;login:
FALL 2017 VOL. 42, NO. 3

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

LISA17
October 29–November 3, 2017, San Francisco, CA, USA
www.usenix.org/lisa17

Enigma 2018
January 16–18, 2018, Santa Clara, CA, USA
www.usenix.org/enigma2018

FAST '18: 16th USENIX Conference on File and Storage Technologies
February 12–15, 2018, Oakland, CA, USA
Submissions due September 28, 2017
www.usenix.org/fast18

SREcon18 Americas
March 27–29, 2018, Santa Clara, CA, USA
www.usenix.org/srecon18americas

NSDI '18: 15th USENIX Symposium on Networked Systems Design and Implementation
April 9–11, 2018, Renton, WA, USA
Paper titles and abstracts due September 18, 2017
www.usenix.org/nsdi18

SREcon18 Asia/Australia
June 6–8, 2018, Singapore
www.usenix.org/srecon18asia

USENIX ATC '18: 2018 USENIX Annual Technical Conference
July 11–13, 2018, Boston, MA, USA
Submissions due: February 6, 2018
www.usenix.org/atc18

USENIX Security '18: 27th USENIX Security Symposium
August 15–17, 2018, Baltimore, MD, USA
Submissions due February 8, 2018

Co-located with USENIX Security '18

SOUPS 2018: Fourteenth Symposium on Usable Privacy and Security
August 12–14, 2018

SREcon18 Europe/Middle East/Africa
August 29–31, 2018, Dusseldorf, Germany
www.usenix.org/srecon18europe

OSDI '18: 13th USENIX Symposium on Operating Systems Design and Implementation
October 8–10, 2018, Carlsbad, CA, USA
www.usenix.org/osdi18

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

;login:
FALL 2017 VOL. 42, NO. 3

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Michele Nelson
michele@usenix.org

COPY EDITORS
Steve Gilmartin
Amber Ankerholz

PRODUCTION
Arnold Gatilao
Jasmine Murcia

TYPESETTER
Star Type
startype@comcast.net

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

www.usenix.org

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

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

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

EDITORIAL

Musings
Theory of Mind

RIK FARROW

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

I am going to depart from the realm of computer science briefly, because I want to discuss a problem which is rampant in software design and papers. The problem involves the Theory of Mind (ToM), the "ability to attribute mental states—beliefs, intents, desires, pretending, knowledge—to oneself and others" [1]. But while ToM generally refers to interpersonal relations or philosophy [2], I am going to focus on the part about attributing knowledge to the people who will use your software or read your papers.

I was talking to an old friend, who had been complaining about his son. My friend said that, unlike his son, he just decided one day that he would focus on work and become responsible. If he could do it, so could his son.

I found myself suggesting that my friend look up Theory of Mind. Just because my friend could resolve to buckle down doesn't mean that other people would, or could, behave just like he did.

I remembered ToM from college psychology classes from many years ago. Today, deficiencies in ToM are now associated with autism among other disorders. That really isn't what I am referring to. Rather, thinking that because you did or know something, so should anyone else, just seemed a bit, well, not quite sane to me.

Theory of Mind and CS

Where ToM and CS intersect is a bit different, having more to do with culture. As an editor, I am constantly running into this, as I read articles or papers where the authors assume that you have the same background and understand the same jargon that they do. After all, all the people they work with speak that jargon and have the same background information, right?

I've written an editorial about the importance of being able to write clearly [3], and ignoring ToM can mean rejected papers. You really shouldn't assume that people will just accept your brilliant research if you can't articulate it clearly.

I wrote a column about a software design issue many years ago [4] that dealt with ToM, but without saying so directly. I wrote that column because I had observed that most people had a difficult time with state machines. I found I could set friends' digital watches for them, even though they couldn't, because I understood the watches (and their two or three buttons) had different purposes depending on their current state. I've since discovered that clocks with four buttons, and no manuals, have so many states that even I have trouble setting them.

Today we get devices, such as smartphones, complete with state machines implementing the user interface. There are no manuals—what you need to do is find someone who has already communicated with someone who knows how the damn things work. Of course, the next update means that what you learned no longer works, and you need to make another social connection to understand the new interface. And speaking of overloaded interfaces, the most popular smartphone uses a single button that has a multitude of different purposes, depending on the software's current state. What an amazing design—for engineers.

Consider how this works in the place where the new interface gets developed. Someone comes up with some new UI widget and shows a coworker how to use it. The knowledge gets spread to others, and if the widget is compelling enough, it appears in the public version. But only insiders initially know how to make it work. It's like building systems where every new feature is an Easter egg [5].

The Lineup

We have many articles related to cloud in this issue. We lead off with an article explaining the design goals and implementation of VFP, Microsoft's very different version of vswitch. I met Daniel Firestone during NSDI '17, where he presented the only industry paper, one which provides more implementation details about VFP.

Reid Priedhorsky and Tim Randles, of Los Alamos National Laboratories, describe their open source solution, Charliecloud. They determined that a lighter-weight solution than Docker would work best for HPC. To help maintain a familiar interface, containers are still built using Docker but are run via Charliecloud. Their article also helped me understand more about containers.

I interviewed James Bottomley. James has written about containers versus VMs for *;login:* [6], and I wanted to probe his viewpoints further now that he has changed jobs. James explains a lot more about the difference between containers and VMs, the Linux system calls used to set containers up, and why containers haven't been embraced by many vendors.

We have another article from NSDI '17. "Knockoff," by Dou, Chen, and Flinn, examines the tradeoff between recomputing data in the cloud and the cost of copying data. Hint: oftentimes, recomputing is both cheaper and faster.

In the system administration and SRE section, we have two articles. Carata, Chick, and Sohan describe Resourceful, a tool they developed for use in OS research at Cambridge and have now open sourced. You use the Resourceful API to instrument apps, allowing you to produce performance data from the kernel about very specific activities relating just to portions of an app.

Roy et al. explain how they instrumented servers and network hardware at Facebook and discovered how they could uncover subtle network problems faster than the current monitoring tools used. Their approach does rely on having a well-balanced workload to start with but should work in any well-tuned environment.

In the security section, Escobedo et al. talk about how the BeyondCorp team at Google worked to make the transition from traditional VPNs to BC easier for both current users and new hires. They share key insights and techniques into how others might smooth the migration of users to a very different method of application and server access.

Geoffroy Couprie and Pierre Chifflier reprise work they have done (and presented at the IEEE LangSec '17 workshop) about making existing software more secure. Rather than attempting the Sisyphean task of rewriting software from scratch, the authors focus on input parsers, using Rust, with a compiler that fails to compile dangerous code by accident, and `nom`, a tool that makes building safe parsers easier.

David Beazley tells us how surprised he was when he witnessed a Python programmer writing code concurrently with a testing framework. David explains how other Python programmers can take advantage of using a very simple technique to improve writing even very simple apps.

David Blank-Edelman has another edition of his "Flying Perl" series. Having read about a programmer who had used Python to answer the question "Which airports are closest to each other?" David shows us how to perform the same task in Perl.

Dave Josephsen wanted to show his coworkers the real value of being able to measure performance. Using Phaser.js for the visualization portion and Go for the server, Dave quickly threw together a tool (demonstrated with a YouTube video) that uncovered bottlenecks caused by unbalanced load.

Chris "Mac" McEniry has taken over the task of writing a Go column. Mac begins by adding TLS support to Kelsey Hightower's `gls`, something Kelsey had wanted to do when he wrote his column. But, as you will see, adding TLS, while easy in Go, deserves its own column.

Margo Seltzer has contributed to what we hope will be a new column on education. Margo has converted her operating system course at Harvard to use *flipping*. Flipping involves swapping the usual way that material is taught, so students begin with self-study, then work on classroom projects. Instead of lecture, which can leave many students lost, flipping means that students' questions and problems become the focus. By the way, I invite other teachers to contribute to future versions of the education column.

Dan Geer has written his annual column about the Index of Cyber Security. The ICS relies on polling security practitioners, and in turn these professionals get to see how the others in their field responded to questions about the same issues. Dan shares the answers about four different questions, including this issue's favorite topic, the cloud.

Robert Ferrell has hopped on the strike-back bandwagon. If your organization is under attack, why wait for government or professional assistance when you can launch attacks yourself against the presumed offender? Robert suggests a handful of attack tools that you can use.

EDITORIAL

Musings: Theory of Mind

Mark Lamourine has two book reviews this month. The first is about using CoreOS and the second about a short book on RESTful standards.

While it was my friend that got me going about the Theory of Mind, I do believe that it is relevant to most people. Theory of Mind applies when giving directions: for example, "Turn right where the Sinclair gas station used to be" relies on local knowledge about something that disappeared long ago. Where I live, you might still get instructions like "Turn left at the 'Y'," an intersection that is now a circle and hasn't been a 'Y' for over 25 years.

In the worlds of our own specialties, we also have the problem of insiders' knowledge. If we intend to communicate effectively, we can't assume that our audience knows what we do. If that were true, why would we even be addressing them? ToM, or rather, the assumption that others have the same knowledge or beliefs that we do, is an all-too-easy trap to fall into. Do us all a favor and write for your audience, not for your own in-group.

References

[1] Wikipedia, "Theory of Mind": https://en.wikipedia.org/wiki/Theory_of_mind.

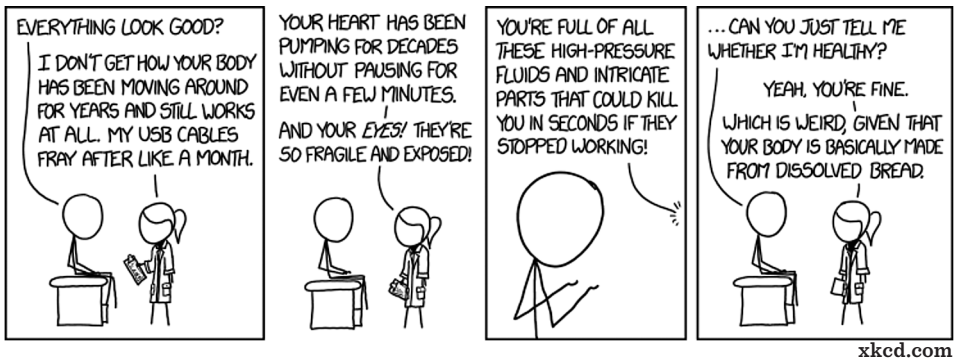
[2] Philosophy and ToM: http://www.iep.utm.edu/theomind/.

[3] R. Farrow, "Musings," *;login:*, vol. 41, no. 2 (Summer 2016): https://www.usenix.org/system/files/login/articles/login_summer16_01_farrow.pdf.

[4] R. Farrow, "Musings," *;login:*, vol. 23, no. 5 (August 1998): http://web.archive.org/web/20111110024139/http://www.usenix.org/publications/login/1998-8/musings.html.

[5] Definition of Easter egg: http://www.webopedia.com/TERM/E/easter_egg.html.

[6] J. Bottomley and P. Emelyanov, "Containers," *;login*, vol. 39, no. 5 (October 2014): https://www.usenix.org/system/files/login/articles/login_1410_02-bottomley.pdf.



ENIGMA

A USENIX CONFERENCE

SECURITY AND PRIVACY IDEAS THAT MATTER

Enigma centers on a single track of engaging talks covering a wide range of topics in security and privacy. Our goal is to clearly explain emerging threats and defenses in the growing intersection of society and technology, and to foster an intelligent and informed conversation within the community and the world. We view diversity as a key enabler for this goal and actively work to ensure that the Enigma community encourages and welcomes participation from all employment sectors, racial and ethnic backgrounds, nationalities, and genders.

Enigma is committed to fostering an open, collaborative, and respectful environment. Enigma and USENIX are also dedicated to open science and open conversations, and will make all talk media freely available on the USENIX Web site.

PROGRAM CO-CHAIRS

Bryan Payne,
Netflix

Franziska Roesner,
University of Washington

The full program and registration will be available in October.

enigma.usenix.org

JAN 16–18, 2018

SANTA CLARA, CA, USA

usenix ASSOCIATION

VFP

A Virtual Switch Platform for Host SDN in the Public Cloud

DANIEL FIRESTONE

Daniel Firestone is the Tech Lead and Manager for the Azure Host Networking group at Microsoft. His team builds the Azure virtual switch, which serves as the datapath for Azure virtual networks, as well as SmartNIC, the Azure platform for offloading host network functions to reconfigurable FPGA hardware and Azure's RDMA stack. Before Azure, Daniel did his undergraduate studies at MIT. fstone@microsoft.com

The Virtual Filtering Platform (VFP) is a cloud-scale programmable virtual switch providing scalable SDN policy to one of the world's largest clouds, Microsoft Azure. It was designed from the ground up to handle the programmability needs of Azure's many SDN applications, the scalability needs of deployments of millions of servers, and to deliver the fastest virtual networks in the public cloud to Azure's VMs through hardware offloads.

We, the VFP team, describe here our goals and motivations in building VFP, VFP's design, and lessons we learned from production deployments. We also compare our design with that of other popular host SDN technologies such as OpenFlow [2] and Open vSwitch (OVS) [3] to show how our constraints in the public cloud can differ from those of popular open source projects. We believe these lessons can benefit the SDN community at large. More details of our design can be found in our recent NSDI paper [1].

The rise of public cloud workloads, such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform, has created a new scale of datacenter computing, with vendors regularly reporting server counts in the millions. These vendors not only have to provide scale and high density of VMs to customers, but must provide rich network semantics, such as private virtual networks with customer supplied address spaces, scalable L4 load balancers, security groups and ACLs, virtual routing tables, bandwidth metering, QoS, and more. This policy is sufficiently complex that it isn't feasible to implement at scale in traditional switch hardware.

Instead this is often implemented using Software-Defined Networking (SDN) on the VM hosts, in the virtual switch (vswitch) connecting VMs to the network, which scales well with the number of servers and allows the physical network to be simple, scalable, and very fast. As a large public cloud provider, Azure has built its cloud network on host-based SDN technologies. Much of the focus around SDN in recent years has been on building scalable and flexible network controllers and services—however, the design of the programmable vswitch is equally important. It has the dual and often conflicting requirements of a highly programmable dataplane, with high performance and low overhead. VFP is our solution to these problems.

Design Goals and Rationale

As a motivating example for VFP, we consider a simple scenario requiring four host policies used for O(1M) VM hosts in a cloud. Each policy is programmed by its own SDN controller and requires both high performance and SR-IOV offload support: the first is virtual networking, allowing a customer to define their own private network with their own IP addresses, despite running on shared multi-tenant infrastructure. Our virtual networks (VNETs) are based on the design from VL2 [4]. Second is an L4 (TCP/UDP connection) load balancer based on Ananta [5], which scales by running the load balancing NAT in the vswitch on end hosts, leaving the in-network load balancers stateless and scalable. We also

include a stateful firewall and per-destination traffic metering for billing.

Originally, we built independent networking drivers for each of these host functions. As host networking became our main tool for virtualization policy, we decided to create VFP in 2011 because this model wasn't scaling. Instead, we created a single platform based on the Match-Action Table (MAT) model popularized by projects such as OpenFlow.

Original Goals

Our original goals for the VFP project were as follows:

1. *Provide a programming model allowing for multiple simultaneous, independent network controllers to program network applications, minimizing cross-controller dependencies.*

Implementations of OpenFlow and similar MAT models often assume a single distributed network controller that owns programming the switch. Our experience is that this model doesn't fit cloud development of SDN—instead, independent teams often build new network controllers and agents for those applications. This model reduces complex dependencies, scales better, and is more serviceable than adding logic to existing controllers. We needed a design that not only allows controllers to independently create and program flow tables, but enforces good layering and boundaries between them (e.g., disallows rules to have arbitrary GOTOs to other tables) so that new controllers can be developed to add functionality without old controllers needing to take their behavior into account.

2. *Provide a MAT programming model capable of using connections as a base primitive, rather than just packets—stateful rules as first-class objects.*

OpenFlow's original MAT model derives historically from programming switching or routing ASICs, and assumes that packet classification is stateless. However, we found our controllers required policies for connections, not just packets—for example, end users often found it more useful to secure their VMs using stateful access control lists (ACLs) (e.g., allowing outbound connections but not inbound ones) rather than stateless ACLs used in commercial switches. Controllers also needed NAT (e.g., Ananta) and other stateful policies. Stateful policy is more tractable in soft switches than in ASIC ones, and we believe a MAT model should take advantage of that.

3. *Provide a programming model that allows controllers to define their own policy and actions, rather than implementing fixed sets of network policies for predefined scenarios.*

Due to limitations of the MAT model provided by OpenFlow (historically, a limited set of actions, limited rule scalability, and no table typing), OpenFlow switches such as OVS have added virtualization functionality outside of the MAT model. For example, constructing virtual networks is accomplished via a virtual tunnel endpoint (VTEP) schema in OVSDB, rather than rules specifying which packets to encapsulate (encap) and decapsulate (decap) and how to do so.

We prefer instead to base all functionality on the MAT model, trying to push as much logic as possible into the controllers while leaving the core dataplane in the vswitch. For instance, rather than a schema that defines what a VNET is, a VNET can be implemented using programmable encap and decap rules matching appropriate conditions, leaving the definition of a VNET in the controller. We've found this greatly reduces the need to continuously extend the dataplane every time the definition of a VNET changes.

Later Goals Based on Production Lessons

Based on lessons from initial deployments of VFP, we added the following goals for VFPv2, a major update in 2013-14, mostly around serviceability and performance:

1. *Provide a serviceability model allowing for frequent deployments and updates without requiring reboots or interrupting VM connectivity for stateful flows, and strong service monitoring.*

As our scale grew dramatically to over O(1M) hosts, more controllers built apps on top of VFP, more engineers joined us, and we found more demand than ever for frequent updates, both features and bug fixes. In Infrastructure as a Service (IaaS) models, we also found customers were not tolerant of taking downtime for individual VMs for updates.

2. *Provide very high packet rates, even with a large number of tables and rules, via extensive caching.*

Over time we found more and more network controllers being built as the host SDN model became more popular, and soon we had deployments with large numbers of flow tables (10+), each with many rules, reducing performance as packets had to traverse each table. At the same time, VM density on hosts was increasing, pushing us from 1G to 10G to 40G and even faster NICs. We needed to find a way to scale to more policy without impacting performance and concluded we needed to perform compilation of flow actions across tables, and use extensive flow caching such that packets on existing flows would match precompiled actions without having to traverse tables.

3. *Implement an efficient mechanism to offload flow policy to programmable NICs, without assuming complex rule processing.*

As we scaled to 40G+ NICs, we wanted to offload policy to NICs themselves to support SR-IOV, which lets NICs indicate packets directly to VMs without going through the host. However, as controllers created more flow tables with more rules, we concluded that directly offloading those tables would require prohibitively expensive hardware resources for server-class NICs. Instead we wanted an offload model that would work well with

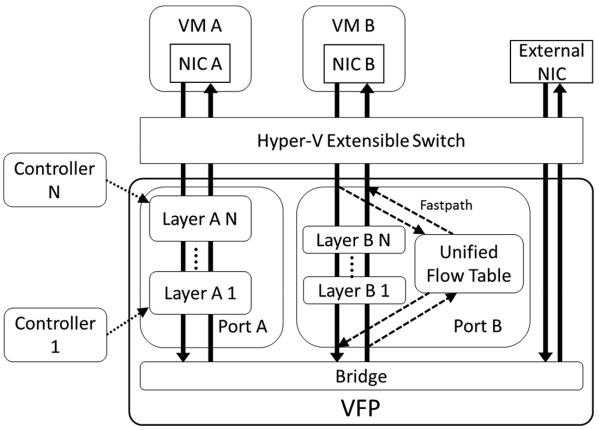


Figure 1: Overview of VFP design

our precompiled exact-match flows, requiring hardware to only support a large table of cached flows in DRAM and our associated action language.

