm deployment consists of primarily two roles: a master node, called Nimbus, and the worker nodes, called Supervisors. Nimbus is the orchestrator of the work that happens in a Storm deployment. Supervisors spin up workers that execute the tasks on the nodes they are running on. Storm uses Zookeeper under the hood for the purpose of coordination and storing operational

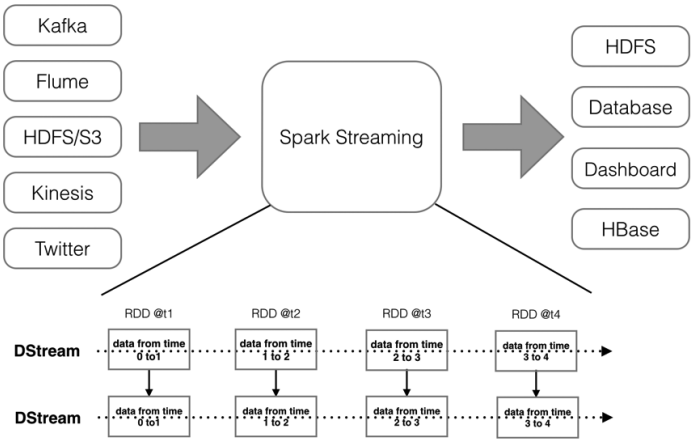


Figure 3: Diagram of Spark Streaming showing Input Data Sources, Spark DStreams, and Output Stores

state. Storing state in Zookeeper allows the Storm processes to be stateless and also have the ability to restart failed processes without affecting the health of the cluster.

Streaming applications in Storm are defined by *topologies*. These are a logical layout of the computation that the application is going to perform for the stream of data coming in. Nodes in the topology define the processing logic on the data, and links between the nodes define the movement of data. The fundamental abstraction in Storm topologies is of a Stream. Streams consist of tuples of data. Fields in a tuple could be of any type. Storm processes streams in a distributed manner. The output of this processing can be one or more streams or be put back into Kafka or a storage system or database. Storm provides two primitives to do the work on these streams—*bolts* and *spouts*. You implement bolts and spouts to create your stream processing application.

A spout is a source of the stream in the Storm topology. It consumes tuples from a stream, which could be a Kafka topic, tweets coming from the Twitter API or any other system that is emitting a stream of events.

A bolt consumes one or more streams from one or more spouts and does work on it based on the logic you've implemented. The output of a bolt could be another stream that goes into another bolt for further processing or could be persisted somewhere. Bolts can do anything from run functions, filter tuples, do streaming aggregations, do streaming joins, talk to databases, and more. A network of bolts and spouts make up a Storm topology (graphically shown in Figure 4) that is deployed on a cluster where it gets executed.

A topology keeps running until you terminate it. For each node, you can set the parallelism and Storm will spawn the required number of threads. When tasks fail, Storm automatically restarts them.

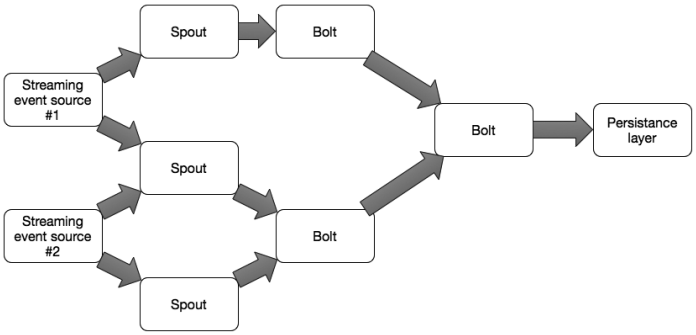


Figure 4: A Storm topology consisting of bolts and spouts

Storm provides three levels of guarantees for tuples in a stream.

- At-most-once processing: this mode is the simplest one and is appropriate in cases where it is required that a tuple be processed not more than once. Zero processing for a tuple is possible, which means message loss is acceptable in this case. If failures happen in this mode, Storm might discard tuples and not process them at all.
- At-least-once processing: this mode is where the application needs tuples to be processed at least one time. This means that more than once is acceptable. If the operations are idempotent or a slight inaccuracy in the results of the processing is acceptable, this mode would work fine.
- Exactly-once processing: this is a more complex and expensive level. Typically, an external system like Trident [6] is used for this guarantee level.

Storm provides users with a simple way to define stream processing topologies with different kinds of configurations. These make for a compelling way to implement a streaming application. Twitter recently announced a new project (Heron [10]) that learns lessons from Storm and is built to be the next generation of Storm.

Apache Flink

Apache Flink, like Spark, is a distributed stream and batch processing platform. Flink's core is a streaming dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations over data streams.

Flink uses streams for all workloads—streaming, micro-batch, and batch. Batch is treated as a finite set of streamed data.

Spark is a batch processing framework that can approximate stream processing; Flink is primarily a stream processing framework that can look like a batch processor.

At its core, Flink has an abstraction of DataStreams for streaming applications. These represent a stream of events of the same type created by consuming data from sources like Kafka, Flume, Twitter, and ZeroMQ. DataStream programs in Flink are regular programs that implement transformations on streams. Results may be written out to files, standard output, or sockets. The execution can happen in a local JVM or on clusters of many machines. Transformation operations on DataStreams include Map, FlatMap, Filter, Reduce, Fold, Aggregations, Window, WindowAll, Window Reduce, Window Fold, Window Join, Window CoGroup, Split, and some more.

Data streaming applications are executed with continuous, long-lived operators. Flink provides fault-tolerance via Lightweight Distributed Snapshots. It is based on Chandy-Lamport distributed snapshots. Streaming applications can maintain custom state during their computation. Flink's checkpointing mechanism ensures exactly-once semantics for the state in the presence of failures.

The DataStream API supports functional transformations on data streams with flexible windows. The user can define the size of the window and the frequency of reduction or aggregation calls. Windows can be based on various policies—count, time, and delta. They can also be mixed in their use. When multiple policies are used, the strictest one controls the elements in the window.

As an optimization, Flink chains two subsequent transformations and executes them within the same thread for better performance. This is done by default if it is possible, and the user doesn't have to do anything extra. Flink takes care of finding the best way of executing a program depending on the input and operations. For example, for join operations, it chooses between partitioning and broadcasting the data, between running a sort merge join and a hybrid hash join.

As you can see, Apache Flink has similar objectives as Apache Spark but different design principles. Flink is more powerful based on the design and capabilities since it can handle batch, micro-batch, and individual event-based processing, all in a single system. As it stands today, Flink is not as mature a platform as Spark and doesn't have the same momentum and user community.

Architectural Patterns

Streaming architectures often consist of multiple systems integrated with each other depending on the desired access patterns. Custom integrations happen at the following stages of a streaming pipeline.

1. Ingestion points
2. Stream processing output points

There are typically two ingestion point integrations in a typical architecture: integration of the message queue (Kafka for the context of this article) with the source system, and integration of the message queue with the stream processing system (Storm, Spark Streaming, or Flink for the context of this article).

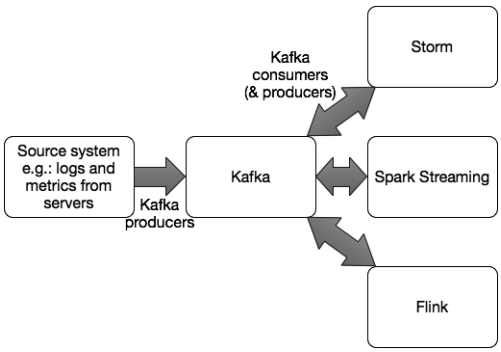


Figure 5: Streaming architecture consisting of Kafka, Storm, Spark Streaming, and Flink

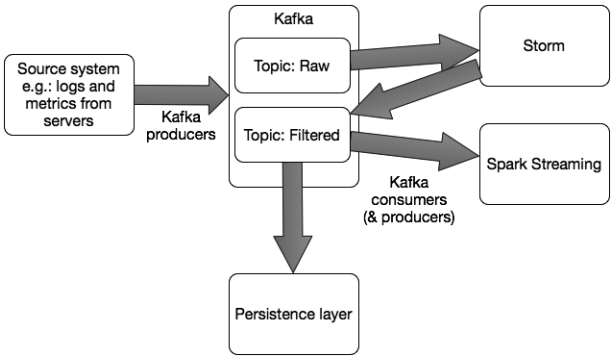


Figure 6: Streaming access pattern showing Storm processing events first, with results then processed by Spark Streaming and also persisted

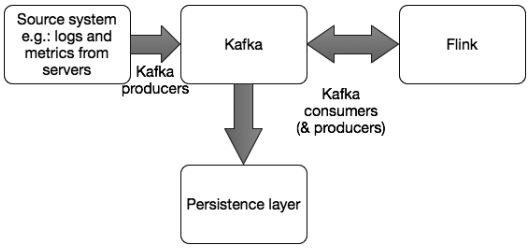


Figure 7: Streaming access pattern showing Flink doing the job of both Storm and Spark Streaming in the use case

As shown in Figure 5, the first level of integration is between the streaming event source and Kafka. This is done by writing Kafka producers that send events to Kafka. The second level of integration is between Kafka and the downstream stream processing systems. The stream processing systems consume events from Kafka, using Kafka consumers, that are written by the user. The processing systems can also write data back into Kafka by implementing Kafka producers. They write data back into Kafka if the output of the stream processing system needs to be put back into the message queue for asynchronous consumption by more than one system thereafter. This approach

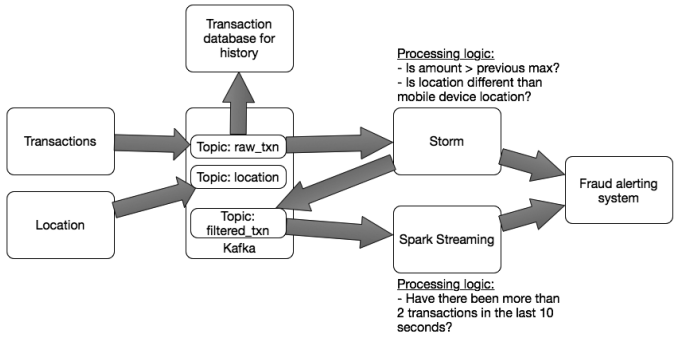


Figure 8: Streaming architecture for detecting fraudulent transactions

offers more flexibility and scalability than a tight wiring between the stream processing system and the downstream persistence layer.

In Figure 5, a possible access pattern is that Storm consumes events from Kafka first, does event-level filtering, enrichment, and alerting, with latencies below 100 ms, and writes the processed events back to Kafka in a separate Kafka topic. Thereafter, a windowing function is implemented in Spark Streaming that consumes the output of the Storm topology from Kafka. Kafka becomes the central piece of this architecture where raw data, intermediate data as well as processed data sets land. Kafka makes for a good hub for streaming data. In this case, the output of the windowing function in Spark Streaming is charted onto graphs and not necessarily persisted anywhere. The filtered events (that were output by Storm into Kafka) are what go into a downstream persistence layer like the Hadoop Distributed File System, Apache HBase, etc. That system would look as shown in Figure 6.

Flink can handle both access patterns, and the above architecture could look like Figure 7 with Flink, eliminating the need to have two downstream stream processing engines.

Let's apply this to a specific (hypothetical) use case—detecting and flagging fraudulent credit card transactions. The source streams for this use case would be the following:

- Transaction information coming in from point-of-sale devices of the merchant
- Mobile device location of the customer

For the sake of the discussion, we'll use the following definition of a fraudulent transaction. These make up the rules for our stream processing application.

1. Two or more transactions performed in a span of 10 seconds
2. Transaction amount greater than the previous max done by the given customer
3. If the mobile device location of the customer is different from the location of the transaction

To solve this use case, we need two kinds of access patterns:

1. Transaction-level processing to detect breach of rules 2 and 3
2. Detection of breach of rule 1 over a period of time, potentially across multiple transactions

You could implement this architecture as shown in Figure 8.

Note that this is a hypothetical case to show how the different systems would be used together to solve the complete problem.

Conclusion

More organizations are incorporating streaming in their data pipelines. We discussed Kafka for stream ingestion and Spark, Storm, and Flink for stream analytics. Using the right mix of streaming systems and architectures based on the use case leads to scalable and successful implementations. We hope this article provides enough information for you to select, architect, and start implementing your streaming systems.

References

[1] Apache Kafka—http://kafka.apache.org/.

[2] Apache Spark Streaming—http://spark.apache.org/streaming/.

[3] Apache Storm—http://storm.apache.org/.

[4] Apache Flink—https://flink.apache.org/.

[5] Apache Spark—https://spark.apache.org/.

[6] Trident—http://storm.apache.org/documentation/Trident-tutorial.html.

[7] Apache Hadoop YARN—https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[8] Apache Mesos—http://mesos.apache.org/.

[9] Spark RDDs—http://spark.apache.org/docs/latest/programming-guide.html#resilient-distributed-datasets-rdds.

[10] Heron stream processing system by Twitter—https://blog.twitter.com/2015/flying-faster-with-twitter-heron.

Pivot Tracing

Dynamic Causal Monitoring for Distributed Systems

JONATHAN MACE, RYAN ROELKE, AND RODRIGO FONSECA

Jonathan Mace is a PhD student in computer science at Brown University, advised by Rodrigo Fonseca. His research interests include end-to-end tracing, runtime debugging, and resource management in distributed systems. jcmace@cs.brown.edu

Ryan Roelke received a master's degree in computer science from Brown University in 2015 and is currently a Software Engineer at HP Vertica. rroelke@cs.brown.edu

Rodrigo Fonseca is an Assistant Professor at Brown University's Computer Science Department. He holds a PhD from UC Berkeley, and prior to Brown was a visiting researcher at Yahoo! Research. He is broadly interested in networking, distributed systems, and operating systems. His research involves seeking better ways to build, operate, and diagnose distributed systems, including large-scale Internet systems, cloud computing, and mobile computing. He is currently working on dynamic tracing infrastructures for these systems, on new ways to leverage network programmability, and on better ways to manage energy usage in mobile devices. rfonseca@cs.brown.edu

Pivot Tracing is a monitoring framework for distributed systems that can seamlessly correlate statistics across applications, components, and machines at runtime without needing to change or redeploy system code. Users can define and install monitoring queries on-the-fly to collect arbitrary statistics from one point in the system while being able to select, filter, and group by events meaningful at other points in the system. Pivot Tracing does not correlate cross-component events using expensive global aggregations, nor does it perform offline analysis. Instead, Pivot Tracing directly correlates events as they happen by piggybacking metadata alongside requests as they execute—even across component and machine boundaries. This gives Pivot Tracing a very low runtime overhead—less than 1% for many cross-component monitoring queries.

Monitoring and Troubleshooting Distributed Systems

Problems in distributed systems are many and varied: component failures due to hardware errors, software bugs, and misconfiguration; unexpected overload behavior due to hot spots and aggressive tenants; or simply unrealistic user expectations. Due to designs such as fault-tolerance and load balancing, the root cause of an issue may not be immediately apparent from its symptoms. However, while troubleshooting distributed systems is inherently challenging, many of the monitoring and diagnosis tools used today share two fundamental limitations that further exacerbate the challenge.

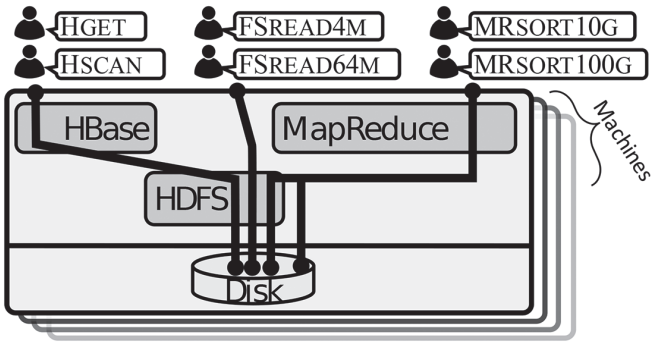
One Size Does Not Fit All

First, many tools only record information that is selected a priori at development or deployment time. Even though there has been great progress in using machine-learning techniques and static analysis to improve the quality of logs, they still carry an inherent tradeoff between recall and overhead. The choice of what to record must be made a priori, so inevitably the information needed to diagnose an issue might not be reported by the system. Even if a relevant event is captured in a log message, it can still contain too little information; similarly, performance counters may be too coarse grained or lack the desired filters or groupings.

On the other hand, if a system does expose information relevant to a problem, it is often buried under a mountain of other irrelevant information, presenting a "needle in a haystack" problem to users. Any time a user or developer patches a system to add more instrumentation, they contribute to this information overload. They also potentially add performance overheads for any monitoring that is enabled by default. Unsurprisingly, developers are resistant to adding additional metrics or groupings, as can be observed in a plethora of unresolved and rejected issues on Apache's issue trackers.

Crossing Boundaries

Second, many tools record information in a component- or machine-centric way, making it difficult to correlate events across these boundaries. Since today's datacenters typically host a wide variety of interoperating components and systems, the root cause and symptoms of an



HGET	10 kB row lookups in a large HBase table
HSCAN	4 MB table scans of a large HBase table
FSREAD4M	Random closed-loop 4 MB HDFS reads
FSREAD64M	Random closed-loop 64 MB HDFS reads
MRSORT10G	MapReduce sort job on 10 GB of input data
MRSORT100G	MapReduce sort job on 100 GB of input data

Figure 1: Six client workloads access the disks on eight cluster machines indirectly via HBase, a distributed database; HDFS, a distributed file system; and MapReduce, a data processing framework.

issue often appear in different processes, machines, and application tiers. A user of one application may need to relate information from some other dependent application in order to diagnose problems that span multiple systems. To do this manually is cumbersome, and in many cases impossible, because it depends on sufficient execution context having been propagated across software component and machine boundaries.

Dynamic Instrumentation and Causal Tracing

Pivot Tracing overcomes these challenges by combining two key techniques: dynamic instrumentation and causal tracing. Dynamic instrumentation systems, such as DTrace [1], Fay [2], and SystemTap [6], let users defer until runtime their selection of information reported by the system. They allow almost arbitrary instrumentation to be added dynamically at runtime as needed, and have proven extremely useful in diagnosing complex and unanticipated system problems. Pivot Tracing also uses dynamic instrumentation, enabling users to specify new monitoring queries at runtime. Pivot Tracing queries are dynamically installed without the need to change or redeploy code.

Dynamic instrumentation alone does not address the challenge of correlating events from multiple components. To address this challenge, Pivot Tracing adapts techniques presented in the causal tracing literature by systems such as X-Trace [3] and Dapper [7]. These systems maintain a notion of context that follows an execution through events, queues, thread pools, files, caches, and messages between distributed system components. Likewise, Pivot Tracing propagates a tracing context alongside requests. Unlike end-to-end tracing, Pivot Tracing does not record or reconstruct traces of executions for offline analysis. Instead, its tracing context is a means for propagating a small amount of state directly along the execution path of requests, including when they cross component and machine boundaries.

Pivot Tracing

Pivot Tracing exposes these two features by modeling system events as the tuples of a streaming, distributed data set. Users can write relational queries about system events using Pivot Tracing's LINQ-like query language. Pivot Tracing compiles queries into instrumentation code and dynamically installs the code at the sources of events specified in the query. Each time one of the events occurs, the instrumentation code is also invoked.

Happened-Before Join

In order to reason about causality between events, Pivot Tracing introduces a new relational operator, the "happened-before join," $\overrightarrow{\bowtie}$, for joining tuples based on Lamport's happened-before relation [4]. For events a and b occurring anywhere in the system, we say that a happened before b and write $a \rightarrow b$ if the occurrence of event a causally preceded the occurrence of event b and they occurred as part of the execution of the same request. Using the happened-before join, users can write queries that group and filter events based on properties of events that causally precede them in an execution. Pivot Tracing evaluates the happened-before join by putting partial query state into the tracing contexts propagated alongside requests. This is an efficient way to evaluate the happened-before join, because it explicitly follows the happened-before relation. It drastically mitigates the overhead and scalability issues that would otherwise be required for correlating events globally.

Pivot Tracing in Action

To motivate Pivot Tracing's design and implementation, we present a brief example of Pivot Tracing with a monitoring task in the Hadoop stack. Suppose we are managing a cluster of eight machines and want to know how disk bandwidth is being used across the cluster. On these machines, we are simultaneously running several clients with workloads in HBase, MapReduce,

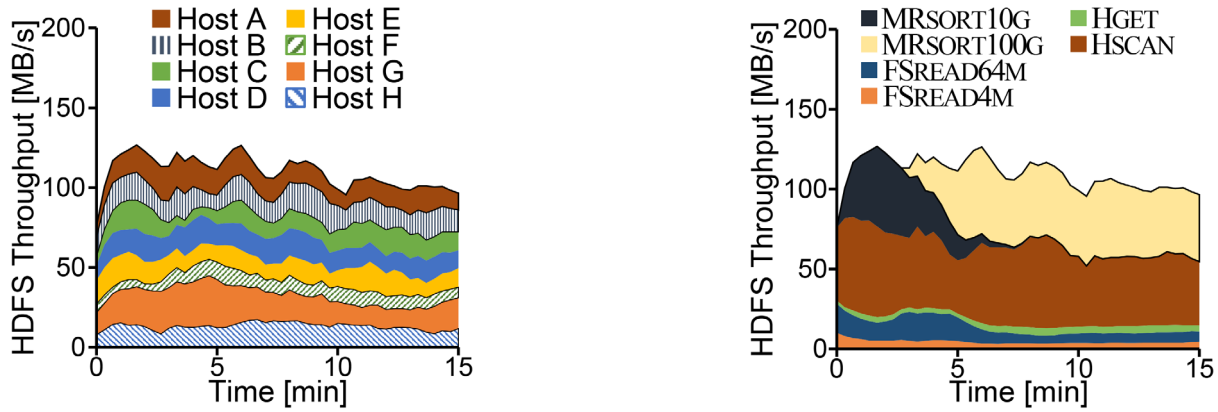


Figure 2: In this example, Pivot Tracing dynamically instruments HDFS to expose read throughput grouped by client identifiers from other applications.

and HDFS. It suffices to know that HBase is a distributed database that accesses data through HDFS, a distributed file system. MapReduce, in addition to accessing data through HDFS, also accesses the disk directly to perform external sorts and to shuffle data between tasks. Figure 1 depicts this scenario.

By default, our distributed file system HDFS already tracks some disk consumption metrics, including disk read throughput aggregated on each of its DataNodes. To reproduce this metric with Pivot Tracing, we can define a *tracepoint* for the method `incrBytesRead(int delta)` in the `DataNodeMetrics` class in HDFS. A tracepoint is a location in the application source code where instrumentation can run. We then run the following query in Pivot Tracing's LINQ-like query language:

```
Q1: From incr In DataNodeMetrics.incrBytesRead
GroupBy incr.host
Select incr.host, SUM(incr.delta)
```

This query causes each machine to aggregate the delta argument each time `incrBytesRead` is invoked, grouping by the host name. Each machine reports its local aggregate every second, from which we produce the time series in Figure 2a.

Things get more interesting if we wish to measure the HDFS usage of each of our client applications. HDFS only has visibility of its direct clients, and thus it only has an aggregate view of all HBase and all MapReduce clients. At best, applications must estimate throughput client side. With Pivot Tracing, we define tracepoints for the client protocols of HDFS (`DataTransferProtocol`), HBase (`ClientService`), and MapReduce (`ApplicationClientProtocol`), and use the name of the client process as the group-by key for the query. Figure 2b shows the global HDFS read throughput of each client application, produced by the following query:

```
Q2: From incr In DataNodeMetrics.incrBytesRead
Join cl In First(ClientProtocols) On cl -> incr
GroupBy cl.procName
Select cl.procName, SUM(incr.delta)
```

The -> symbol indicates a happened-before join. Pivot Tracing's implementation will record the process name the first time the request passes through any client protocol method and propagate it along the execution. Then, whenever the execution reaches `incrBytesRead` on a DataNode, Pivot Tracing will emit the bytes read or written, grouped by the recorded name. This query exposes information about client disk throughput that cannot currently be exposed by HDFS.

Design and Implementation

We opted to implement our Pivot Tracing prototype in Java in order to easily instrument the aforementioned open source distributed systems. However, the components of Pivot Tracing generalize and are not restricted to Java—a query can even span multiple systems written in different programming languages. Full support for Pivot Tracing in a system requires two basic mechanisms: dynamic code injection and causal metadata propagation. For full details of Pivot Tracing's design and implementation, we refer the reader to the full paper [5] and project Web site, http://pivottracing.io/.

Figure 3 presents a high-level overview of how Pivot Tracing enables queries such as Q2. We will refer to the numbers in the figure (e.g., ①) in our description.

Writing Queries

Queries in Pivot Tracing refer to variables exposed by one or more tracepoints (①)—places in the system where Pivot Tracing can insert instrumentation. Tracepoints export named variables that can be accessed by instrumentation. However, the definitions of tracepoints are not part of the system code but, rather, instructions on where and how Pivot Tracing can add instrumentation. Tracepoints in Pivot Tracing are similar to pointcuts from aspect-oriented programming and can refer to arbitrary interface/method signature combinations. Pivot Tracing's LINQ-like query language supports several typical operations including projection, selection, grouping, aggregation, and happened-before join.