VFP Overview

Figure 1 shows a model of the VFP design, which is described in subsequent sections. VFP operates on top of Hyper-V's extensible switch as a packet filter. Its programming model is based on layers, MATs that support a multi-controller model. VFP's packet processor includes a fastpath through Unified Flow Tables and a classifier used to match rules in the MAT layers.

The core VFP model assumes a switch with multiple ports that are connected to virtual NICs (VNICs). VFP filters traffic from a VNIC to the switch, and from the switch to a VNIC. All VFP policy is attached to a specific port. From the perspective of a VM with a VNIC attached to a port, ingress traffic to the switch is considered to be "outbound" traffic from the VM, and egress traffic from the switch is considered to be "inbound" traffic to the VM. VFP's API and its policies are based on the inbound/outbound model.

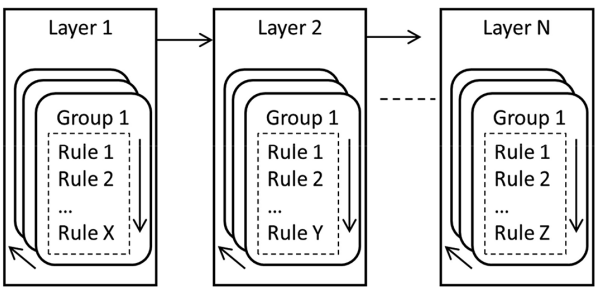


Figure 2: VFP objects: layers, groups, and rules

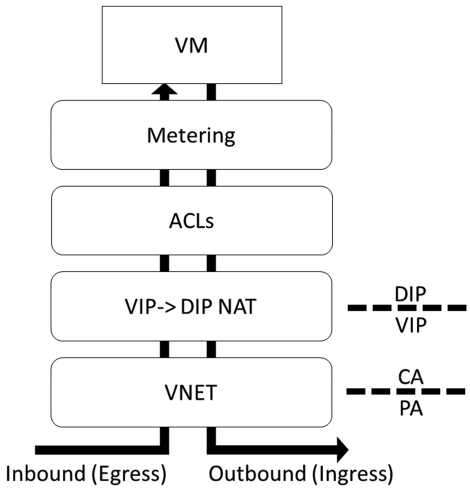


Figure 3: Example VFP layers with boundaries

Programming Model

VFP's core programming model is based on a hierarchy of VFP objects that controllers can create and program to specify their SDN policy, with ports containing layers of policy made up of groups of rules.

Layers

VFP divides a port's policy into layers. Layers are the basic Match Action Tables that controllers use to specify their policy. They can be created and managed separately by different controllers. Logically, packets into a VM go through each layer one by one, matching rules in each based on the state of the packet after the action performed in the previous layer, with returning packets coming back in the opposite direction.

Figure 3 shows layers for our SDN deployment example. A VNET layer creates a customer address (CA) / physical address (PA) boundary by having encapsulation rules on the outbound path and decapsulation rules on the inbound path. In addition, an ACL layer for a stateful firewall sits above our Ananta NAT layer. The security controller, having placed it here with respect to those boundaries, knows that it can program policies matching dynamic IP addresses (DIPs) of VMs in CA space. Finally, a metering layer used for billing sits at the top next to the VM, where it can meter traffic exactly as the customer in the VM sees it.

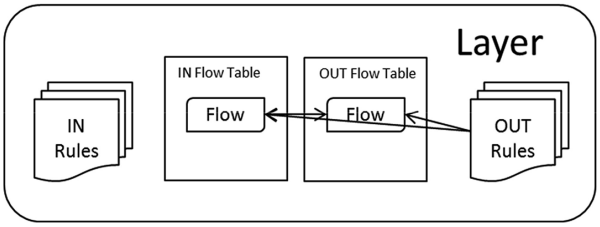


Figure 4: A layer with a stateful flow

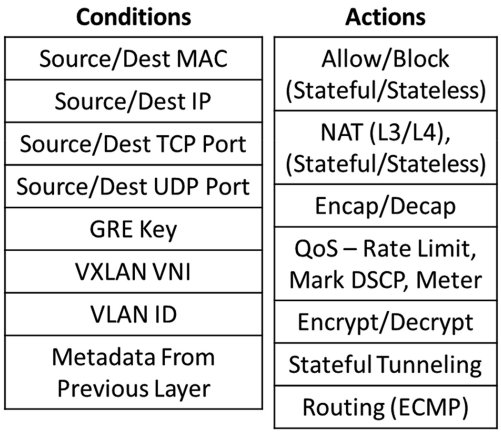


Figure 5: Example conditions and actions

Layering also gives us a good model on which to implement stateful policy. We keep flow state on a layer with a hash table tracking all TCP, UDP, or RDMA connections in either direction. When a stateful rule is matched, it creates both an inbound and outbound flow in the layer flow tables, with appropriate actions in each direction (e.g., NAT or ACL).

Rules

Rules are the entities that perform actions on matching packets in the MAT model. Per original goal 3, rules allow the controller to be as expressive as possible while minimizing fixed policy in the dataplane. Rules are made up of two parts: a condition list, specified via a list of conditions, and an action. Example conditions and actions are listed in Figure 5.

Rules can be organized into groups for purposes of doing transactional update/replace operations, or to split a port into sub-interfaces (e.g., allow creation of independent policies for multiple Docker-style containers behind a single port).

Packet Processor and Flow Compiler

A primary innovation in VFPv2 was the introduction of a central packet processor. We took inspiration from a common design in network ASIC pipelines e.g.,—parse the relevant metadata from the packet and act on the metadata rather than on the packet, only touching the packet at the end of the pipeline once all decisions have been made. We compile and store flows as we see packets. Our just-in-time flow compiler includes a parser, an action language, an engine for manipulating parsed metadata and actions, and a flow cache.

Unified FlowIDs

VFP's packet processor begins with parsing. One each of an L2/L3/L4 header (as defined in Table 1) form a header group, and the relevant fields of a header group form a single FlowID. The tuple of all FlowIDs in a packet is a Unified FlowID (UFID)—the output of the parser.

Header	Parameters
Ethernet (L2)	Source MAC, Dest MAC
IP (L3)	Source IP, Dest IP, ToS (DSCP+ EC)
Encapsulation (L4)	Encapsulation Type Tenant ID, Entropy (Optional)
TCP/UDP (L4)	Source Port, Dest Port, TCP Flags (note: does not support Push/Pop)

Table 1: Valid parameters for each header type

Action	Notes
Pop	Remove this header.
Push	Push this header onto the packet. All header parameters for creating the new header are specified.
Modify	Modify this header. All header parameters needed are optional, but at least one is specified.
Ignore	Leave this header as is.

Table 2: Header Transposition actions

Header	NAT	Encap	Decap	Encap+NAT
Outer Ethernet	Ignore	Push (SMAC, DMAC)	Pop	Push (SMAC, DMAC)
Outer IP	Modify (SIP, DIP)	Push (SIP, DIP)	Pop	Push (SIP, DIP)
GRE	Not Present	Push (Key)	Pop	Push (Key)
Inner Ethernet	Not Present	Modify (DMAC)	Ignore	Modify (DMAC)
Inner IP	Not Present	Ignore	Ignore	Modify (SIP, DIP)
TCP/ UDP	Modify (SPt, DPt)	Ignore	Ignore	Modify (SPt, DPt)

Table 3: Example Header Transposition

Header Transpositions

Our action primitives, Header Transpositions (HTs), so called because they change or shift fields throughout a packet, are a list of paramaterizable header actions, one for each header. Actions (defined in Table 2) are to *Push* a header (add it to the header stack), *Modify* a header (change fields within a given header), *Pop* a header (remove it from the header stack), or *Ignore* a header (pass over it). Table 3 shows examples of a NAT HT used by Ananta, and encap/decap HTs used by VL2.

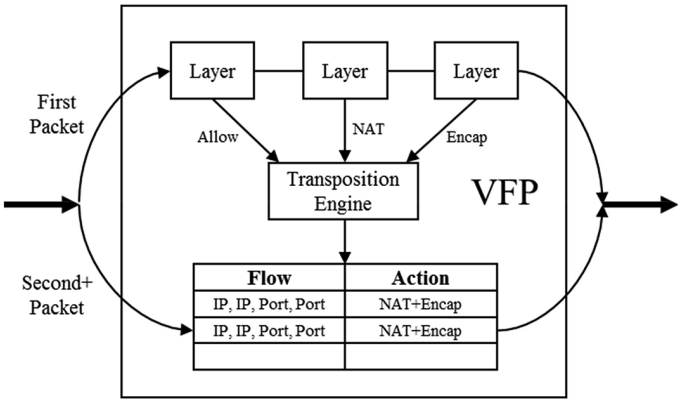


Figure 6: VFP Unified Flow Table

VFP creates an action for a UFID match by composing HTs from matched rules in each layer. For example, a packet passing the example Ananta NAT layer and the VL2 VNET encap layer may end up with the composite Encap+NAT transposition in Table 3.

Unified Flow Tables and Caching

The intuition behind our flow compiler is that the action for a UFID is relatively stable over the lifetime of a flow—so we can cache the UFID with the resulting HT from the engine. The resulting flow table where the compiler caches UFs is called the Unified Flow Table (UFT).

With the UFT, we segment our datapath into a fastpath and a slowpath. On the first packet of a TCP flow, we take a slowpath, running the transposition engine and matching at each layer against rules. On subsequent packets, VFP takes a fastpath, matching a unified flow via UFID and applying a transposition directly. This operation is independent of the layers or rules in VFP.

Operationalizing VFP

As a production cloud service, VFP's design must take into account serviceability, monitoring, and diagnostics. During update, we first pause the datapath, then detach VFP from the stack, uninstall VFP (which acts as a loadable kernel driver), install a new VFP, attach it to the stack, and restart the datapath. This operation looks like a brief connectivity blip to VMs, while the NIC stays up. To keep stateful flows alive across updates, we support serialization and deserialization for all policy and state in VFP on a port. VFP also supports live migration of VMs. During the blackout time of the migration, the port state is serialized out of the original host and deserialized on the new host.

VFP implements hundreds of performance counters and flow statistics, on per port, per layer, and per rule bases, as well as extensive flow statistics. This information is continuously uploaded to a central monitoring service, providing dashboards on which we can monitor flow utilization, drops, connection resets, and more, either on a VM or aggregated on a cluster/host/VNET basis. VFP also supports remote debugging and tracing for rules and policies as part of its diagnostics suite.

Hardware Offloads and Performance

VFP has long used standard stateless offloads (VXLAN/NVGRE encapsulation, QoS bandwidth caps, and reservations for ports, etc.) to achieve line rate with SDN policy. But to enable added goal 3 of full SR-IOV offload and host bypass, we built logic to directly offload our unified flows. These are exact-match flows representing each connection on the system, so they can be implemented in hardware via a large hash table, typically in inexpensive DRAM. In this model, the first packet of a new flow goes through software classification to determine the UF, which is then offloaded.

We've used this mechanism to enable SR-IOV in our datacenters with VFP policy offload on custom FPGA-based SmartNICs we've deployed on all new Azure servers. As a result we've seen bidirectional 32Gbps+ VNICs with near-zero host CPU and <25µs end-to-end TCP latencies inside a VNET.

Experiences

We have deployed 22 major releases of VFP since 2012. VFP runs on all Azure servers, powering millions of VMs, petabits per second of traffic, and providing load balancing for exabytes of storage, in hundreds of datacenters in over 30 regions across the world. In addition, we are releasing VFP publicly as part of Windows Server 2016 for on-premises workloads, as we have seen it meet all of the major goals listed above in production.

Over six years of developing and supporting VFP, we learned a number of lessons of value:

- **L4 flow caching is sufficient**. We didn't find a use for multi-tiered flow caching such as OVS megaflows. The two main reasons: being entirely in the kernel allowed us to have a faster slowpath, and our use of a stateful NAT created an action for every L4 flow and reduced the usefulness of ternary flow caching.
- **Design for statefulness from day 1.** The above point is an example of a larger lesson: support for stateful connections as a first-class primitive in a MAT is fundamental and must be considered in every aspect of a MAT design. It should not be bolted on later.
- **Layering is critical.** Some of our policy could be implemented as a special case of OpenFlow tables with GOTOs chaining them together, with separate inbound and outbound tables. But we found that our controllers needed clear layering semantics or else they couldn't reverse their policy correctly with respect to other controllers.

- **GOTO considered harmful.** Controllers will implement policy in the simplest way needed to solve a problem, but that may not be compatible with future controllers adding policy. We needed to be vigilant in not only providing layering but enforcing it. We see this layering enforcement not as a limitation compared to OpenFlow's GOTO table model but, instead, as the key feature that made multi-controller designs work for multiple years running.
- **IaaS cannot handle downtime.** We found that customer IaaS workloads cared deeply about uptime for each VM, not just their service as a whole. We needed to design all updates to minimize downtime and provide guarantees for low blackout times.
- **Design for serviceability.** Serialization is another design point that turned out to pervade all of our logic—in order to regularly update VFP without impact to VMs, we needed to consider serviceability in any new VFP feature or action type.
- **Decouple the wire protocol from the dataplane.** We've seen enough controllers/agents implement wire protocols with different distributed systems models to support O(1M) scale that we believe our decision to separate VFP's API from any wire protocol was a critical choice for VFP's success. For example, bandwidth metering rules are pushed by a controller, but VNET required a VL2-style directory system (and an agent that understands that policy comes from a different controller than pulled mappings) to scale.
- **Everything is an action.** Modeling VL2-style encap/decap as actions rather than tunnel interfaces was a good choice. It enabled a single table lookup for all packets—no traversing a tunnel interface with tables before and after. The resulting HT language combining encap/decap with header modification enabled single-table hardware offload.
- **Design for end-to-end monitoring.** Determining network health of VMs despite not having direct access to them is a challenge. We found many uses for in-band monitoring with packet injectors and auto-responders implemented as VFP rule actions. We used these to build monitoring that traces the E2E path from the VM-host boundary. For example, we implemented Pingmesh-like [6] monitoring for VL2 VNETs.
- **Commercial NIC hardware isn't ideal for SDN**. Despite years of interest from NIC vendors about offloading SDN policy with SR-IOV, we have seen no success cases of NIC ASIC vendors supporting our policy as a direct offload. Instead, large multicore NPUs are often used. We used custom FPGA-based hardware to ship SR-IOV in Azure, which we found was lower latency and more efficient.

Conclusions and Future Work

We introduced the Virtual Filtering Platform (VFP), our cloud scale vswitch for host SDN policy in Microsoft Azure. We discussed how our design achieved our dual goals of programmability and scalability. We discussed concerns around serviceability, monitoring, and diagnostics in production environments, and provided performance results, data, and lessons from real use. Future areas of investigation include new hardware models of SDN and extending VFP's offload language.

References

[1] D. Firestone, "VFP: A Virtual Switch Platform for Host SDN in the Public Cloud," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*: https://www.usenix.org/system/files/conference/nsdi17/nsdi17-firestone.pdf.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review,* vol. 38, no. 2 (April 2008): http://ccr.sigcomm.org/online/files/p69-v38n2n-mckeown.pdf.

[3] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, M. Casado, "The Design and Implementation of Open vSwitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*: https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-pfaff.pdf.

[4] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," in *Proceedings of the ACM Conference on Data Communication (SIGCOMM '09),* pp. 51–62: https://www.researchgate.net/publication/234805283_VL2_A_Scalable_and_Flexible_Data_Center_Network.

[5] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, N. Karri, "Ananta: Cloud Scale Load Balancing," in *Proceedings of the ACM Conference on Data Communication (SIGCOMM '13),* pp. 207–218: http://conferences.sigcomm.org/sigcomm/2013/papers/sigcomm/p207.pdf.

[6] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z. Lin, V. Kurien, "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis," in *Proceedings of the ACM Conference on Data Communication (SIGCOMM '15)*: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/11/pingmesh_sigcomm2015.pdf.

Linux Containers for Fun and Profit in HPC

REID PRIEDHORSKY AND TIM RANDLES

Reid Priedhorsky is a Staff Scientist at Los Alamos National Laboratory. Prior to Los Alamos, he was a Research Staff member at IBM Research. He holds a PhD in computer science from the University of Minnesota and a BA, also in computer science, from Macalester College. His work focuses on large-scale data analysis from both systems and applications perspectives. Recent lines of research include using social media and Web traffic to monitor and forecast the spread of disease as well as developing technology to bring data-intensive computing and user-defined software stacks to existing high-performance computing systems. In his spare time, he enjoys reading, bicycling, hiking (especially in the mountains and deserts of the American West), tinkering with things, photography, and hanging out with his wife and son. reidpr@lanl.gov

Tim Randles has been working in scientific, research, and high-performance computing for many years, first in the Department of Physics at the Ohio State University, then at the Maui High Performance Computing Center, and most recently as a member of the HPC Division at Los Alamos National Laboratory. His current work is focused on the convergence of the high performance and cloud computing worlds. When Tim isn't working, he enjoys brewing beer, cheesemaking, taking hikes, and working on computer games. He lives in Santa Fe with his wife and three cats. In an ideal world they'd also have a few goats and some chickens. trandles@lanl.gov

This article outlines options for user-defined software stacks from an HPC perspective. We argue that a lightweight approach based on Linux containers is most suitable for HPC centers because it provides the best balance between maximizing service of user needs and minimizing risks. We discuss how containers work and several implementations, including Charliecloud, our own open-source solution developed at Los Alamos.

Innovating Faster in HPC

Users of high performance computing resources have always been asking for more, better, and different software environments to support their scientific codes. We've identified four reasons why:

- **Software dependencies** not provided by the center. Examples include libraries that are numerous, unusual, or simply newer or older; configuration incompatibilities; and build-time resources such as Internet access.
- **Portability** of environments between resources. For example, it is helpful to have the same environment across development and testing workstations, local compute servers for small production runs, and HPC resources for large runs.
- **Consistency** of environments to promote reproducibility. Examples include validated software stacks standardized by a field of inquiry and archival environments that remain consistent into the future.
- **Usability** and comprehensibility for meeting the above.

These needs for flexibility have been traditionally addressed by sysadmins installing various software upon user request; users can then choose what they want with commands such as `module load`. However, only software with high demand justifies the sysadmin effort for installation and maintenance. Thus, more unusual needs go unmet, whether innovative or crackpot—and it's hard to tell which is which beforehand. This can create a chicken-and-egg problem: a package has low demand because it's unavailable, and it's unavailable because it has low demand.

This motivates empowerment of users with "bring your own software stack" functionality, which we call *user-defined software stacks* (UDSS). The basic notion is to let users install software of their choice, up to and including a complete Linux distribution, and run it on HPC resources.

Of course, this approach has drawbacks as well. We've identified three potential pitfalls:

- **Security:** By introducing very flexible new features, UDSS can expand a center's attack surface, especially if they depend on privileged or trusted functionality.
- **Missing functionality:** Separation from the native software stack can interfere with features such as file systems, accelerator hardware, and high-speed interconnects that make HPC centers interesting and special.
- **Performance:** Implementations must take care to avoid introducing overhead that meaningfully impacts performance.

Options for User-Defined Software Stacks

We believe the needs and pitfalls above lead to three design goals for an HPC-focused UDSS implementation.

First, it should provide a standard and reproducible workflow. A standard workflow reduces training and development costs while enhancing the portability of staff skill sets; a reproducible workflow, in contrast with a "tinker 'til it's ready, then freeze," makes the creation of UDSS images simpler and more robust.