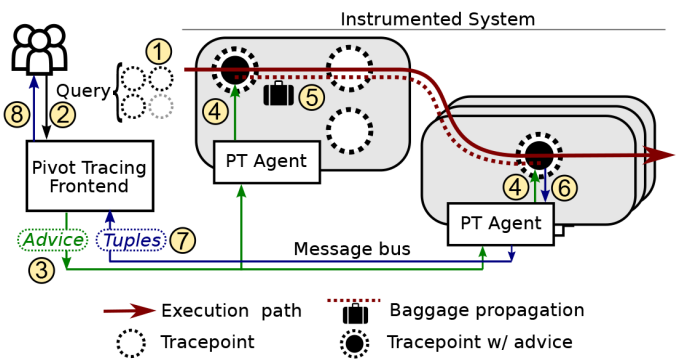


Figure 3: Pivot Tracing overview

Compiling Queries

Users submit queries to the Pivot Tracing front-end (②), which is responsible for optimizing queries using some simple static rewriting rules, pushing projection, selection, and aggregation as close as possible to the source tracepoints. The front-end then compiles queries into advice, an intermediate representation of the system-level instrumentation needed to evaluate the query. Advice specifies the operations to perform at each tracepoint used in a query.

Installing Queries

The Pivot Tracing front-end distributes advice to local Pivot Tracing agents running in each process (③). Pivot Tracing agents are responsible for dynamically instrumenting the running system so that advice is invoked at tracepoints. The agents *weave* advice into tracepoints (④) by: (1) generating code that implements the advice operations; (2) configuring the tracepoint to execute that code and pass its exported variables; (3) activating the necessary tracepoint at all locations in the system. Later, requests executing in the system will invoke the installed advice every time their execution reaches the tracepoint.

Crossing Boundaries

In order to implement the happened-before join, advice invoked at one tracepoint needs to make information available to advice invoked at other tracepoints later in a request's execution. For example, in Q2, advice at the `ClientProtocols` tracepoint needs to make its `procName` available to later advice invoked at the `DataNodeMetrics` tracepoint. This is done through Pivot Tracing's *baggage* abstraction, which uses causal metadata propagation (⑤). Baggage is a per-request container for tuples that is propagated alongside a request as it traverses thread, application, and machine boundaries. At any point in time, advice can put tuples in the baggage of the current request, and retrieve tuples that were previously placed in the baggage by other advice.

Evaluating Queries

Advice uses a small instruction set to evaluate queries and maps directly to the code that local Pivot Tracing agents generate. Advice operations are as follows: advice can create a tuple from tracepoint-exported variables (Observe); filter tuples by a predicate (Filter); and output tuples for global aggregation (Emit). Advice can put tuples in the baggage (Pack) and retrieve tuples from the baggage (Unpack). Unpacked tuples are joined to the observed tuples (i.e., if t_0 is observed and t_{u1} and t_{u2} are unpacked, then the resulting tuples are $t_0 t_{u1}$ and $t_0 t_{u2}$). Both Pack and Emit can group tuples based on matching fields and perform simple aggregations such as SUM and COUNT.

Query Results

Advice can emit tuples as output of a query using the Emit instruction (⑥). Pivot Tracing first aggregates emitted tuples locally within each process, then reports results globally at a regular interval, e.g., once per second (⑦). The Pivot Tracing front-end collects and forwards query results to the user (⑧). Process-level aggregation substantially reduces traffic for emitted tuples; Q2 is reduced from approximately 600 tuples per second to six tuples per second from host.

Pivot Tracing Example

Recall query Q2 from our earlier Hadoop example:

```
Q2: From incr In DataNodeMetrics.incrBytesRead
    Join cl In First(ClientProtocols) On cl -> incr
    GroupBy cl.procName
    Select cl.procName, SUM(incr.delta)
```

Q2 compiles to two advice specifications, A1 and A2, to be invoked at the `ClientProtocols` and `DataNodeMetrics` tracepoints, respectively:

```
A1: OBSERVE procName
    PACK procName

A2: UNPACK procName
    OBSERVE delta
    EMIT procName, SUM(delta)
```

When a request invokes any of the `ClientProtocols` methods, the instrumented code will invoke advice A1. The advice will observe the value of the `procName` variable and pack a tuple into the request's baggage, e.g., `<procName="HGet">`. The request will continue execution, carrying this tuple in its baggage. If the request subsequently invokes the `DataNodeMetrics.incrBytesRead` method, the instrumented code will invoke advice A2. The advice will unpack the previously packed `procName` and observe the local value of the delta variable, e.g., `<delta=10>`. The advice will then join the unpacked procName with the observed delta and emit the result as output, e.g., `<procName="HGet", delta=10>`. The output tuple will be aggregated with other tuples in the process's Pivot Tracing agent and included in the next interval's query results.

Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems

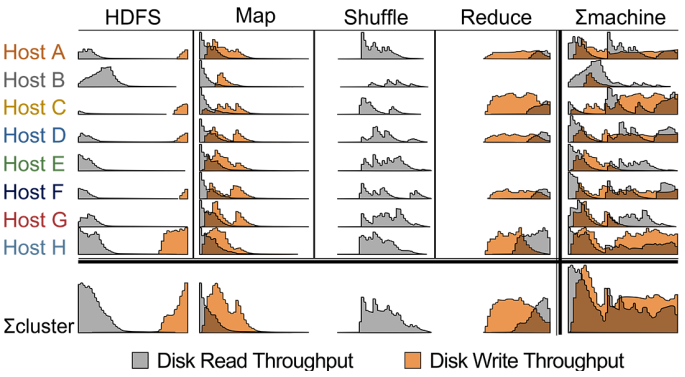


Figure 4: Pivot table showing disk read and write sparklines for MRsort10g. Rows group by host machine; columns group by source process. Bottom row and right column show totals, and bottom-right corner shows grand total.

Figure 4 gives a final demonstration of how Pivot Tracing can group metrics along arbitrary dimensions. It is generated by two queries similar to Q2 that instrument Java's `FileInputStream` and `FileOutputStream`, still joining with the client process name. We show the per-machine, per-application disk read and write throughput of `MRsort10g` from the same experiment. This figure resembles a pivot table, where summing across rows yields per-machine totals, summing across columns yields per-system totals, and the bottom-right corner shows the global totals. In this example, the client application presents a further dimension along which we could present statistics.

Summary

In this article we gave an overview of how Pivot Tracing can evaluate cross-component monitoring queries dynamically at runtime using a combination of dynamic instrumentation and causal tracing. For full details of Pivot Tracing's design and implementation, we refer the reader to the full paper [5] and project Web site. In our full evaluation, we present several case studies where we used Pivot Tracing to successfully diagnose root causes, including real-world issues we encountered in our cluster. We also evaluate the overheads imposed by Pivot Tracing, including the additional costs of invoking advice and the overheads of propagating tuples alongside requests at runtime. Of the examples presented in this article, Q2 only required the propagation of a single tuple per request, and imposed less than 1% overhead in terms of end-to-end latency on several application-level HDFS benchmarks.

Pivot Tracing is the first monitoring system to combine dynamic instrumentation with causal tracing. Its novel happened-before join operator fundamentally increases the expressive power of dynamic instrumentation and the applicability of causal tracing. Pivot Tracing enables cross-tier analysis between any interoperating applications, and the overheads of evaluating the happened-before join are sufficiently low that we believe Pivot Tracing is suitable for production systems, both for high-level standing queries and for digging deeper when necessary. Ultimately, its power lies in the uniform and ubiquitous way in which it integrates monitoring of a heterogeneous distributed system.

References

[1] Bryan Cantrill, Michael W Shapiro, and Adam H Leventhal, "Dynamic Instrumentation of Production Systems," in *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*, 2004, pp. 15–28.

[2] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz, "Fay: Extensible Distributed Tracing from Kernels to Clusters," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 4, 2012.

[3] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica, "X-Trace: A Pervasive Network Tracing Framework," in *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[4] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, 1978, pp. 558–565.

[5] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca, "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[6] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen, "Locating System Problems Using Dynamic Instrumentation," in *Proceedings of the Ottawa Linux Symposium (OLS)*, 2005.

[7] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google Research, 2010.

Doing research in the security field? Consider submitting your work to these upcoming events.

ASE '16: 2016 USENIX Workshop on Advances in Security Education

August 9, 2016, Austin, TX
Paper submissions due: May 3, 2016
www.usenix.org/ase16

The 2016 USENIX Advances in Security Education Workshop (ASE '16) is a new workshop, co-located with the 25th USENIX Security Symposium, designed to be a top-tier venue for cutting-edge research, best practices, and experimental curricula in computer security education.

CSET '16: 9th Workshop on Cyber Security Experimentation and Test

August 8, 2016, Austin, TX
Submissions due: May 3, 2016
www.usenix.org/cset16

The CSET workshop invites submissions on cyber security evaluation, experimentation, measurement, metrics, data, simulations, and testbeds.

SOUPS 2016: Twelfth Symposium on Usable Privacy and Security

June 22-24, 2016, Denver, CO
Poster submissions due: May 16, 2016
Lightning Talks and Demos early submissions due: May 16
www.usenix.org/soups2016

Posters: High-quality poster presentations are an integral part of SOUPS. We seek poster abstracts describing recent or ongoing research related to usable privacy and security. SOUPS will include a poster session in which authors will exhibit their posters. Accepted poster abstracts will be distributed to symposium participants and made available on the symposium Web site. Interactive demos alongside posters are welcome and encouraged. We also welcome authors of recent papers on usable privacy and security (2015 to 2016) to present their work at the SOUPS poster session.

Lightning Talks: A continuing feature of SOUPS is a session of 5-minute talks and 5- to 10-minute demos. These could include emerging hot topics, preliminary research results, practical problems encountered by end users or industry practitioners, a lesson learned, a research challenge that could benefit from feedback, a war story, ongoing research, a success, a failure, a future experiment, tips and tricks, a pitfall to avoid, exciting visualization, new user interface or interaction paradigm related to security and privacy. etc. Demo presentations should convey the main idea of the interface and one or more scenarios or use cases.

WOOT '16: 10th USENIX Workshop on Offensive Technologies

August 8-9, 2016, Austin, TX
Submissions due: May 17, 2016
www.usenix.org/woot16

The USENIX Workshop on Offensive Technologies (WOOT) aims to present a broad picture of offense and its contributions, bringing together researchers and practitioners in all areas of computer security. Offensive security has changed from a hobby to an industry. No longer an exercise for isolated enthusiasts, offensive security is today a large-scale operation managed by organized, capitalized actors. Meanwhile, the landscape has shifted: software used by millions is built by startups less than a year old, delivered on mobile phones and surveilled by national signals intelligence agencies.

FOCI '16: 6th USENIX Workshop on Free and Open Communications on the Internet

August 10, 2016, Austin, TX
Submissions due: May 19, 2016
www.usenix.org/foci16

The 6th USENIX Workshop on Free and Open Communications on the Internet (FOCI '16), to be held on August 8, 2016, seeks to bring together researchers and practitioners working on means to study, detect, or circumvent practices that inhibit free and open communication on the Internet.

www.usenix.org/cfp

Interview with Doug McIlroy

RIK FARROW

Doug McIlroy, now an Adjunct Professor at Dartmouth, headed a small computing research department at Bell Labs. The department hired great people to do their thing, combining theory and practice. Seven of them, including Doug, have been elected to the National Academy of Engineering. UNIX was born on his watch.
doug@cs.dartmouth.edu

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

Over the years, I've had occasion to exchange email with Doug McIlroy. I always found Doug friendly and have long wanted to interview him.

When I finally got around to asking him, Doug anticipated that I would be interested in the role he played during the early years of UNIX and pointed me to a document he wrote in the late '80s about the research versions of UNIX [1]. The first 15 pages cover a lot of the early history of UNIX, from 1969 onward, and I really wish I had had this document when I was first researching UNIX in 1982. Doug answers a lot of questions I had then, as well as solving some mysteries that I've managed to hold on to.

The full title of this work mentions "Annotated Excerpts," and most of this document is just that: sections of early UNIX manuals. When I first encountered the UNIX manuals, reading them all was actually quite possible: there were just two volumes, perhaps a stack of paper about three inches tall (excluding the binders they were in). By the late '80s, I recall that Sun Microsystems would ship two crates of documentation about SunOS: one box full of paper and the second full of binders, perhaps 20 in all. Things have only gotten more complex since then.

But early UNIX had both a simplicity and an elegance to it that persists even to this day in the command line tools. And that's where Doug played some of his biggest roles.

Rik: I read the Research UNIX Reader, and wondered if the v8 and v9 refer to commercial versions of UNIX called System III and V? I am familiar with v6 and v7 UNIX, with most people having heard of Lions' Commentary [2], which was based on v6. And v7 became the basis for BSD UNIX.

Doug: The research and commercial systems evolved separately after v7, although not without some cross-fertilization. One more research version, v10, was documented before attention turned to Plan 9 [3]. It is a shame that only some of the good ideas of Plan 9 were adopted by the UNIX community at large. Networking would be far more transparent had Plan 9's inherently distributable architecture caught on.

Rik: You mention that you were a manager, but you were also responsible for writing some code. While most of what you wrote I don't recognize, such as your compiler-compiler (TMG), other tools would likely be familiar to command line users and script writers today, like `echo`, `tr`, and `spell`.

One thing I noticed about early UNIX tools were the short names. I used to tell people, in a joking manner, that the reason for the short names was that using Teletypes [4] for command input encouraged brevity. Even the clock daemon's name was shortened (from the prefix *chron-*) to *cron*. But I am guessing there are other reasons for short names.

Doug: Typing long names is slow on any keyboard, whether teletype or smartphone. I know of no other reason for short names. Whatever regret Ken has for quirky contractions like *creat* and *cron* is fully compensated by *grep*, a euphonious coinage so useful that it made its way into the OED as both noun and verb.

I can't help noting that `vi` commands are even shorter, and are invisible to boot—too cryptic for the taste of most of us in the UNIX room, who never strayed from `ed` until `sam` came along.

Rik: You also wrote, in the Reader, that the first shell, written by Ken Thompson and Dennis Ritchie, was very simple as it had only eight kilobytes of RAM to run in. That sounds very tough, but that limitation also seems almost unbelievable. I had more usable memory in the computer I built from a kit in 1979!

Doug: The sheer fun and productivity of UNIX inveigled lucky people to switch whatever programming they could from the megabyte address-spaces available in Bell Labs computer centers to the mere 8K on PDP-11 UNIX, and forced everyone to distill projects to their essentials [5]. Remember, though, that the 8K was backed by 16K of highly useful operating system—concentrated fare that was a far cry from today's diluted offerings. What fraction of Linux's more than 450 system calls do most users know about, much less use?

Also, 8K was much bigger back then. I just rewrote `echo`. By the time it was linked in Cygwin, its 25 machine instructions had exploded into an 8K (stripped) object file. In early UNIX, it might have been a few hundred bytes.

What bigger programs could fit in 8K bytes? The assembler, for one. Also the `roff` text-formatter—an application used by secretaries as well as researchers. And B, the ace up Ken's sleeve. This word-oriented forerunner of C produced threaded code that could run with software paging, which in particular allowed B to recompile itself.

As an aside, I remember the great sense of roominess that the 2KB memory of MIT's Whirlwind II inspired after experience with the 24-word data memory of an IBM CPC.

Rik: You also had a large role in the design of pipes, a method for joining commands, so the output of one command becomes the input to the next command. Where did the idea of the pipe come from? And wasn't the original notation different from the symbol we use today?

Doug: Pipes came out of an interest in coroutines, which had fascinated me ever since Bob McClure introduced me to Melvin Conway's concept [6]. Coroutine connections look very much like I/O. This led me to write (in a 1964 Multics document) about screwing processes together like garden hose. Joe Ossanna intended to enable reconfigurable interprocess connections in Multics, but with Bell Labs' withdrawal from Multics, I believe that did not come into use.

From time to time I toyed with (unsatisfactory) syntaxes for connecting processes in a Multics-like shell; and I repeatedly suggested that UNIX should support direct interprocess I/O. Eventually, I came up with a concrete proposal for a system call with the catchy name "pipe," and a shell syntax (exemplified by command>command>file) to exploit it. This time Ken responded, "I'll do it!"

Ken did it all in one night: creating the system call, teaching the shell to use it, and fixing programs that previously handled only named files to also deal with standard input and standard output. Pipes were an instant success. Subsequently, Ken polished the implementation by introducing the distinctive pipe symbol, "|", and revising details of the system call.

Pipes hit a design sweet spot. The world is generally unaware today (as we were then) of an earlier and more ambitious mechanism for process-to-process I/O. "Communication files" in the Dartmouth time-sharing system allowed processes to handle the entire open-file interface. They were used to implement a few multiuser services. But communication files were too arcane to make their way into programmers' mental toolkits and were never used to enable UNIX-like pipelines. In interprocess I/O, UNIX simplicity again upstaged elaborate capability.

Confession: besides fussing around for years before finding a very simple answer, I totally failed to perceive the fact that connecting processes via pipes is logically more powerful than via stored serial files. You can replace an intermediate file with a pipe, but not always vice versa. An interactive session, such as `dc|speak` (a talking desk calculator), won't work if it has to treasure up all the output of `dc` before running speak. Bob Morris pointed this out on the very day pipes first worked. Had Ken and I been conscious of it, UNIX might have gotten some pipelike facility—perhaps not so simple—much earlier.

Resources

[1] D. M. McIlroy, "A Research UNIX Reader": https://archive.org/details/a_research_unix_reader.

[2] J. Lions, *A Commentary on the Sixth Edition UNIX Operating System,* 1977 (out of print): http://www.lemis.com/grog/Documentation/Lions/book.pdf.

[3] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom, "Plan 9 from Bell Labs," *Computing Systems,* vol. 8, no. 3, Summer 1995: https://www.usenix.org/legacy/publications/compsystems/1995/sum_pike.pdf.

[4] ASR 33 Teletype Information: http://www.pdp8.net/asr33/asr33.shtml.

[5] Gerard Holzmann, "Code Inflation," *IEEE Software*, March/April 2015: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7057573.

[6] Melvin E. Conway, "Design of a Separable Transition-Diagram Compiler," *Communications of the ACM*, vol. 6, no. 7, July 1963, pp. 396–408.

BeyondCorp
Design to Deployment at Google

BARCLAY OSBORN, JUSTIN MCWILLIAMS, BETSY BEYER, AND MAX SALTONSTALL

Barclay Osborn is a Site Reliability Engineering Manager at Google in Los Angeles. He previously worked at a variety of software, hardware, and security startups in San Diego. He holds a BA in computer science from the University of California, San Diego. barclay@google.com

Justin McWilliams is a Google Engineering Manager based in NYC. Since joining Google in 2006, he has held positions in IT Support and IT Ops Focused Software Engineering. He holds a BA from the University of Michigan, Ann Arbor. jjm@google.com

Betsy Beyer is a Technical Writer for Google Site Reliability Engineering 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

Max Saltonstall is a Program Manager for Google Corporate Engineering in New York. Since joining Google in 2011 he has worked on advertising products, internal change management, and IT externalization. He has a degree in computer science and psychology from Yale. maxsaltonstall@google.com

The goal of Google's BeyondCorp initiative is to improve our security with regard to how employees and devices access internal applications. Unlike the conventional perimeter security model, BeyondCorp doesn't gate access to services and tools based on a user's physical location or the originating network; instead, access policies are based on information about a device, its state, and its associated user. BeyondCorp considers both internal networks and external networks to be completely untrusted, and gates access to applications by dynamically asserting and enforcing levels, or "tiers," of access.

We present an overview of how Google transitioned from traditional security infrastructure to the BeyondCorp model and the challenges we faced and the lessons we learned in the process. For an architectural discussion of BeyondCorp, see [1].

Overview

As illustrated by Figure 1, the fundamental components of the BeyondCorp system include the Trust Inferer, Device Inventory Service, Access Control Engine, Access Policy, Gateways, and Resources. The following list defines each term as it is used by BeyondCorp:

- Access requirements are organized into ***Trust Tiers*** representing levels of increasing sensitivity.
- ***Resources*** are an enumeration of all the applications, services, and infrastructure that are subject to access control. Resources might include anything from online knowledge bases, to financial databases, to link-layer connectivity, to lab networks. Each resource is associated with a minimum trust tier required for access.
- The ***Trust Inferer*** is a system that continuously analyzes and annotates device state. The system sets the maximum trust tier accessible by the device and assigns the VLAN to be used by the device on the corporate network. These data are recorded in the Device Inventory Service. Reevaluations are triggered either by state changes or by a failure to receive updates from a device.
- The ***Access Policy*** is a programmatic representation of the Resources, Trust Tiers, and other predicates that must be satisfied for successful authorization.
- The ***Access Control Engine*** is a centralized policy enforcement service referenced by each gateway that provides a binary authorization decision based on the access policy, output of the Trust Inferer, the resources requested, and real-time credentials.
- At the heart of this system, the ***Device Inventory Service*** continuously collects, processes, and publishes changes about the state of known devices.
- Resources are accessed via ***Gateways***, such as SSH servers, Web proxies, or 802.1x-enabled networks. Gateways perform authorization actions, such as enforcing a minimum trust tier or assigning a VLAN.

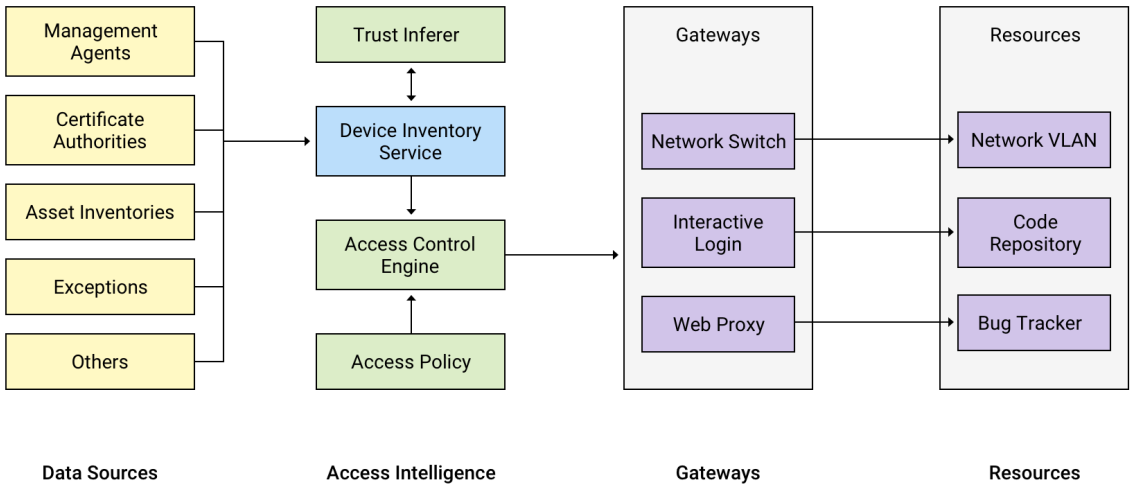


Figure 1: Architecture of the BeyondCorp Infrastructure Components

Components of BeyondCorp

Using the components described below, BeyondCorp integrated various preexisting systems with new systems and components to enable flexible and granular trust decisions.

Devices and Hosts

An inventory is the primary prerequisite to any inventory-based access control. Depending on your environment and security policy, you may need to make a concerted effort to distinguish between devices and hosts. A *device* is a collection of physical or virtual components that act as a computer, whereas a *host* is a snapshot of the state of a device at a given point in time. For example, a device might be a laptop or a mobile phone, while a host would be the specifics of the operating system and software running on that device. The Device Inventory Service contains information on devices, their associated hosts, and trust decisions for both. In the sections below, the generic term "device" can refer to either a physical device or a host, depending on the configuration of the access policy. After a basic inventory has been established, the remainder of the components discussed below can be deployed as desired in order to provide improved security, coverage, granularity, latency, and flexibility.

Tiered Access

Trust levels are organized into *tiers* and assigned to each device by the Trust Inferer. Each resource is associated with a minimum trust tier required for access. In order to access a given resource, a device's trust tier assignment must be equal to or greater than the resource's minimum trust tier requirement. To provide a simplified example, consider the use cases of various employees of a catering company: a delivery crew may only require a low tier of access to retrieve the address of a wedding, so they don't need to access more sensitive services like billing systems.

Assigning the lowest tier of access required to complete a request has several advantages: it decreases the maintenance cost associated with highly secured devices (which primarily entails the costs associated with support and productivity) and also improves the usability of the device. As a device is allowed to access more sensitive data, we require more frequent tests of user presence on the device, so the more we trust a given device, the shorter-lived its credentials. Therefore, limiting a device's trust tier to the minimum access requirement it needs means that its user is minimally interrupted. We may require installation of the latest operating system update within a few business days to retain a high trust tier, whereas devices on lower trust tiers may have slightly more relaxed timelines.

To provide another example, a laptop that's centrally managed by the company but that hasn't been connected to a network for some period of time may be out of date. If the operating system is missing some noncritical patches, trust can be downgraded to an intermediate tier, allowing access to some business applications but denying access to others. If a device is missing a critical security patch, or its antivirus software reports an infection, it may only be allowed to contact remediation services. On the furthest end of the spectrum, a known lost or stolen device can be denied access to all corporate resources.