Second, it should run on existing, minimally modified HPC hardware and software resources. This is for two reasons. First, the pitfalls above are already well-controlled in HPC centers; smaller modifications add fewer risks than larger ones. Second, the challenges of orchestrating large parallel applications are well-addressed by HPC centers. We have good resource managers (Slurm, Moab, Torque, PBS, etc.), good high-performance parallel file systems (Lustre, Panasas), good high-speed networks (InfiniBand, OPA), and more. These solutions need not be reimplemented and reoptimized using novel technology.

Finally, it should be as simple as is practical while still delivering the necessary features. This is in keeping with the UNIX philosophy to "make each program do one thing well" [2].

We see three basic options for implementing UDSS: self-compile, virtual machines, and Linux containers.

Compile It Yourself

The traditional method for users to take care of themselves is to simply compile what they need in a home directory or other user area. This is available almost everywhere already, employs only unprivileged functionality, and yields direct access to all center resources. However, it is also tedious and error-prone, hard to update, and does not provide portability or consistency of environments. In principle, users can self-compile arbitrary software; in practice, its difficulty is very limiting.

Virtual Machines and Public/Private Cloud

A *virtual machine* (VM) is a program that emulates a physical computer. One then installs an operating system and applications into this emulator. This is appealing because it gives users ultimate flexibility and strong isolation; it is reasonable to let them install even non-UNIX operating systems and have full administrative privileges. Modern virtual machines perform excellently for things needed by industry, such as CPU-bound tasks and Ethernet networking.

However, the approach has challenges. Performance is often an issue for things uncommon in industry, such as HPC high-speed networks; this can sometimes be mitigated by compromising on isolation. Virtual machines must be provisioned with a complete OS, including kernel and system daemons, and the support infrastructure such as virtual networking is complex.

There is a view that HPC should become more like cloud computing, which offers on-demand, loosely coupled virtual machines. However, this approach requires that either users or sysadmins reimplement and reoptimize much of the functionality that HPC centers already offer.

Our belief is that HPC centers should offer virtual machines only if credible UDSS require not only a custom user space but a custom kernel as well. Otherwise, its disadvantages dominate.

Linux Containers

A middle approach is containers, which share "the only" kernel with the native software stack, accomplishing isolation with Linux *namespaces* and related features. (For further reading, we recommend Michael Kerrisk's series in *Linux Weekly News* [1]as well as `namespaces(7)` and related man pages.)

Note that *container* is a widely used term with varying definitions. The view outlined here is the one we find most sensible.

Privileged Linux Namespaces

Linux has six namespaces that isolate different classes of kernel resources; processes in one namespace see a different view of system state than processes in another. Five namespaces are what we call *privileged*, needing root to create; the sixth, unprivileged one, is covered in the next section. The privileged namespaces are:

1. **Mount:** File-system tree and mounts
2. **PID:** Process IDs—a process in a PID namespace has a different PID inside and outside the namespace
3. **UTS:** Host name and domain name (the name deriving from "UNIX time-sharing system")
4. **Network:** All other network-related resources, including network devices, ports, routing tables, and firewall rules
5. **IPC:** Inter-process communication, both System V and POSIX

The six namespaces can be mixed and matched, but there are quirks. For example, a mount namespace cannot create a new `/sys` unless it is also a network namespace, because `/sys` includes files that can be used to manipulate the network configuration.

Namespaces are always active, i.e., all Linux processes have namespace IDs for all six namespaces (try `ls -l /proc/self/ns`). Namespaces form a tree, with parent/child relationships, and everything is owned by a namespace. For example, though it cannot create its own, a mount namespace can bind-mount its parent's, to which the parent namespace controls access.

Namespaces are manipulated by three system calls: `unshare(2)` puts an existing process into new namespaces, `clone(2)` can put a new child process into new namespaces, and `setns(2)` joins an existing namespace.

These features are useful for UDSS because they allow any directory to become the file-system root of a mount-namespaced process, and the other namespaces can be added for additional isolation as needed.

The Unprivileged User Namespace

The sixth namespace, *user*, was added starting in Linux 3.8. Its goal is to give unprivileged processes access to traditionally privileged functionality in specific contexts when doing so is safe. This is accomplished with namespace-specific capabilities and user/group IDs.

The first process in a new user namespace has all capabilities in the new namespace, but none in the parent user namespace, even if created by root.

The relationship between child and parent namespace UIDs is controlled by a one-to-one mapping defined during namespace setup. The situation with GIDs is analogous. A common use is to map one's normal, unprivileged UID to 0 inside the namespace, thus appearing to be root inside the namespace.

If the namespace is created by an unprivileged user, the parent side of this map may only be the existing EUID. This limits access to things already accessible, because while any UID can be selected in the child namespace, it must map to the user's existing, real UID. Also, all access using unmapped UIDs will be rejected. For example, `setuid(2)` cannot be used to access another user's files, because the other user's UID grants no access if unmapped and cannot be set on the parent side of the map.

This one-to-one mapping is used to translate UIDs in both directions. When a UID-based access decision is initiated inside the namespace, the map translates the in-container UID up through the namespace tree to its corresponding base UID, and the latter is used for access control. For example, bind-mounting any directory into the container is safe, because it is the user's real, unprivileged IDs on the host, not the fictional ones in the user namespace that control access. In the opposite direction, for example, files owned by the user will be translated from the user's real UID to the in-container UID. Thus, with the mapping to UID 0 described above, all of a user's files will appear to be owned by root when listed inside the namespace.

Thus, processes and kernel resources inside the user namespace can be manipulated arbitrarily, but only in ways that do not affect the parent namespace—privilege is an illusion.

```
#define _GNU_SOURCE
#include <fcntl.h>
#include <sched.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
  uid_t euid = geteuid();
  int fd;

  printf("outside userns, uid=%d\n", euid);

  unshare(CLONE_NEWUSER);
  fd = open("/proc/self/uid_map", O_WRONLY);
  dprintf(fd, "0 %d 1\n", euid);
  close(fd);
  printf("in userns, uid=%d\n", geteuid());

  execlp("/bin/bash", "bash", NULL);
}
```

Listing 1: Hello world implementation of a user namespace, available as `examples/syscalls/userns.c` in the Charliecloud source code. This program creates the namespace with `unshare(2)`, maps within-namespace UID 0 to the invoking user's EUID by writing `uid_map`, and then starts the world's most useless root shell.

Listing 1 illustrates a hello-world user namespace implementation. This is an unprivileged, untrusted, non-setuid program; given kernel support, any user can run it, or the more complete implementations in Charliecloud, with no sysadmin assistance.

User namespaces are a powerful tool for implementing container-based UDSS tools because they let a normal, unprivileged user create an independent file-system tree and safely access host resources, even if he or she holds "privileges" inside the container, without depending on the container implementation for security.

Additional Components

Other Linux features commonly used in container implementations include:

- `cgroups(7)`, which track and limit resource consumption of processes. This can be useful in multi-tenant settings to keep users from stomping on each other.
- `prctl(2)` with `PR_SET_NO_NEW_PRIVS`, which prevents `execve(2)` from increasing privileges. This can protect against some privilege escalation bugs, e.g., in setuid binaries.
- `seccomp(2)` filters system calls, thus mitigating security issues in the excluded calls.
- SELinux and AppArmor have various features that can change what the processes may do.

These features can be applied to processes in general, not just containers. For example, if a `seccomp(2)` filter increases the security of container jobs, why not apply it to all jobs? That said, it may be reasonable for container implementations to use these tools under a "belt and suspenders" philosophy, if the benefit outweighs the complexity gain.

Container Implementations

There are many container implementations. We divide them generally into two categories, full-featured and lightweight, which serve different use cases.

Full-Featured

Full-featured container implementations have (shockingly!) lots of features, for example some subset of:

- Image building
- Image management (e.g., storage, caching, tagging, signing)
- Images stored in custom formats
- Image sharing (repository/registry, search, Web site)
- Orchestration
- Storage management (overlay management, back-end drivers)
- Runtime setup (default command, start-up script, `inetd`-type functionality)
- Stateful containers that can be started and stopped
- Supervisor daemons, e.g., to proxy signals as required by PID namespace

Typically, these implementations comprise a security boundary.

Examples from industry include Docker/runC, rkt, and LXC, along with perhaps `systemd-nspawn(1)` and NsJail; examples from HPC include NERSC's Shifter and LBNL's Singularity.

These many features are implemented because they are useful, but there are drawbacks. For example, access to the `docker` command is equivalent to root by design [4]. One could write a wrapper, but input sanitization is a perilously difficult problem.

All these features must be supported for configuration, security, and user support. For example, Docker comprises 133,000 lines of code, some of which are privileged, and Docker is written in Go, a language HPC centers tend to lack expertise in.

It can be done, of course, but it's a major step for an HPC center and must be done with great care. We believe that deploying a lightweight solution is an easier path.

Lightweight

In contrast, lightweight implementations have few features. Most basically, given an image, they run a containerized process within that image. Typically, image building is delegated to other tools, whether designed for containers or not (e.g., `debootstrap(8)`).

Lightweight implementations minimize security responsibility, and they have fewer lines of code to evaluate, support, and secure. This makes deployment lower cost and easier for HPC centers to justify.

Examples from industry include `unshare(1)` from `util-linux`, along with perhaps `systemd-nspawn(1)` and NsJail. In HPC, we are aware of only our own Charliecloud, discussed below.

We believe that lightweight implementations are best for HPC centers. They bring the most important dimensions of cloud-like flexibility without compromising the existing tools and strengths of HPC centers or demanding their reimplementation and reoptimization.

Charliecloud

Our basic design is motivated by two observations. First, full-featured implementations are not a good fit for HPC centers. However, some of their features are really important: most importantly, image building and image sharing.

```
$ cd charliecloud/examples/hello
$ ch-build -t hello ../..
Sending build context to Docker daemon 12.24 MB
[...]
Successfully built 2972e7281f75
$ ch-docker2tar hello /var/tmp
57M /var/tmp/hello.tar.gz
$ ch-tar2dir /var/tmp/hello.tar.gz /var/tmp/hello
/var/tmp/hello unpacked ok
$ ch-run /var/tmp/hello -- echo "I'm in a container"
I'm in a container
```

Listing 2: Building and running "hello world" in Charliecloud requires only a few simple commands. The tarball image created in Step 3 can be run on any host where the Charliecloud runtime is installed; Docker is no longer needed once the image is built.

Thus, our open-source, lightweight container implementation takes a dual approach. We put building and sharing in a sandbox that is separate from HPC center resources. This could be a user workstation or a virtual machine: somewhere safe to give the user root. In this sandbox, Charliecloud wraps Docker for image building, and the other Docker tools are also available, including sharing via pull/push to any Docker Hub repository.

Running images uses our own runtime that is unprivileged and independent of Docker. This can be on center resources or anywhere else with the Charliecloud runtime installed, such as the same sandbox for development and testing. Listing 2 is an example of this workflow.

This brings us back to our three design goals:

1. A standard, reproducible workflow is accomplished by using Docker for image building. This enables use of Dockerfiles, an industry standard for reproducible builds. Working atop Docker for image management also integrates our solution into the robust Docker image ecosystem.
2. Running on existing HPC resources is accomplished with our `ch-run` runtime, which provides just enough isolation using the mount and user namespaces to run a container image. Similarly to `time(1)`, which provides an environment that records resource usage, `ch-run` provides a container environment.

   `ch-run` requires no privilege and depends on the Linux kernel for security, just like any other user process. Performance is the same as native in our tests, modulo noise, because minimal isolation yields direct access to all resources: compute, network, file systems, accelerators, and the rest. `ch-run` scales using standard HPC tools. For example, a large application can be started simply with `mpirun -np $BIGNUM ch-run bigprog`.
3. Simplicity: Charliecloud is a collection of five shell scripts and two C programs totaling roughly 900 lines of code. For comparison, NsJail is 4,000 lines, Singularity 11,000, Shifter 19,000, and Docker 133,000.

We have recently deployed Charliecloud in production and are working with Los Alamos scientists on its use and performance for real-world science code. We look forward to sharing these results.

If you'd like to learn more, Charliecloud's source code is available from GitHub (https://github.com/hpc/charliecloud), and its documentation is on the Web (https://hpc.github.io/charliecloud). Further technical detail is available in our forthcoming Supercomputing paper [3].

References

[1] Michael Kerrisk, "Namespaces in Operation, Part 1: Namespaces Overview," *Linux Weekly News*, January 4, 2013: https://lwn.net/Articles/531114/.

[2] Doug McIlroy, E. N. Pinson, and B. A. Tague, "UNIX Time-Sharing System," Foreword, *Bell System Technical Journal*, vol. 67, no. 6, 1978.

[3] Reid Priedhorsky and Tim Randles, "Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC," in *Supercomputing*, 2017 (forthcoming).

[4] Reventlov's Silly Hacks, "Using the Docker Command to Root the Host (Totally Not a Security Issue)," April 2015: http://reventlov.com/advisories/using-the-docker-command-to-root-the-host.

Interview with James Bottomley

RIK FARROW

James Bottomley is a Distinguished Engineer at IBM Research, where he works on cloud and container technology. He is also a Linux kernel maintainer of the SCSI subsystem. He has been a Director on the Board of the Linux Foundation and Chair of its Technical Advisory Board. He went to Cambridge University for both his undergraduate and doctoral degrees after which he joined AT&T Bell Labs to work on distributed lock manager technology for clustering. In 2000 he helped found SteelEye Technology, a high availability company for Linux and Windows, becoming Vice President and CTO. He joined Novell in 2008 as a Distinguished Engineer at SUSE Labs, Parallels (later Odin) in 2011 as CTO of server virtualization, and IBM Research in 2016. james.bottomley@hansenpartnership.com

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

I first met James Bottomley during a Linux File System and Storage workshop that took place before FAST in 2007. James' focus has been on the SCSI subsystem of Linux. But, as the CTO of Parallels, James has also worked on containers. James and Pavel Emelyanor wrote an article comparing containerization to virtualization for *;login:* back in 2014 [1].

While attending LISA '16, I heard many conversations from people in the hallway that suggested that they understood neither the purpose of containers nor how they were implemented. And, it turns out, I didn't understand how containers work under Linux either.

Rik Farrow: Looks like you may not be at Parallels anymore.

James Bottomley: That's right...I'm at IBM Research now.

RF: My problem is that lots of people don't consider container tech important.

JB: Heh, well, there's a strong political reason for that: the main contenders vying to be the enterprise container power have no expertise in the core technology of containers (OS virtualization), so they're anxious to concentrate on stuff they can control. Plus if you look at what industry is after with container technology, development process simplification and agility, although these are enabled by OS virtualization, they're nowhere directly connected to virtualization.

RF: By "main contenders," you mean Docker, Red Hat, Core, and some others I am not thinking of?

JB: Yes: other orchestration companies like Mesos, Joyent, and now even VMware.

RF: You include VMware in the list of companies offering orchestration. Could you clear that up for me?

JB: Yes, VMware's province is still very much hypervisors and thus hardware virtualization not OS virtualization. Admittedly, VMware does have a Linux kernel team, which gives them the capacity to get into the OS virtualization infrastructure in Linux very quickly unlike most of the other orchestration owners, but there's little sign (from kernel commit logs) that they're doing this.

RF: I think that industry wants what you suggest, simpler development and more agility, but they also appreciate having containers that are much lighter weight than VMs.

JB: Remember, I worked for Parallels, which was a container company before it was fashionable. In 2004, Parallels tried to sell containers to the enterprise in place of VMs on the grounds that they were faster and more lightweight. Parallels failed primarily because that's not what the enterprise wanted.

Enterprise CIOs have a problem they try to conceal with excess hardware capacity; something that uses capacity more efficiently is really an unwelcome technology.

The first company to have a genuine need for lightweight virtualization technology was Google in around 2006-2007 because they realized that to run a service at cloud scale you require this type of transactional efficiency—that's when they adopted containers wholesale. Very few traditional enterprises are building out cloud-scale datacenters still.

RF: I've heard that you can run 10 times as many containers as VMs on the same hardware. And they can spin up containers much faster, too.

JB: Yes, that's because there's a single kernel doing all the resource management. Containers are essentially small groups of UNIX processes, so if you want to run 100 Apache servers, it's far cheaper in resources to run 100 Apache processes each in a container than to run 100 VMs with a full OS complement.

Full operating systems are very complex and resource-intensive beasts. The person who just wants to run *X* applications really doesn't care what the OS is doing and really doesn't want to manage it, which is the Achilles' heel of VMs. The world wants to move away from infrastructure, but a VM is anchored there.

RF: I attended a workshop (HotCloud '14), where they broke up into groups discussing different topics. I attended the Container group, and one thing some Google person said stuck in my mind: we run associated containers within a VM, and we use VMs for security isolation. I thought about that a lot.

JB: Google has a particular problem: being the first adopters, they bent the technology to serve themselves. Google actually hired about everyone they could who was working on Linux cgroups in 2006. The Google datacenters grew to be container-centric but supported Google written workloads. The Google cloud allowed you to bring your data but not your code in those days. If you write all the code, you can take a lot of shortcuts with security (which Google did).

Then when they wanted to offer a-bring-your-own-code service, Google App Engine, they had to turn to some external technology to add security. This problem is unique to Google. But every former or current hypervisor company is trying to also smear container security because they fear it's the only way they'll stay in the game, so you hear this type of statement from a lot of sources.

The reality is, of course, that containers were being sold as hypervisor replacements to the hosting industry by Parallels from about 2001 on. With no need of any VM to provide security. The technology itself can be made secure enough on bare metal.

The key phrase is "can be made." The problem with container technology is that it's not all or nothing like VM technology. You can't really emulate just some virtual hardware, so if you don't turn on the OS virtualizations securely, you don't get security. Most of the modern application packaging container technology, like Docker, doesn't turn all the security features on.

RF: In the article you and Pavel wrote [1], you explained that containers are based on cgroups and namespaces. Cgroups (control-groups) provide limits to resource usage, and namespaces limit access to, well, namespaces, such as files, directories, devices, and networks. Is that a good description of how containers work?

JB: Sort of. The problem is that the OS itself has no concept at all of a "container": all the OS knows is that there are a group of processes for which certain OS virtualization features have been set up. So the way "containers" work is potentially hugely variable. For instance, the Kubernetes concept of a "pod" means a set of "containers" that share certain namespaces, like network or IPC (meaning they see each other's network interface, and you can set up IPC message passing between them).

All container systems without exception use the core Linux APIs of namespaces and cgroups, but they can use them in very different ways (so LXC is very different from, say, Docker in how it sets up what it thinks of as a container).

RF: There must also have been some API support added, so a root-EUID process could start up containers.

JB: Actually, the largest amount of work in Linux is going on in the realm of what are called unprivileged containers. This means OS virtualization that can be controlled by non-root users.

What you say above is currently true—most orchestration systems do run as root, but that causes security problems, so they'd actually also be interested in running unprivileged.

RF: I'm guessing that this is involved in orchestration schemes, but there must be more to orchestration than just firing up containers. You need a way to keep track of them, as well as methods for both connecting them as well as constraining them through the orchestration system.

JB: Right. Usually the way an orchestration system keeps track of containers to think of each container as being a collection of processes. Usually the container has some unique ID, and each process within the container carries it as either a mark or a mapping. Most often the way you can see this from outside is that each container is a separate PID namespace. So Docker uses UUIDs, and it keeps a runtime map of UUID<->PID namespace (which changes every time you start and stop a container) so that it can uniquely identify every process in a container by interrogating the PID namespace.

Now that I've told you the above, I have to confess that when I set up my architecture emulation containers, I don't actually use a PID namespace, so the above isn't universal (but realistically nothing in containers is).