In addition to providing tier assignments, the Trust Inferer also supports network segmentation efforts by annotating which VLANs a device may access. Network segmentation allows us to restrict access to special networks—lab and test environments, for example—based on the device state. When a device becomes

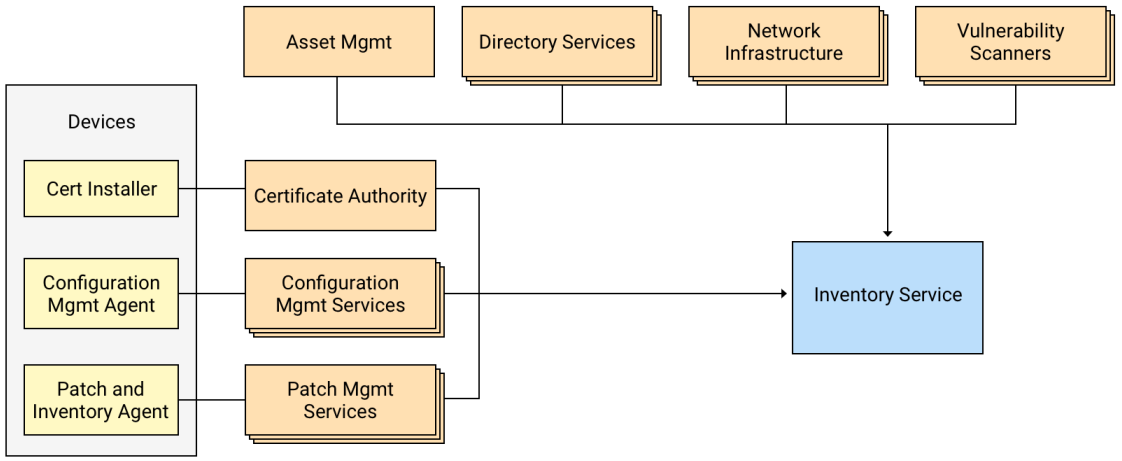


Figure 2: Device Inventory Service

untrustworthy, we can assign it to a quarantine network that provides limited resource access until the device is rehabilitated.

Device Inventory Service

The Device Inventory Service (shown in Figure 2) is a continuously updated pipeline that imports data from a broad range of sources. Systems management sources might include Active Directory, Puppet, and Simian. Other on-device agents, configuration management systems, and corporate asset management systems should also feed into this pipeline. Out-of-band data sources include vulnerability scanners, certificate authorities, and network infrastructure elements such as ARP tables. Each data source sends either full or incremental updates about devices.

Since implementing the initial phases of the Device Inventory Service, we've ingested billions of deltas from over 15 data sources, at a typical rate of about three million per day, totaling over 80 terabytes. Retaining historical data is essential in allowing us to understand the end-to-end lifecycle of a given device, track and analyze fleet-wide trends, and perform security audits and forensic investigations.

Types of Data

Data come in two main flavors: observed and prescribed.

Observed data are programmatically generated and include items such as the following:

- The last time a security scan was performed on the device, in addition to the results of the scan
- The last-synced policies and timestamp from Active Directory
- OS version and patch level
- Any installed software

Prescribed data are manually maintained by IT Operations and include the following:

- The assigned owner of the device
- Users and groups allowed to access the device
- DNS and DHCP assignments
- Explicit access to particular VLANs

Explicit assignments are required in cases of insufficient data or when a client platform isn't customizable (as is the case for printers, for example). In contrast to the change rate that characterizes observed data, prescribed data are typically static. We analyze data from numerous disparate sources to identify cases where data conflict, as opposed to blindly trusting a single or small number of systems as truth.

Data Processing

TRANSFORMATION INTO A COMMON DATA FORMAT

Several phases of processing are required to keep the Device Inventory Service up to date. First, all data must be transformed into a common data format. Some data sources, such as in-house or open source solutions, can be tooled to publish changes to the inventory system on commit. Other sources, particularly those that are third party, cannot be extended to publish changes and therefore require periodic polling to obtain updates.

CORRELATION

Once the incoming data are in a common format, all data must be correlated. During this phase, the data from distinct sources must be reconciled into unique device-specific records. When we determine that two existing records describe the same device, they are combined into a single record. While data correlation may appear straightforward, in practice it becomes quite complicated because many data sources don't share overlapping identifiers.

For example, it may be that the asset management system stores an asset ID and a device serial number, but disk encryption escrow stores a hard drive serial number, the certificate

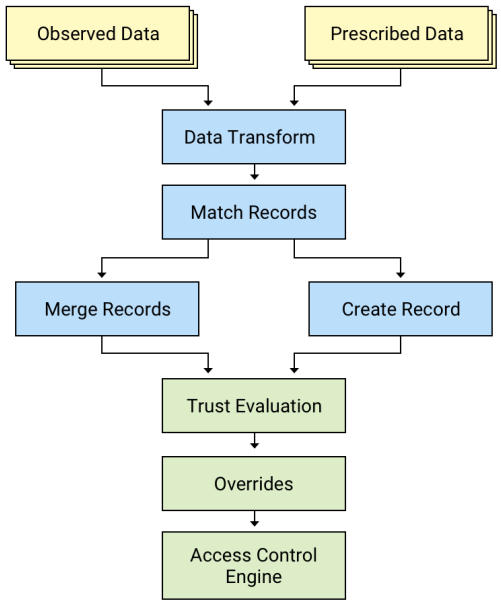


Figure 3: The data processing pipeline

authority stores a certificate fingerprint, and an ARP database stores a MAC address. It may not be clear that deltas from these individual systems describe the same device until an inventory reporting agent reports several or all of these identifiers together, at which point the disjoint records can be combined into a single record.

The question of what, exactly, constitutes a device becomes even more muddled when you factor in the entire lifecycle, during which hard drives, NICs, cases, and motherboards may be replaced or even swapped among devices. Even more complications arise if data are manually entered incorrectly.

TRUST EVALUATION

Once the incoming records are merged into an aggregate form, the Trust Inferer is notified to trigger reevaluation. This analysis references a variety of fields and aggregates the results in order to assign a trust tier. The Trust Inferer currently references dozens of fields, both platform-specific and platform-agnostic, across various data sources; millions of additional fields are available for analysis as the system continues to evolve. For example, to qualify for a high level of trust, we might require that a device meets all (or more) of the following requirements:

- Be encrypted
- Successfully execute all management and configuration agents
- Install the most recent OS security patches
- Have a consistent state of data from all input sources

This precomputation reduces the amount of data that must be pushed to the gateways, as well as the amount of computation that must be expended at access request time. This step also allows us to be confident that all of our enforcement gateways are using a consistent data set. We can make trust changes even for inactive devices at this stage. For example, in the past, we denied access for any devices that may have been subject to Stagefright [2] before such devices could even make an access request. Precomputation also provides us with an experiment framework in which we can write pre-commit tests to validate changes and canary small-percentage changes to the policy or Trust Inferer without impacting the company as a whole.

Of course, precomputation also has its downsides and can't be relied on completely. For example, the access policy may require real-time two-factor authentication, or accesses originating from known-malicious netblocks may be restricted. Somewhat surprisingly, latency between a policy or device state change and the ability of gateways to enforce this change hasn't proven problematic. Our update latency is typically less than a second. The fact that not all information is available to precompute is a more substantial concern.

EXCEPTIONS

The Trust Inferer has final say on what trust tier to apply to a given device. Trust evaluation considers preexisting exceptions in the Device Inventory Services that allow for overrides to the general access policy. Exceptions are primarily a mechanism aimed at reducing the deployment latency of policy changes or new policy primitives. In these cases, the most expedient course of action may be to immediately block a particular device that's vulnerable to a zero-day exploit before the security scanners have been updated to look for it, or to permit untrusted devices to connect to a lab network. Internet of Things devices may be handled by exceptions and placed in their own trust tier, as installing and maintaining certificates on these devices could be infeasible.

Deployment

Initial Rollout

The first phase of the BeyondCorp rollout integrated a subset of gateways with an interim meta-inventory service. This service comprised a small handful of data sources containing predominantly prescribed data. We initially implemented an access policy that mirrored Google's existing IP-based perimeter security model, and applied this new policy to untrusted devices, leaving access enforcement unchanged for devices coming from privileged networks. This strategy allowed us to safely deploy various components of the system before it was fully complete and polished and without disturbing users.

In parallel with this initial rollout, we designed, developed, and continue to iterate a higher-scale, lower-latency meta-inventory solution. This Device Inventory Service aggregates data from

over 15 sources, ingesting between 30–100 changes per second, depending on how many devices are actively generating data. It is replete with trust eligibility annotation and authorization enforcement for all corporate devices. As the meta-inventory solution progressed and we obtained more information about each device, we were able to gradually replace IP-based policies with trust tier assignments. After we verified the workflows of lower-tiered devices, we continued to apply fine-grained restrictions to higher trust tiers, proceeding to our ultimate goal of retroactively increasing trust tier requirements for devices and corporate resources over time.

Given the aforementioned complexity of correlating data from disparate sources, we decided to use an X.509 certificate as a persistent device identifier. This certificate provides us with two core functionalities:

- If the certificate changes, the device is considered a different device, even if all other identifiers remain the same.
- If the certificate is installed on a different device, the correlation logic notices both the certificate collision and the mismatch in auxiliary identifiers, and degrades the trust tiers in response.

Thus, the certificate does not remove the necessity of correlation logic; nor is it sufficient to gain access in and of itself. However, it does provide a cryptographic GUID which enforcement gateways use to both encrypt traffic and to consistently and uniquely refer to the device.

Mobile

Because Google seeks to make mobile a first-class platform, mobile must be able to accomplish the same tasks as other platforms and therefore requires the same levels of access. It turns out that deploying a tiered access model tends to be easier when it comes to mobile as compared to other platforms: mobile is typically characterized by a lack of legacy protocols and access methods, as almost all communications are exclusively HTTP-based. Android devices use cryptographically secured communications allowing identification of the device in the device inventory. Note that native applications are subject to the same authorization enforcement as resources accessed by a Web browser; this is because API endpoints also live behind proxies that are integrated with the Access Control Engine.

Legacy and Third-Party Platforms

We determined that legacy and third-party platforms need a broader set of access methods than we require for mobile devices. We support the tunneling of arbitrary TCP and UDP traffic via SSH tunnels and on-client SSL/TLS proxies. However, gateways only allow tunneled traffic that conforms with the policies laid out in the Access Control Engine. RADIUS [3] is one special case: it is also integrated with the device inventory, but it receives VLAN assignments rather than trust-tier eligibility semantics from the Trust Inferer. At network connection time, RADIUS dynamically sets the VLAN by referencing Trust Inferer assignments using the certificate presented for 802.1x as the device identifier.

Avoiding User Disruptions

One of our biggest challenges in deploying BeyondCorp was figuring out how to accomplish such a massive undertaking without disrupting users. In order to craft a strategy, we needed to identify existing workflows. From the existing workflows, we identified:

- Which workflows we could make compliant with an unprivileged network
- Which workflows either permitted more access than desirable or allowed users to circumvent restrictions that were already in place

To make these determinations, we followed a two-pronged approach. We developed a simulation pipeline that examined IP-level metadata, classified the traffic into services, and applied our proposed network security policy in our simulated environment. In addition, we translated the security policy into each platform's local firewall configuration language. While on the corporate network, this measurement allowed us to log traffic metadata destined for Google corporate services that would cease to function on an unprivileged network. We found some surprising results, such as services that had supposedly been decommissioned but were still running with no clear purpose.

After collecting this data, we worked with service owners to migrate their services to a BeyondCorp-enabled gateway. While some services were straightforward to migrate, others were more difficult and required policy exceptions. However, we made sure that all service owners were held accountable for exceptions by associating a programmatically enforced owner and expiration with each exception. As more services are updated and more users work for extended periods of time without exercising any exceptions, the users' devices can be assigned to an unprivileged VLAN. With this approach, users of noncompliant applications are not overly inconvenienced; the pressure is on the service providers and application developers to configure their services correctly.

The exceptions model has resulted in an increased level of complexity in the BeyondCorp ecosystem, and over time, the answer to "why was my access denied?" has become less obvious. Given the inventory data and real-time request data, we need to be able to ascertain why a specific request failed or succeeded at a specific point in time. The first layer of our approach in answering this question has been to craft communications to end users (warning of potential problems, and how to proceed with self-remediation or contact support) and to train IT Operations staff. We also developed a service that can analyze the Trust Inferer's

decision tree and chronological history of events affecting a device's trust tier assignment in order to propose steps for remediation. Some problems can be resolved by users themselves, without engaging support staff with elevated privileges. Users who have preserved another chain of trust are often able to self-remediate. For example, if a user believes his or her laptop has been improperly evaluated but still has a phone at a sufficient trust tier, we can forward the diagnosis request to the phone for evaluation.

Challenges and Lessons Learned

Data Quality and Correlation

Poor data quality in asset management can cause devices to unintentionally loose access to corporate resources. Typos, transposed identifiers, and missing information are all common occurrences. Such mistakes may happen when procurement teams receive asset shipments and add the assets to our systems, or may be due to errors in a manufacturer's workflow. Data quality problems also originate quite frequently during device repairs, when physical parts or components of a device are replaced or moved between devices. Such issues can corrupt device records in ways that are difficult to fix without manually inspecting the device. For example, a single device record might actually contain data for two unique devices, but automatically fixing and splitting the data may require physically reconciling the asset tags and motherboard serial numbers.

The most effective solutions in this arena have been to find local workflow improvements and automated input validation that can catch or mitigate human error at input time. Double-entry accounting helps, but doesn't catch all cases. However, the need for highly accurate inventory data in order to make correct trust evaluations forces a renewed focus on inventory data quality. Our data are the most accurate they've ever been, and this accuracy has had secondary security benefits. For example, the percentage of our fleet that is updated with the latest security patches has increased.

Sparse Data Sets

As mentioned previously, upstream data sources don't necessarily share overlapping device identifiers. To enumerate a few potential scenarios: new devices might have asset tags but no hostnames; the hard drive serial might be associated with different motherboard serials at different stages in the device lifecycle; or MAC addresses might collide. A reasonably small set of heuristics can correlate the majority of deltas from a subset of data sources. However, in order to drive accuracy closer to 100%, you need an extremely complex set of heuristics to account for a seemingly endless number of edge cases. A tiny fraction of devices with mismatched data can potentially lock hundreds or even thousands of employees out of applications they need to be productive. In order to mitigate such scenarios, we monitor and verify that a set of synthetic records in our production pipeline, crafted to verify trust evaluation paths, result in the expected trust tier results.

Pipeline Latency

Since the Device Inventory Service ingests data from several disparate data sources, each source requires a unique implementation. Sources that were developed in-house or are based on open source tools are generally straightforward to extend in order to asynchronously publish deltas to our existing pipeline. Other sources must be periodically polled, which requires striking a balance between frequency of polling and the resulting server load. Even though delivery to gateways typically takes less than a second, when polling is required, changes might take several minutes to register. In addition, pipeline processing can add latency of its own. Therefore, data propagation needs to be streamlined.

Communication

Fundamental changes to the security infrastructure can potentially adversely affect the productivity of the entire company's workforce. It's important to communicate the impact, symptoms, and available remediation options to users, but it can be difficult to find the balance between over-communication and under-communication. Under-communication results in surprised and confused users, inefficient remediation, and untenable operational load on the IT support staff. Over-communication is also problematic: change-resistant users tend to overestimate the impact of changes and attempt to seek unnecessary exemptions. Overly frequent communication can also inure users to potentially impactful changes. Finally, as Google's corporate infrastructure is evolving in many unrelated ways, it's easy for users to conflate access issues with other ongoing efforts, which also slows remediation efforts and increases the operational load on support staff.

Disaster Recovery

Since the composition of the BeyondCorp infrastructure is non-trivial, and a catastrophic failure could prevent even support staff from accessing the tools and systems needed for recovery, we built various fail-safes into the system. In addition to monitoring for potential or manifested unexpected changes in the assignment of trust tiers, we've leveraged some of our existing disaster recovery practices to help ensure that BeyondCorp will still function in the event of a catastrophic emergency. Our disaster recovery protocol relies on a minimal set of dependencies and allows an extremely small subset of privileged maintainers to replay an audit log of inventory changes in order to restore a previously known good state of device inventory state and trust evaluations. We also have the ability in an emergency

to push fine-grained changes to the access policy that allow maintainers to bootstrap a recovery process.

Next Steps

As with any large-scale effort, some of the challenges we faced in deploying BeyondCorp were anticipated while others were not. An increasing number of teams at Google are finding new and interesting ways to integrate with our systems, providing us with more detailed and layered protections against malicious actors. We believe that BeyondCorp has substantially improved the security posture of Google without sacrificing usability, and has provided a flexible infrastructure that will allow us to apply authorization decisions based on policy unencumbered by technological restrictions. While BeyondCorp has been quite successful with Google systems and at Google scale, its principles and processes are also within the reach of other organizations to deploy and improve upon.

Resources

[1] Architectural discussion of BeyondCorp: http://research.google.com/pubs/pub43231.html.

[2] Stagefright: https://en.wikipedia.org/wiki/Stagefright_(bug).

[3] RADIUS: https://en.wikipedia.org/wiki/RADIUS.

Writing for *;login:*

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

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

PROPOSALS

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

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

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

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

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

UNACCEPTABLE ARTICLES

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

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

FORMAT

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

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

DEADLINES

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

COPYRIGHT

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

Talking about Talking about Cybersecurity Games

MARK GONDREE, ZACHARY N J PETERSON, AND PORTIA PUSEY

Mark Gondree is a security researcher with an interest in cybersecurity games for education and outreach. With Zachary Peterson, he co-founded 3GSE, a USENIX workshop dedicated to the use of games for security education, and released [d0x3d!], a board game about network security to promote interest and literacy in security topics among young audiences. Gondree is a Research Professor at the Naval Postgraduate School in Monterey, CA. gondree@gmail.com

Zachary Peterson is an Assistant Professor of Computer Science at Cal Poly, San Luis Obispo. He has a passion for creating new ways of engaging students of all ages in computer security, especially through the use of games and play. He has co-created numerous non-digital security games, including [d0x3d!], a network security board game, and is co-founder of 3GSE, a USENIX workshop dedicated to the use of games for security education. znjp@calpoly.edu

The recent explosion of cybersecurity games not only reflects a growing interest in the discipline broadly, but a recognition that these types of games can be entertaining as well as useful tools for outreach and education. However, cybersecurity game terminology—those terms used to describe or communicate a game's format, goals, and intended audience—can be confusing or, at worst, misleading. The result being a potential to disappoint some players, or worse, misrepresent the discipline and discourage the same populations we intend to attract. The year 2015 marked the second USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE), co-located again with the USENIX Security Symposium. At the event, we invited a community conversation about terminology for cybersecurity games. The conversation was the seed of a draft vocabulary report to be presented to the Cybersecurity Competition Federation for comment and possible adoption. In this article, we summarize some of the issues arising from that discussion.

Cybersecurity competitions are growing in both popularity and diversity. The Web site CTFtime [1] reports that there have been an average of 56 events per year since 2013; this is over one game every week. The International Capture the Flag (iCTF) competition has seen participation steadily increase, with the past five years averaging more than double the participation seen in prior years. There are at least three separate US leagues where bracketed, regional play culminates in a national competition. DARPA's Cyber Grand Challenge is the latest variation; it is "research in CTF form." During DEFCON 2016, participants will engage in a technology demonstration in a game format. In the midst of this cybersecurity game renaissance, we see designers, organizers, and researchers facing a semantic gap when describing and discussing cyber competitions.

Some terms used to describe cybersecurity games are based on analogy, sometimes stretched to where the relationship becomes weak: capture the flag (CTF), *Jeopardy*-style, quiz bowl, etc. Other terminology is invented but without wide adoption and therefore still evolving in meaning: e.g., hack-quest, inherit-and-defend, hack-a-thon. Certainly, game format can be a deciding factor for players, who may be unable to participate in person for non-virtual events, may be unable to assemble a group for team play, or may be unavailable to engage in a full-day, synchronous competition. Thus, at the very least, a common lexicon would help players and teams to identify competitions aligned with their interests and abilities.

Generating such a lexicon is non-trivial, however, as players come to games from different backgrounds, with various motivations and desired outcomes [3]. Players may be novice learners seeking to build new skills or practice learned skills. These players may only want to play if they know solutions or write-ups will be released after the event. Others may want challenges to persist after the competition, allowing players to complete them outside the competition or present their solutions to a class or study group. Experts may want harder challenges to demonstrate skills for bragging rights or increasingly large prizes.

Portia Pusey provides educational research and research development services for projects that improve our national preparedness to protect our digital infrastructure by enriching the engagement and professional skills of cybersecurity learners and professionals. Her research interests center on cybersecurity competitions as a sport and the potential of competitions to function as professional development, learning environments, and assessment. She specializes in leading the design, conducting, and performing analysis of research that strengthens practice in formal and informal cybersecurity learning situations. She also designs outreach experiences that promote cybersecurity careers and awareness for all k-career stakeholders. She is fluent in academic and technical jargon and often serves as a bridge when working with interdisciplinary academic and professional teams in technical fields. edrportia@gmail.com

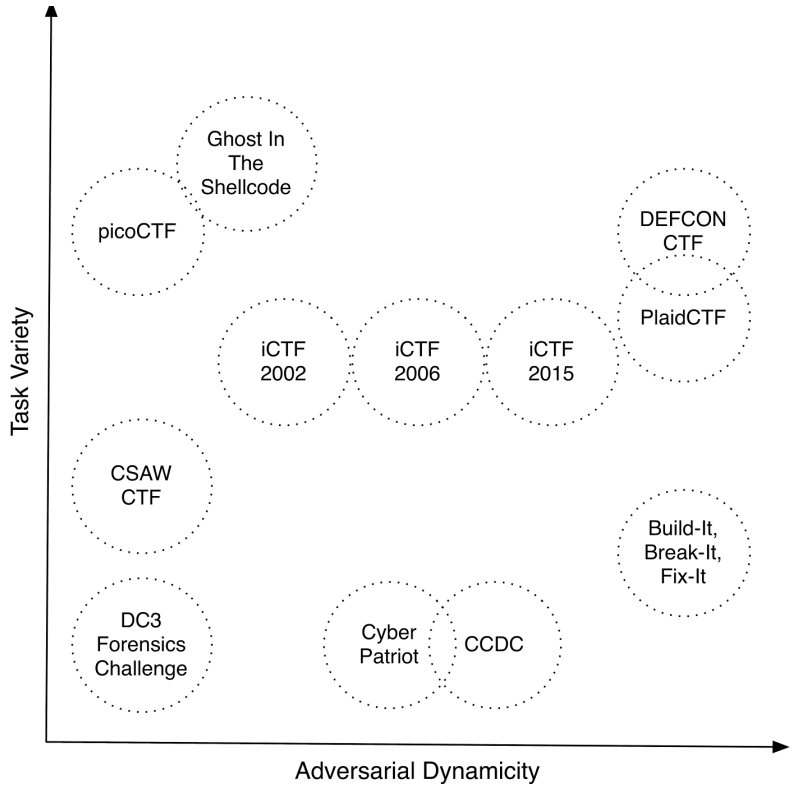


Figure 1: A common but somewhat misleading characterization of cybersecurity games, which ignores a game's intended audience, re-playability, and usefulness in an education setting—all identified as meaningful qualities by the security game community.

Taxonomies for Cybersecurity Games

No game on its own can possibly satisfy all the demands of every player. Imprecision in communicating requirements, outcomes, and mechanics means some players may not be able to identify games appropriate to their goals. To avoid player disappointment, competition Web sites sometimes identify both what they are and what they are not, clarifying where established language is imprecise and terminology is confusing. The "capture the flag" term has become especially problematic within the community; it is a powerful descriptor for a wide audience but too broad for players seeking a specific type of game or experience.

The two factors of cybersecurity games most frequently discussed, either explicitly or implicitly via comparison, are (1) whether the player will be either attacking or defending a network, service, or digital asset, and (2) whether the player will be attacking other players. While these factors are more easily characterized at their extremes, they can be imagined as a continuum, encompassing the dimensions of task variety and adversary dynamicity (see Figure 1). Task variety considers the types of knowledge, skills, and abilities players need to demonstrate during the competition. At one end of task variety are games that mix attack-defend mechanics with a variety of domain-specific challenges, typically requiring a team due to complexity and scope; at the other end are games that focus on a narrower variety of skills, like service hardening or reverse-engineering challenges. At one end of adversary dynamicity are games featuring pre-created challenges, where the game adversary's strategy is "baked" into the competition by the designer; at the other end are games where opposing players control the game adversary's strategy, allowing it to be arbitrarily complex and highly dynamic.

Characterizing games along these two dimensions, however, may be overly simplistic, artificially constraining, and misrepresent the quality of the event. Indeed, we believe all the games identified in Figure 1 are fun, effective, and enjoyable to a variety of audiences. What's more, our community discussion at 3GSE '15 highlighted that players care about many game attributes beyond these dimensions. Novice players want exercises that progressively build technical skills and self-efficacy in an environment that is unintimidating. Instructors seeking games to complement the curriculum want challenges that highlight specific learning objectives and persist after the competition ends, allowing continued use in the classroom. Designers want to develop entirely new genres that share and play with traditional CTF ideas, without fear of mischaracterizing themselves. Normative, secondary terminology could acknowledge and highlight these features, when present.