RF: That really helped me understand containers: that the UUIDs that Docker creates is just the Docker tool's own way of identifying a group of processes. I found myself wondering whether there was a "create container" system call. Instead I discovered that most of the work is done by clone() by setting certain flags when creating a new process.

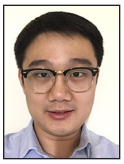
JB: Yes, there are essentially two namespace creation system calls, clone() and unshare(), and one namespace entry system call, setns(). Cgroups don't have any system calls at all; it's currently all done by manipulating files in the cgroup file systems, which are usually mounted under /sys/fs/cgroup.

Reference

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

Knockoff
Cheap Versions in the Cloud

XIANZHENG DOU, PETER M. CHEN, AND JASON FLINN



Xianzheng Dou is a PhD student in computer science and engineering at the University of Michigan, Ann Arbor. In general, his research interests include file systems, operating systems, and distributed file systems. More specifically, he has been focusing on how to reduce communication and storage costs for distributed file systems and how to speed up computation via memorization. xdou@umich.edu

Peter M. Chen is an Arthur F. Thurnau Professor in the Computer Science Division at the University of Michigan. He is an ACM and IEEE Fellow and served as the Editor-in-Chief of *ACM Transactions on Computer Systems* from 2009–2013. In 2007, he received the ACM SIGOPS Mark Weiser Award "for creativity and innovation in operating systems research." His research interests include operating systems, computer security, and fault-tolerant computing. He is currently investigating how to improve software reliability for multicore computers and how to integrate new types of persistent memories into computer systems. He regularly teaches a senior course on operating systems and a first-year course on computer engineering. pmchen@umich.edu

Jason Flinn is a Professor of Computer Science and Engineering and Director of the Software Systems Laboratory at the University of Michigan, whose research interests include operating systems, distributed systems, and mobile computing. He is a fellow of the ACM, and his research has been recognized with an NSF CAREER award and eight Best Paper awards at SOSP, OSDI, ASPLOS, FAST, and MobiSys. jflinn@umich.edu

Cloud-based storage provides reliability and ease-of-management. Unfortunately, it can also incur significant costs for both storing and communicating data. These costs increase when systems retain past versions of files for data recovery, auditing, and forensic troubleshooting. While techniques such as chunk-based deduplication and delta compression have proven very effective in reducing bytes stored and sent over the network, further optimizations to these techniques are yielding increasingly incremental benefits. We argue that it is time to consider additional strategies for reducing storage costs. In our current work, we are demonstrating that one such strategy, deterministic recomputation of data, can substantially reduce the cost of cloud storage. Our distributed file system, Knockoff, selectively substitutes nondeterministic inputs for file data. Our results show that this reduces the cost of sending files to the cloud without versioning by 21–24%; the relative benefit is substantially greater when past versions are retained.

Deterministic Recomputation

Knockoff leverages an unconventional method for communicating and storing file data. In lieu of the actual data, it selectively represents a file as a log of the nondeterministic inputs needed to recompute the data (e.g., system call results, thread scheduling, and external data read by a process). With such a log, a cloud file server can deterministically replay the computation that originally produced the data to recreate the data. We call the observation that one can represent data generated by computation either by value or by the log of inputs needed to reproduce the computation the *principle of equivalence* (between values and computation); the principle has been observed and used in many settings such as fault tolerance and state machine replication.

Representing data as a log of nondeterminism leads to several benefits for a distributed file system. First, it substitutes (re)computation for communication and storage, and this can reduce total cost because computation in cloud systems is less costly than communication and storage. Second, it can reduce the number of bytes sent over the network when the log of nondeterminism is smaller than the data produced by the recorded computation. For the same reason, it can reduce the number of bytes stored by the cloud storage provider. Finally, representing data as a log of nondeterminism can support a wider range of versioning frequencies than prior methods.

Although similar ideas have been previously applied to distributed storage, the computation has either been assumed to be deterministic given its command line and file inputs [4] or given a specific sequence of user-interface events [1]. Unfortunately, neither a log of shell commands nor a log of user activity is sufficient to reproduce the computation of modern, general-purpose programs, especially due to the shift to multithreaded computation running on multiprocessors, as well as a growing diversity in execution environments and corresponding dependencies on operating systems, libraries, and installed application versions.

Knockoff uses deterministic record and replay to guarantee that data produced by all data-race free programs can be reproduced. Rather than capture a subset of nondeterministic inputs, it uses the Arnold [2] system to record all nondeterministic data entering each process that executes on a file system client, including the results of system calls (such as user and network input), the timing of signals, and real-time clock queries. Arnold enables deterministic replay of multithreaded programs by recording all synchronization operations (e.g., `pthread_lock` and atomic hardware instructions). This recording has minimal overhead (8% or less in our experiments). Because it supplies recorded values on replay rather than re-executing system calls that interact with external dependencies, Arnold can trivially record an application on one computer and replay it on another. The only requirements are that both computers run the Arnold kernel and have the same processor architecture (x86).

For example, consider a simple application that reads in a data file, computes a statistical transformation over that data, and writes a timestamped summary to an output file. The output data may be many megabytes in size. However, the program itself can be reproduced given a small log of determinism, as shown in Figure 1 (for clarity, the log has been simplified).

The log records the results of system calls (e.g., `open`) and synchronization operation (e.g., `pthread_lock`). The first entry in Figure 1 records the file descriptor (rc=3) chosen by the operating system during the original execution. Parameters to the open call do not need to be logged since they will be reproduced during a deterministic re-execution. The second entry records the mapping of the executable; replaying this entry will cause the exact version used during recording to be mapped to the same place in the replaying process address space. Lines 4 and 5 read data from the input file, line 6 records the original timestamp, and lines 7 and 8 write the transformation to the output file. Data read from the file system is not in the log since Knockoff is a versioning file system that can reproduce the desired version on demand. Also, the data written to the output file need not be logged since it will be reproduced exactly as a result of replaying the execution.

With aggressive compression [2], a log for this sample application can be only a few hundred bytes in size, as contrasted with the megabytes of data that the execution produces. The output data is reproduced by starting from the same initial state, re-executing the computation, and supplying values from the log for each nondeterministic operation. Since the log contains references to executable and shared library versions, as well as all interactions with the operating system, the complex environmental dependencies of an application are automatically resolved as part of the replay process. For instance, the replay starts from the same executable, loads the same versions of shared libraries, and sees the same results of IPC and network operations that were seen during recording.

	Log entry	Values
1	open	rc=3
2	mmap	file=<id,version>
3	pthread_lock	
4	open	rc=4
5	read	rc=<size>, file=<id,version>
6	gettimeofday	rc=0, time=<timestamp>
7	open	rc=5
8	write	rc=<size>
9	pthread_unlock	

Figure 1: Sample log of nondeterminism

Additionally, just as deduplication and compression of file data can reduce bytes stored and sent over the network for file data, we have found that applying these techniques to logs of nondeterminism can also provide similar savings by exploiting similarities in computation across executions of the same application. In particular, Knockoff achieves an additional 42% reduction in bytes stored and communicated by using delta compression on the logs of nondeterminism.

Writing Data to the Cloud

To propagate modifications to the cloud, Knockoff first calculates the cost of sending and replaying the log of nondeterminism given a pre-defined cost of communication ($cost_{comm}$) and computation ($cost_{comp}$):

$$cost_{log} = size_{log} * costcomm + time_{replay} * cost_{comp} \quad (1)$$

$size_{log}$ is determined by compressing the log of nondeterminism for the application that wrote the file and measuring its size directly. To estimate $time_{replay}$, Knockoff records the user CPU time consumed so far by the recorded application with each log entry that modifies file data. This is a very good estimate for the time needed to replay the log on the client [6]. To estimate server replay time, Arnold multiplies this value by a conversion factor to reflect the relative CPU speeds of the client and server.

Knockoff calculates the cost of sending file data as:

$$cost_{data} = size_{chunks} * cost_{comm} \quad (2)$$

Knockoff implements the chunk-based deduplication algorithm used by LBFS [5] to reduce the cost of transmitting file data. It breaks all modified files into chunks, hashes each chunk, and sends the hashes to the server. The server responds with the set of hashes it has stored. $size_{chunks}$ is the size of any chunks unknown to the server that would need to be transmitted; Knockoff uses gzip compression to reduce bytes transmitted for such chunks.

If $cost_{log} < cost_{data}$, Knockoff sends the log to the server. The server spawns a replay process that consumes the log and replays the application. When the replay process executes a system call that modifies a target file, it updates the current version, and potentially retains the past version as described below.

Replay is guaranteed to produce the same data if the application being replayed is free of data races. Data-race freedom can be guaranteed for some programs (e.g., single-threaded ones) but not for complex applications. Knockoff therefore ships a SHA-512 hash of each modified file to the server with the log. The Knockoff server verifies this hash. If verification fails, it asks the client to ship the file data. Such races are rare since the replay system itself acts as an efficient data-race detector. All subsequent replays are guaranteed to produce the same data as the first replay, so once Knockoff verifies that the replay produces the desired data, it need not do so again.

If $cost_{data} < cost_{log}$, then Knockoff could reduce the cost of the current transaction by sending the unique chunks to the server. However, for long-running applications, it may be the case that sending and replaying the log collected so far would help reduce the cost of future file modifications that have yet to be seen (because the cost of replaying from this point is less than replaying from the beginning of the program). Knockoff predicts this by looking at a history of $cost_{data}/cost_{log}$ ratios for the application. If sending logs has been historically beneficial and current application behavior is similar (the ratios differ by less than 40%) to past executions, it sends the log. Otherwise, it sends the unique data chunks.

Storing Data in the Cloud

Knockoff may store file data on the server either by value (as normal file data) or by operation (as the log of nondeterminism required to recompute that data). If the log of nondeterminism is smaller than the file data it produces, then storing the file by operation saves space and money. However, storing files by operation delays future reads of that data, since Knockoff will need to replay the original computation that produced the data. In general, this implies that Knockoff should only store file data by operation if the data is very cold, i.e., if the probability of reading the data in the future is low.

Knockoff currently stores the current version of all files by value so that its read performance for current file data is the same as that of a traditional file system. Knockoff may store past versions by operation if the storage requirements for storing the data by log are less than those of storing the data by value. However, Knockoff also has a configuration parameter that sets a maximum *materialization delay*, which is the time to reconstruct any version stored by operation. The default materialization delay is 60 seconds.

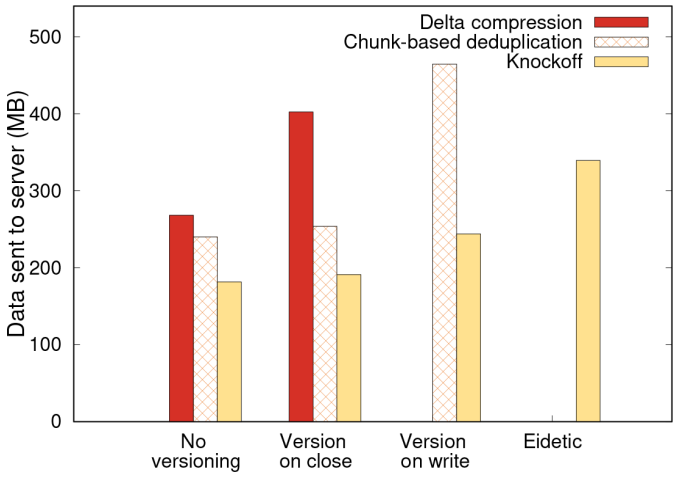


Figure 2: Total bytes sent to the server across all user study participants. We compare Knockoff with two baselines across all relevant versioning policies.

When replaying a log to regenerate data, Knockoff may find that some of the input files for the computation being replayed are also stored by operation rather than by value. In this case, it recursively replays those logs to reproduce the input data needed to regenerate the target data. Knockoff tracks such recursive dependencies in a data structure called the *version graph*. When storing data, it ensures that any path of recomputation in this graph does not exceed the materialization delay, and this guarantees that the total time to reproduce any file is no greater than that bound.

Fine-Grained Versioning

Past file versions have many uses: recovery of lost or overwritten data, reproduction of the process by which data was created, auditing, and forensic troubleshooting. These benefits increase as versions are retained more frequently. For instance, if versions are retained every time a file is closed, the user may have a snapshot of file data with each save operation. However, many applications only close files on termination, so versioning on every file write may be required to provide snapshots of intermediary states. However, such a policy would not capture intermediary states from modifications to memory-mapped files.

When storing and communicating file data by value, more frequent versioning substantially increases costs due to a greater amount of data sent over the network and saved to disk. However, when Knockoff represents file data by operation, its deterministic recomputation can produce *any* version of file data written by that computation at no additional cost. This means that Knockoff has much lower costs for retaining past versions of file data than traditional storage systems.

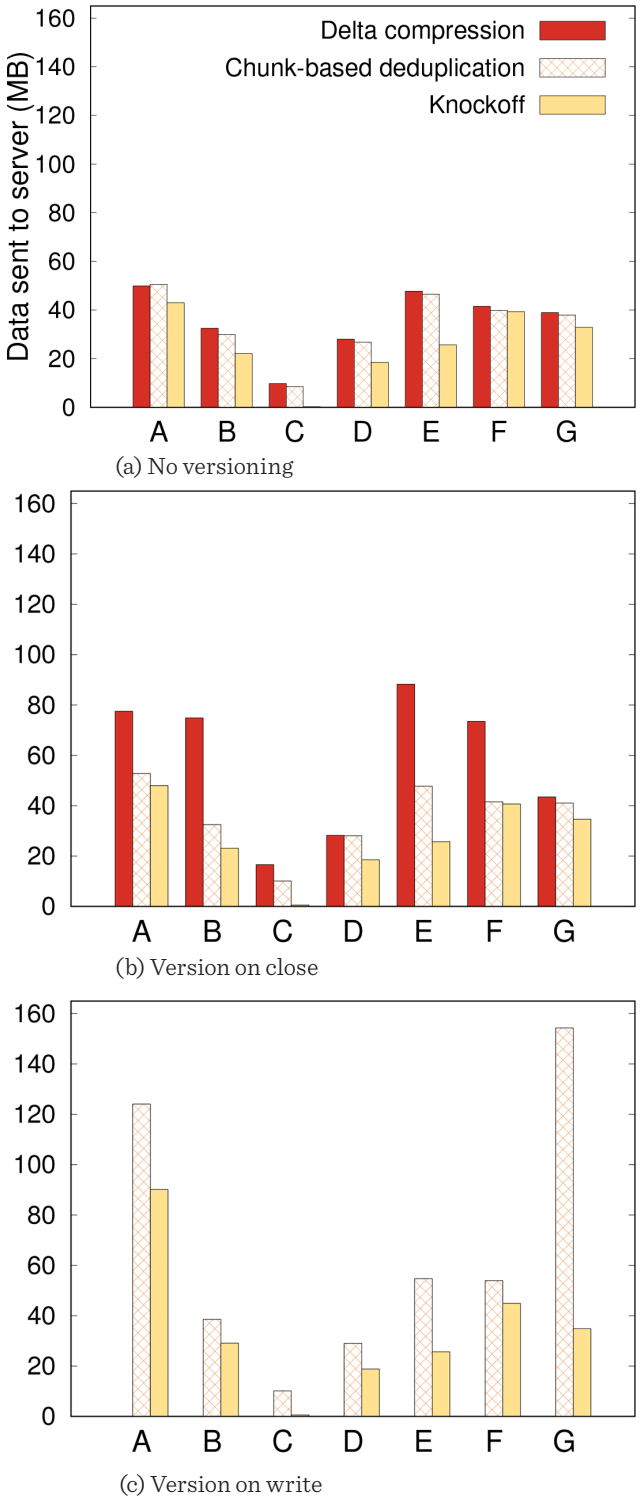


Figure 3: Bytes sent to the server for each individual user-study participant (A-G). We compare Knockoff with two baselines across all relevant versioning policies.

As a result, Knockoff currently supports four different versioning policies:

- **No versioning**: Knockoff retains only the current version of all files.
- **Version on close**: Knockoff retains all past versions at close granularity; for past versions, Knockoff may store the actual data or the logs required to regenerate the data.
- **Version on write**: Knockoff retains all past versions at write granularity.
- **Eidetic**: Knockoff retains all past versions at instruction granularity. It can reproduce versions of a memory-mapped file by replaying the computation up to a specified point and redoing the individual store instructions that modified the file.

User Study Results

As part of a detailed evaluation of Knockoff [3], we recruited eight graduate students to use Knockoff for software development tasks. We asked participants to write software to perform several simple tasks, e.g., converting a CSV file to a JSON file; each participant could spend up to an hour solving the problem. We did not dictate how the problem should be solved. Participants used various Linux utilities, text editors, IDEs, and programming languages. They used Web browsers to visit different Web sites such as Google and StackOverflow, as well as sites unrelated to the assignment (e.g., Facebook and CNN News). Almost all files accessed during the study are stored in Knockoff (exceptions include the tmp directory and system configuration files), and almost all file modifications are therefore persisted in the cloud. One of the eight participants was unable to complete the programming assignment and quit right away. We show results for the seven participants who attempted the tasks; four of these finished successfully within the hour.

Figure 2 summarizes the results by aggregating the bytes sent to a cloud server by Knockoff and the baseline file systems across all seven users. Even without retaining past versions, Knockoff is surprisingly effective in reducing bytes sent over the network for non-versioning file systems. Compared to chunk-based deduplication, Knockoff reduces communication by 24%. Compared to delta compression, it reduces communication by 32%. Note that these baselines are already very effective in reducing bandwidth; without compression, this workload requires 1.9 GB of communication, so delta compression alone achieves an 86% reduction in network bandwidth, and chunk-based deduplication achieves an 87% reduction.

The benefit of Knockoff increases substantially as past versions are maintained more frequently. For instance, Knockoff reduces bytes sent by 47% compared to chunk-based deduplication for a `version on write` policy. In fact, versioning on write with Knockoff uses less bandwidth than the baselines without versioning.

At the limit, the eidetic policy, which can reproduce any past version even for memory-mapped files, is completely infeasible with current storage systems that store data by value. Knockoff can support this granularity of versioning while sending only 41% more bytes to the cloud than chunk-based deduplication *without versioning* in the user study and storing only 134% more bytes in the cloud to retain this state in another longitudinal study (not shown).

A surprising result from this study was that the effectiveness of Knockoff varied tremendously across users, as shown in Figure 3 (each individual study participant is labeled A-G in each graph). For participant C, Knockoff achieves a 97% reduction in bandwidth for the `no versioning` policy and a 95% reduction for the `version on write` policy compared to chunk-based deduplication. On the other hand, for participant F, the corresponding reductions are 2% and 17%. This shows the orthogonal nature of Knockoff's cost savings. When the mix of tools and workloads is better for operation shipping than it is for deduplication or compression, Knockoff produces large savings. In cases where operation shipping is not economical, Knockoff can detect this and revert to more traditional forms of bandwidth and storage reduction.

Summary

Operation shipping has long been recognized as a promising technique for reducing the cost of distributed storage. However, using operation shipping in practice has required onerous restrictions about application determinism or standardization of computing platforms, and these assumptions make operation shipping unsuitable for general-purpose file systems. Knockoff leverages recent advances in deterministic record and replay to lift those restrictions. It can represent, communicate, and store file data as logs of nondeterminism. This saves network communication and reduces storage utilization, leading to cost savings.

In the future, we hope to extend the ideas in Knockoff to other uses; one promising target is reducing cross-datacenter communication. We are also investigating whether it is feasible to generate logs of nondeterminism from which data can be reproduced by observing only a portion of those nondeterministic inputs and synthesizing likely values for the rest. This could represent a promising middle ground between Knockoff and prior operation shipping systems in which one could still guarantee that data can always be reproduced once a successful recomputation has been generated, but such guarantees could be achieved without running a full-scale deterministic recording system such as Arnold on each client.

Acknowledgments

This work has been supported by the National Science Foundation under grants CNS-1513718 and CNS-1421441 and by a gift from NetApp.

References

[1] T.-Y. Chang, A. Velayutham, and R. Sivakumar, "Mimic: Raw Activity Shipping for File Synchronization in Mobile File Systems," in *Proceedings of the 2nd International Conference on Mobile Systems, Applications and Services* (June 2004), pp. 165–176.

[2] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic Systems," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)* (October 2014): https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-devecsery.pdf.

[3] X. Dou, P. M. Chen, and J. Flinn, "Knockoff: Cheap Versions in the Cloud," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)* (February 2017): https://www.usenix.org/system/files/conference/fast17/fast17-dou.pdf.

[4] Y.-W. Lee, K.-S. Leung, and M. Satyanarayanan, "Operation Shipping for Mobile File Systems," in *IEEE Transactions on Computers,* vol. 51, no. 12 (December 2002), pp. 1410–1422.

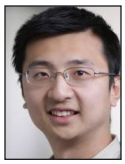
[5] A. Muthitacharoen, B. Chen, and D. Mazières, "A Low-Bandwidth Network File System," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (October 2001), pp. 174–187: https://pdos.csail.mit.edu/papers/lbfs:sosp01/lbfs.pdf.

[6] A. Quinn, D. Devecsery, P. M. Chen, and J. Flinn, "JetStream: Cluster-Scale Parallelization of Information Flow Queries," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)* (November 2016): https://www.usenix.org/system/files/conference/osdi16/osdi16-quinn.pdf.

Passive Realtime Datacenter Fault Detection and Localization

ARJUN ROY, HONGYI ZENG, JASMEET BAGGA, AND ALEX C. SNOEREN

Arjun Roy is a graduate student in the Computer Science and Engineering Department at the University of California, San Diego, in the Systems and Networking Research Group. His research interests include datacenter networking, particularly when it comes to fault detection, localization, and mitigation. arroy@cs.ucsd.edu



Hongyi (James) Zeng is an Engineering Manager and Research Scientist on the Facebook Net Systems team. He works on intra- and inter-DC network monitoring and analytics, troubleshooting tools, and security tools. He received his PhD from Stanford University, co-advised by Professor Nick McKeown and Professor George Varghese. His current research interests include software-defined networks, network verification, and programmable hardware. zeng@fb.com

Jasmeet Bagga received his master's from the University of Southern California in 2005 and then worked at an early-stage startup building network analytics software. Jasmeet is currently a Software Engineer at Facebook, working on Facebook's in-house router/switch software called FBOSS. jasmeetbagga@fb.com

Alex C. Snoeren is a Professor in the Computer Science and Engineering Department at the University of California, San Diego, where he is a member of the Systems and Networking Research Group. His research interests include oper-ating systems, distributed computing, and mobile and wide-area networking. snoeren@cs.ucsd.edu

Datacenters are characterized by their large scale, comprising a large number of network links and switches. However, these hardware components can develop intermittent faults, resulting in randomly occurring packet drops or delays that harm application performance—several such faults occur daily in large production datacenters. Since the effects are intermittent, traditional detection techniques involving host and router statistics or active probe traffic can fall short in their ability to identify and locate these errors. In this article, we present our passive hybrid approach that combines network path information with host-based statistics to rapidly detect and pinpoint the location of datacenter network faults inside a production Facebook datacenter.

Modern datacenters continue to increase in scale, speed, and complexity. Unfortunately, experience indicates that modern datacenters are rife with hardware and software failures—indeed, they are designed to be robust to large numbers of such faults. The large scale of deployment both ensures a non-trivial fault incidence rate and complicates the localization of these faults. Recently, authors from Microsoft described [9] a rogue's gallery of datacenter faults: dusty fiber-optic connectors leading to corrupted packets, switch software bugs, hardware faults, incorrect ECMP load balancing, untrustworthy counters, and more. Confounding the issue is the fact that failures can be intermittent and partial: rather than failing completely, a link or switch fault might only affect a subset of traffic, complicating detection and diagnosis. To illustrate this difficulty, the authors of NetPilot [8] describe how a single link dropping a small percentage of packets, combined with cut-through routing, resulted in degraded application performance and a multiple-hour network goose chase to identify the faulty device.

We present our approach [5] to detect and localize such faults by providing greater visibility into the fate of application traffic once it is injected into the network—specifically, by exposing network path information for all datacenter traffic to the hosts. This allows us to correlate poor network performance observed at each host to the specific component in the network that is responsible passively, without any probe traffic overhead. Furthermore, we find that the vast amount of data available—we use TCP state machine data for every flow on every host—allows us to do so fairly rapidly.

Current Methods

Commonly deployed network monitoring approaches include host monitoring (e.g., RPC latency and TCP retransmits) and switch-based monitoring (e.g., drop counters and queue occupancies). However, such methods can fall short for troubleshooting datacenter-scale networks. Host monitoring alone lacks specificity in the presence of large numbers of alternative paths, which is characteristic of datacenter topologies [2, 7]. An application suffering from dropped packets or increased latency does not give any insight on where the fault is located, or whether a given set of performance anomalies are due to the same faults. Similarly, if a switch drops a packet, the operator is unlikely to know which application's traffic was impacted or, more importantly, what is to blame. Even if a switch samples dropped

packets, the operator might not have a clear idea of what traffic was impacted. Due to sampling bias favoring high volume flows, mouse flows experiencing loss might be missed. Switch-counter-based approaches are further confounded by cut-through forwarding and unreliable hardware [8, 9]. Recent work [9] uses detailed path tracing of a subset of network traffic, combined with a modicum of active probe traffic, to debug where packets are dropped in the network and why. We argue that, for the highly regular topologies used by current datacenter networks, it should be possible to determine path information for all traffic. Luckily, common datacenter topologies are particularly amenable to providing this functionality.

Getting Path Information Scalably

Facebook's datacenters consist of thousands of hosts and hundreds of switches grouped into a multi-rooted, multi-level tree topology [2]. Figure 1 describes a simplified view of this topology, focusing on one of the several identical "pods" in the network. Each pod consists of several tens of Top-of-Rack (ToR) switches, each responsible for a few tens of servers. Each pod also contains four aggregation switches (Aggs) that enable inter-rack communication; every ToR connects to every Agg in the pod. Pods are in turn interconnected by a layer of core switches; each Agg is connected to a disjoint subset of core switches.

The network uses equal-cost multipath (ECMP) routing. When a host communicates with a host in another rack, a hash function at the ToR switch determines which ToR-to-Agg uplink the packets traverse based on fields such as source and destination IP addresses and network ports. Similarly, when a host communicates with a host in another pod, a hash function at the Agg switch determines which Agg-to-Core uplink is used.

For cross-pod traffic, once a packet reaches the core layer, there is only one path leading to the destination server. Thus, if we know the start and end point of a packet (from the source and destination IP address) and the core switch it transits, the receiving host can learn the entire path traversed. Thus, we assign an ID to each core switch and install a rule on the core switch instructing it to stamp every packet it forwards with this ID (see Figure 2).

Pinpointing a Fault to a Link

Once we have full path information, we could theoretically associate packet loss with a particular network path. However, this doesn't tell us which link along the path is responsible. An observation about the traffic engineering employed at Facebook aids us, however.

A significant amount of engineering effort has been targeted at calming hotspots at the application level to ensure that no particular server is overloaded by requests [4]. Specifically, for a front-end datacenter containing Web and cache servers (which

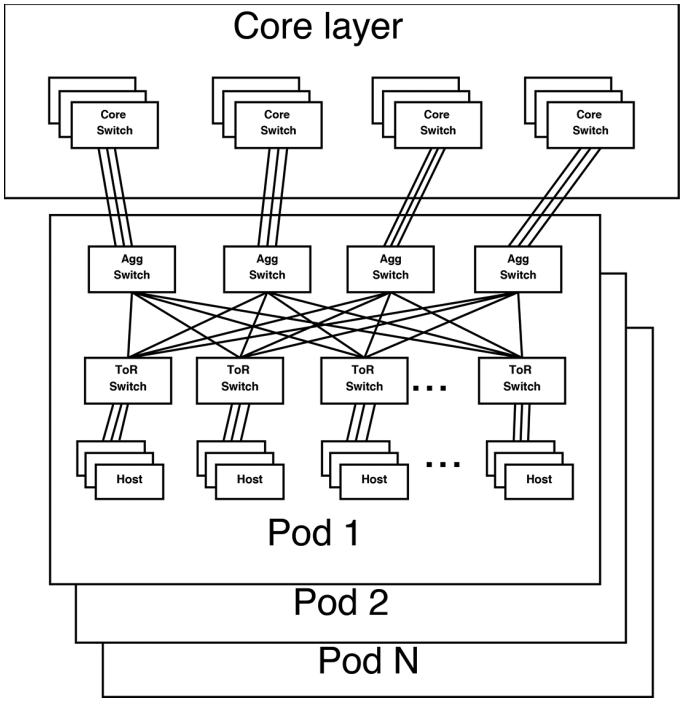


Figure 1: Facebook datacenter topology

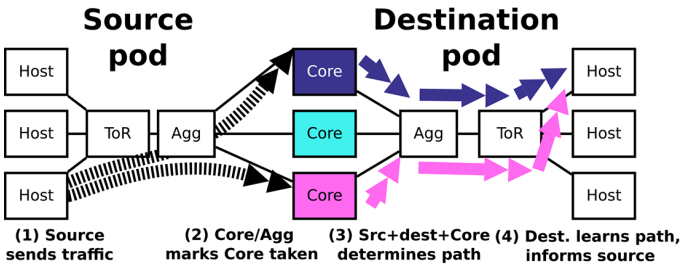


Figure 2: Determining flow network path

cache user data stored by back-end databases), every Web server spreads its requests for user data across all the cache servers in the datacenter. Furthermore, these requests are individually quite small and evenly spread, but in aggregate constitute the bulk of traffic within the network. This application-level load balancing is in addition to normal network load balancing techniques like ECMP routing.

Consequently, if we look at a level of the multirooted tree topology (for example, every Agg to Core link in a pod, or every ToR to Agg link in a pod), every link in the group we examine has a very even load—both in terms of number of flows and number of bytes handled—on short timescales of just a few seconds. The aggregate performance of the flows for any given link is similar to that of the flows of any other link within the set—we call it an "equivalence set" of links.

On the flip side, if one of the links is faulty, it sticks out like a sore thumb—the aggregate flow performance diverges compared

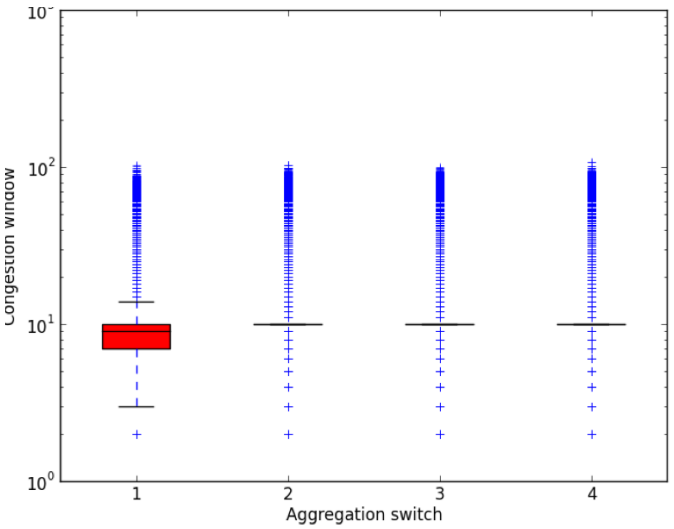


Figure 3: TCP congestion window distribution per ToR uplink for cache server. The ToR to aggregation switch 1 link has 0.5% randomized packet loss, which has shifted the distribution towards smaller values.

to the other links in the set. Thus, if we pick a metric that is correlated with packet loss—for example, TCP retransmits or congestion window—we can compare the distributions of the metric across links and perform outlier analysis to pinpoint the faulty link. Figure 3 depicts the distributions for TCP congestion window for each ToR to Agg uplink traversed by traffic destined for a single cache server, where link 1 (out of four total) has a 0.5% induced packet loss rate; note the significant skew to lower values present for the distribution corresponding to that link.

Outlier Analysis

While it is visually apparent that the distributions of performance metrics across links are impacted by the presence of a fault, we need a way to automate the process of using this information to generate a verdict for every combination of (host, link) to determine whether the link is faulty or not. Fundamentally, the question we are asking is: does a particular link have more retransmits (or say, smaller flow congestion windows) than the others? If so, maybe there is a fault at that link!

To answer this question, we use the Student's t-test. The t-test determines whether a given distribution has a mean that is higher than another distribution. It is amenable to efficient streaming computation (a prototype implementation of our system uses approximately 0.5% of CPU on a production Web server) and runs on every host, with each host examining its own traffic. Note, however, that this raises the chance of false positives, where a link might temporarily have worse performance distributions, possibly due to effects like transient congestion. Given a large number of hosts, it is certain that some subset of them will incorrectly flag a link as faulty. We have to account for these false positives.

The observation we leverage is that false positives, in the absence of an actual fault, ought to be evenly distributed among links due to the high degree of load balancing. Thus, we aim to filter out the false positives by asking the question: are links being claimed as faulty roughly evenly, or is there a particular link (or group of links) that is (are) being accused more than the others? For that we use the chi-squared test, which is used to determine whether the frequency distribution of a set of events matches some theoretical distribution. The chi-squared test sees whether the claims that a link is faulty is evenly spread among all the links considered, or if a particular subset of links have a significantly higher percentage of hosts claiming fault. It outputs a p-value ranging from 0 to 1. If the outputted p-value is "close" to 0 (a common cutoff is 0.05), then the link with the most "faulty verdicts" is considered to actually be faulty. In the case of multiple errors, that link can be removed from the set considered, and the chi-squared test can be run again.

Putting It All Together

We combine the functional components described thus far into an always-active fault detection system. Our system involves functional components at all servers, a subset of switches, and a centralized aggregator, depicted step-by-step in Figure 4. Switches mark packets (1) to indicate network path as described before. Hosts then independently compare the performance of their own flows to generate a host-local decision (a "verdict") about the health of all network components (2), performing outlier analysis using the Student's t-test on metrics such as TCP retransmits. Specifically, every host will output a verdict for the ToR-to-Agg and Agg-to-Core links in its own pod once every 10 seconds. These verdicts are sent (3) to a central aggregator, which filters false positives to arrive at a final set of faulty components using the chi-squared test (4), aggregating data and outputting a result once every 10 seconds (configurable to increase sensitivity as a tradeoff to reaction time) as well. We do not consider host-to-ToR uplinks in this system.

Detecting Faults

To validate our approach, we deployed a prototype of our fault detection system inside a production Facebook front-end datacenter serving user Web traffic. For the sake of reproducible experiments, we primarily focus on injected synthetic failures, which we describe momentarily. We also discuss experience gained in tracking down naturally occurring partial faults.

Induced Faults

Within one of Facebook's datacenters, we instrumented 86 Web servers spread across three racks with the monitoring infrastructure described previously. Path markings are provided by a single Agg switch, which sets DSCP bits based on the core switch from which the packet arrived. To inject faults, we use

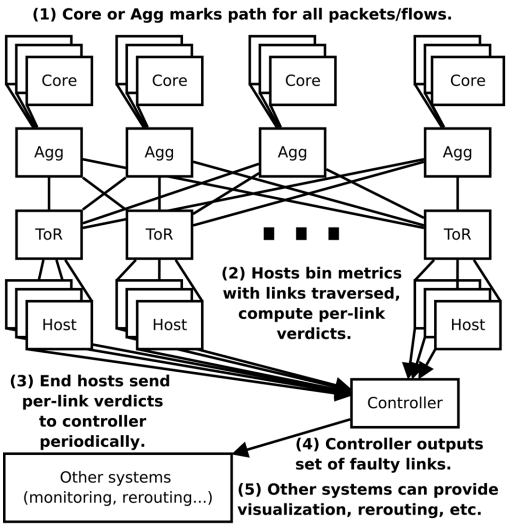


Figure 4: High-level system overview (single pod depicted)

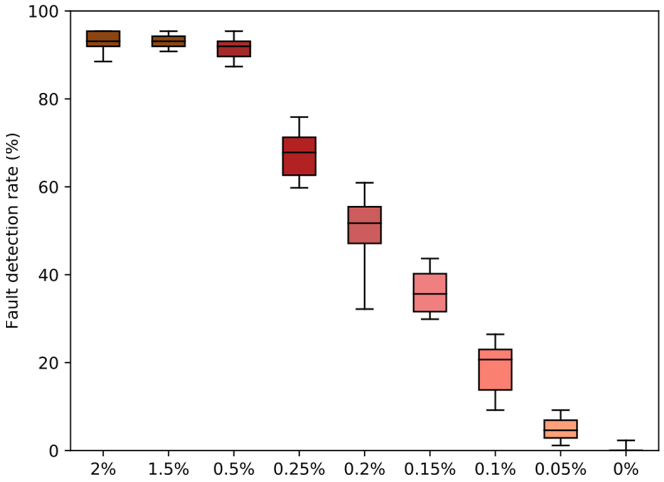


Figure 5: Single fault loss rate sensitivity

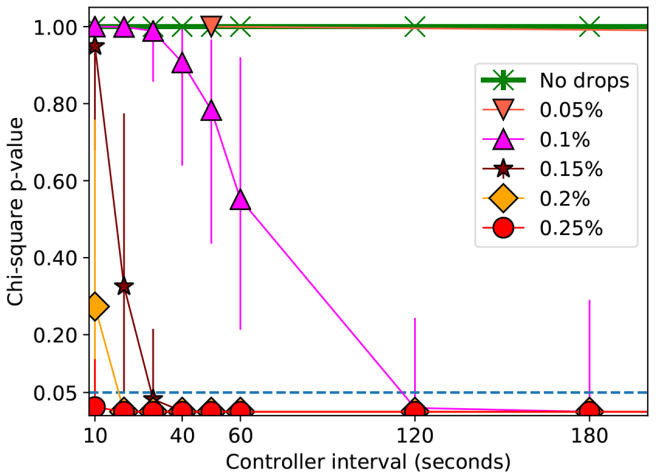


Figure 6: Controller interval required to find single fault vs. packet loss rate

iptable rules installed at hosts to selectively drop inbound packets that traversed specific links (according to DSCP markings). For example, we can configure a host to drop 0.5% of all inbound packets that transited a particular core-to-Agg link.

First, for a single faulty core-to-Agg link, we depict the percentage of hosts that flag the faulty link as having an error as a function of the packet loss rate over consecutive 10-second intervals in Figure 5. For drop rates at 0.5% and higher, close to all of the hosts flag the link as faulty. For drop rates below 0.5%, we observe a linear drop off in the percentage of hosts that catch the fault.

Recall the aggregator, which gives us the overall verdict on per-link health, looks for a non-trivial difference in the number of hosts that claim that a link is faulty before marking it faulty. Thus, over a 10-second interval, it might not decide that the fraction of hosts marking a link as faulty is significant for the smaller magnitude errors. Note, however, that since a fault is likely to persist for longer periods of times, we can simply run the aggregator for longer to catch an error. Instead of processing N verdicts over 10 seconds, we could aggregate and operate on $3N$ verdicts over 30 seconds. We find that this allows us to reliably catch the smaller magnitude errors as well, without inducing false positives in the no-error case.