One problem with characterizations of task variety is that they tend to perpetuate a false dichotomy between attack and defense. Some games designers feel obligated to limit themselves to defense-only skills or sysadmin skill building. This may encourage some players to participate, communicating that game skills are relevant to an accessible, well-defined profession, such as "network security administrator," compared to the less understandable profession of "security consultant." This may also be to avoid any impression of "hacker training" or otherwise serving as a training ground for unethical skills. Limiting tasks in this way, however, likely underestimates the value and mischaracterizes the intent of offensive skills. As with all types of games, offensive and defensive skills are very related—some experts claim learning to attack is prerequisite to effectively defending. Learning to analyze and patch a vulnerable binary is, perhaps, an improperly structured version of the exercise in which one analyzes a binary, demonstrates how to exploit it, and then patches it. Further, characterizing games along this continuum may underemphasize essential technical and social skills exercised during the game, such as writing code in a team (e.g., Build-it, Break-it, Fix-it [3]) or reasoning about game-theoretic cost-benefit tradeoffs (e.g., 2011 iCTF's point-laundering scoring mechanism [4]).

The problem with characterizations of adversary dynamicity is that they tend to perpetuate the myth that human opponents are more dynamic, less predictable, and more skilled than the non-player adversaries encoded in challenges. Automated systems can be dynamic and arbitrarily complex. The term "adaptation" is employed for games where the obstacle is changed to challenge the player at an appropriate level, creating an experience of flow. In contrast, player adversaries could be considered "poorly designed": they can become distracted, become disengaged, be offline for significant portions of the competition, be over-skilled (or under-skilled) compared to other players, etc. The systems performing in DARPA's Cyber Grand Challenge are demonstrations, in some ways, comparable to IBM's Watson competing on *Jeopardy*. Their performance may hint, among other things, at the potential for non-player adversaries in cybersecurity games. Perhaps, in the future, some of the most dynamic, educational, fun and challenging experiences may be *Jeopardy*-style "beat the expert system" competitions.

One factor of frequent discussion for cybersecurity games is their potential relationship to education and training. Organizers are certainly designing in such opportunities, despite the lack of appropriate terminology. The NSA's Codebreaker challenge is one such example. It is a multi-month, online, *Jeopardy*-style, reverse-engineering competition where challenges are parametrized for each player. Correct solutions yield links confirming completion, making it possible for instructors to assign the challenges as extra credit and get proof of student achievement.

One might try to develop a taxonomy characterizing the role of a cybersecurity game in instruction or its placement within formal educational curricula; however, to date, games have yet to evolve into full, online courseware. Instead, it may be more appropriate to consider cybersecurity games as "informal learning spaces," like museums, libraries, and makerspaces [5]. They can be practice spaces for hands-on activities—opening up opportunities for tinkering, improvisation, failure, and sharing—in an authentic yet safe environment. They can be enriching virtual environments with embedded opportunities that teachers may leverage, while avoiding the suggestion that games supplement instruction or shoulder specific classroom goals. Just as teachers need to develop strategies to adjust instruction to get the most out of a field trip, the same may be true for cybersecurity games. Those game designers seeking to curate such an environment may benefit from lessons learned by other informal learning spaces. For example, the idea of participatory experiences and co-creative design may help designers evolve the game in response to individual and community goals [6].

While a community discussion about terminology may appear pedantic to some, it has highlighted some essential questions and core values about game objectives (which is, perhaps, a separate and similarly controversial subject). The discussion demonstrates the struggles our community faces when presenting new games to established players, designing games to reach new players, and interfacing with educators for use in clubs and classrooms. It further suggests missing research on who players are and what they need from the cybersecurity community. Ultimately, discourse that includes building a common body of terminology also will help us to be more aware of our values and goals.

ASE and the Future of 3GSE

In response to the USENIX community's interest in security education research, more broadly, the 3GSE workshop has been expanded and rebranded as the USENIX Workshop on Advances in Security Education (ASE), a new USENIX workshop designed to welcome a wider range of contributions to security education research. ASE '16 will be co-located with the 25th USENIX Security Symposium, to be held in Austin, TX in August. We hope to see you there!

Acknowledgments

The authors would like to thank the National Science Foundation for their generous contributions to 3GSE, through awards #1140561 and #1419318.

References

[1] CTFtime: ctftime.org.

[2]Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Atif Memon, Jandelyn Plane, and Piotr Mardziel, "Build It Break It: Measuring and Comparing Development Security," *Proceedings of the USENIX Workshop on Cyber Security Experimentation and Test (CSET),* 2015: https://www.usenix.org/conference/cset15/workshop-program/presentation/ruef.

[3] Masooda Bashir, Jian Ming Colin Wee, April Lambert, and Boyi Guo, "An Examination of the Vocational and Psychological Characteristics of Cybersecurity Competition Participants," *Proceedings of the USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE),* 2015: https://www.usenix.org/conference/3gse15/summit-program/presentation/bashir.

[4] Yan Shoshitaishvili, Luca Invernizzi, Adam Doupe, and Giovanni Vigna, "Do You Feel Lucky? A Large-Scale Analysis of Risk-Rewards Trade-Offs in Cyber Security," *Proceedings of the 29th Annual ACM Symposium on Applied Computing,* 2014.

[5] Andrew Richard Schrock, "'Education in Disguise': Culture of a Hacker and Maker Space," *InterActions: UCLA Journal of Education and Information Studies,* vol. 10, no. 1, 2014.

[6] Nina Simon, *The Participatory Museum,* Museum 2.0, 2010.

Interview with Lixia Zhang and kc claffy

RIK FARROW

Lixia Zhang is a Professor in the Computer Science Department of UCLA. She received her PhD in computer science from MIT, worked at Xerox PARC as a member of the research staff before joining UCLA. She has been leading the Named Data Networking (NDN) project development since 2010. lixia@cs.ucla.edu

kc claffy is founder and director of the Center for Applied Internet Data Analysis (CAIDA), a resident research scientist of the San Diego Supercomputer Center at UC, San Diego, and an Adjunct Professor in the Computer Science and Engineering Department at UC, San Diego. kc@caida.org

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

I first heard about Named Data Networking (NDN) several years ago and just couldn't get excited about it. The very notion of replacing the protocols that underpin the Internet appeared to me to be a Sisyphean task—hopeless, yet time-consuming. Yet today I find myself feeling very differently about NDN.

When I listened to kc claffy's talk during LISA15 [1], I learned three things: NDN researchers have produced working prototypes that solve real problems; NDN secures data itself, instead of securing communication channels as we do today with TLS; and the way we use TCP/IP today is nothing at all like the tasks that protocol was designed for. Let's consider the third point first.

When TCP/IP was developed in the late '70s and early '80s, computers were terribly slow. Networks were proprietary, which meant that only computers from the same vendor could communicate over networks. Computers were also tremendously expensive, making the notion of sharing them across great distances very desirable. The two goals of early sharing were the ability to log in remotely and upload or download files. With the addition of email and netnews, that's exactly what the Internet was used for—until 1994.

Today, over half of Internet traffic is streaming data, with remote login being just a tiny fraction of all traffic. File copying is still common, although a lot of files are copied using BitTorrent clients. As soon as I shared my short introduction with kc claffy and Lixia Zhang, they let me know that I was understating the current state of the Internet.

kc: Yes, and the important bit isn't that it's BitTorrent or streaming: the important bit is that *we don't care exactly where the data comes from, so long as we can verify its provenance and integrity*. So the IP network architecture forces on us something we typically don't need, a point-to-point communications abstraction, and denies us something we typically do need: data integrity/provenance mechanisms.

IP is, without a doubt, not how one would design a network architecture today to serve the current world's communication needs.

With respect to *Sisyphean task*, we recognize the project is no slam dunk, but let's also not forget that in the 1980s and even early 1990s, many people, including those employed by large telecommunications companies in particular, thought that replacing the PSTN (public switched telephone network) network architecture with something as ephemeral and unmanageable as a packet-switching architecture was the sort of pipe dream only academics could afford to pursue. Within 30 years, "impossible" became "inevitable." A thoroughly pessimistic view of this challenge ignores the empirical reality we enjoy every day.

Lixia: Related to the network architecture revolution: one needs to pay attention to the related device revolution. The proliferation of devices in recent years, most of them mobile, from cell phones to cars to billions of sensors, makes address configuration management absolutely intractable. Note that it's the computer revolution that led to packet switching, both using computers to do the switching and to connect computers to each other. A network

architecture has to fit the communication needs of the devices, and applications running on them, that the network connects.

Rik: The people who designed the IP protocols weren't aware of the types of security problems that would face the Internet [6]. When computers were rare, and the network paid for by the Department of Defense, there really wasn't a lot of concern about sharing these resources with people who couldn't be trusted.

NDN researchers have been building applications, on top of protocols, that are both data-centric and security-aware. I must admit that it was the addition of security as a major part of NDN that got my attention. Perhaps the security was there all along and I just missed it. But now, today, we could really use a network that includes scalable security as part of the base protocols.

Lixia: Yes, NDN was designed with essential security building blocks that were neither affordable nor necessary 40 years ago. The research challenge we are pursuing now is making these building blocks easy to use by app developers and end users.

Rik: After reading some of the information found on the NDN site [2], I'd like to know if NDN runs on top of IP and whether the intention is to run NDN alongside IP or to replace IP altogether.

Lixia: Designed as the new narrow waist of the Internet, NDN runs over *anything* that can move packets from one NDN node to another. Not only IP, but WiFi, Bluetooth, Ethernet, IP, UDP/TCP tunnels, or even over tin-can link as shown in this old photo [3].

The current NDN testbed runs software on commodity Linux hosts connected by IP/UDP tunnels, which is exactly how IP started: running over whatever existing communication infrastructure, which was telco wires at the time.

Rik: NDN relies on requests, called *Interests*, which name the data the client is interested in. Can you tell us more about the naming scheme?

Lixia: The easiest analogy is the HTTP request: the URL names the object the browser requests. In a URL, the early part includes the domain name information, followed by application-specific information.

URLs are coded with conventions: for example, by default an HTTP connection uses port 80. NDN namespace structures are conceptually similar, but the naming conventions matter more because applications, as well as the network itself, use names to fetch data. Our work over the last few years has included developing naming conventions to facilitate both application development and automation of signing and verification of data for a few general classes of applications.

Rik: In NDN, returned data is called *Content*, and all Content is signed, which I think is a wonderful idea. But signing relies on having a secure and manageable method of acquiring the public keys of the signing parties. How will NDN deal with this?

kc: That is an excellent and essential question, and sometimes followed by, "You've reduced NDN security to a problem we have utterly failed to solve for any application in TCP/IP: key management and distribution!"

But I think the systems and networking administration community can appreciate more than most that a data-centric security approach can convert hard security problems (e.g., host security) into relatively easier ones (crypto key management). NDN security principal investigator Alex Halderman from the University of Michigan gave a great talk on this last year [4].

Lixia: In a nutshell, securely acquiring the public keys requires one to first establish trust anchors. Today's Internet already has some ways to acquire public keys from trust anchors, i.e., using CAs (certificate agents/authority) to set up secure communication channels. There are new ways to build CAs: for example, DANE (DNS-based Authentication of Named Entities) and Let's Encrypt (https://letsencrypt.org/).

NDN is also developing new approaches to trust anchor establishments, establishing local trust anchors (e.g., the IoT management for all UCLA buildings can configure UCLA rootkey as a trust anchor), and then establishing trusts across local trust anchors. There was early work in this direction by Rivest (Simple Distributed Security Infrastructure (SDSI) [5], but today's IP architecture did not have easily available building blocks to realize it. NDN does.

Rik: The design of IP relies on relatively simple routers performing simple operations—the delivery of a packet to the next hop. NDN seems to require that routers become a lot more intelligent, in that not only must they learn routes and maintain state, they must also intelligently perform caching. Historically, TCP/IP routers have been like switches—hardware designed to pass packets out the correct interface quickly—with relatively slow CPUs to handle routing updates. Will NDN routers require serious processing support, as well as lots of storage?

Lixia: Another very good question. First, today's routers are only simple in textbooks—in the wild, they maintain ridiculous amounts of state and complexity: MPLS for traffic engineering, multicast state for multicast, VPN state for private communications, etc. Why? Because the simple and elegant IP communications model can no longer meet people's needs! For example, delivering CNN news to millions of viewers using point-to-point connections is clearly inefficient, so people want multicast. IP's single best path forwarding can't make good use of today's high-density connectivity, so people want traffic engineering; communication over the public Internet is not secure, so people need VPN; and then there is the eternal promise of QoS.

The complexity that has evolved in response to these needs is not at all elegant. As IP's simple forwarding capabilities were overtaken by the world's needs, people hashed out patch after patch in a reactive and incremental mode. We have ended up not only with many states, but many orthogonal states. Each one does its own job; not only does this not help with other goals, but their interactions may lead to more complex pictures (e.g., MPLS has to design solutions for MPLS multicast!).

Stepping up a level to look at the whole picture: it is not a question of whether data plane needs state, but what is the right data plane state that can address everyone's needs.

The datagram is still the basic unit in packet switched networks. An NDN router keeps just one forwarding state: the Pending Interest Table (PIT). Each entry in PIT records one interest packet that has been forwarded to the next hop(s) and is waiting for a reply. If another request for the same data arrives (a PIT hit), the router simply remembers this new interest's incoming interface so that it'll send a copy of data there when the data arrives. So when the requested data comes back, the router can measure the throughput and RTT in retrieving data; if the data does not come back within the expected RTT, the router can quickly try another neighbor. This router PIT provides per-datagram, per-hop state that gives a network the most flexibility to support a wide variety of functions.

Together with the Content store (cache at each NDN node), the PIT enables:

1. Loop-free, multipath data retrieval
2. Native support of synchronous and asynchronous multicast (i.e., servicing requests from multiple consumers that come at the same time or at different times)
3. Efficient recovery from packet losses (a retransmitted interest finds the data right after the lossy link)
4. Effective flow balancing (i.e., congestion avoidance by regulating how fast to forward Interests to each neighbor node)
5. Real-time recovery from network problems, such as link or node failures (i.e., reducing reliance on slowly converging routing protocols)

People have been trying for years to achieve every one of the above, in separation. NDN uses PIT to get them all in a coherent way.

Rik: Many current applications use a push model, like media streaming or video conferencing, while NDN uses a pull model. The NDN project team has a video conferencing application, ndnrtc, as well as a chat application, ChronoChat. Could you explain how NDN handles these applications using a pull model?

Lixia: The so-called "push model" is only an illusion. There is never any application that uses only one-way packet push (how would the sender know anyone received anything?). All communications are about the receivers, what receivers want (the sender does not gain anything by sending), how well the receivers get it (flow/congestion control). So a receiver has to want some data first. For example, when one wants to join a conference, the receiver sends a request and data will come. That's how all today's conference applications work. Netflix does not stream a movie to you without your request, nor does Hangout push video to your laptop before you click join. NDN does per-packet pulling—that is, one interest pulls one data packet, which leads to the advantages we talked about earlier.

Rik: The NDN project team has also created prototype applications for the Internet of Things. I am guessing that naming could be a great aid in setting up IoT networks. Can you tell us more about how this works?

Lixia: Yes indeed, NDN's use of names in building IoT systems is an ideal fit (as compared to IPv6's way of using addresses to connect IoT devices). Essentially, each IoT device, when installed, will get a name based on where it is installed and what it does, which enables devices to communicate and controllers to send commands to devices using names directly. NDN's built-in security support is also a big plus, as the security solutions for the wired Internet (TLS) really do not fit the IoT environment due to multiple factors: communication overhead, computation complexity, unreliable wireless links, and energy consumption. One writes IoT applications using meaningful names, so names are there already; it's just that today's Internet protocol stack requires a lot of IP-address-related overhead before communication can start.

Rik: I've also heard that there is a lot of interest in NDN by big data communities like meteorology and physics. Why is NDN a better fit for sharing large amounts of data than TCP/IP?

Lixia: The first and foremost factor in data sharing is data naming. If everyone names data in some arbitrary way, it won't be easy for others to figure out what data is available and where; one has to build some lookup tables to describe that file named "foobar" actually contains Los Angeles weather data for January 2016. So even before climate researchers started working with the NDN team, they already recognized their need for a common hierarchical naming structure to facilitate data sharing, so LA weather can be named something like /collectionXX/LA/2016/jan (the name includes other metainfo too).

Second, we are talking about sharing *large* amounts of data, so one certainly wants to fetch data in the most resilient and efficient manner. Resiliency means data fetching can proceed in the face of losses and partial failures. FTP cannot do it because when a TCP connection breaks mid-transfer, everything already fetched is gone (TCP semantics: a connection either successfully closed or aborted). In contrast, NDN names data packets, and every data packet arrived is received; if the current data path

failed, NDN forwarders can quickly and automatically find an alternative path if any exists. Efficiency means fetching data from the nearest location where it is available, multicast delivery with caching—all NDN's built-in functionality.

Third are data security and integrity. Even if big data may not be sensitive to privacy issues, big data users care about data integrity and provenance, and both are ensured by NDN's per data packet crypto signature. Every data packet is assigned a unique name, which ensures that everyone gets the same data if they send requests using the same name.

Rik: Anything else you'd like to say about NDN?

Lixia: We often hear people's concern about any notion of replacing IP: "See how difficult it has been to roll out IPv6, which is only a different version of IP, let alone alone a drastically different architecture."

We want to make very clear that NDN won't *change* the deployed IP infrastructure, v4 or v6, in any way. Rolling out IPv6 has been a challenge precisely because it requires *changes* to the deployed IPv4 infrastructure. NDN, on the other hand, simply makes best use out of IP delivery to move NDN packets, if direct Layer 2 channels are not available. IP got rolled out in the same way over PSTN.

If people say IP replaced PSTN, that is not because IP ever attempted to kick PSTN out, but rather because PSTN went off the stage on its own, as new applications were developed to run over IP. And all legacy applications (e.g., voice call) eventually moved over to IP as well, making IP the global communication infrastructure.

Resources

[1] kc claffy, "A Brief History of a Future Internet: The Named Data Networking Architecture" (slides), USENIX LISA15: http://www.caida.org/publications/presentations/2015/brief_history_future_internet_lisa/brief_history_future_internet_lisa.pdf.

[2] Home site for NDN: http://named-data.net/; FAQ pages: http://named-data.net/project/faq/.

[3] Jon Postel, Steve Crocker, and Vint Cerf portray TCP/IP's ability to be carried by any media, in 1994: http://www.internetsociety.org/what-we-do/grants-and-awards/awards/postel-service-award/photo-gallery.

[4] Alex Halderman, "NDN: A Security Perspective" (slides): http://named-data.net/wp-content/uploads/2015/06/fiapi-2015-security-perspective.pdf.

[5] R. Rivest, B. Lampson, "SDSI—A Simple Distributed Security Infrastructure": https://people.csail.mit.edu/rivest/sdsi10.html.

[6] Craig Timburg, "Net of Insecurity," *Washington Post,* May 30, 2015: http://www.washingtonpost.com/sf/business/2015/05/30/net-of-insecurity-part-1/.

A Brief POSIX Advocacy

Shell Script Portability

ARNAUD TOMEÏ

Arnaud Tomeï is a self-taught system administrator who first worked for the French social security administration as a consultant, where he discovered portability issues between AIX and RHEL Linux. He currently works for a hosting and services company in the south of France, specializing in Debian GNU/Linux and OpenBSD, administering 600 systems and the network. arnaud@tomei.fr

Automating things is the most important task a system administrator has to take care of, and the most practical or at least widespread way to do that is probably by writing shell scripts. But there are many flavors of shell, and their differences are a big concern when you have a heterogeneous environment and want to run the same script with the same result on every machine (that's what any sane person would expect). One option is to write POSIX-compliant shell scripts, but even the name might be confusing because POSIX normalizes a lot of UNIX-related things, from system APIs to standard commands, so I will try to clear things up.

The Bestiary of /bin/sh

One remarkable characteristic of Unixes since their beginning is the separation between the base system and the command interpreter. Beside architectural considerations, it allowed a wide diversity of programs to exist, and even to coexist on the same system with the choice given to the users on which one to use.

The first shell available was not surprisingly developed by Ken Thompson, and while it remained the default only for a couple of years, it laid the basis for the functionalities we use today: pipes, redirections, and basic substitutions. It was rapidly improved by Steve Bourne [1] in 1977 and developed into the now widely known Bourne shell. But another competing implementation was released in 1978, the C shell, written by Bill Joy to be closer to the C syntax and to have more interactive features (history, aliases, etc.). Sadly, those two syntaxes were incompatible.

That's when the Korn shell emerged; developed by David Korn and announced at the 1983 summer USENIX conference, it was backward-compatible with the Bourne shell syntax and included a lot of the interactive features from the C shell. Those two main characteristics made ksh the default shell on many commercial versions of UNIX, and made it widely known and used. No major alternative shell was written, and a stable base was reached with the release of ksh88. A new version was shipped in 1993, ksh93, which brought associative arrays and extensibility of built-in commands. Due to its popularity, the Korn shell has seen a lot of forks, including the "Public Domain KSH" pdksh, which shipped on OpenSolaris, most of the open source BSD variants, and even graphic-enabled versions like dtksh [2] and tksh [3].

It took until the late '80s and the beginning of the '90s to see two new shells released: bash in 1989 and zsh in 1990. The first was an effort from the GNU Project to have a free software equivalent of the Bourne shell for the GNU operating system, and the second was a student project of Paul Falstad's [4]. They are both backward-compatible with the Bourne shell but aim at providing more advanced functions and better usability.

A Step by Step Normalization

Back in 1988, the IEEE Computer Society felt the need to standardize tools and APIs to maintain compatibility between systems and started to write what was going to be commonly known as "POSIX," the IEEE Std 1003.1-1988, or ISO/IEC 9945 standard. This document defined very low-level mandatory characteristics of what could be called a UNIX, and

was the foundation of what we now know. It was further expanded to the point where four standards were necessary: POSIX.1, POSIX.1b, POSIX.1c, and POSIX.2, with even longer official denominations. The interesting part for our purposes is the 1992 revision (POSIX.2 also known as IEEE Std 1003.2-1992), which defined the behavior of the shell and the syntax of the scripting language. This norm is based on what was the most available shell at the time which, given the time frame, was still ksh88.

All those standards were finally merged as the result of a vendor consortium (if you thought it was already complex, search for The Open Group history) into one document in 1994: the Single UNIX Specification. The standards have since all become available under the same IEEE Std 1003.1 standard, divided into four sections. The shell scripting language is defined by the XCU chapter, along with standard tools (e.g., grep, sed, or cut) with their options, and those specifications are now maintained both by the IEEE Computer Society and by The Open Group.

Testing Code Portability

Modern shells like bash, zsh, or ksh will all be able to run POSIX-compatible scripts with no modifications, but will not fail when facing nonstandard options or constructs. For example, bash has a POSIX-compatibility mode that can be triggered in three different ways: calling it directly with the `--posix` argument, setting the `POSIXLY_CORRECT` environment variable, and calling `set -o posix` in an interactive session; none of these methods, however, will cause bash to fail to run a script containing a test between double brackets, a bash-only construct, or use the `-n` argument for `echo`. Reading the full XCU specification before writing a script is not even remotely conceivable: the specification's table of contents alone is already 4867 lines long (I'm serious) [5].

> Although setting the `POSIXLY_CORRECT` variable will not make bash behave as a strictly POSIX shell, it will enable other GNU tools like df or tar to use 512-byte blocks (as specified by the norm) instead of one kilobyte by default, which might be useful for a backup script designed to run between Linux and BSD, for example.

Installing all available shells and running the intended script with all of them might sound crazy but is a serious option if you want to look after really specific cases where strict POSIX compliance is not mandatory but portability is.

But for a more generic situation, using a minimal Bourne-compatible shell is a quicker solution: if you are using Debian or a derivative you can use dash, which is installed by default now, or even install posh (Policy-compliant Ordinary SHell) to test the script against, as they will exit with an error when encountering a nonstandard syntax. On almost all other systems (e.g., AIX, HP-UX, *BSD, and Solaris/Illumos), a ksh derivative will be available. Since the XCU standard was written when ksh88 was the most widespread interpreter, chances are that your script will be well interpreted on most platforms if it runs with ksh: granted it is ksh88 and this might not be the case on all systems.

One other option, coming again from the Debian project, is the Perl script `checkbashisms` [6], originally designed to help the transition of the default system shell from bash to dash. It allows for some exceptions by default, as it checks for conformance against the Debian policy [7] first (which allows `echo -n`, for example), but can be forced to be strictly POSIX:

```
$ checkbashisms --posix duplicate-fronted.sh
possible bashism in duplicate-frontend.sh line 144 (echo -n):
      echo -n "Updating server list ..."
possible bashism in duplicate-frontend.sh line 157 (brace
expansion):
mkdir -p $wwwpath/{www,log,stats}
[...]
```

`checkbashisms` has one big limitation, however: it does not check for external tools and their arguments, which can be nonportable.

Finally, there is Shellcheck [8], a tool that does a lot more than just checking portability but also warns you about stylistic errors, always true conditions, and even possible catastrophic mistakes (`rm $VAR/*`). Shellcheck also has an online version with a form to submit the script if you don't want to install the Haskell dependencies required to run Shellcheck.