Figure 6 depicts the amount of time needed by the aggregator to catch errors ranging from 0.25% packet loss down to 0.1%. Recall that a chi-squared test outputs a p-value, where if the p-value is "close" to 0 (we arbitrarily use 0.05 as our cutoff for "close") it means that the link with the most faulty verdicts from the hosts is likely to, in fact, be faulty. Thus, we depict the p5, p50, and p95 for the p-values outputted by the aggregator for each packet loss rate. We see that a 20-second interval will reliably catch a 0.25% error—in other words, the aggregator almost always outputs a p-value of less than 0.05. However, a 0.15% packet loss rate requires 40 seconds, and we receive an intermittent signal for a 0.1% error—at least some portion of the time the aggregator will find no fault.

Naturally Occurring Faults

To determine whether our system can successfully detect and localize network anomalies in the wild, we deployed our system on 30 Web servers for a two-week period in early 2017 without inducing any synthetic errors. On January 25, 2017, the software agent managing a single switch linecard that our system was monitoring failed. The failure had no immediate impact on traffic, since the existing switch rule set installed by the agent remained in effect. Roughly a minute later, however, as the BGP peerings between the linecard and its neighbors began to time out, traffic was preemptively routed away from the impacted linecard.

Our system observed that as traffic was routed away from the failed linecard, the distributions of TCP's congestion window and slow start threshold metrics for the traffic remaining on the faulty linecard's links rapidly diverged from those associated with non-faulty linecards (Figure 7). The deviations are immediate and significant, with the mean congestion window for the faulty linecard dropping over 10% in the first interval after the majority of traffic is routed away, and continually diverging from the working links thereafter. Furthermore, the volume of measured flows at each host traversing the afflicted linecard rapidly drops from O(1000s) to O(10s) per link.

By contrast, one of Facebook's monitoring systems, NetNORAD [1], took several minutes to detect the unresponsive linecard control plane and raise an alert. It is important to note that in this case, we did not catch the underlying software fault ourselves; that honor goes to BGP timeouts. However, we do observe a sudden shift in TCP statistics in real time as traffic is routed away, as our system was designed to do. Thus, this anecdote shows that our system can complement existing fault-detection systems and provide rapid notification of significant changes in network conditions on a per-link or per-device basis.

Caveats

A couple of caveats apply to this methodology. First, it is conceivable that for more complicated topologies, a single stamp on a packet might not be enough to uniquely resolve the path. Prior work [6] has explored marking multiple packets with partial path information, such that the overall network path can be recovered by examining enough packets. We leverage a similar technique to generalize our packet-stamping mechanism. Suppose there is a maximum of H hops in the network between any pair of communicating servers. We provide every switch an ID instead of a select few. Suppose a flow has H or more packets.

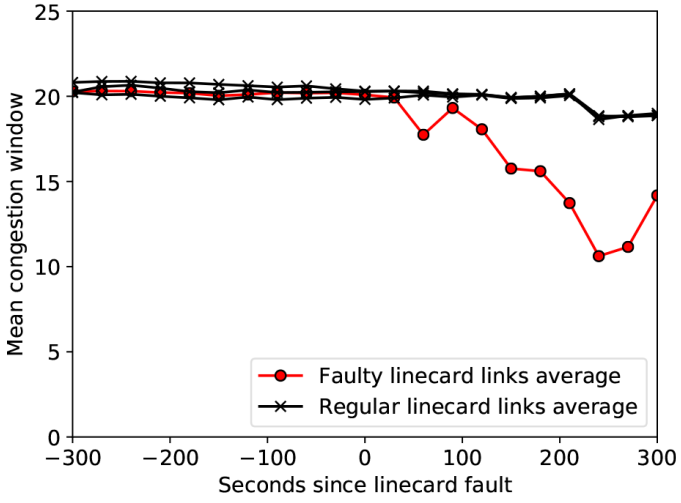


Figure 7: Mean cwnd per (host, link) during linecard fault

The first packet can be marked by the sender with some bits that instruct the first switch in the path to stamp the packet only—for example, we might use the IP TTL field to arrange this. The second can be marked so the second switch in the path marks it, and so on until we send H packets to recover the full path.

Second, when it comes to applying per-switch stamps, we need to choose where in the packet we apply it. Our prototype stamps the packet DSCP field, but this is limited since it is only 6-bits wide and frequently has other uses—typically for choosing switch-queueing policies. A solution that could scale to much larger networks would be to write to the IPv6 flow label field in the packet header, which is 20-bits wide. We are necessarily limited to what switch ASICs support, though the capabilities of switch ASICs have been progressively improving.

Future Directions

While our existing prototype has leveraged some favorable characteristics of the Facebook datacenter environment—most notably, the heavily load balanced traffic distribution—we are optimistic that our approach can generalize to datacenters with different and more variable application traffic patterns, such as Hadoop cluster workloads. Additionally, we hope that our findings incentivize router manufacturers to provide more options to allow packet header manipulation, perhaps through mechanisms such as P4 [3].

References

[1] A. Adams, P. Lapukhov, and H. Zeng, "NetNORAD: Troubleshooting Networks via End-to-End Probing," Facebook Code blog, Feb. 18, 2016: https://code.facebook.com/posts/1534350660228025/netnorad-troubleshooting-networks-via-end-to-end-probing/.

[2] A. Andreyev, "Introducing Data Center Fabric, the Next-Generation Facebook Data Center Network," Facebook Code blog, Nov. 14, 2014: https://code.facebook.com/posts/360346274145943.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communi-cation Review*, vol. 44, no. 3 (July 2014), pp. 87–95: http://www.sigcomm.org/sites/default/files/ccr/papers/2014/July/0000000-0000004.pdf.

[4] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network's (Datacenter) Network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*, pp. 123–137: http://cseweb.ucsd.edu/~snoeren/papers/fb-sigcomm15.pdf.

[5] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, "Passive Realtime Datacenter Fault Detection and Localization," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, pp. 595– 612: https://www.usenix.org/system/files/conference/nsdi17/nsdi17-roy.pdf.

[6] S. Savage, D. Wetherall, A. Karlin, and T. Anderson, "Practical Network Support for IP Traceback," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '00)*, pp. 295–306: http://cseweb.ucsd.edu/~savage/papers/Sigcomm00.pdf.

[7] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*, pp. 183–197: http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/p183.pdf.

[8] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang, "NetPilot: Automating Datacenter Network Failure Mitigation," in *Proceedings of the ACM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*, pp. 419–430: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/netpilot.pdf.

[9] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, "Packet-Level Telemetry in Large Datacenter Networks," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*, pp. 479–491: https://www.cs.ucsb.edu/~ravenben/publications/pdf/everflow-sigcomm15.pdf.

Resourceful
Monitoring under the Microscope

LUCIAN CARATA, OLIVER R. A. CHICK, AND RIPDUMAN SOHAN

Lucian Carata is a Research Associate at the Computer Laboratory, University of Cambridge. He has done work in the area of provenance, root-cause analysis, I/O performance, and system measurement. lucian.carata@cl.cam.ac.uk

Oliver R.A. Chick is a passionate hacker of all things, from compilers to lower levels in the software stack. He earned his PhD from University of Cambridge with a thesis on understanding the impact of running complex workloads in virtualized environments. In the process, he devised new methods to achieve low-side-effect tracing (shadow kernels). oliver.chick@cl.cam.ac.uk

Ripduman Sohan is a Senior Researcher in the systems area at the Computer Laboratory, University of Cambridge. He has done work in the area of storage, virtualization, end-host networking, energy-efficient computing, provenance, and instrumentation. ripduman.sohan@cl.cam.ac.uk

Typically, monitoring systems record system-wide and application-level metrics *separately*, with significant time and expertise being invested in understanding how one affects the other when diagnosing complex issues. Resourceful, our open source project, bridges the gap between the two by allowing applications to record the system-level metric changes caused by each of their actions. For example, a Web server could record "the time spent in the TCP stack for servicing a request." We discuss the ideas that support this approach and provide a number of use cases showing how they can be useful in the real world.

The Usual Suspects

"Why is it slow?" (with the dreaded variant, "Why is it *sometimes* slow?") is a question that sysadmins have been asking ever since computer systems grew complex enough to run software. In response, common wisdom suggests deploying monitoring solutions such as Nagios and Munin to understand the status and evolution of production systems. More recently, open-source tools such as Prometheus, Heka, and Bosun have become popular by introducing ideas on tracking multi-dimensional time series that were battle-tested in companies with large computing infrastructures [6, 7]. They provide APIs with which software engineers can instrument their code to expose metrics for the monitoring system. The data ends up in customizable dashboards where it can be queried, used for alerts, or archived.

While there have been significant improvements in the number of available tools and low-overhead introspection mechanisms (perf, SystemTap [5], DTrace [1], eBPF), easily tying together the resources used and code paths touched inside the kernel while an application performs arbitrarily defined activities (such as executing a db query and sending back a response) remains a challenge, one which Resourceful (rscfl) sets up to solve. This is not about "fixing everything without waking sysadmins up," but exploring new design points and tradeoffs in the monitoring/debugging space that will make your life easier.

Key to this is programmability: we should start using tools that provide their results in ways that can be naturally consumed, either by dashboards, complex analysis tools, or by applications themselves, while placing everything they measure *in context*: in the context of what other applications/VMs are doing, competing workloads, and lack of perfect isolation. *No metric should be recorded without tracking the circumstances and effect it has on other metrics within the same time period.*

While at first sight simple, those initial ideas have led us to some less obvious design and implementation choices. By open-sourcing Resourceful, we hope both to start a wider discussion and to show the ability of solving some difficult real-world problems.

Resourceful: The Ideas

At its core, Resourceful allows applications to express interest in the measurement of fine-grained kernel-side metrics in order to understand the side effects of userspace actions when

interacting with the OS: Where was most of the time spent? Was their execution interrupted by the scheduler, and for how long? How did the statistics of the TCP stack (retransmits, bytes sent) change during this time? This takes an application-centric view, like a monitoring API would, but the measurements are about the OS and its resource sharing and multiplexing.

Exposing this data in a monitoring context allows gaining insights about real-time application behavior. Consider the case of a simple Web server: how would you track *per-request* page cache misses, time spent in the TCP stack, time spent doing I/O, or interactions with the servicing of other requests?

Measurements in Context

One of the significant differences between OS-level debugging tools and application-monitoring frameworks is the amount of detail they have about the running application: a monitoring framework may collect custom metrics specific to the application such as "number of client transactions per second," "time taken to run database queries fetching the front page," or "number of 404 errors per minute." This data may be collected together with per-system global metrics such as "TCP traffic," "I/O wait times," and "CPU load" and be displayed on the same dashboard for at-a-glance sanity checking.

However, once problems appear, it becomes *somebody's* (hopefully, somebody else's) task to figure out how things went wrong. How useful are dashboards in figuring out the problem? If "system-wide I/O wait times have increased while the number of transactions per second have dropped," do we have a better idea on where to look for what's causing the issue? Likely so, but only with enough experience and intuition about where the problem might be. That or a lot of trial-and-error. This is the stuff sysadmin "war stories" are made of.

We propose that it would be helpful to bridge the gap between application-specific and system-wide metrics. What if you could collect changes to system-wide metrics *in the context* of an application-specific one? What if you could have a metric of "I/O wait times for each request"? This is what rscfl is implementing through its API: applications declare the boundaries of interesting actions ("the request") and "announce" when they switch from one action to another, while an rscfl kernel module measures their kernel side effects. This can also be framed as a way of understanding what system resources are used by application-specific actions.

Integration as a Monitoring Solution

Although closer in implementation and low-level mechanisms to existing tracing tools, rscfl integrates with applications as a monitoring system would: it provides an API for collection of fine-grained metrics and allows applications to instrument code paths implementing a high-level functionality or activity (i.e., a Web server declaring "this is code for processing a Web request"). The resulting data can be further exported to inherently distributed monitoring systems such as Prometheus and be integrated in its larger monitoring infrastructure. Creating a root-cause diagnosis system like the one discussed by Ostrovski et al. [4] around this is definitely possible, and we have already built a prototype [8].

This position as the middle-man requires thought about programmability and efficiency: rscfl allows applications to access measurement results by sharing a region of memory between them and the kernel, giving direct access to results without extra copying or parsing of data. Do I hear you say, "That poses security issues"? We have looked at protecting the data as well: measurements are accessible as normal data structures within the application's address space, but by default no other applications have access to it.

Targeting the Kernel

The point at which any application interacts with the world outside its own memory address space is through the OS kernel: whether it is performing I/O, being scheduled together with other applications, or dealing with hardware failure, the kernel is the one doing the management. Our experience has been that these kernel interactions are typically some of the hardest to understand: the kernel is usually part of the code base that developers and sysadmins would like to treat as a black box that "just works."

On the other hand, you might be forced to learn about details inside the box once an application is not behaving as expected, and you're trying to find its bottlenecks. We propose solving this disconnect by explicitly exposing the notion of a kernel subsystem when returning measurement data. It seems like the right level of abstraction to talk about the kernel from an application's perspective: "It has spent this much time in the TCP subsystem"; "The file was not cached, so reading from it used the block subsystem."

In terms of actual measurements, the kernel remains the ideal place to understand the side effects of application actions: it is where resources (CPU, memory, disk) are being shared and time-multiplexed among multiple processes. However, adding lots of instrumentation can be costly. In existing probing mechanisms, the time taken to execute some measurement also depends on the total number of probes that are active. This is why rscfl uses a new type of low-overhead probing, called a KAMprobe (Kernel Advanced Measurement probe). Its execution time only depends on the complexity of the code being run inside the probe, with no dependency on how many other probes are active. We've been running kernels with tens of thousands of active measurement points without a significant performance impact.

Virtualization Awareness (Alpha)

Virtualization introduces new challenges in the picture, with multiple containers or VMs isolated to various degrees from each other on the same host, but introducing resource sharing (time, page cache, memory) that applications or OSes are not directly aware of. We have added hypervisor and containerized kernel support in Resourceful in order to be able to track those elements in the context of application actions: with this support, applications are aware (for example) of the time the VM was not scheduled in as part of the time added to the latency of particular actions. This introduces security considerations for cloud environments, but exploring this area for better understanding of workload co-location properties is very important.

Case Studies

Beyond the general configurable framework that allows anybody to extend Resourceful for tracking custom-defined kernel subsystems, we have investigated a number of use cases, generally connected to making our own systems research and problem troubleshooting easier. They are useful as examples of how the ideas presented above come together in a coherent manner.

Advanced Cache Monitoring

In a production system, caches are some of the key elements for maintaining good performance, yet keeping track of their behavior under complex workloads remains painful, with only coarse-grained summaries available at the OS-level. Collecting fine-grained information is unpopular because nobody likes slow caches: any measurement performed on them is by definition on a hot code path, where every cycle spent counts.

What about measuring things in a test environment? That doesn't often work since it's impossible to know and replicate production cache behavior—especially for shared environments like the cloud. Thus, monitoring by getting periodic snapshots of metrics like hit/miss ratios and eviction rates is typically the only realistic option. Still, wouldn't it be nice to be able to dig deeper and drive optimizations by having a map of what files were hit/missed in OS caches during different operations performed by your application?

We thought the same and leveraged Resourceful's low overhead probing mechanisms to define a PAGE_CACHE measurement subsystem. As the name implies, it tracks the OS-level page cache (normally used for file I/O, mmap, or fs metadata). Developers can choose to monitor the full cache or restrict the parts of the cache that are tracked (not interested in mmaps? why pay the overhead?). On the application side, data collection for this subsystem can be enabled through the API. When per-action aggregations are needed, their boundaries will need to be marked by API calls as well (e.g., for a Web server, mark parts of the code servicing a request or switching between them). Table 1 shows a more detailed comparison with other available mechanisms.

The result allows an application to record per-file cache statistics and give a better idea of when I/O latency degradation happens due to cache trashing. Knowing which files have incurred the most misses in the context of a particular action allows you to make informed compromises: does ensuring a particular file is cached make the code path you're interested in faster?

We have used the same functionality to characterize slowdowns caused by workload transitioning from being fully served from the cache to requiring disk accesses. In such cases, bottlenecks can shift (e.g., network-bound operations becoming disk-bound) for just part of your application, making diagnosis hard.

We're currently working to add visibility into evictors (who eliminated the cache entry that caused my process to miss?) and virtualized environments that hide shared caches (containers).

Hidden Work

The Linux kernel is able to run its own long-lived threads (kthreads) that are treated by the scheduler as any other process. They are used by the kernel to deal with long-running work (e.g., writing dirty page cache entries back to disk) or with work that cannot be completed immediately in regions where blocking is not allowed (such as interrupt service routines). In the latter case, the kernel provides a general mechanism that drivers can use to schedule delayed work: work queues.

However, work queues use a thread-pooling model where a number of long-lived kthreads wait for work to be enqueued from various subsystems and take on the execution of callback functions doing the actual work as needed. Due to this multiplexing of work belonging to different kernel subsystems and drivers, and due to the inherent asynchronicity, it is quite challenging to get a high-level understanding of what work is being carried out by a given work queue/kthread at a given time and to determine what high-level userspace action might have required its triggering.

We have defined a custom rscfl subsystem named TRACK_WORKQUEUE to help us in understanding why applications using an nvme device driver we extended were not achieving the expected throughput and latency figures. It has allowed us to monitor the creation and queuing of work inside kernel work queues as I/O requests from a benchmarking framework (fio) were issued.

This targeted investigation (monitoring the calls into the work-queue subsystem from within just a single application as opposed to system-wide) has allowed us to quickly determine that instead of using the inherent nvme parallelism, our modified driver was serializing block device requests through a single-threaded work queue. Having identified the bottleneck, it was an easy fix to increase the number of dedicated workers for that work queue, leading to significantly improved performance.

	Operation Selection	Aggregation	Metrics	Breakdown
rscfl	• selective (on file read, write, mmap) • all	• per-app • per-app action (programmer defined)	• hit/miss ratio • eviction rate • dirty entries • cached size	• per-file • summary
OS	non-selective (all)	system-wide, per-cgroup		summary
Tracing	non-selective (all)	system-wide, per-app	custom	summary

Table 1: Options offered by rscfl when monitoring caches, in comparison to default OS metrics and tracing mechanisms such as SystemTap, DTrace, or eBPF

Latency in Context

As a fully application-facing example, we have modified a non-blocking Web server to use the rscfl API for tracking resources consumed while servicing each client request. The resulting data is both pushed into Prometheus as a time series and used by a monitoring dashboard in order to understand variations in latency on a per-request basis. We've called this "Latency Explorer," a tool that dynamically allows us to compare high-latency requests with low-latency ones and try to determine where the differences appear. This provides more visibility into one of the areas of high interest for understanding any high fan-out architecture, where tail latency matters greatly [2].

In Figure 1, two views of the system are made available: a latency distribution and a per-request resource consumption breakdown based on Resourceful data. Each of the parallel axes in the bottom graph identifies a consumed resource or metric specific to the application activity (here, responding to an HTTP request). A given request is thus represented on the graph as a line linking the corresponding measured values (the dashed line in Figure 1). An idea of visual analysis using this data is to allow the selection of different intervals in the latency histogram while coloring the corresponding requests differently in the resource consumption graph (Figure 2). Further filtering is available on each of the resource axes.

Using Resourceful

Until recently, Resourceful was developed at the University of Cambridge, and while we spoke openly about the tool, its implementation was considered too immature for release to the wider world. Realizing the buzz that Resourceful was building in academic circles, we have been hard at work for the past 18 months and are now in a position where we are open-sourcing Resourceful so it can be used to increase observability in production systems. Our project is available at github.com/lc525/rscfl, and we're accepting both suggestions and contributions. If you have a monitoring problem where you believe the existing tooling is inadequate or might benefit from the ideas presented here, we would welcome your contributions.