Built-ins

Some of the errors the previous tools would point out are frequently part of the shell syntax itself, which is often extended for ease of use, but at the expense of compatibility.

read

The `-p` option of `read` is a good example of an extended shell built-in that is frequently used in interactive scripts to give input context to the user:

```
read -p "Enter username: " username
echo "$username"
```

On bash or zsh, it would output something like this:

```
$ ./test.sh
Enter username: foo
foo
```

But it will fail on dash, posh, or ksh because `-p` is not available:

```
$ ./test.sh
read: invalid option -- 'p'
```

Another nonstandard extension of `read` is the special variable `$REPLY`, which contains user input if no variable name is provided:

```
read -p "Enter username: "
echo "$REPLY"
```

This code will also fail on other interpreters:

```
$ ./test.sh
test.sh:2: read: mandatory argument is missing
```

A better version of the above examples would be to use `printf` and explicitly name the variable:

```
printf "Enter username: "
read username
```

Which will give the same output as `read -p` on all shells.

echo

On the last example given with `read`, an alternative would have been to use the following code:

```
echo -n "Enter username: "
read username
```

because `echo -n` does not output a newline. But this option is not portable either, and interpreters on which it is available will likely support the `-p` option of `read`. Actually, the POSIX `echo` does not support any option: as stated by The Open Group, "Implementations shall not support any options."

Some operands are supported, however, and a workaround to suppress the newline would be to insert `\c` at the end: `echo` immediately stops outputting as soon as it reads this operand. But this method, although POSIX-compliant, is not portable either, at least with bash and zsh (only when zsh is called as /bin/sh):

```
$ ./test.sh
Enter username: \c
foo
```

Those two interpreters don't process operands unless echo is followed by the `-e` option, in contradiction with the POSIX specification. That's why it's often recommended to use `printf` instead of `echo`. A rule of thumb is to use `echo` only when no option or operand is needed, or to print only one variable at a time.

getopts

Yes, with an s. Unlike `getopt`, the platform-dependent implementation, `getopts` is well defined and will behave consistently across different systems, with one big limitation: long options are not supported.

test

The `test` built-in, or `[ ]`, obviously has many useful options, too many to be listed here, but two of them were deprecated (actually they were not part of the POSIX norm but of the XSI extension) and are still in use: the binary operators AND and OR, noted `-a` and `-o`.

```
[ "$foo" = "$bar" -a -f /etc/baz ]
[ "$foo" = "$bar" -o -f /etc/baz ]
```

Because they were ambiguous, depending on their position in the expression, and could be confused by user input, they have been marked obsolescent. Moreover, they could be easily replaced by the equivalent shell operators: `&&` and `||`. Another nonportable syntax often used is the bash extended `test`, delimited by double brackets, which must also be avoided for POSIX scripts.

Don't Forget the Standard Tools

The shell language on its own would not have met its success without all the tools it can use to process files and streams. Did I mention that such tools as grep, sed, and cut are mandatory in the XCU standard? They are, and their necessary options are even listed. But we're used to some options not necessarily being available on all systems.

cut

I've never used this option, but I've seen it in others' scripts a couple of times, so I guess it is worthy to mention that `--output-delimiter` is GNU-specific:

```
cut -f 1,2 -d ':' --output-delimiter ',' foo
```

will work with GNU coreutils but will throw an error on other systems:

```
cut: unknown option -- -
```

The alternative in this case is pretty obvious and straightforward: pipe it to sed.

```
cut -f 1,2 -d ':' foo | sed -e 's/:/,/g'
```

sed

One really useful flag of `sed`, the `-i` option, is sadly not defined, and that can lead to some surprising errors even on systems supporting it: for example, a small script I wrote on my Linux machine to run on my girlfriend's Mac produced the following:

```
$ ./spectro-split.sh lipo-ctrl_1.csv
sed: 1: "lipo-ctrl_1.csv": invalid command code .
[...]
```

In the script, `sed` was used to replace the decimal mark in spectrometry raw data, for later analysis by another tool:

```
sed -i 's/,/./g' $column.txt
```

With GNU `sed`, the `-i` option takes an optional string as a suffix for a backup copy of the file being edited, but on Mac (and FreeBSD) the suffix is mandatory even if empty; here the substitution pattern was misunderstood, so I had to use this more portable (but still non-POSIX) syntax:

```
sed -i '' -e 's/,/./g' $column.txt
```

`read -p "Input:" variable`	The `-p` option is not portable. Actually, the only POSIX option to read is `-r`.
`read; echo $REPLY`	The `$REPLY` special variable is interpreter-specific and is not always available.
`echo -n Foo`	Portable `echo` does not support any option; `printf` should be preferred.
`sed -i "s/foo/bar/" file`	Although really useful, this option is not standard and behaves differently depending on the system.
`cp /etc/{passwd,shadow}`	Brace substitutions are commonly used with bash and zsh but are not available on ksh and POSIX.
`if [[ -e /tmp/random-lock ]]`	Double brackets are bash-specific.
`touch /tmp/$RANDOM.tmp`	The special variable `$RANDOM` is not available everywhere.
`if [ $var1 == $var2 ]`	String comparison takes only one equals sign. Moreover, doubling it might be interpreted as a variable (named "=") assignment, which can't be done in a test.
`foo () { local var1=bar }`	Scoped variables are not defined by the XCU. The `unset` routine might be used instead if necessary.
`foo=((foo++))`	Works only with bash, should be replaced by `foo=$((foo+1))` or `$((foo=foo+1))` when used in another expression (for example, `ls -l $((foo=foo+1))`).
`[ "$foo" = "$bar" -a -f /etc/baz ]`	Should be replaced by `([ "$foo" = "$bar" ] && [ -f /etc/baz ])`
`[ "$foo" = "$bar" -o -f /etc/baz ]`	Should be replaced by `([ "$foo" = "$bar" ] || [ -f /etc/baz ])`
`ls -1 ~/foo`	Often used in interactive sessions, the tilde should be banned from script as it is not expanded by all shells.

Table 1: A cheat-sheet with a quick check for the most common errors in shell scripts

This syntax will run, at least on Linux, Mac, FreeBSD, and OpenBSD, but it will throw an error on AIX, Solaris, and HP-UX, whose `sed` does not know the in-place editing option. An alternative would be to use `perl` if available:

```
perl -pi -e 's/,/./g' $column.txt
```

Or to rely only on POSIX tools:

```
sed -e 's/,/./g' $column.txt > $column.txt.new && mv
$column.txt.new $column.txt
```

This last option might not be prettiest, but it is the most portable and reliable: it does not require external tools or nonstandard options, and it actually does the same as the in-place argument, without risking silent corruption in case of disk space exhaustion.

sort and uniq

Sort is often used in conjunction with uniq, which can only process adjacent lines, and, contrary to most of the previous options we've seen, one of sort's options is often wrongly thought to be nonportable although it is perfectly standard and more efficient:

```
sort -u foo.txt -o bar.txt
```

which is POSIX-compliant and portable, and is more elegant than piping the result into uniq before redirecting the output.

Common Mistakes

Table 1 provides a small cheat-sheet to quickly check for the most common errors in shell scripts:

Conclusion

Even if it requires greater discipline, writing POSIX-compliant scripts, as well as knowing the syntax and the options of the tools used, is a good starting point for portability: it will produce higher quality scripts and, in some marginal cases, might even lead to better performance by using a limited but optimized interpretation. Of course, as in the echo example, even with standards some specific features can interfere, but by sticking closely to the norm, those situations will be limited and trivial to correct most of the time.

Resources

[1] http://www.unix.org/what_is_unix/history_timeline.html.

[2] dtksh was a fork able to manipulate Motif widgets and was included with the CDE desktop.

[3] tksh, like dtksh, was a fork adding graphic capabilities to ksh but with the Tk widget toolkit instead of Motif.

[4] http://zsh.sourceforge.net/FAQ/zshfaq01.html.

[5] http://pubs.opengroup.org/onlinepubs/9699919799/utilities/contents.html.

[6] http://sourceforge.net/projects/checkbaskisms/.

[7] http://www.debian.org/doc/debian-policy/ch-files.html.

[8] http://www.shellcheck.net/about.html.

System Administration in Higher Education Workshop at LISA15

JOSH SIMON

Josh Simon is a Senior Systems Administrator for the University of Michigan's College of Literature, Science, and the Arts. In addition to higher education, he's worked for hardware manufacturers, software companies, and multinational financial institutions. He's also a long-time *;login:* contributor. jss@umich.edu

The System Administration in Higher Education Workshop asked what's different and what's the same for system administration in higher education versus industry, including the challenges faced by practitioners. About half of the attendees worked in central IT at their institution as opposed to a distributed IT shop such as college (within a university) or department level. Most of us were at bigger institutions...or ones that felt bigger.

Improving Support

Our first topic of discussion was practical ideas for improving support. One manager reported that he has a team of five engineers and 12 students. Outsourcing work to students works really well except during finals week. They have about 40 to 50 products in their service portfolio, most of which are commercial off-the-shelf products, and half his staff spend most of their time maintaining them. He wants to move towards a service center model so that his engineers can be freed up to work on problems more interesting than provisioning and day-to-day operational tasks. Suggestions included providing self-service access for faculty and students to do certain provisioning tasks themselves, abstracting the work into smaller chunks, generalizing (e.g., "students" as opposed to "art students" and "engineering students"), and retiring obsolete services or combining common services in the portfolio.

Keeping staff engaged by handing off the routine stuff to the help desk helps. What are other ways to keep staff engaged? Automate (or "provide consistent service delivery for") the daily-operations tasks so that people can work on projects instead. Over time, people find their interests and that's okay. Another attendee's organization is stable: employees have been there for 35 years, so there aren't a lot of new people. Some people want to be more engaged, but the institutionalists don't want people to be engaged because that means the old-timers would have to change.

In another case, someone is unwilling to disengage when workload says he should. Someone wants to be the de facto SPOF: he's holding knowledge and won't document or disengage or relinquish tasks. This needs management buy-in and culture change (no one product owner). There may still be specialization, but information needs to be shared (e.g., SMEs are okay).

An attendee suggested switching jobs and not asking each other for help, doing the routine tasks, and just using the documentation. Mentoring was raised as another possibility.

How do we measure success and translate that into the right operational changes? Ticketing systems can provide some metrics. Surveys to faculty ("How're we doing?") can be useful, especially if repeated so that you can measure change over time. Justify new projects or products by (faculty) demand. What if your goals ("faculty: keep this guy here") conflict with the dean's ("move this FTE to Central")? If you do surveys, who gets to see the results (raw data, analysis, future actions or priorities)? Some people wish they got more negative feedback than they do, even if it's "I like this but..."—we're not perfect.

Setting Priorities

Our next discussion was on priorities. Other than incident handling, how do you prioritize what's important? It's not unique to higher education but we put a twist on it: there's no obvious budget bottom line to point to. A lot of institutions of higher learning care about teaching and research; how do you measure that?

In an ideal world, priorities would be obvious, and management would help with guidance. Our priorities should align with those of the college or university, which is usually about teaching and learning, research, and service, depending on your environment. Those areas are inherently messy and can't be planned the way "build a building" would be (which is messy but in a constrained way).

Can you abstract priorities to "my faculty, students and staff"? Not entirely. You still need to plan for end-of-life and capacity changes. Ask faculty if there'll be other changes (e.g., Java to C or whether Eclipse will go away). Remember that priorities may be different for the group (maintain stable network) and for yourself (continually learn, teach, and research in your own field). Regardless of that, you need to make sure things keep working. Build things to stay stable 24/7 in a one-person shop yet move technology ahead.

One team has a goal of stability (changing hardware or software is declined), and they do trouble tickets for issues and weekly meetings with the researchers for possible future planning. Another team is cleaning up after years of non-management.

In a department of 21 (plus students) on a four-person infrastructure team, someone went from taking direction from their boss into creating a feedback loop—providing ideas for improvement, simplifying workflows, presenting new ways of contributing (including beyond their own group).

As an ITIL teacher, the business drives the priorities, and it's based on urgency and impact in the operational work. Trouble-ticketing systems are a good start for incident handling.

For another attendee, it varies at the university (research, teaching, and patient care), college (research and teaching), department (projects based on survey results as defined by the director), and team (e.g., infrastructure) levels. Having regular one-on-one meetings is essential.

Individual priorities are yours regardless. On a professional basis, what you're prioritizing needs to align with the rest of your institution. We need to provide clear advice and recommendations to senior management for them to draw on in making decisions; we shouldn't be making decisions at our levels.

You need to be sensitive to the unwritten rules: what about those with bad histories (e.g., faculty person A has more problems than another faculty person, or there are HR issues behind the scenes)? Can VIPs be flagged in the system? If your manager is not setting your priorities, let them know what you are prioritizing.

We need to set priorities because we don't have enough resources. Kanban is a way to organize and prioritize work and can help with communications (in all directions).

Security

Next we talked about security. Universities aren't really that much like businesses. What are the unique aspects of higher ed? Some can't say "No" (e.g., "no porn" or "no Netflix"), but some rate-limiting may be useful.

Someone thought they had a security problem and hired a CISO. Their only directive is "Security." Issues of privacy are being disregarded in the name of security.

Some of the challenges: research institutions have short-timers—but IT isn't told when and where they went. How do we ensure accounts are closed when they should be? What about when credentials or machines are compromised (and three-letter agencies come for it)? How do you get those with prestigious awards to choose longer passwords without writing both ID and password on a sticky note?

Other challenges include personally owned devices ("BYOD"), application hosting (where the institution provides containers and infrastructure but the customers build their own insecure front-ends with SQL injection possibilities), worldwide collaborators (so the institution can't block countries known as threats—which won't work long-term anyway because the threats move), and senior faculty who don't want to change what they've been doing.

A policy or advisory group that meets regularly can help write the policies and make them sane and applicable *with buy-in from the relevant sources.* Have the CISO keep the chancellor or executives quiet until they're ready to act. Remember that "declare by edict" often doesn't work in academia; there's no boss-employee relationship here. We don't have the ability to tell faculty how to do things; we can make recommendations, but they are responsible for their data.

Some places are trying top-down edicts, and IT is having to dance around the push-to-centralize. "Academic freedom" is a red herring: we're not trying to prevent faculty or researchers from doing their work. In reality, a "grant" is a contract between the granting agency and the university and has requirements that may include security. Some grants have specific security requirements (including FISMO). One is "You might have to monitor logs"—but they could use that requirement to justify it for a Splunk license...to monitor those logs as well as everything else.

What's at risk? Intellectual property of research and the reputation of the researcher and institution. It seems like the lawyers and executives are finally catching up to what we understood 15 years ago about how dangerous technology can be. They seem to be much more interested in a vendor-provided solution or service, shifting the responsibility and blame (and liability) to someone else. Is that good or bad?

Some places use a combination of vendor and internal tools, VLAN segmentation, and SQL injection review. Even with all those, you have to use them correctly. Remember, though, that regardless of whether it's internal or vendor, it's still your institution's name above the fold of *The New York Times* front page.

There are some types of liability you can't get away from. Some have to store SSN, PCI, or PPI somehow. Some business processes need to be fixed (e.g., an SSN is needed in one place but stored in multiple places).

Budget

Our penultimate discussion was about budget. Some places have an adequate budget overall, some adequately budget for equipment but not for people, and some just don't adequately budget.

One place is moving towards cloud-based services like AWS. They also let their Symantec contract expire and moved to Sophos. They could move from hosting their own to AWS, which is PCI-compliant. It shifted the expense from capital expenditure (CapEx) to operational (OpEx) and freed up FTE resources internally. CapEx is almost always easier to justify than OpEx.

Do an honest analysis: Are you *the* service provider or *a* service broker? Can you manage external services? Remember you may be a customer not a provider. Recommending others' stuff instead of your own may be hard. Remember that doing customer support is hard when you're at the mercy of the third-party provider.

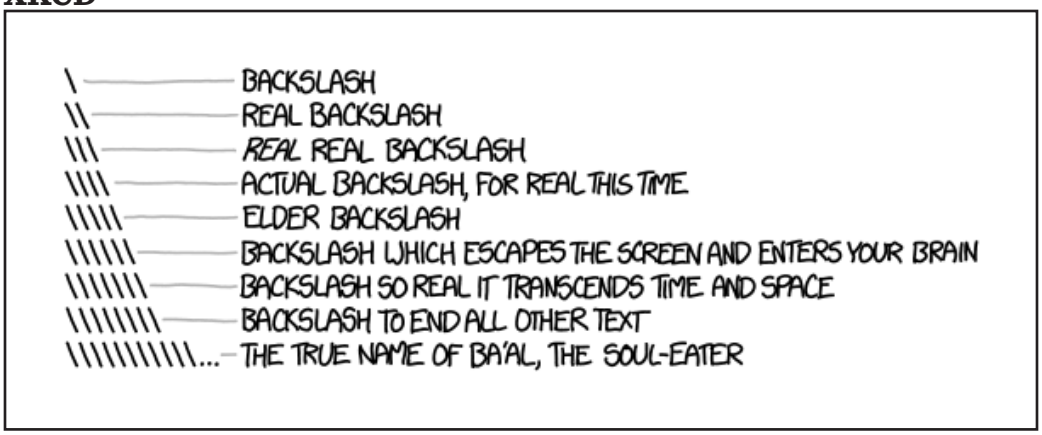
Monitoring and alerting (your internal people) is not necessarily possible when you're not the provider. How do you monitor cloud-based services? (You don't.) You may or may not have lowered your users' service level.

Handing off the "fun" stuff to the cloud and being a service broker can lead to disengagement of the IT staff. It might save time and money (at least CapEx), but it loses the staff engagement. Handing off some stuff to the cloud lets you focus on the stuff that you're keeping.

Campus Participation

Our last topic was participation on campus. We generally advised everyone to get involved on campus-level committees, both technical and nontechnical. Faculty and staff boundaries may be problematic but making the connections is very valuable. There are also off-campus activities like ACM, EduCause, IEEE, LISA, LOPSA, USENIX, and so on. Find those formal and informal networking groups that work for you and participate.

XKCD



xkcd.com

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

If your campus does not have a representative and you or someone you know would like to represent USENIX on your campus, please contact the Campus Rep Administrator, campusrep@usenix.org.

www.usenix.org/students

Crossing the Asynchronous Divide

DAVID BEAZLEY

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

The addition of improved support for asynchronous I/O in Python 3 is one of the most significant changes to the Python language since its inception. However, it also balkanizes the language and libraries into synchronous and asynchronous factions—neither of which particularly like to interact with the other. Needless to say, this presents an interesting challenge for developers writing a program involving I/O. In this article, I explore the problem of working in an environment of competing I/O models and whether or not they can be bridged in some way. As a warning, just about everything in this article is quite possibly a bad idea. Think of it as a thought experiment.

Pick a Color

I recently read an interesting blog post "What Color Is Your Function?" by Bob Nystrom [1]. I'm going to paraphrase briefly, but imagine a programming language where every function or method had to be assigned one of two colors, blue or red. Moreover, imagine that the functions were governed by some rules:

- The way in which you call a function differs according to its color.
- A red function can only be called by another red function.
- A blue function can never call a red function.
- A red function can call a blue function, but unknown bad things might happen.
- Calling a red function is much more difficult than calling a blue function.

Surely such an environment would lead to madness. What is the deal with those difficult red functions? In fact, after a bit of coding, you'd probably want to ditch all of the red code and its weird rules. Yes, you would, except for a few other details:

- Some library you're using has been written by someone who loves red functions.
- Red functions offer some advantages (i.e., concurrency, less memory required, more scalability, better performance, etc.).

Sigh. So, those red functions really are annoying. However, you're still going to have to deal with them and their weird rules in some manner.

Although this idea of coloring functions might seem like an invention of evil whimsy, it accurately reflects the emerging reality of asynchronous I/O in Python. Starting in Python 3.5, it is possible to define asynchronous functions using the `async` keyword [2]. For example:

```
async def greeting(name):
    print('Hello', name)
```

If you define such a function, it can be called from other asynchronous functions using the `await` statement.

```
async def spam():
    await greeting('Guido')
```

However, don't dare call an asynchronous function from normal Python code or from the interactive prompt—it doesn't work:

```
>>> await spam()
  File "<stdin>", line 1
    await spam()
          ^
SyntaxError: invalid syntax
>>>
```

You might think that you could make it do something if you take away the `await` statement. However, if you do that, you won't get an error—instead nothing happens at all.

```
>>> spam()
<coroutine object spam at 0x101a262b0>
>>>
```

To get an asynchronous function to run, you have to run it inside a separate execution context such as an event-loop from the `asyncio` library [3].

```
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(spam())
Hello Guido
>>>
```

The `async` functions are clearly the red functions. They have weird rules and don't play nicely with other Python code. If you're going to use them, there will be consequences. Pick a side. You're either with us or against us.

An Example

To better illustrate the divide and to put a more practical face on the problem, suppose you were writing a network application that involved some code for sending JSON-encoded objects. Maybe your code involved a function such as this:

```
import json

def send_json(sock, obj):
    data = json.dumps(obj)
    sock.sendall(data.encode('utf-8'))
```

As written here, the function has been written in a synchronous manner. You could use it in a program that uses normal functions, threads, processes, and other Python features. For example, this function waits for a connection and sends back a JSON object in response:

```
def stest():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind(('',25000))
    s.listen(1)
    c,a = s.accept()
    request = {
        'msg': 'Hello World',
        'data': 'x'
        }
    send_json(c, request)
    c.close()
    s.close()
```

To test this code, run `stest()` and connect using `nc` or `telnet`. You should see a JSON object sent back.

Now, suppose you wrote an asynchronous version of the `stest()` function:

```
import asyncio

async def atest(loop):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind(('',25000))
    s.listen(1)
    s.setblocking(False)
    c,a = await loop.sock_accept(s)
    request = {
        'msg': 'Hello World',
        'data': 'x'
        }
    send_json(c, request)       # Dicey! Danger!
    c.close()
    s.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(atest(loop))
```

In this code, you will notice that the `send_json()` function is being called directly. This is allowed by the rules (red functions can call blue functions). If you test the code, you'll find that it even appears to "work." Well, all except for the hidden time bomb lurking in the `send_json()` function.

Time bomb you say? Try changing the `request` to some large object like this:

```
request = {
    'msg': 'Hello World',
    'data': 'x'*10000000
}
```

Now, run the test again. You'll suddenly find that the program blows up in your face:

```
>>> loop.run_until_complete(atest(loop))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "python3.5/asyncio/base_events.py",
       line 342, in run_until_complete return future.result()
  File "python3.5/asyncio/futures.py", line
       274, in
       result raise self._exception
  File "python3.5/asyncio/tasks.py", line 239,
       in _step
       result = coro.send(value)
  File "j.py", line 32, in atest
    send_object(c, request)
  File "j.py", line 7, in send_object
    sock.sendall(data.encode('utf-8'))
BlockingIOError: [Errno 35] Resource temporarily unavailable
>>>
```

Silly you—that's what you get for filling up all of the I/O buffers in a function that was never safe to use in an asynchronous context. I hope you enjoy your 3:15 a.m. phone call about the whole compute cluster mysteriously going offline.

Of course, this problem can be fixed by writing a separate asynchronous implementation of the `send_json()` function. For example:

```
async def send_json(sock, obj):
    loop = asyncio.get_event_loop()
    data = json.dumps(obj)
    await loop.sock_sendall(sock, data.encode('utf-8'))
```

In your asynchronous code, you would then use this function by executing the following statement:

```
await send_json(c, request)
```

It's almost too simple—except for the fact that it's actually horrible.

Interlude: The Horror, the Horror

In the previous example, you can see how the code is forced to pick an I/O model. Interoperability between the two models isn't really possible. If you are writing a general-purpose library, you might consider supporting both I/O models by simply providing two different implementations of your code. However, that's also a pretty ugly situation to handle. Changes to one implementation would probably require changes to the other. Working with the two factions of your library is going to be a constant headache.

If you were working on a larger library or framework, you would likely find that your code base splits along the synchronous/asynchronous divide whenever I/O is involved. It would probably result in a gigantic mess. You might just abandon one of the sides altogether.

Even if you can manage to hold the whole ball of mud together in your mind, a library mixing synchronous and asynchronous code together is fraught with other problems. For example, users might forget to use the special `await` syntax in asynchronous calls. Synchronous calls executed from asynchronous functions may or may not work—with a variety of unpredictable consequences (e.g., blocking the event loop). Debugging would likely be fun.

Thought Experiment: Can You Know Your Color?

Needless to say, working in a mixed synchronous/asynchronous world has certain difficulties. However, what if functions could somehow determine the nature of the context in which they were called? Specifically, what if a function could somehow know whether it was called asynchronously?

As it turns out, this can be determined with a clever bit of devious frame hacking. Try defining this function:

```
import sys

def print_context():
    if sys._getframe(1).f_code.co_flags & 0x80:
        print('Asynchronous')
    else:
        print('Synchronous')
```

Just from the fact that the function uses a hardwired mysterious hex constant (0x80), you know that it's going to be good. Actually, you might want to ignore that part. However, try it out with this example:

```
>>> def foo():
...     print_context()
...
>>> foo()
Synchronous
>>> async def bar():
...     print_context()
...
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(bar())
Asynchronous
>>>
```

This is interesting. The function `print_context()` is a normal Python function, yet it can determine the nature of the environment from which it was called. Naturally, this raises further questions about what might be possible with such information.