Requirements

The core of Resourceful is a system that modifies your running kernel to insert instrumentation. In order to safely apply this instrumentation we require some capabilities that may not be accessible on some systems:

- **Elevated access.** Resourceful can be run on any Linux kernel without requiring a reboot or modification to the kernel as stored on disk. This is made possible by Resourceful scanning the running kernel, determining the parts of the running code that should be measured, and then applying itself to these regions. Doing that typically requires some form of elevated privileges. However, once rscfl is running it can be used by any application.
- **Kernel debug symbols.** Resourceful has an automated analysis that determines boundaries in the kernel that should be measured. To enable us to perform this analysis, Resourceful requires access to the kernel's debug symbols. In most Linux distributions these can be obtained as a separate package that does not modify the kernel that is running (i.e., the debug symbols live in a separate file and do not affect the running kernel).

Installation

At present Resourceful must be built from Source, however we are considering packaging it for some distributions. We maintain and provide a full set of up-to-date instructions on running Resourceful on our GitHub page, but here we outline the sets required at the time of writing.

- Installation requires you have Git, Wget, and Python 2.7 installed. We expect these will be installed on most Linux boxes.
- Beyond that, it should be as simple as running `make` and `make install`.

Modifying Programs to Use Resourceful

Resourceful supplies a C/C++ API with which userspace programs specify where they start and stop processing a given activity. While this does mean that applications need to be modified in order to use Resourceful, the changes in practice are often trivial and can be added to commonly used remote procedure call libraries in an elegant fashion. The context for mea-

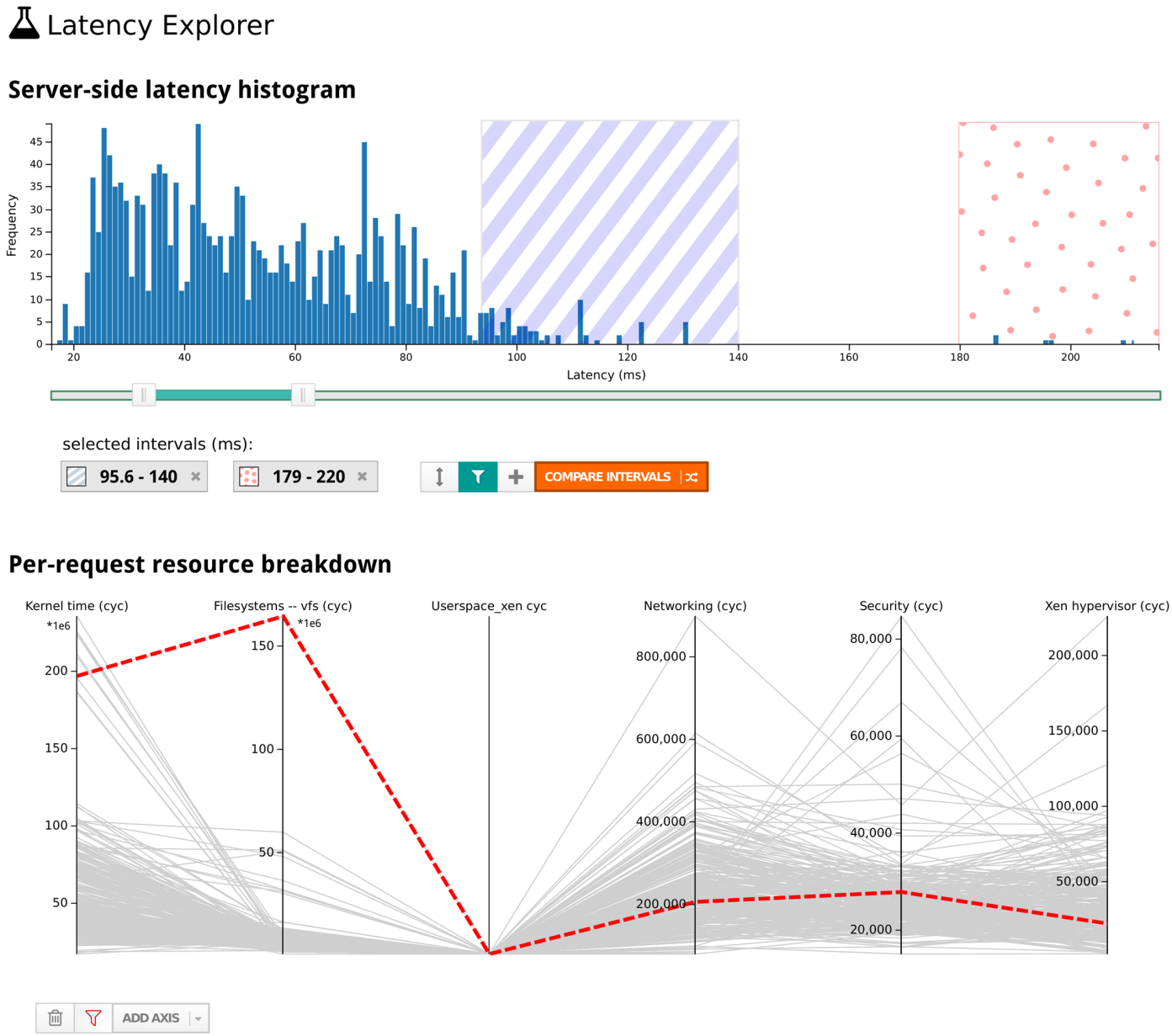


Figure 1: Latency Explorer, a visual analysis tool prototype

surements is being kept by communicating some opaque tags to the kernel. This is not unlike the strategy taken by other systems such as XTrace [3], but we are considering asynchronous behavior in greater detail. When receiving a tag that is the same as one seen before, our kernel module knows that any metric changes should be accounted to the same activity, and it can perform the aggregation directly in kernel space.

The general steps for using the API would be as follows:

1. Initialize Resourceful in your program. This creates a Resourceful "handle," which is much like a traditional file descriptor. It is passed to the other Resourceful functions and contains state about the innards of Resourceful.

```
rhdl_t rhdl = rscfl_init( ) ;
```

2. When your application starts a new activity (i.e., receives a user request), it can request a "token" for it and start accounting the resources it uses:

```
token_t token ;
rscfl_acct(rhdl, token, ACCT_START);
```

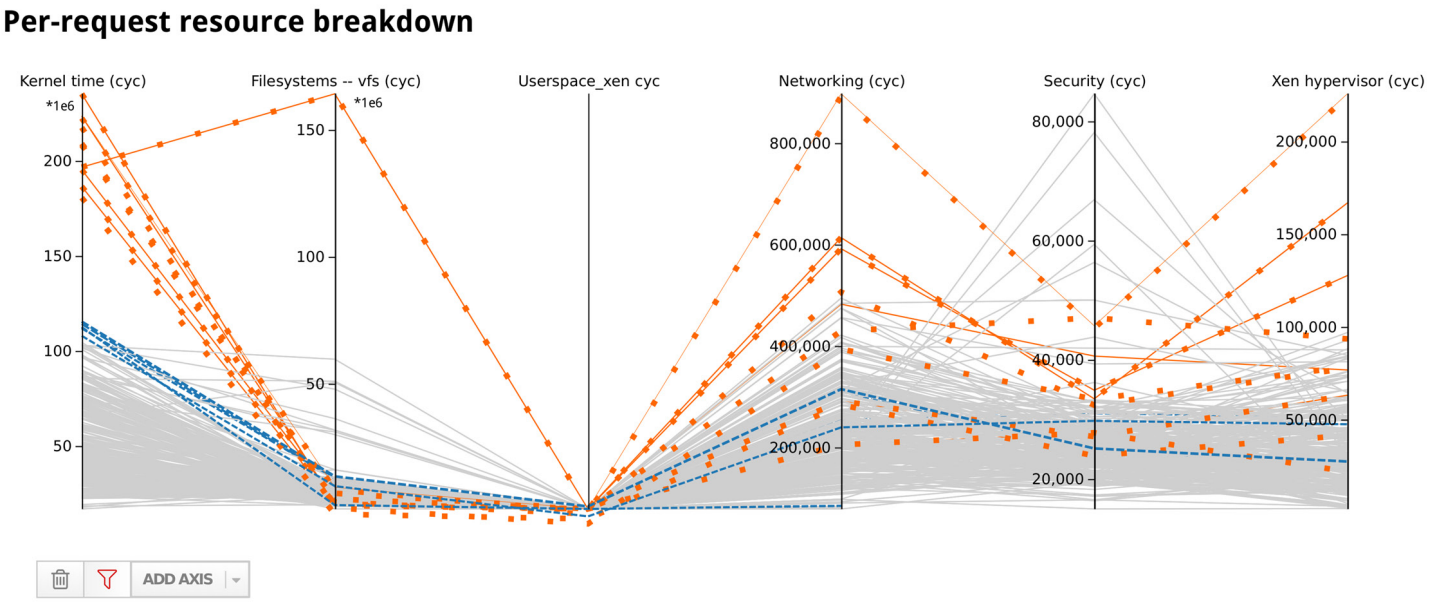


Figure 2: Latency Explorer, interactive filtering for comparing the latencies of requests selected in the Figure 1 histogram: tail latency (dotted) vs latencies between 95.6 and 140 ms (dashed). Each axis can be further filtered, and that in turn updates the histogram (how does the histogram of response times look for requests that spent a lot of time in the Networking layer?).

3. When the activity stops (i.e., the request has been sent), we can stop recording the resources used and read out the values:

```
rscfl_acct(rhdl, token, ACCT_STOP);

// Read the accounting information that we recorded.
rscfl_account_t rscfl_results;
rscfl_read_acct(rhdl, &rscfl_results);
```

`rscfl_results` is a structure from which you can read the kernel resources used in the processing of your request. This is a broken down per-kernel subsystem. For this example, we have measured a default list of performance measurement counters, however Resourceful also has APIs that can be used to measure specific resources. Resourceful also contains some magic higher-order functions that let you perform advanced aggregation of resources used across many requests (map-fold-filter).

4. In modern systems, processing often takes place in asynchronous event loops. This means the application activity might complete in stages. If this happens you can tell Resourceful to apply the resources used to a new activity by switching token:

```
rscfl_switch_token(rscfl_hdl, new_token);
```

The API also provides features for storing arbitrary application-specific metrics together with the kernel-recorded measurements, which is extremely useful when performing a detailed analysis.

Upcoming Features, Conclusion

Resourceful's API is currently available for C and C++ only, but we hope to add wrappers for other popular languages soon. In particular, this presents a good opportunity for instrumenting runtimes that provide green threads. Those can be tricky to monitor at present, and by instrumenting at the runtime level we would also limit the amount of required changes to application code. Other planned features target the extension of our visibility into virtualized environments, and we already have promising research results in that area.

We're not aiming to produce just another tool for debugging/monitoring applications. Instead, we're hoping to restart a discussion on what is needed to advance this area in ways that are helpful to practitioners. Download from github.com/lc525/rscfl and let us know what you think.

References

[1] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," in *Proceedings of the USENIX Annual Technical Conference (ATC '04)*, pp. 2–2: https://www.usenix.org/legacy/event/usenix04/tech/general/full_papers/cantrill/cantrill_html/.

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

[3] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-Trace: A Pervasive Network Tracing Framework," in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation (NSDI '07)*, pp. 20–20: https://www.usenix.org/legacy/events/nsdi07/tech/full_papers/fonseca/fonseca.pdf.

[4] K. Ostrowski, G. Mann, and M. Sandler, "Diagnosing Latency in Multi-Tier Black-Box Services," in *Proceedings of the 5th Workshop on Large Scale Distributed Systems and Middleware (LADIS '11)*: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/37477.pdf.

[5] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and B. Chen, "Locating System Problems Using Dynamic Instrumentation," in *Proceedings of the 2005 Ottawa Linux Symposium (OLS '05)*, pp. 49–64: https://www.kernel.org/doc/ols/2005/ols2005v2-pages-57-72.pdf.

[6] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers," *IEEE Micro,* vol. 30, no. 4 (July/August 2010), pp. 65–79.

[7] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google Technical Report, 2010: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/36356.pdf.

[8] J. Snee, L. Carata, O. R. A. Chick, R. Sohan, R. M. Faragher, A. Rice, and A. Hopper, "Soroban: Attributing Latency in Virtualized Environments," in *Proceedings of the 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '15)*: https://www.usenix.org/system/files/conference/hotcloud15/hotcloud15-snee.pdf.

BeyondCorp 5
The User Experience

VICTOR ESCOBEDO, BETSY BEYER, MAX SALTONSTALL, AND FILIP ŻYŹNIEWSKI

Victor Escobedo is a Corporate Operations Engineer at Google in Mountain View. Originally joining Google in 2010 through the ITRP Program, he now focuses on change and impact management. He holds a BS in computer science from CSU Fullerton. victore@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 Technical Director in the Google Cloud Office of the CTO in New York. Since joining Google in 2011, he has worked on video products, internal change management, IT externalization, and coding puzzles. He has a degree in computer science and psychology from Yale. maxsaltonstall@google.com

Filip Żyźniewski is a Site Reliability Engineer at Google in Dublin and the lead of BeyondCorp's portal project. He previously worked as a Performance Engineer at Sabre Holdings. He holds a master's degree in computer science from the University of Lodz. zyzniewski@google.com

Previous articles in the BeyondCorp series discuss aspects of the technical challenges we solved along the way [1–3]. Beyond its purely technical features, the migration also had a human element: it was vital to keep our users constantly in mind throughout this process. Our goal was to keep the end user experience as seamless as possible. When things did go wrong, we wanted users to know exactly how to proceed and where to go for help. This article describes the experience of Google employees as they work within the BeyondCorp model, from onboarding new employees and setting up new devices, to what happens when users run into issues.

Enabling a Seamless New Hire Experience

For many new employees, the idea of a BeyondCorp model is quite foreign: they're used to accessing the tools they need for their day-to-day work through VPNs, "corp wireless," and other privileged environments. When we initially rolled out BeyondCorp, many new hires continued to request VPN access from our help desk team (internally known as Techstop). From past experiences, they assumed they needed to jump through a few IT hoops if they planned to work while away from the office. The architects of BeyondCorp mistakenly assumed that users would try to access internal resources while away from the office and notice that things "just worked"—no access requests from users and no support load for Techstop would be a win-win!—but old habits die hard.

New Hire Orientation

We clearly needed to reach users earlier in their IT journey at Google, so we began introducing BeyondCorp in new hire orientation. During orientation, we explicitly avoid explaining the technical aspects of the model and instead focus on the end user experience. We emphasize that users don't need VPNs and that they're "automatically" granted remote access; they can work from the office, from their home, on a plane, or in a coffee shop without changing their workflows. During this short training, we show users the BeyondCorp Chrome extension—the most common user-facing expression of the BeyondCorp access model (for more details on the extension, see "The BeyondCorp Extension," below)—and the icon that represents a "good" connection within BeyondCorp (see Figure 2). We explain that from a good connection, they can access the vast majority of the tools and resources they need from any network connection.

New Device Setup

When users log in to their corporate devices with their corporate credentials the first time, their access settings are automatically configured. To enable this seamless onboarding experience, inventory processes and platform management tools work behind the scenes to configure a new hire device for initial setup. As described in [1], we infer device trust based on a number of signals, some observed (last security scan, patch level, installed software, etc.) and some prescribed (assigned owner, VLAN, etc.). To handle this complexity, our inventory teams follow an automated provisioning process to ensure that new hire devices are correctly trusted at first login. Once the necessary user credentials are validated,

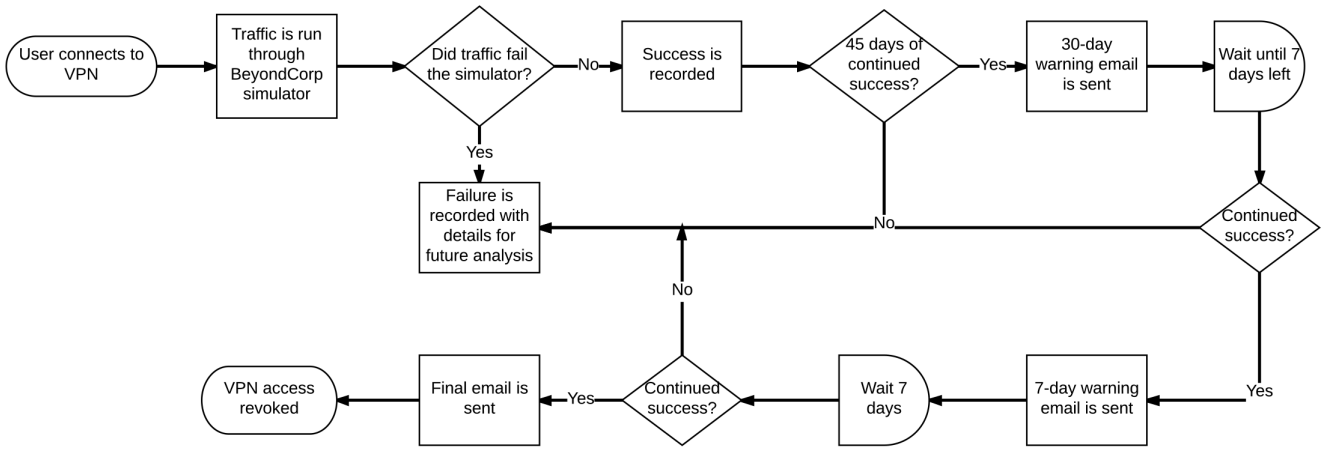


Figure 1: Automated analysis and revocation of employee VPN usage

we automatically push our custom Chrome extension to the machine.

From the user's perspective, as long as they see the green icon in the extension, they know they can access their corporate resources. By explaining the BeyondCorp Chrome extension in new hire training, we have virtually eliminated new hire confusion and support requests relating to remote access.

VPN Reduction

Although new hires learn about BeyondCorp during orientation, their first few days at Google can be a somewhat overwhelming torrent of information. Because we don't expect every person to recall every detail they learn that first day, we modified our VPN request processes and tools to emphasize the concepts introduced in orientation.

Since new hires aren't given access to our VPN gateways by default, they must request VPN access through an online request portal. On this portal, we clearly remind users that BeyondCorp is automatically configured and that they should try to access the resources they need before requesting VPN access.

As shown in the flowchart in Figure 1, if the user skips this warning, we also perform automated analysis on the services users access through the VPN tunnel. If a user hasn't accessed a single corporate service not available within the BeyondCorp model within 45 days, we send them an email. The email explains that because all the corporate resources they've accessed are supported through BeyondCorp, their VPN access will expire in 30 days unless they access a service that isn't supported by BeyondCorp. We send one more notification seven days before their VPN access expires, and then revoke permission to the VPN gateway at the end of the seventh day. This automated process allows us to proactively cull unnecessary usage of legacy access infrastructure, and will eventually allow us to turn down our VPN infrastructure entirely.

Loaners

As a side benefit of the automatic configuration implemented for BeyondCorp, we've also improved other technology experiences for our users. One of the most visible improvements is our loaner laptop program. Like many modern companies, our employees are quite mobile and freely work from their desks, meeting rooms, lounges, or their homes. Mobile devices—specifically, laptops—are incredibly vital to their productivity. To handle cases of forgotten, misplaced, or stolen laptops, we have a self-service loaner laptop program that gets users up and running again as soon as possible.

Using custom-built Chromebook loaner stations deployed around the world, any user can temporarily assign a loaner laptop to themselves for a period of up to five days. Users benefit from the ability to simply pick up a laptop and get back to work within a matter of minutes. Techstop benefits from fewer requests for loaners, which frees up their time to work on other issues. When the user returns the device or the loaner period expires, the system automatically revokes the certificate and demotes the device's trust, leaving it ready for the next user to reinitiate the loaner process.