For example, could you use this in various metaprogramming features such as decorators? If so, maybe you can change some of the rules. Maybe you don't have to play by the rules.

Walled Gardens

Suppose you wanted to more strongly isolate the world of synchronous and asynchronous functions to prevent errors and undefined behavior. Here is a decorator that more strictly enforces the underlying I/O model on the calling context:

```
from functools import wraps
import sys
import inspect

def strictio(func):
   # Determine if func is an async coroutine
   is_async = inspect.iscoroutinefunction(func)
   @wraps(func)
   def wrapper(*args, **kwargs):
      called_async = sys._getframe(1).f_code.co_flags & 0x80
      if is_async:
         assert called_async, "Can't call async function here"
      else:
         assert not called_async, "Can't call sync function here"
      return func(*args, **kwargs)
   return wrapper
```

To use this decorator, simply apply it to either kind of function:

```
@strictio
def foo():
   print('Synchronous')

@strictio
async def bar():
   print('Asynchronous')
```

Attempts to call these functions from the wrong context now result in an immediate assertion error. For example:

```
>>> bar()
Traceback (most recent call last):
...
AssertionError: Can't call async function here
>>>

>>> async def test():
       foo()

>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(test())
Traceback (most recent call last):
...
AssertionError: can't call sync function here
>>>
```

As you can see, this is an even stronger version of the partisan rules—crossing the asynchronous divide is simply not allowed. If you applied this to the earlier `send_json()` function, you might have been able to prevent a hidden time bomb from showing up in your code. So that's probably a good thing.

Adaptive I/O

Rather than strictly separating the two worlds, another approach might be to adapt the execution of a function to the current I/O environment. For example, consider this decorator:

```
import sys
import inspect
import asyncio
from functools import partial

def adaptiveio(func):
   is_async = inspect.iscoroutinefunction(func)
   @wraps(func)
   def wrapper(*args, **kwargs):
      called_async = sys._getframe(1).f_code.co_flags & 0x80
      if is_async and not called_async:
         # Run an async function in a synchronous context
         loop = asyncio.new_event_loop()
         return loop.run_until_complete(func(*args,
             **kwargs))
      elif not is_async and called_async:
         # Run a sync function in an asynchronous context
         loop = asyncio.get_event_loop()
         return loop.run_in_executor(None, partial(func,
             *args, **kwargs))
      else:
         return func(*args, **kwargs)
   return wrapper
```

Unlike the previous example, this decorator adapts a function to the calling context if there is a mismatch. If called from an asynchronous context, a synchronous function is executed in a separate thread using `loop.run_in_executor()`. An asynchronous function called synchronously is executed using an event loop. Let's try it:

```
@adaptiveio
def foo():
   print('Synchronous')

@adaptiveio
async def bar():
   print('Asynchronous')
```

Now, make some calls:

```
>>> foo()
Synchronous
>>> bar()          # Adapted async -> sync
Asynchronous
>>> async def test():
      await foo()    # Adapted sync -> async
      await bar()

>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(test())
Synchronous
Asynchronous
>>>
```

A possible benefit of an adapted function is that a single implementation could be used seamlessly in either a synchronous or asynchronous context (the function would just "work" regardless of where you called it). A downside is that the mismatched use case might suffer a hidden performance penalty of some kind: for instance, the extra overhead of passing an operation over to a thread-pool or in creating the event loop. Perhaps the decorator could be extended to issue a warning message if this was a concern.

A subtle feature of this decorator is that an adapted function must use the normal calling convention of the current I/O context. So, if you had this function:

```
@adaptiveio
def send_json(sock, obj):
   data = json.dumps(obj)
   sock.sendall(data.encode('utf-8'))
```

you would use `send_json()` in synchronous code, and you would use `await send_json()` in asynchronous code.

Dual Implementation

Another possible strategy might be to bind separate synchronous and asynchronous functions to a common name. The following decorator allows an "awaitable" asynchronous implementation to be attached to an existing synchronous function.

```
import sys
import inspect
from functools import wraps

def awaitable(syncfunc):
   def decorate(asyncfunc):
      assert (inspect.iscoroutinefunction(asyncfunc) and
            not inspect.iscoroutinefunction(syncfunc))
      @wraps(asyncfunc)
      def wrapper(*args, **kwargs):
         called_async = sys._getframe(1).f_code.co_flags
               & 0x80
         if called_async:
            return asyncfunc(*args, **kwargs)
         else:
            return syncfunc(*args, **kwargs)
      return wrapper
   return decorate
```

With this decorator, you write two functions as before, but give them the same name. The appropriate function is used depending on the calling context. For example:

```
>>> def spam():
...     print('Synchronous')
...
>>> @awaitable(spam)
... async def spam():
...     print('Asynchronous')
...
>>> spam()
Synchronous
>>> async def test():
...     await spam()
...
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(test())
Asynchronous
>>>
```

The main benefit of such an approach is that you could write code with a uniform API—the same function names would be used in either synchronous or asynchronous code. Of course, it doesn't solve the problem of having repetitive code. For example, the `send_json()` function would have two implementations like this:

```
def send_json(sock, obj):
   data = json.dumps(obj)
   sock.sendall(data.encode('utf-8'))

@awaitable(send_json)
async def send_json(sock, obj):
   loop = asyncio.get_event_loop()
   data = json.dumps(obj)
   await loop.sock_sendall(sock, data.encode('utf-8'))
```

Of course, all of this might just be a bad idea as heads explode while trying to figure out which function is being called during debugging. It's hard to say.

Thoughts

At this point, it's too early to tell how Python's emerging asynchronous support will play out except that libraries will likely have to side with one particular approach (asynchronous or synchronous). It seems possible that various metaprogramming techniques might be able to make the overall environment slightly more sane: for example, preventing common errors, adapting code, or making it easier to present a uniform programming interface. However, to my knowledge, this is not an approach being taken at this time.

Somewhere in the middle of this mess are libraries such as gevent [4]. gevent provides support for asynchronous programming but implements its concurrency at the level of the interpreter implementation itself (as a C extension). As a result, there is no obvious distinction between synchronous and asynchronous code—in fact, the same code can often run in both contexts. At this time, support for Python 3 in gevent is a bit new, and its whole approach runs in a different direction from the built-in `asyncio` library. Nevertheless, there's still a distinct possibility that this approach will prove to be the most sane in light of the difficulties associated with having code split into asynchronous and synchronous factions. Saying more about gevent, however, will need to be the topic of a future article.

In the meantime, if you're looking for some uncharted waters, you should definitely take a look at Python's emerging asynchronous I/O support. May your code be interesting.

References

[1] What Color Is Your Function: http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function.

[2] PEP 0492 - Coroutines with async and await syntax: https://www.python.org/dev/peps/pep-0492.

[3] asyncio module: https://docs.python.org/3/library/asyncio.html.

[4] gevent: http://www.gevent.org.

Practical Perl Tools
With Just a Little Bit of a Swagger

DAVID N. BLANK-EDELMAN

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

I've already come out as an API-phile in this column, so I suspect no one will be shocked that we're going to dive into yet another API-related topic this column. Just like the TV show where you recognize everyone in the bar (and they all know your name), we'll be back among old friends like REST and JSON. The one thing I perhaps should provide a trigger warning for is we're going to be mentioning Java in the column. If that's not your cup of you know what, then you may want to skip ahead in the magazine. If it is any comfort, we won't see any actual Java code in the column, just a bit of the tooling.

Why Are APIs Important?

Even though this column is a bit of a drinking game where every time I say "API," you drink, I don't think we've ever discussed why APIs are important. A (good) API can be seen as a contract between the person who is writing the code to provide a service and the person who is writing the code to consume that service. This is true even if that turns out to be the same person, because all you need is a little time to pass for it to be easy to forget just how two components were supposed to work together. Essentially the contract says, "If you send me a request of this form, I promise to respond (ideally with the data that was requested) in a way that you will expect." It helps to ensure the principle of least surprise, leading to (more) stable and reliable software. An API also encourages software authors to think up front about how pieces of a system should interact. I say "encourages" just because we have all dealt with an API at one time or another that wasn't as well defined or thought out as we might like.

APIs also make it possible to write "loosely coupled" components that interact only through their API, à la the microservices concept that is all the rage. I won't go into more detail here about why loosely coupled services make for a better system, but if you haven't heard that gospel yet, be sure to take some time to look up "microservices." I joke, but this idea is super serious. If you haven't read Steve Yegge's post [1] that included Jeff Bezos' big mandate about APIs, be sure to do so.

And finally, in an ideal world, part of creating a good API is the process of documenting it well. A well-documented API makes things better for everyone (the people who wrote it, the people who use it, the people who are thinking about using it, people who want to learn from it, and so on). And this is where Swagger comes in.

And Now We Swagger

Here's how the official Web site [2] defines it:

> The goal of Swagger™ is to define a standard, language-agnostic interface to REST APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined via Swagger, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Similar to what interfaces have done for lower-level programming, Swagger removes the guesswork in calling the service.

> Technically speaking - Swagger is a formal specification surrounded by a large ecosystem of tools, which includes everything from front-end user interfaces, low-level code libraries and commercial API management solutions.

So what does this standard look like? At the moment, you can write Swagger specifications in either JSON (the original format) or YAML (recently added). To get a sense of what it actually looks like, here's the Hello World-ish sample in YAML format from the "Getting Started with Swagger—What Is Swagger?" article on the official Web site:

```
swagger: "2.0"
info:
  version: "1.0"
  title: "Hello World API"
paths:
  /hello/{user}:
    get:
      description: Returns a greeting to the user!
      parameters:
        - name: user
          in: path
          type: string
          required: true
          description: The name of the user to greet.
      responses:
        200:
          description: Returns the greeting.
          schema:
            type: string
        400:
          description: Invalid characters in "user" were provided.
```

This defines a REST interface that has exactly one endpoint, `/hello/{user}` (as in /hello/rik). The username at the end is defined to be a required string. If a valid username was given, the API promises to return a 200 return code (success) followed by a greeting to that user (in string form). If there is a problem with the username, an error code (400) is returned.

This by itself, besides being a reasonable documentation format, isn't the cool part. The cool part is when you bring the tools written around the Swagger specification into the picture. Let's quickly mention the non-Perl-related tools and then take a look at how Swagger plays in the Perl space.

Two of the more interesting non-Perl tools are Swagger Editor (running sample at http://editor.swagger.io/) and Swagger UI (running sample at http://petstore.swagger.io/). Swagger Editor lets you compose your YAML or JSON in real time and see how it will look as a fully formatted (and purdy, they've done a nice job with the design) documentation page generated on the fly. The editor also has some options for code generation—more on that in a moment.

Equally interesting is the Swagger UI tool, which generates a Web application that lets people not only read the documentation, but try API calls right from the documentation page. If you've ever tried something like Google's API Explorer or Spotify's API Console [3] you'll have a sense of what Swagger UI provides. And if you haven't, you really should because they are both very useful tools.

Generating Code

So now we step closer to the promise of Perl code. It's cool that we now have a good format for specifying our REST API. It is even cooler that we can process that specification and produce good-looking (and even interactive) documentation. But even better would be to run that specification through a post-processor that actually writes the code for us to make use of the specification. Why is this cool? It means that your API documentation and your API code won't get out of sync, because one begets the other. As an aside before we go deeper into this: I have seen efforts that allow people to take existing code and generate a Swagger spec (i.e., go the other way). I think it is cleaner to write the doc first, but I can see how going in the opposite direction could be beneficial in certain cases.

There are two kinds of code we could think about generating: client and server. We'll look at both separately. If we are continuing our look at "official" tools, we should start with Swagger Codegen (http://swagger.io/swagger-codegen/). Swagger Codegen is primarily meant to produce client code in a wide range of languages/frameworks from a Swagger spec. It manages this by making it relatively easy to add your own modules/templates.

At the moment, it knows how to output clients using these languages/frameworks:

```
[
  "android",
  "async-scala",
  "csharp",
  "dart",
  "flash",
  "java",
  "objc",
  "perl",
  "php",
  "python",
  "qt5cpp",
  "ruby",
  "scala",
  "dynamic-html",
```

```
  "html",
  "swagger",
  "swagger-yaml",
  "swift",
  "tizen",
  "typescript-angular",
  "typescript-node",
  "akka-scala",
  "CsharpDotNet2"
]
```

This is output from the online Codegen tool at https://generator.swagger.io, essentially a pretty-printed version of the output of:

```
curl -X GET --header "Accept: application/json"
  "https://generator.swagger.io/api/gen/clients"
```

To use Swagger Codegen, you have to install a particular version of Java (7 as of this writing), Apache maven, and the tool itself. I used homebrew on my Mac to install all of these components, including Java. Java 7 gets installed in a homebrew-specific place via its Cask mechanism since downloading that version from Oracle's Web site isn't easy. All in all, the process of bringing up the necessary Java toolchain wasn't as painful as I expected, but your mileage may vary.

Once you have everything installed, you can process a Swagger specification. Swagger ships with sample specs (for example, one based on an API for a pet store because, um, every modern pet store needs an API, I guess?) and scripts that process them to generate sample code for each language. Rather than using that sample spec, let's stick to the simpler "hello world" example shown earlier. To process it, we might type something like:

```
$ swagger-codegen generate -i ./test.yaml -l perl -o perl-test
```

The output will look something like this:

```
reading from ./test.yaml
[main] INFO io.swagger.codegen.DefaultCodegen -
        generated operationId helloUserGet
        for Path: get /hello/{user}
writing file /tmp/perl-test/lib/WWW/SwaggerClient/DefaultApi.pm
writing file /tmp/perl-test/lib/WWW/SwaggerClient/ApiClient.pm
writing file /tmp/perl-test/lib/WWW/SwaggerClient/
        Configuration.pm
writing file /tmp/perl-test/lib/WWW/SwaggerClient/Object
        /BaseObject.pm
```

As you can see, four separate files have been generated to form a module we can use (WWW::SwaggerClient). Of these, three are "support" files and one has the code specific to the defined REST API. That info is in DefaultApi.pm. In it we find the following code (slightly reformatted to fit on the page):

```
#
# hello_user_get
#
#
#
# @param string $user The name of the user to greet. (required)
# @return string
#
 sub hello_user_get {
   my ($self, %args) = @_;

   # verify the required parameter 'user' is set
   unless (exists $args{'user'}) {
     croak("Missing the required parameter 'user' when calling
        hello_user_get");
   }

   # parse inputs
   my $_resource_path ='/hello/{user}';
   # default format to json
   $_resource_path =~ s/{format}/json/;

   my $_method ='GET';
   my $query_params = {};
   my $header_params = {};
   my $form_params = {};

   # 'Accept' and 'Content-Type' header
   my $_header_accept =
    $self->{api_client}->select_header_accept();
   if ($_header_accept) {
      $header_params->{'Accept'} = $_header_accept;
   }
   $header_params->{'Content-Type'} =
     $self->{api_client}->select_header_content_type();

   # path params
   if ( exists $args{'user'}) {
      my $_base_variable = "{" . "user" . "}";
      my $_base_value =
        $self->{api_client}->to_path_value($args{'user'});
      $_resource_path =~ s/$_base_variable/$_base_value/g;
   }


   my $_body_data;

   # authentication setting, if any
   my $auth_settings = [];

   # make the API Call
   my $response =
```

```
    $self->{api_client}->call_api($_resource_path,
                                  $_method,
                                  $query_params,
                                  $form_params,
                                  $header_params,
                                  $_body_data,
                                  $auth_settings);
    if (!$response) {
        return;
    }
    my $_response_object =
        $self->{api_client}->deserialize('string', $response);
    return $_response_object;


}
```

This code is a little gnarly (as are the other files). The generated code is meant to handle more sophisticated specs, so it looks a bit like overkill at first glance. It definitely represents a certain set of opinions and programming choices of the template author. The generated code includes a bunch of Perl modules you may or may not have installed (e.g., Log::Any), so be prepared to work a bit if you are going to use the code right out of the box.

Given all of this, let me highlight one small part of the code above. In it you can see that it has defined a hello_user_get subroutine. This is the one you are going to call as a method to perform the actual call from the Swagger spec. To use all of this, we would write code like this:

```
use lib 'perl-test/lib';
use WWW::SwaggerClient::DefaultApi;
''
my $api = WWW::SwaggerClient::DefaultApi->new();

my $greet = $api->hello_user_get('user' =>'rik' );
```

If I just run this code from my laptop without any other preparation, I get the following error:

```
API Exception(500): Can't connect to localhost:443 at
perl-test/lib/WWW/SwaggerClient/DefaultApi.pm line 100.
```

because there is nothing currently listening on my laptop for connections from a client (i.e., no server). If I ran this under the debugger, I'd see more detail about what was being attempted (here's the key line of the output):

```
API Exception(500): Can't connect to localhost:443 at
perl-test/lib/WWW/SwaggerClient/DefaultApi.pm line 100.
 at perl-test/lib/WWW/SwaggerClient/ApiClient.pm line 124.
WWW::SwaggerClient::ApiClient::call_api(WWW::SwaggerClient::
 ApiClient=HASH(0x7fc1339f7d98), "/hello/rik", "GET",
 HASH(0x7fc133a76fb0), HASH(0x7fc133a76f98),
 HASH(0x7fc133a76fc8), undef, ARRAY(0x7fc1320083c0))
 called at perl-test/lib/WWW/SwaggerClient/DefaultApi.pm line 100
```

From this line you can see that it was going to attempt to connect to a server and issue a GET request for the path /hello/rik just as we'd hoped it would. If you'd like to see a more sophisticated example, I recommend you dissect the sample apps that come with Swagger Codegen (e.g., the pet store one) to see how it works. If the generated code isn't to your liking, you may want to consider creating a custom plugin that outputs the kind of code you seek.

Another possibility is to use the module we are about to see for server code: Swagger2. Swagger2 ships with a Swagger2::Client module, which lets you write code that looks like this:

```
use Swagger2::Client;

$ua = Swagger2::Client->generate("/tmp/test.yaml");
$ua->base_url->host("other.server.com");

# yes, this is ugly. If our spec had an operationId parameter,
# the name of the method would be based on it instead
$ua->_hello__user_({'user'=>'rik'})
```

But let's move away from the client code possibilities and think a little bit about the server side of things instead. Swagger Codegen has limited support for server code (e.g., it can create server stubs for NodeJS, Python Flask, Ruby Sinatra, and so on) but nothing for Perl-based servers. For that we're going to have to go a little further off the ranch and use the Swagger2 module.

Probably the easiest path is to use Mojolicious::Plugin::Swagger2, which ships with the Swagger2 Perl module. With this plugin, we can use the Mojolicious Web framework we've seen in past columns. If you add code like this to the startup routine of your Web app:

```
$app->plugin(Swagger2 =>
          {url => "file:///path/to/test.yaml"});
```

it will automatically add routes and validation to your Web app (providing it has operationId info in the spec). The paths defined in the Swagger spec will automatically become routes that require the parameters mentioned in the spec. There's a lovely example of how this works in the author's tutorial on his blog [4]. Swagger2 with Mojolicious isn't the only game in town for Swagger (for example, there is the REST API framework "raisin" that also integrates with Swagger), but I think it is a lovely combination.

So with that, I think you've got at least a small peek at Swagger and how it can improve your API life. Take care, and I'll see you next time.

Resources

[1] Steve Yegge's Google Platform Rant: https://plus.google.com/+RipRowan/posts/eVeouesvaVX.

[2] Swagger: http://swagger.io.

[3] Google's API explorer: https://developers.google.com/apis-explorer; Spotify's API Console: https://developer.spotify.com/web-api/console/.

[4] Mojolicious Swagger2 tutorial: http://thorsen.pm/perl/programming/2015/07/05/mojolicious-swagger2.html.

Modern System Administration with Go and Remote Procedure Calls (RPC)

KELSEY HIGHTOWER

Kelsey Hightower has worn every hat possible throughout his career in tech, and enjoys leadership roles focused on making things happen and shipping software. Kelsey is a strong open source advocate focused on building simple tools that make people smile. When he is not slinging Go code, you can catch him giving technical workshops covering everything from programming to system administration and distributed systems.
kelsey.hightower@gmail.com

The datacenter is the new computer and it's time to look past the UNIX shell for building tools and utilities. While the programming environment outside the shell is different, the UNIX philosophy is still applicable: the tools and utilities you build should have a single purpose and support composition through clean inputs and outputs that allow users to build larger systems and custom workflows.

In the early days of UNIX, stdin and stdout streams allowed us to chain specialized tools and compose various workflows to suit our needs. For example, processing HTTP logs was as simple as running the following command:

```
$ grep 'html HTTP' /var/log/apache.log | uniq -c
```

What an easy way to build a data pipeline with very little code, but there are a few minor problems. The above solution only works for a single machine running specific versions of the UNIX utilities used in the pipeline. Running the same command on another flavor of UNIX is not guaranteed to work, or even worse, might yield different results. On top of everything else, the data between grep and uniq is often unstructured, which means ad hoc text parsing will be required to extract specific fields before data processing can continue.

To overcome these challenges, a programming language with a little more power, such as Go, can be used to model data using modern serialization formats such as JSON, which can improve interoperability between command line tools and services over a network. Expanding beyond a single system does introduce another set of challenges, such as invoking code over a network and handing failures without introducing too much overhead or complexity. One way of doing this is to use remote procedure calls (RPCs) between clients and servers.

Go and its robust standard library provide everything you need to build tools ranging from simple command line utilities to microservices that scale horizontally across a cluster of machines. The remainder of this article will focus on Go's native syscall interface, RPC mechanisms, and standard libraries you can use to ship robust sysadmin tools in little to no time.

Remote Procedure Calls (RPC)

When creating system administration tools that need to scale beyond a single host, RPC should be strongly considered. While there are other platforms for building services, I feel that RPC maps closest to task originated tools built by most system administrators and provides better performance by avoiding the unnecessary overhead required by protocols such as HTTP.

What Are Remote Procedure Calls?

As the name implies, RPC is about calling procedures (functions) remotely. RPC aims to ease the development of client-server applications by reusing native-language semantics and sharing code between both client and server. The learning curve for RPC is relatively low because there is no need to learn new ways of interacting with remote services outside of the native function calling conventions and error handling of the language you're programming in.

gls: A Distributed ls Service

To demonstrate the simplicity of Go and RPC for system administration tasks, we are going to reimplement the classic UNIX tool ls—with a twist. gls is a distributed tool for collecting file attributes for a given file system on a remote system.

The remainder of this article will walk you through building gls from the ground up. The source code for gls is hosted on GitHub [1], but I encourage you to type the commands by hand as you follow along—of course, this assumes you have a working Go installation [2].

The gls Package

At the heart of the gls server is the gls package, which holds common code shared by the gls server and client. Create the gls package directory under the GOPATH. We'll get into the details later, but type exactly what you see here for now:

```
$ mkdir -p $GOPATH/src/github.com/kelseyhightower/gls
```

Next, change into the gls package directory and save the following code snippet to a file named gls.go:

```
$ cd $GOPATH/src/github.com/kelseyhightower/gls
$ vim gls.go
package gls

import (
      "os"
      "path/filepath"
)

type Files []File

type File struct {
      Name     string
      Size     int64
      Mode     string
      ModTime  string
}

type Ls struct{}

func (ls *Ls) Ls(path *string, files *Files) error {
      root := *path
      err := filepath.Walk(*path, func(path string, info
os.FileInfo, err error) error {
            if err != nil {
                  return err
            }

            file := File{
                  info.Name(),
                  info.Size(),
                  info.Mode().String(),
                  info.ModTime().Format("Jan _2 15:04"),
            }
            *files = append(*files, file)
            if info.IsDir() && path != root {
                  return filepath.SkipDir
            }
            return nil
      })
      if err != nil {
            return err
      }
      return nil
}
```

Let's walk through the gls package to see what's happening.

First, we declare the gls package and import the `os` and `filepath` packages from the standard library:

```
package gls

import (
      "os"
      "path/filepath"
)
```

Next, we define two types, a File type, which holds file metadata, and a Files type, which holds a list of File objects:

```
type Files []File

type File struct {
      Name     string
      Size     int64
      Mode     string
      ModTime  string
}
```

Finally, we define the Ls type for the sole purpose of defining the Ls method, which is responsible for gathering metadata from files under a specific directory path. For each file found, the name, size, permissions, and last modified time are captured and appended to a files list that will ultimately be returned to the caller.

```
type Ls struct{}

func (ls *Ls) Ls(path *string, files *Files) error {
      ...
}
```

There are a couple of things to note here. First, Ls is a method and not a function. Second, Ls takes two arguments and returns a single error value. This is not arbitrary, but a requirement of Go's RPC support, which provides access to exported methods of an object over a network. Only methods that meet the following requirements can be exposed as RPC methods:

- The method's type is exported.
- The method is exported.
- The method has two arguments, both exported (or built-in) types.
- The method's second argument is a pointer.
- The method has return type error.

In the case of the gls package, the exported type is the Ls struct and the exported method is the Ls method. In order to meet the RPC requirements, the Ls method takes two arguments—the path to search for files, and a pointer to a files list to store file metadata—and returns a single error value.

In Go, this is not the typical way methods or functions are written. If the Ls method was not exposed as a RPC method, then it would have been written like this:

```
func (ls *Ls) Ls(path *string) (*Files, error)
```

The set of constraints imposed by Go's RPC support may seem odd at first, but when you think about it, it all makes sense. Requiring all RPC methods to have a similar signature, two arguments and a single return value, means it's much easier to encode and decode the communication between the client and server over the network.

Complex arguments can be expressed using a complex type. For example, if we wanted to include a pattern to limit which files are inspected by the Ls method, we could use the Options type in place of the original path argument.

```
type Options struct {
	Path    string
	Pattern string
}

func (ls *Ls) Ls(options *Options, files *Files) error {...}
```

The gls Server

With the gls package in place, it's time to create the gls server, which is responsible for exposing the Ls method from the gls package over RPC.

Start in the gls package directory created earlier:

```
$ cd $GOPATH/src/github.com/kelseyhightower/gls
```

Create a new directory named server to hold the gls server binary:

```
$ mkdir server
```

Next, change into the server directory and save the following code snippet in a file named main.go.

```
$ cd server
$ vim main.go
package main

import (
	"log"
	"net"
	"net/rpc"

	"github.com/kelseyhightower/gls"
)

func main() {
	log.Println("Starting glsd..")
	ls := new(gls.Ls)
	rpc.Register(ls)

	l, err := net.Listen("tcp", "0.0.0.0:8080")
	if err != nil {
		log.Fatal(err)
	}

	for {
		conn, err := l.Accept()
		if err != nil {
			log.Println(err)
		}
		rpc.ServeConn(conn)
		conn.Close()
	}
}
```

Let's quickly walk through what's going on here. First, we import a few packages from the Go standard library, including the `net/rpc` package, which provides support for exposing methods over RPC, and the gls package, which holds the definition of the Ls method.

Before we move on it's important to note the full name of the gls package: `github.com/kelseyhightower/gls`. This name was chosen to match where the gls package will be hosted on the Internet—on GitHub under the username kelseyhightower. Go's tooling has native support for working with packages hosted on remote repositories such as GitHub, and it's common to see packages named using this convention. The package name is important: because we cannot simply import "gls", we must use the complete import path where the gls package lives in relation to the GOPATH or our program will fail to compile. Learn more about Go's import semantics from the official docs [3].

With the gls package imported, we are ready to export the gls.Ls method by registering it using the `net/rpc` package.

```
ls := new(gls.Ls)
rpc.Register(ls)
```

The rest of the code creates a listener which binds to port 8080 on all available network interfaces and waits for RPC requests from clients.

The gls Client

The gls client is responsible for making requests to the gls server and printing the results to stdout. Create the gls client by running the following commands:

Start in the gls package directory created earlier:

```
$ cd $GOPATH/src/github.com/kelseyhightower/gls
```

Create a new directory-named client to hold the gls client binary:

```
$ mkdir client
```

Next, change into the client directory and save the following code snippet in a file named main.go.

```
$ cd client
$ vim main.go
package main

import (
	"fmt"
	"log"
	"net/rpc"
	"os"

	"github.com/kelseyhightower/gls"
)

func main() {
	client, err := rpc.Dial("tcp", "127.0.0.1:8080")
	if err != nil {
		log.Fatal(err)
	}

	files := make(gls.Files, 0)
	err = client.Call("Ls.Ls", os.Args[1], &files)
	if err != nil {
		log.Fatal(err)
	}
	for _, f := range files {
		fmt.Printf("%s %10d %s %s\n", f.Mode, f.Size,
f.ModTime, f.Name)
	}
}
```

As with the gls server, we import a few packages from the standard library and the gls package, which in the case of the gls client provides access to the gls.Files type. Remember the gls.Files type is defined in the gls package:

```
package gls

type Files []File

type File struct {
	Name    string
	Size    int64
	Mode    string
	ModTime string
}
```

In order to communicate with the gls server, we need an RPC client and must establish an RPC connection:

```
client, err := rpc.Dial("tcp", "127.0.0.1:8080")
if err != nil {
	log.Fatal(err)
}
```

Before making the call to the remote Ls method, we must initialize an empty gls.Files slice to hold the results from the gls server:

```
files := make(gls.Files, 0)
```

Now we are ready to make our RPC call and print the results.

```
err = client.Call("Ls.Ls", flag.Args()[0], &files)
if err != nil {
	log.Fatal(err)
}

for _, f := range files {
	fmt.Printf("%s %10d %s %s\n", f.Mode, f.Size, f.ModTime,
f.Name)
}
```

Also, notice how we are using the first positional command line argument identified by `flag.Args()[0]` as the path argument to the Ls method. This will allow us to use the gls client binary like the standard ls UNIX command. For example, to list files in the tmp directory, we can run the gls client like this:

```
$ gls /tmp/
```

The string "/tmp/" will be stored at the first position of the slice returned by the flag.Args() function.

At this point, we are code complete and are ready to build and deploy the gls client and server.

Build and Deployment

Now that we have written and understand the code behind the gls client and server, let's turn our attention to the build and deployment process. Go is a compiled language, which means we must run our source code through a compiler before we can run it.

Building the gls client and server is as simple as running the following commands from the gls package directory:

```
$ cd $GOPATH/src/github.com/kelseyhightower/gls
```

Build the gls client using the `go build` command:

```
$ go build -o gls client/main.go
```

Build the gls server using the `go build` command:

```
$ go build -o glsd server/main.go
```

Running the above commands results in the following binaries:

```
gls
glsd
```

One thing to note about the gls and glsd binaries (and Go binaries in general) is that they are self-contained. This means each binary can be copied to a similar OS and architecture and be run without the need to install Go on the target system. In a future article, I'll cover how cross-compiling in Go works, which allows you to develop applications on one platform (Linux) and compile them to run on another (Windows).

You are now ready to launch the gls server:

```
$ ./glsd
2015/12/23 07:50:06 Starting glsd..
```

At this point the gls server is ready to accept RPC requests on port 8080.

Open a new terminal window and use gls client to get a directory listing of your home directory from the gls server:

```
$ ./gls ~/
drwx------   170  Nov 28 20:23  Applications
drwxr-xr-x   102  Dec 20 01:52  Desktop
drwx------  1122  Dec 20 11:57  Documents
drwx------   340  Dec 22 11:30  Downloads
...
```

The gls client is hardcoded to communicate with the gls server over localhost (127.0.0.1) on port 8080. This is being done because the gls server is not protected by any form of authentication or encryption such as TLS. In a future article, we will revisit extending the gls client and server to support encryption, authentication, and communication over any IP/port combination using a set of command line flags.

Conclusion

The way we think about computers is changing, and this is the perfect time to rethink the way we approach systems programming in general. Go has native RPC support and low-level syscall functionality, which allows us to build enhanced versions of UNIX classics such as ls or new tools that perform tasks that meet the challenges of today while leveraging the timeless UNIX philosophy that has defined computing for decades.

Resources

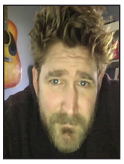
[1] GitHub for the sources in this column: https://github.com/kelseyhightower/gls.

[2] Installing Go: https://golang.org/doc/install.

[3] Docs for understanding package paths: https://golang.org/doc/code.html#PackagePaths.

iVoyeur
We Don't Need Another Hero

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

Someone recently asked me this question: "What's the first thing that comes to your mind when you hear the word 'DevOps'?"

A loaded question, I agree, and of course I lied, and made up something about the "first way." I mean really, if you want a possibly embarrassing answer to a loaded question, you really should confront me face-to-face with it in a public place. If the asker of that question had done so, I would have had to answer honestly that the first thing I think about when I hear the word "DevOps" is Brent from The Phoenix Project novel [1].

If you haven't read it, let me explain: Brent is probably you. The one person who knows how all the stuff actually works, and who everyone depends on to fix things when they go sideways. Brent is a hero. And because the book is about DevOps, and DevOps abhors constraints and local optimization (in other words, because DevOps hates heroes), Brent is basically a huge organizational problem.

I was deeply hurt by this plot device on my first reading. In fact, if I hadn't been trapped on a plane with nothing else to read, I probably would not have finished *The Phoenix Project* because of it, which would have been my loss. It's just hard to wrap your head around how a hero like Brent could be bad for the IT organization as a whole (especially when I relate so strongly to him). As a result, I also had a hard time wrapping my head around the endgame. Sometimes it seems like all anybody talks about is "improvement-kata" and "getting there." What it looks like day-to-day once you've arrived is something you almost never hear about.

I write this in the still-warm afterglow of LISA15, where I gave a talk about (surprise) metrics and monitoring. In that talk, I had two big bones to pick. The first was to attempt to fill that gap, basically to show off a bit of what the DevOps endgame looks like for operations folks like ours, who still work to solve real problems day to day. The second was to make the point that DevOps is mostly still getting "monitoring" wrong, because rather than working monitoring in to the the rest of the improvement processes they practice, DevOps seems intent on treating monitoring as a "heroic" discipline. Creating an ever-increasing litany of new, specialized monitoring categories, which in turn silos the resulting telemetry data in ways that limit its potential to the rest of the organization.

During the talk, I shared several chat transcripts with diagrams, like the one in Figure 1. Each of these represented a real problem in several wildly varying contexts that our engineering teams had encountered in the weeks leading up to LISA. My thinking at the time was that presenting a breadth of different problems instead of a depth of one would better illustrate the point that, since our monitoring data was *not* being siloed, it was therefore more useful than this sort of data is at other shops. It was frustrating to me, however, that I couldn't dive as deeply into the actual problems as I would have liked to in the time I had on stage.

So in this issue, I'd like to choose just one of these and dive a little more deeply into it, so you can really get a solid feel for how our engineers are using the telemetry system in the context of detecting and tracking imbalances in the system that will eventually lead to catastrophic outages if left unchecked. It's a pervasive belief today that metrics are not yet useful for very

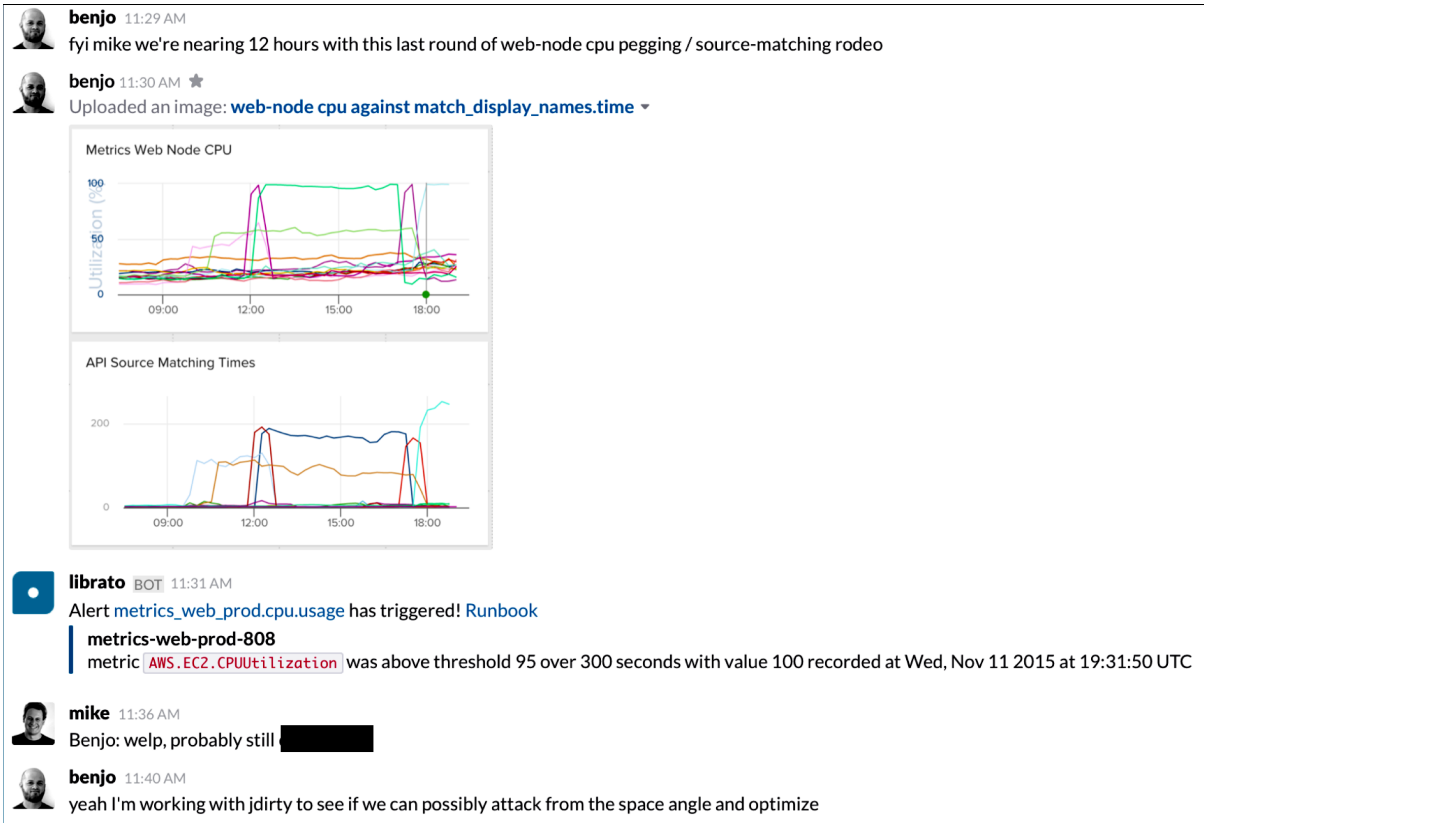


Figure 1: Some internal chat where Ben notices a CPU spike, posts those graphs to the chat, and involves engineers from other areas

early problem detection of this sort—that advanced aberrant detection algorithms need to be developed to help in this regard. I'm going to use the problem depicted in Figure 1, which is also the first problem I presented in my LISA talk.

Let's talk for a second about how these graphs even exist, much less exist in the same system. The first graph is the age-old OPS CPU graph. We get ours by way of collectd, and I'm sure you have them too. The second graph was put in place by the data engineering team because they needed to quantify the amount of time we lose looking up metric and source names in our various index DBs. I couldn't cover in the time I had at LISA what that means, but here I can spare the space, so let's digress into that for just a few paragraphs.

One big problem with designing time-series databases is that the pattern of reads and writes is very different. If you think about writing time-series measurements into a database, you'll see that we're writing into columns. That is, if you think about it like a spreadsheet, you have a column for every second of lapsed time, and then a row for each metric name. We can only write one column at a time, because the future hasn't happened yet, but we can write to multiple rows in that column at once because we have many different metrics to track, and all of those numbers are coming in at once.

But when you read from a time-series data store, you read *rows*. That is to say, you're never interested in a search that gives you a single data point for *every* metric you're tracking at a single point in time. You always want 60 or 90 seconds' worth of data for one metric. So you have to read out rows.

This is usually okay, because in TSDB land, you write a lot more often than you read, and columnar writes are pretty efficient. But reads are exquisitely painful. Think about it—as time progresses, the rows keep getting longer and longer. Soon you need to search through increasingly gigantic rows in order to isolate the set of data you need and then extract it. So you wind up spinning cycles in linearly increasing seek time to find and retrieve your data as the rows grow. Not good.

One way we get around the long-row read problem is with a rotating row-key. Imagine that, instead of the row being the name of the metric, it was a number that changes every six hours. That way, we create a new row for each metric every six hours, and we store it as a name/number tuple and our seek time never goes above a certain predictable value.

"Predictable" is something of a keyword there, because really what row keys buy you is a heap of different kinds of very important predictability. With row keys, we know, for example, that

our rows will also always be a predictable size (in bytes), which is the size of a measurement times the number of measurements in a row-key interval. From there we can extrapolate how much data we'll need to put on the wire for client reads, and how much storage and processing power our data tier will require. The point is, we can make really important decisions by relying on the predictability that row keys provide.

But here's the problem (well, *one* problem) with running a shared storage back-end for a multi-tenant system: users have the power to name things. So if you think back to how we're now storing our data as name/measurement tuples, we no longer have a predictably sized data structure because end-users can create *really* big names if they want to.

So when you use a row key to buy predictable computational quantities in the data tier, you also often have to pay some taxes in the form of lookup-tables, or indexes if you prefer. In this example, we're going to need two indexes, one to keep track of where in our storage tier a given six-hour block of measurements is stored (because our rows are named after rotating numbers now, so we need some way to actually find the right data when a user asks for 60-minutes of metric:foo), and another to map user-generated variable-length names into either hashes or some other unique identifier (so we know what to even search for in the first index when a user asks for 60 minutes of metric:foo).

Okay, now we know pretty much everything we need to reexamine the first problem I shared in my LISA presentation on metrics. Ben, the Ops guy in that conversation, is tracking what he considers to be a resource allocation problem. The symptom Ben is reacting to here is high CPU utilization, which is actually something of a rarity for us, but it's also why I included it here (everybody can relate to a good-ole CPU utilization graph). Our problems are often CPU bound—that is, a problem in something we've built will often result in high CPU utilization, but it's rare for us to discover it by way of a CPU graph.

The second graph is the one we can now fully understand given our short discussion about time-series data stores above: briefly, the second graph is timing how long it takes us to perform an index lookup on a metric. You'll recall that I said rotating row keys come at the cost of index tables. Well, you can think of the second graph here as quantifying the cost of our row-key taxes. Literally, when we want to retrieve 60 seconds of metric:foo, how long does it take us to convert "metric:foo" into "ID12345" so we can use the ID to retrieve the data from the data storage tier?

If you're asking yourself why we don't cache this stuff, the answer is we do. So this problem—the one Ben and Mike are discussing in chat—is extraordinary for that reason alone. It never happens except in the event of a cache-miss. And for it to be this bad, and progress for this length of time, an awful lot of recurrent individual cache misses need to happen, and that strongly implies that we're encountering a new end-user behavior here. Either these services don't work like we think they do (not the case here), or an end-user is doing something to generate an inordinate amount of cache-missing index lookups. Problems like this one are, as you can probably imagine, very interesting to us. They inevitably teach us something we didn't already know about our systems, by showing us either a gap in our understanding or a path through our system that we didn't anticipate.

I think it's fair to say that many engineers believe that being good at Web-operations engineering, or really any kind of high-availability engineering, means building solid infrastructure and reacting quickly and effectively to blocking-outages as they occur. But in my opinion, being really excellent at operations engineering is 99% about being interested in problems like this one: problems that are non-blocking, that are not currently causing anything close to a catastrophic outage. Annoyances really, but exactly the right sort of annoyance. The annoying little imbalances in the system that teach you something you didn't know, or hadn't anticipated about the thing your organization created. Problems that you, by definition, can't actually fix by yourself.

You might be tempted to call it preventive maintenance, but the big difference between the problem we're looking at here and preventative maintenance, in my opinion, is the fact that Ben can't solve this problem alone. In fact, I think probably the most important aspect of this issue—certainly the thing that made me want to share it with the world—is that Ben the operations hero can't solve it by himself. These issues are *so* important, they are the bubbling spring from which catastrophic torrents are born (especially in distributed systems). Like Muhammad Ali said, it's always the punch you didn't see coming that knocks you out, and this one certainly would have eventually been a knockout punch if Ben hadn't seen it coming. The problem is, you need not just interdisciplinary know-how, and a toolchain to work problems like this one, but a culture that encourages cooperation over heroism.

First, Ben needs to know enough about the system and the monitoring data it generates to even discover that this problem is there, that it's worth working on, and what's causing it. Then he has to track it—quantify its occurrence long-term over weeks and months. He needs to understand the nuances of the system from which it has emerged well enough to even know whom to work together with. And, finally, he needs the interpersonal skills to get other engineers involved, as he's done here, as well as the toolchain to allow him to easily share the data he's looking at.

In the snippet, he's giving Mike, a data engineering guy, a heads up about it. We can tell they've spoken about it before, and the eventual fix will certainly be a data-engineering endeavor. At the very bottom, you'll see that Ben is also working with Jason on some UI tweaks (our UI is called "spaces").

So if you're a system administrator, or otherwise consider yourself operations or SRE, try to put yourself in Ben's shoes. You're looking at a CPU spike that correlates to the amount of time it takes us to run an index lookup on a metric name. I think many of us wouldn't have made it that far. Past versions of myself certainly wouldn't have made it that far, because I wouldn't have even had access to the data that would have allowed me to discover that correlation. I certainly wouldn't have had the domain knowledge about our product to know what that correlation meant.

Anyway, rather than opening a ticket and throwing it over the fence, Ben personally gets data engineering involved, helps them help him further understand the problem, and then proceeds to update them periodically as he sees it reoccur. Together they establish a pattern of behavior and narrow it to a handful of customers (that's the little black box that was redacted from the chat transcript).

I personally have never made it that far. I've never been able to create a strong enough rapport with engineers outside my own discipline to be able to work together to understand an issue like this one. I think my own super-hero-thinking is a huge contributing factor in this unfortunate truth. Any problem I couldn't solve by myself with open source software just wasn't worth my time and should be thrown over this or that fence. Made someone else's problem. And every time I pulled that rip cord, I gave up any chance I had of learning about how the things my organization cared about actually worked.

But Ben never stops. In fact, looking at the transcript, he takes it one step further even than *that*. Well, he reasons, if customers use our UI to retrieve data from the data tier, then really it's the *UI* that's triggering all the index-churning. Therefore, it's possible that some light UI optimizations might help alleviate part of this index lookup churn. Maybe index lookups would be faster if we batched them, or maybe we can pre-fetch them based on user behavior? I don't know if those particular optimizations are what Ben had in mind when he roped in the UI team, but as an engineer they occur to me as distinct possibilities.

And therein lies the thing about this conversation that *really* fascinates me. I've been calling this stuff "domain knowledge" and saying that it's "interdisciplinary," but really none of these notions are *so* abstract or complex that I can't understand them with a few minutes of consideration. It only took me a few hundred words to bring you fully up to speed with respect to index timing and what that means in our TSDB, and of course we all know, or can infer, that a user-interface might be able to batch-query things to avoid a multitude of individual requests, so why, in the 20 years I've been doing operations work, have I never stepped over that line to work together with other departments toward a common solution to problems like this one?

One answer is certainly that my monitoring data has always been trapped in silos. I didn't have a means of sharing engineering data with other teams. Just being able to see what the data engineering team has chosen to measure inside a newly constructed service teaches me as an operations engineer an awful lot about that service. It also provides an opportunity for me to formulate questions that I can ask those engineers when I see them at the coffee pot or the bar after-hours. Those conversations build rapport, and rapport is exactly what you're seeing there between Ben and Mike.

Monitoring data shouldn't belong to anyone, and it certainly shouldn't be a magical contraption reserved for a select few heroes who have bothered to understand how it works. I really hope this little play-by-play helps illustrate what I mean by that, and also what I mean when I say we don't need another hero, we just need (as always) to let the data be free.

Resource

[1] Gene Kim, Kevin Behr, and George Spafford, *The Phoenix Project*: http://itrevolution.com/books/phoenix-project-devops-book/.

For Good Measure

Betting on Growth versus Magnitude

DAN GEER

Dan Geer is the CISO for In-Q-Tel and a security researcher with a quantitative bent. He has a long history with the USENIX Association, including officer positions, program committees, etc. dan@geer.org

Whether you are Werner Heisenberg or Janet Yellen, your field of study includes measurements of position and velocity and how they interact. Poor measurements may be unable to nail down either the one or the other (much less both), but even if only the one is measured, there is likely to be some prediction that you will be able to make. In hard-to-measure situations, consistency of error distribution can be your friend—consistent errors help you to find the message in the body of noisy data. The reader probably knows all that.

Let's look at some data where position and velocity have been charted for enough years to get a feel for what is going on, and then we'll discuss what use we can make of it.

The (US) Federal Trade Commission has a program known as Sentinel [1]. Quoting its introduction,

> Sentinel is the unique investigative cyber tool that provides members of the Consumer Sentinel Network with access to millions of consumer complaints. Consumer Sentinel includes complaints about:
>
> - Identity Theft
> - Do-Not-Call Registry violations
> - Computers, the Internet, and Online Auctions
> - Telemarketing Scams
> - Advance-fee Loans and Credit Scams
> - Immigration Services
> - Sweepstakes, Lotteries, and Prizes
> - Business Opportunities and Work-at-Home Schemes
> - Health and Weight Loss Products
> - Debt Collection, Credit Reports, and Financial Matters
>
> Consumer Sentinel is based on the premise that sharing information can make law enforcement even more effective. To that end, the Consumer Sentinel Network provides law enforcement members with access to complaints provided directly to the Federal Trade Commission by consumers, as well as providing members with access to complaints shared by data contributors.

What Sentinel then produces are interpretive text reports backed by spreadsheets, all freely available for aggregate data. Querying the backing database for individual complaints is limited to law enforcement with a need to know.

Because the Sentinel data files are categorized enumerations of consumer complaints, there will be some errors. Let us assume that the errors are relatively constant over time, which is to say that trendlines and rank orderings are going to be instructive even if, say, there is persistent undercounting. The categories of consumer complaints for which we have at least six years of data are

- Advance Payments for Credit Services
- Auto-Related Complaints
- Banks & Lenders
- Business & Job Opportunities
- Buyers' Clubs
- Charitable Solicitations
- Computer Equipment & Software
- Credit Bureaus, Information Furnishers & Report Users
- Credit Cards
- Debt Collection
- Education
- Foreign Money Offers & Counterfeit Check Scams
- Grants
- Health Care
- Home Repair, Improvement & Products
- Identity Theft
- Impostor Scams
- Internet Auction
- Internet Services
- Investment-Related Complaints
- Magazines & Books
- Mortgage Foreclosure Relief & Debt Management
- Office Supplies & Services
- Prizes, Sweepstakes & Lotteries
- Real Estate
- Shop-at-Home & Catalog Sales
- Tax Preparers
- Telephone & Mobile Services
- Television & Electronic Media
- Travel, Vacations & Timeshare Plans

As of the time of writing, the most recent Sentinel data set available is for 2014. Let's think of the number of complaints as "position" and the rate of growth as "velocity." Converting both position and velocity into rank order and graphing them in the typical high/low quadrant style, we see considerable spread in Figure 1.

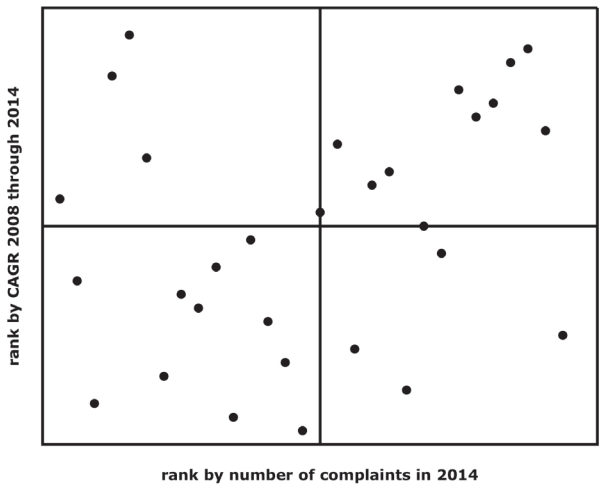


Figure 1: Size versus trajectory spread using the data from Sentinel categories with at least six years of data

The smallest and slowest growing are in the lower left quadrant, the biggest and fastest growing are in the upper right quadrant, etc.; you've seen this kind of graph before. The dot closest to the lower left corner is that for Real Estate, ranked as third smallest both in numbers of complaints (4,952) and in compound annual growth rate (or CAGR, which is actually declining at -5%). The dot closest to the upper right corner is that for Impostor Scams, ranked third highest in numbers of complaints (276,622) and first highest in growth rate of complaints (CAGR of 182%). If you were the person in charge, it is pretty clear that you'd put more manpower into impostor scams than into real estate fraud. That's an easy call.

On the other hand, the dot farthest to the right in the lower right quadrant is Identity Theft, number one in total complaints (332,646 in 2014 meaning one every 20 seconds) but with the eighth-slowest rate of growth (CAGR of not quite 1%). Similarly, the dot closest to the boundary of the upper left quadrant is Tax Preparers, fifth smallest in total complaints (6,418) but with the highest growth rate of all (CAGR of 292%). For this pair of Identity Theft versus Tax Preparers, which one is more deserving of investment?

Some readers will look at that graph and ask, "Is there any correlation here?" No, there isn't—Spearman's $\rho = 0.280$, meaning there is nothing worth talking about, as you can see: one-third of all categories are well off the diagonal, i.e., categories come and go, which is no surprise when you have sentient opponents. Again, where would you put your money when you are in charge?

The three most common complaints are Identity Theft (17.7%), Debt Collection (15.0%), and Impostor Scams (14.8%), which together comprise almost half of all complaints. The three fastest growing are Tax Preparers (291% CAGR), Impostor Scams

(182% CAGR), and Telephone and Mobile Services (47% CAGR). Is it position or velocity—count or CAGR—that tells you where to put your money?

Suppose the current observed compound annual growth rates are sustained until 2020. By then, the three most common complaints would be Impostor Scams #1, Telephone and Mobile Services #2, and Debt Collection #3. Identity Theft would have fallen to #6 (behind Banks and Lenders #4 and Auto-Related Complaints #5). If you imagine that any serious countermeasure takes, say, five years to actually work, then should we be spending our money now on the predicted top three in 2020? Or does the sentient opponent make a five-year plan an exercise as useless as the Five Year Plans that many national bureaucracies so love?

We are not alone in facing this kind of problem, namely, do you spend your money (or other scarce resource) to solve problems you actually have now or to stave off problems you are going to have later? As the old saying goes, a stitch in time saves nine, so prevention is likely the better buy, but it is never the easy sell... When there is not enough vaccine to go around, do you vaccinate those most likely to soon sicken and die (children, perhaps) or those most likely to soon become transmission vectors (clinic workers, perhaps)? In the Middle East, Western governments are invited to choose between stability and justice. If you choose stability, then you must reinforce dictators' grip on power, regardless of how they treat their people. This was the West's policy during the Cold War—and it is Vladimir Putin's policy today. If, however, you choose justice, you must side with the crowds trying to throw off their rulers, even if this triggers the collapse of order [2].

I've written over and over on this problem from every different angle, including the disastrous practice of vendors abandoning code bases they don't want to support yet simultaneously refusing to open source the code they are abandoning. Or how many platforms in common use are provisioned with software that its maker can no longer build? Or the longer a deployed device stays deployed, the more likely it is that it cannot be found and the more certain it is that it cannot be updated if found; should we be putting money into having a, say, 20-year guarantee for updatability of autonomous devices with network connectivity, or is the embedding of sensors in damned near everything already past the point where such decisions are even relevant?

Nassim Taleb (in *The Black Swan,* for example) argues that when a distribution is fat-tailed, estimations of parameters based on historical experience will inevitably mislead, which means

> [we are] undergoing a switch between [continuous low grade volatility] to...the process moving by jumps, with less and less variations outside of jumps. [3]

As I ponder that, I am more inclined to put my money on identifying, as best I can, problems that will grow than on problems that have grown. Easy for me to say, but killing dragons in their cribs beats dealing with them later on and, by and large, I can avoid the dragons that are already full size by just not doing the things that make me look like lunch. If you study the full Sentinel reports, you'll see what the demographics that spell "lunch" look like, such as the order of magnitude greater rate of identity theft in Miami (340.4 per 100,000 population per year) than in Bismarck, ND (27.9 per 100,000 population per year), or how demographics predict whether it is one's government benefits or one's credit cards that you are most likely to lose to an identity thief.

References

[1] https://www.ftc.gov/enforcement/consumer-sentinel-network; https://www.ftc.gov/enforcement/consumer-sentinel-network/reports.

[2] http://www.telegraph.co.uk/news/worldnews/middleeast/12046082/Tony-Blair-has-learnt-important-lessons-from-Iraq.-Its-a-shame-no-one-wants-to-listen.html.

[3] N. N. Taleb, "On the Super-Additivity and Estimation Biases of Quantile Contributions": www.fooledbyrandomness.com/longpeace.pdf.

/dev/random

ROBERT G. FERRELL

Robert G. Ferrell is an award-winning author of humor, fantasy, and science fiction, most recently *The Tol Chronicles* (www.thetolchronicles.com).

Lest I give the impression of being some horrible Luddite for what I will say later in this column, let me tell you a story from the before times. Way back in the second half of the last decade of the twentieth century (because saying it that way takes more words than simply writing "the late 1990s," and that means I have to come up with that much less actual content), I was involved in the Internet Engineering Task Force. I say "involved" but mostly what I did was join working groups and then sit in the back wishing I knew enough about whatever they were talking about to participate in some meaningful way. I was trying to be dutiful and read all of the drafts as they came out—or at least as many as I could without suffering debilitating brain damage. That got to be a little confusing and tedious after a while, so I decided to organize them a little better.

I created a simple HTML-based interface using tables for sorting and display and stuck it on a local Apache server. After a while I decided to share this tool with the world just to be a good netizen, not really expecting many to make use of it. I had also created a couple of the early MAC address OUI lookup and IANA Assigned Port Number sites. I was fond of that sort of thing back in the day. Of course, everything was written in Perl and used flat databases exclusively, so scalability was nonexistent; I did all content generation and maintenance manually, but I'd always enjoyed that stuff so it wasn't a problem...at first.

I called my IETF draft site *The Internet Report* because the name was sort of catchy and being a federal government employee I wasn't allowed to be overly creative or possess a functional imagination. And after all, it *was* a report on something rather closely connected with the Internet as an organism. Truth in advertising.

The *Report* proved to be much more popular than I had anticipated, probably because it allowed draft monkeys to skim the steady stream of documents coming out of the IETF more easily. Eventually, after I'd had the site up for a couple of years, the Internet Society took an interest and started asking me to provide various features and improvements. I really didn't have additional time to devote to the project or, in truth, expertise to do a lot more than I already had, so after a few months of this I suggested they just take the whole thing over, which they did. It may still be in existence, for all I know. I haven't looked for it in a number of years, but I hope it illustrates that I'm not in any way against technology or the Internet.

This issue of *;login:* marks a transition from bimonthly to quarterly. Coincidentally, it also marks my tenth anniversary as a columnist for this august publication. In that decade we have seen a lot of what I hesitate to call "progress" in regard to the cyberverse. While it has always been a vast wasteland, the landscape of our shared system of tubes within tubes has convolved: hundreds of petabytes of cute cat videos, ad hominem pejoratives, memes about memes about memes, and that execrable monument to self-absorption and bad photography, the ridiculous selfie.

Perhaps most significant has been the rise of social media, towering like a gossip-fueled Godzilla over the Tokyo skyline of our online existence. We now know far more about everyone on social media than we do about our neighbors and often even our own family (unless they happen to be our Facebook friends, too). Why people feel it incumbent upon themselves to share every last detail of their daily lives with the entire Internet is quite frankly beyond my meager comprehension. Not only is social media an exercise in grossly inflated oversharing, it is an addiction for many people that takes over their lives as surely as will heroin or gambling or collecting pop culture memorabilia still in the original packaging.

I know too many people who, were I to suggest that they leave their phone on the table and go for a walk, would look at me with the expression of grave concern usually reserved for a friend suddenly struck mentally ill or react with horror as though I had asked them to remove an appendage (of their own) with a rusty steak knife. People are hooked on connectivity, often to the exclusion of even basic needs such as hygiene. This dependence goes very deep. I have seen grievously injured people posting about the motor vehicle accident in which they were just involved as a result of texting while driving, without ever seemingly realizing the two events were inextricably linked. The discomfort of being unable to access the grid even for a short time is so pronounced that almost no price is too great to pay to avoid that heinous fate. I don't know whether this is a fad or the next inevitable step in our social evolution, but if the latter is the case, the world portrayed in *The Matrix* may well be more predictive documentary than dystopian fiction.

Admittedly, I do carry a smartphone and participate in various forms of social media, but were I not a novelist with pesky marketing/branding responsibilities to worry about, I would probably be far less well-connected. I heartily enjoy my sessions of glorious unplugged solitude, the boundaries of which the latest idiotic political pronouncement or news of the massive identity theft du jour cannot penetrate. It's just me, my meandering thoughts, and that strange little gray alien who taunts me with encoded millimeter-wave transmissions from behind trees and shrubs.

It's not that I am virulently opposed to all manifestations of the social media demon. I in fact enjoy chatting with my friends and seeing their little triumphs and challenges chronicled: that sort of thing is an integral part of what it means to participate in human society. The idea that we must never be more than a hair's breadth distance from the global rete or somehow wither away does disturb me greatly, however. Breathing and posting to Instagram are not synonymous, the collective wisdom of the World Wide Web notwithstanding.

This obsession with constant interaction is, I suppose, a logical step on the path of human evolution. The science fiction archetype of the futuristic human with a huge pulsing-veined cranium is being replaced by one where the giant cranium is the Internet itself, with humans serving merely as data acquisition nodes, sensors the sole purpose of which is to feed the insatiable information appetite of our distributed id juggernaut. Eventually, analysis and retrieval of that information will fade in importance as mandatory incessant data collection becomes the goal in and of itself. Machines will not merely control us: they will define us as a species. In many ways they already do.

Oops, my phone just chirped. Gotta reply to this moron's comment about my new cat meme video. Later.

Book Reviews

MARK LAMOURINE

The Logician and the Engineer: How George Boole and Claude Shannon Created the Information Age
Paul J. Nahin
Princeton University Press, 2013, 228 pages
ISBN 978-0-691-15100-7

When I started reading this book, I didn't know that I would finish it about the time of George Boole's 200th birthday, but it was a nice note. Claude Shannon worked during my lifetime and probably most of yours. The two of them don't get the attention that other luminaries of computing do, but their contributions rank with Charles Babbage, Ada Lovelace, Alan Turing, and John von Neumann. In some ways their work is more significant because of its cross-over application to physics as well as computation.

Nahin is a fan of both Boole and Shannon, and in this book he shows how Shannon's work built on Boole's to bring us the fundamental logical basis for modern computing hardware. He also wants to help the reader understand the formal results of their work. In that, he can only be as successful as the reader is dedicated. He is clear in the introduction that a certain level of mathematical background will be needed and that the reader will need to take care to follow along to get the most from the book.

In the first third of the book Nahin offers a brief biography of both Boole and Shannon. He's clearly not happy with at least Shannon's treatment by some popular modern writers. He takes a shot at James Gleick in the opening paragraph of the first chapter, quoting a somewhat disparaging comment about Shannon's sense of humor. Nahin also goes to some lengths later to highlight some of what would today be known as geeky humor. Neither biography is particularly deep or insightful, but they do give a sense of the time and influences on the men.

The middle section considers Boole's contribution to computing, the algebra of two-value logic. All of this should be familiar to anyone who's studied programming in any formal way. For a non-programmer, the discussion of De Morgan's Theorem and Karnaugh maps will give some sense of how to combine Boolean operators. Sometimes I think some programmers should remember how to reduce logical operators.

The last and largest section shows how Shannon picked up where Boole left off. Electrical relays didn't exist in Boole's time, and transistors were new during Shannon's career. Nahin explains how Shannon discovered the way in which Boole's logic could be expressed in terms of relays. It was adapted naturally to electronic circuits. This is something I did learn in college as part of a computer science course. I'm not sure whether this is commonly taught as a core course anymore, but if not, this would be a great section for a curious coder or admin to read. But, while Shannon's own work was in computer engineering, the implications didn't end there.

Shannon also analyzed the theory of signaling, describing what it meant to "send a message" in the most fundamental terms. His goal was to understand the limits of logical expression in his circuits, where mechanical relays often fail. In the process he created the field of information theory, which brings together mathematics, physics, and computation. In combination with Boole's binary logic, he produced a way of understanding the logic of quantum mechanics.

Here I agree with Nahin, that Shannon's work is underappreciated. While he didn't set out to found a new mathematical field, his straightforward inquiry has had an outsized influence, in both theory and practice, on numerous fields. Nahin's book on the contributions of Shannon and Boole is both timely and overdue.

NOTES

USENIX Member Benefits

Members of the USENIX Association receive the following benefits:

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

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

Discounts on registration fees for all USENIX conferences

Special discounts on a variety of products, books, software, and periodicals: www.usenix.org/member-services/discount-instructions

The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers

For more information regarding membership or benefits, please see www.usenix.org/membership/or contact office@usenix.org. Phone: 510-528-8649

USENIX Board of Directors

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

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

What USENIX Means to Me

by Daniel V. Klein, USENIX Board

Thirty-five years ago, I attended my first USENIX conference in Austin, TX. That's a date in the previous century, longer ago than some of my readers have been alive. USENIX had only been around for five years, and Unix itself was less than a decade old. Dennis Ritchie (and either Ken Thompson or Brian Kernighan, I forget which) were in attendance, as were other legendary computer scientists, and I was awed that they were open, approachable, and willing to talk with me, a then-young graduate student with myriad questions.

Dennis is sadly no longer with us, but Ken and Brian now work "with" me at Google. USENIX celebrated its 40th anniversary last year, along with the version of Unix I started with: V5. The longevity of the Unix OS is amazing, but even more amazing is that the CLI looks very similar to (but a lot snazzier and faster than) that early version. But if you think that the remainder of this article is going to consist of reminiscences, you're wrong. This article is about the future.

The past is as dust and the future is not yet born, and the present is all we can change. So really what I want to get you to think about is the promise of the future, based on the history I have personally witnessed, and circle back to why my present day continues to include The USENIX Association, even after 35 years. It all boils down to this:

Magic, wonder, and play.

Let's face it, computers are magic. The phone in my pocket is what we used to refer to as a supercomputer (except my phone is substantially more powerful than a Cray Y-MP). In my graduate student days, we had dreams and visions of systems that would

recognize and parse connected speech, see and analyze images, drive cars, and sift through vast amounts of data, and today we have them. We got them through incremental hard work and information sharing, with USENIX providing the medium for sharing. What is yet to come is more magic, more dreams and visions, and when you attend USENIX conferences, you get to share those dreams and hear about the visions that others have. And you get it first, because among other things, USENIX is known for firsts. This is why, at every conference I attend, for every proceedings I read, for every invited talks track I watch, I have a renewed sense of wonder. I am seeing a hint of the future with the newest magic for today.

When people ask me what I do for a living, I often tell them that I am paid to play. Sure, sometimes the work is hard, the hours long, but creating software is a game to me. I often ask myself, "Do they really pay me to do this?" because that daily sense of wonder and magic makes my job fun! The more I contribute, the more I get to appreciate the work of others, because we share the benefits of each other's work. And much of that sharing is facilitated by USENIX, by mechanisms pioneered by USENIX and its members.

Which brings me to the question of "Why USENIX, and why for 35 years?" Because we (the Association, its board, staff and members, the authors and attendees) make the magic, we share the wonder, and we play well together. USENIX is not simply about "open source"; that is only part of the equation. We are, and in my opinion always have been, about "open access." USENIX is about making everyone's job easier and more productive, because we don't hide our magic, we show it and share it. Our vast archive of (often groundbreaking) technical papers is free and open to the public. Our conferences reveal new and innovative technologies, irrespective of corporate, political, or OS bias. And perhaps most importantly, the luminaries are still open, approachable, and willing to talk.

USENIX doesn't just talk the talk; USENIX walks the walk. And the Association's mission statement is and always has been my mission statement: foster technical excellence and innovation; support and disseminate research with a practical bias; provide a neutral forum for discussion of technical issues; and encourage computing outreach into the community at large. USENIX is something I believe in, whether as an attendee, an author, a speaker, as staff, or as a board member. It's been 35 years and it's still fun because I can't wait to see what the future will bring. And I know that I'll see a lot of that future at a USENIX conference.

Refocusing the LISA Community

by Casey Henderson, USENIX Executive Director

For 24 years, the LISA Special Interest Group for Sysadmins (LISA SIG, formerly known as SAGE) has been a resource and virtual meeting ground for the sysadmin community at USENIX. Despite its sometimes tumultuous history, dedicated members have provided content for Short Topics books, shared insight with colleagues via mailing lists, and helped advance the state of the profession via the creation of the System Administrators' Code of Ethics, contributions to salary surveys, postings to colleagues via the Jobs Board, and nominations for the Outstanding Achievement Award.

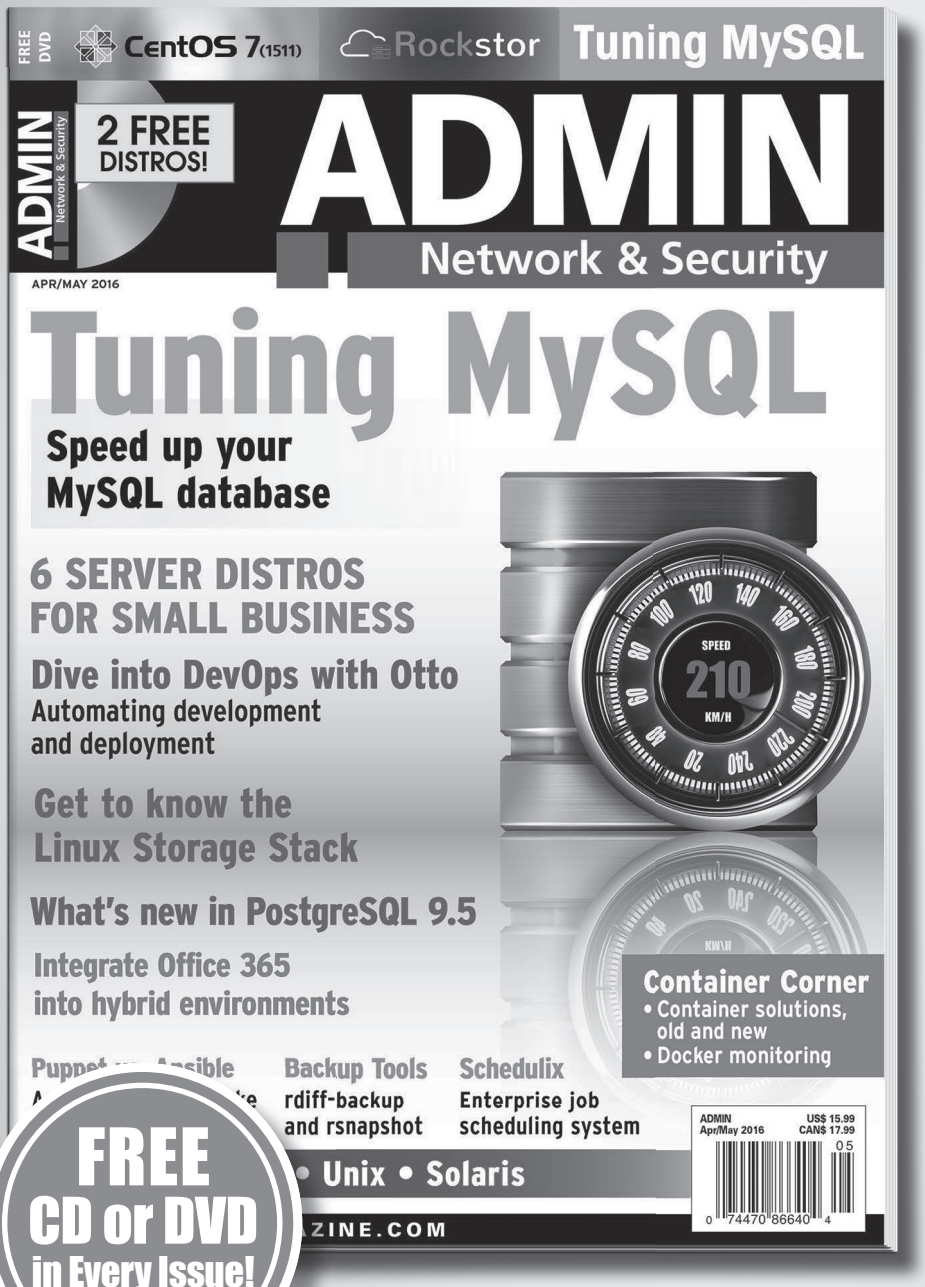
With full recognition of this history and value of the LISA SIG, USENIX has made the decision to retire it at the end of 2016. In recent years, our efforts to serve the sysadmin community have focused on reengineering and revitalizing the annual LISA conference to ensure its relevance and long-term sustainability, as well as creating and nurturing SREcon for the emerging, related field of site reliability engineering. These ongoing efforts have been successful and well received by the community, so this is where we are going to focus our energies to help support the sysadmin community.

The USENIX Board of Directors and staff did not make this decision lightly. To inform our deliberation, we convened a committee comprised of community members to explore the possible future paths of the SIG. The committee surveyed SIG members, analyzed the results, and presented their recommendations to us. After weighing all the factors, we determined that the best path forward is to continue building community through the LISA conference itself.

LISA SIG resources, including the Short Topics books and Code of Ethics, will continue to be available on the USENIX Web site. All active memberships will continue to receive the current slate of benefits, including the LISA conference discount, through the end of the year.

We look forward to continuing to serve this community that continues to be an integral part of USENIX, and hope to see you at LISA16 in Boston, December 4–9, 2016.

REAL SOLUTIONS FOR REAL NETWORKS



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 Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

POSTMASTER
Send Address Changes to *;login:*
2560 Ninth Street, Suite 215
Berkeley, CA 94710

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES

SAVE THE DATE!

2016 USENIX Annual Technical Conference

JUNE 22–24, 2016 • DENVER, CO
www.usenix.org/atc16

USENIX ATC '16 brings leading systems researchers together for cutting-edge systems research and unlimited opportunities to gain insight into a variety of must-know topics, including virtualization, system administration, cloud computing, security, and networking.

Co-located with USENIX ATC '16:

SOUPS 2016

Twelfth Symposium on Usable Privacy and Security

JUNE 22–24, 2016
www.usenix.org/soups2016

SOUPS 2016 will bring together an interdisciplinary group of researchers and practitioners in human computer interaction, security, and privacy. The program will feature technical papers, workshops and tutorials, a poster session, panels and invited talks, and lightning talks.

HotCloud '16

8th USENIX Workshop on Hot Topics in Cloud Computing

JUNE 20–21, 2016
www.usenix.org/hotcloud16

Researchers and practitioners at HotCloud '16 share their perspectives, report on recent developments, discuss research in progress, and identify new/emerging "hot" trends in cloud computing technologies.

HotStorage '16

8th USENIX Workshop on Hot Topics in Storage and File Systems

JUNE 20–21, 2016
www.usenix.org/hotstorage16

HotStorage '16 is an ideal forum for leading storage systems researchers to exchange ideas and discuss the design, implementation, management, and evaluation of these systems.