The BeyondCorp Extension

By more or less eliminating the need for a VPN client, we can encapsulate almost all access needs—whether remote or onsite—through one entry point, the BeyondCorp Chrome extension. The extension automatically manages a user's Proxy Auto-Config (PAC) files that explicitly route special cases through the Access Proxy [2]. When a user connects to a network, the extension automatically downloads the most current PAC file and displays the good connection icon. Rules in the PAC file automatically route requests to corporate services through the Access Proxy. This allows our internal developers to deploy internal corporate Web services without explicitly configuring client access: they

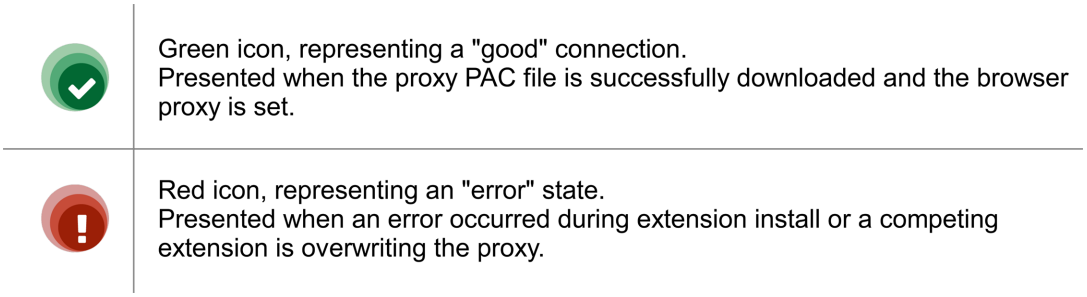


Figure 2: Icons in Chrome extension that indicate authentication state

deploy a service that will have a CNAME DNS entry in the public address space that resolves to the Access Proxy. The Access Proxy then automatically handles the user authentication and authorization.

Since the BeyondCorp extension routes all traffic through our Access Proxy, users can't communicate with devices that the Access Proxy can't reach. Additionally, the extension must be able to download a correct PAC file in order to route their traffic appropriately. This setup causes issues with common technologies like captive portals or when users need to communicate with devices on private local networks without routing through the Access Proxy. We needed a way to explain these scenarios and remediation steps to users, ideally without increasing load on Techstop. The Chrome extension's authentication state icons (shown in Figure 2) provide a gateway to further troubleshooting information.

When Things Go Wrong

What happens when things break or users run into complicated corner cases? By acknowledging that users will run into problems, we can identify the most common scenarios and develop plans to resolve them as smoothly as possible. Empowering our users to understand the problem and self-remediate when possible is our constant overarching goal.

Issues That Can Be Self-Remediated

Captive Portals

Because we're a global company with many traveling employees, users commonly encounter captive portals when working from airports, hotels, and coffee shops. These portals are usually implemented on the default gateway of a private network. When a user connects to this network, the BeyondCorp Chrome extension attempts to download the PAC file, but the captive portal prevents a successful download.

To resolve this issue, whenever the extension detects a network state change, we determine whether the device is behind a captive portal: we simply attempt to retrieve the Web page at http://clients3.google.com/generate_204, which is an empty page that always returns an HTTP 204. If we receive anything other than an HTTP 204 (most commonly, an HTTP 302), we assume that the device is connected to a captive portal. We then fall back to a predefined PAC file that we store within the extension itself and alert the user.

Users confronted with a captive portal can click on the Chrome extension icon, where we let them know that this issue is common when trying to authenticate to networks at airports or hotels. BeyondCorp is working as intended, and they just need to change the BeyondCorp setting to **Off: Direct**. Users can then complete the authentication through the captive portal, at which point the extension can successfully download the latest PAC file. This simple flow allows users to completely self-remediate with minimal downtime and no support load on our Techstop.

Local Network Devices

Users also frequently attempt to access devices on private address spaces. Many Google employees use their corporate laptops for tasks like configuring personal printers or other networking equipment. However, since we route all connections through the Access Proxy, access fails when the BeyondCorp extension is enabled. Similar to the captive portal use case, the solution is to change the BeyondCorp setting to **Off: Direct**. Unlike the previous case, we can't easily detect this failure state. Typically, users in this scenario have an active and functioning Internet connection. From the extension's point of view, everything is working normally and the user can access all corporate resources, so there is no reason to raise an alert.

To figure out how to effectively interface with users in this situation, we worked through a representative user journey: an engineer takes their corporate laptop home and wants to use it to change a setting on their home printer, which they connect to via its IP address. The user connects to their home network, and the BeyondCorp extension connects successfully, downloads the latest PAC file, and configures the browser's proxy. When the user enters the printer's IP address in a new browser tab, the request is sent to our Access Proxy along with all other private address space traffic. The routing request fails and the user gets an error.

We came up with a solution to this user journey by focusing on the end result: an error page from the Access Proxy. We created a custom HTTP 502 error message to insert into our error pages when certain conditions are met—specifically, whenever we return an HTTP 502 and the user was attempting to reach an RFC1918 or RFC6598 address. The error message explains to the user that if they were trying to access a local network device such as a home router or printer (the two most common cases we found), they need to switch the BeyondCorp extension to **Off: Direct**. In this way, we were able to use already existing infrastructure and processes to allow users to self-remediate the issue.

Custom Proxy Settings

Our employees sometimes need to set custom proxies to test ads in foreign countries. If a user installs multiple extensions that each try to set the proxy, the extensions collide with each other, which can confuse users and break their access to corporate resources.

We approached this use case with two solutions. First, we integrated foreign country proxy settings directly into the BeyondCorp extension. When users have a business need to egress from a specific location, they can select that location from a dropdown of supported countries directly within the extension. This provides our users a single extension that manages their most common business proxy needs.

Additionally, when a user has a valid need to run a secondary proxy management extension, their BeyondCorp icon switches from green to red. We then give them an option to change their state to **Off: System Alternative** and explain when they want to use this setting. Again, this process allows the user to self-remediate, increasing their productivity and reducing queries to our support teams.

Explaining Complicated Failures: The Portal

For simple cases, like those described above, we could empower users to self-remediate using quick customizations to our error pages or the Chrome extension. However, in cases of legitimate denials of access, we knew that users and support teams would want or need to know why they were denied. The complex, multi-layered ACL logic in our back-end infrastructure can make understanding the logic behind a specific decision difficult for users and support teams alike. It might take even a seasoned SRE multiple minutes of querying many internal services to identify the cause of a single 403 error page. Given the volume of 403 error pages served by our Access Proxy daily (~12M for HTTP/S alone), human involvement in troubleshooting is unscalable and impractical.

To facilitate diagnosing and troubleshooting more complicated BeyondCorp access issues, we designed a single portal to assist both users and support teams. Instead of just telling a user that they were denied access to a resource with a generic error code, we explain why they were denied and how to resolve the issue. The portal is standalone, rather than integrated directly in the Access Proxy, because it uses more granular ACLs that depend upon the end user's current trust level. Since the Access Proxy is available publicly by design, we wanted to limit the amount of knowledge an attacker can gain from the 403 error pages.

Architecture

The portal is roughly split into a front end and a back end, with an API that communicates between the two.

- The front end is an interactive Web service. It issues requests against the back-end API based upon input from the user.
- The back end can query multiple infrastructure services involved in access decisions. It deliberately omits various caching layers so users receive fresh information.
- The API between the front end and back end is also exposed for other uses, like batch processing and analysis, or embedding the output in other tools.

Explanation Engine

Beyond querying and surfacing ACLs, the portal also needs to present this information to users in a useful way. We built an explanation engine to provide troubleshooting details in response to parameters of deny requests. It operates by recursively traversing a tree of subsystems that provide authorization decisions.

For example, the Access Proxy ACL might require a device to be fully trusted in order to access a particular URL. Upon retrieving this ACL, the engine contacts our device inference pipeline to retrieve the conditions necessary to access the corporate resource. We then propagate this information to our front end and translate it into plain language, so the user can visit the portal to find out what's wrong with their current state and how to fix the problem.

ACLing the ACLs

While the explanation engine provides users with helpful information, the data it exposes can be sensitive. It reveals the problematic ACLs of protected systems and discloses information about the state of the user's account and device—all useful information for potential attackers. Defining the ACL for this data is a tricky process, as we need to balance tool usability against the need to protect sensitive information.

Depending on the user and device requesting troubleshooting information, we can replace sensitive nodes in the output with

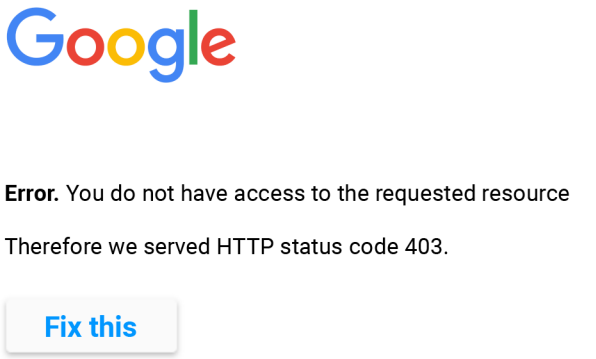


Figure 3: An error page displayed when BeyondCorp blocks a request

less specific variants. In extreme cases, we replace a node with instructions to contact our Techstop. In such cases, our Techstop and SREs can help users without disclosing sensitive information by verifying the user's identity and viewing the relevant information on their behalf.

Access Deny Landing Page

Once we developed the portal, we exposed it to users by integrating it into our Access Proxy error messages. When a user hits an HTTP 403 error, they see a button routing them to the portal, where we've automatically forwarded all relevant error details (see Figure 3). The portal then replays the access request against the back end and explains exactly what caused the issue.

For example, if a resource requires membership in a specific group, the portal provides the group name and a handy link to our group management system so the user can request access. Behind the scenes, the portal queries our back-end ACL services to determine the authorization requirements of the resource in question, and compares that information against the user's group memberships. The front end then converts the result of that comparison into a human-understandable statement (see Figure 4). This all happens in a matter of seconds, far faster than it would take the user to puzzle through group membership issues or reach out for assistance.

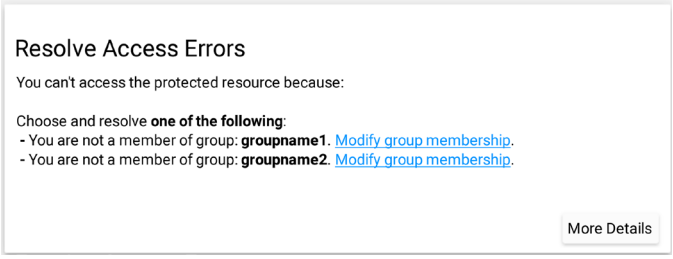


Figure 4: Employee-facing guidance on troubleshooting an access denied error

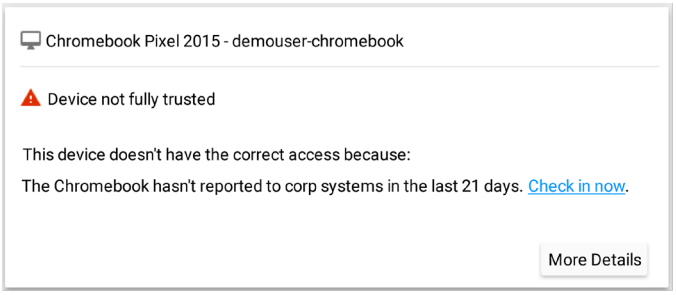


Figure 5: Service desk-facing guidance on troubleshooting an access denied error

Integrating this flow directly into our error messaging allows users to complete this process seamlessly and—most importantly—completely via self-service.

Ad Hoc Troubleshooting

Although we expect most users to access the portal through an error page, we also provide a direct page for more ad hoc troubleshooting. This landing page on our portal front end is customized according to the identity of the user and device accessing it. It presents information about the user and all their devices, and highlights issues that can potentially result in denial of access. By allowing end users to proactively visit this tool to get a global view of all of their devices and potential future access issues, we equip them to remedy issues with any of their devices in one fell swoop. This feature is particularly handy for checking device trust before a trip or demo.

Empowering Support

This front end also empowers our Techstop team to perform detailed troubleshooting quickly by providing immediately actionable steps, which dramatically reduce time to resolution. For example, to explain a 403 error page, techs can use the portal landing page to query for a specific username or device identifier. They can drill down into a specific device to determine whether it's a fully trusted corporate device. If it's not, we present the exact reasons why the device is not trusted and how the tech can resolve the issue (see Figure 5).

Future Goals

Beyond its current functionality, the portal also presents avenues for further automation. In the future, we plan to continuously run checks for potential denial of access issues. We'll notify users of any impending issues they can resolve on their own before those issues manifest in a detrimental way. Similarly, we'll identify critical issues that can't be self-remediated and automatically notify our Techstop with remediation steps. We also hope to expand the range of issues we can solve automatically without human intervention.

Focus on the Experience

Although the migration to BeyondCorp was challenging on multiple technical fronts, it allowed us the freedom to reevaluate our primary user support experience. By focusing on our users during and after the migration, we could deeply integrate processes and features that allow them to navigate the complex network model with ease. We designed our tools so that the user-facing components are clear and easy to use. These interfaces were purpose-built to allow self-remediation whenever possible, freeing up both user time and support channels. When users do need extra help, we provide tools and information to make our Techstop maximally productive.

For the vast majority of users, BeyondCorp is completely invisible. While Google employees worry about their own workflows, the model takes care of any and all access logistics. When users do have issues, we step in quickly and efficiently, giving them just the right information at just the right time to get them up and running again. Then we step back behind the scenes and let them focus on what they do best.

References

[1] B. Osborn, J. McWilliams, B. Beyer, and M. Saltonstall, "BeyondCorp: Design to Deployment at Google," *;login:*, vol. 41, no. 1 (Spring 2016), pp. 28–35: https://www.usenix.org/publications/login/spring2016/osborn.

[2] L. Cittadini, B. Spear, B. Beyer, and M. Saltonstall, "BeyondCorp Part III: The Access Proxy," *;login:*, vol. 41, no. 4 (Winter 2016), pp. 28–33: https://www.usenix.org/publications/login/winter2016/cittadini.

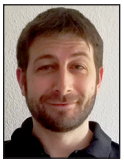
[3] J. Peck, B. Beyer, C. Beske, and M. Saltonstall, "Migrating to BeyondCorp: Maintaining Productivity While Improving Security," *;login:*, vol. 42, no. 2 (Summer 2017), pp. 49–55: https://www.usenix.org/publications/login/summer2017/peck.

Safe Parsers in Rust

Changing the World Step by Step

GEOFFROY COUPRIE AND PIERRE CHIFFLIER

Geoffroy Couprie handles security and quality assurance at Clever Cloud, develops in Rust, and researches on parser security at VideoLAN. He thinks a lot about cryptography, protocol design, and data management.
contact@geoffroycouprie.com



Pierre Chifflier is the head of the intrusion detection lab (LED) at ANSSI (French National Information Security Agency). He is interested in various security topics such as operating systems, boot sequence, compilers and languages, and new intrusion detection methods, and he is also trying to link all these topics by improving detection tools, writing safe parsers, and deploying tools in a secure architecture. Pierre is also a Debian Developer and has been involved in free software for a long time.
chifflier@wzdftpd.net

Parsers are critical parts of applications, exposed to potentially malicious data but also plagued by the same bugs over a period of years, like memory-related problems. Solutions exist but are often not adopted: many of them require rewriting entire software packages. We describe how to leverage Rust's safety features and close integration with C, the strength of the nom [1] parser combinators library, along with a thorough methodology [2] to make existing software much more secure by rewriting critical parts. By surgically replacing functions, we intend to initiate a change towards robust and memory-safe parsers.

A large part of our infrastructure is built on a sand castle. We have been reusing the same code for decades, the same libraries written in the '90s, the same applications, the same operating systems. We tried, and are still trying, to maintain them, patching bit by bit, mostly in reaction to published vulnerabilities, sometimes as a proactive effort. But all that old code is slowing us down.

And if that was not enough, to connect those pieces of code to each other, we have pages and pages of unclear, ambiguous specifications for file formats and network protocols. How can you be sure your implementation is correct when some remove features, some add features, others implement them incorrectly, and there are parts that are completely open to interpretation. Let's also mention that incorrect files generated by one broken application often end up supported by everyone else.

Additionally, most of that software has been written in C (sometimes still written in K&R) and involves unsafe practices and insufficient testing.

One could say it is a miracle that all of this has worked this long, but there is no luck in that. It is the result of incremental work of thousands of developers patiently fixing bugs, and system administrators monitoring failing services. But we are losing the race now.

Attackers only get better: what was previously difficult gets simpler, and the tools only get smarter. More vulnerabilities are published every day, while we keep the same old code and the same development practices.

We Cannot Rewrite Everything

Whatever the quality of all that code, we cannot replace it. Software gets reused over and over, with each generation of developers building upon what the previous one built. There's much more churn in hardware than software: hardware gets replaced, software stays. We can write new software with better solutions, but it would not fix the millions of devices currently in place, or the billions of applications actually running. Our only option is to strengthen the sand castle bit by bit until it can weather the storm.

How can we achieve that? Even rewriting application by application or library by library is a Sisyphean task. Most of those projects are written in C, containing 10k to 10m lines of code. Large parts of that are unmaintained, but there's also a huge domain knowledge embedded in the code. Thousands of bug fixes, improvements, and experimentations with the specifications were done over the years. And the developers themselves carry most of this knowledge. Rewriting a project completely means losing that knowledge and hitting most of those bugs the old project solved. In addition, rewriting the project entirely creates political issues and requires teaching the new ways to developers, all while maintaining the old version. This is impossible to do in most cases.

Here is what we propose: there are specific parts of applications and libraries, weaker than the rest, that could be rewritten, while keeping all of the domain knowledge present in the rest of the code. Since file formats and protocols are the point of entry in most applications, we concentrate on the parsers and state machines, an often overlooked and vulnerable part of the code.

The LangSec approach is in changing the way we view software: we usually see our programs as some kind of engine or industrial machine that we set up and monitor but that, except for the occasional button push, largely runs by itself. That vision is flawed: our computers, operating systems, and programs are designed to modify their behavior in complex ways depending on their input.

The data you feed to your code—be it network packets, files, sensor data—drives your code, not the other way around. That specific bit at that specific address in the file determines whether your code goes into the `if` or the `else` of that specific branch. Your application is in fact a virtual machine, and its language is the input data. What can we do with this language? By modeling that input language correctly, or restricting it to a manageable subset, we can greatly reduce the attack surface of our applications in their most vulnerable elements.

If we replace the parsers and protocols in an existing application, we can better protect it from the attackers' point of entry while keeping the most useful parts of the code running. To that end, we need languages and tools that can easily integrate themselves inside a C application.

Choosing the Tools

We decided to use Rust for various reasons: the language is designed to avoid memory vulnerabilities and development issues frequent in other languages. Rust does not use garbage collection; the compiler is smart enough to know when to allocate and deallocate memory. The compiler will complain if the code is unsafe. With this, the compiler can protect your code from common flaws like double free, use after free, adding bounds check to buffers, etc. Rust is even able to know which part of the code owns which part of the memory, and it warns you when your code manipulates data from multiple threads.

Rust has been available for years now (first stable release in May 2015) and has been steadily improving. Because of the focus on the compiler, instead of fixing a memory safety issue in your code, you can improve the compiler so that nobody will ever get that issue again. Do not fix bugs, fix bug classes.

As you learn more Rust, you tend to rely more and more on the compiler to verify the code, instead of keeping track of dozens of pointers in your head, thus freeing you to think about the most valuable parts of the application.

Along with those features, Rust can work at the same level as C applications. There's no runtime. There is no garbage collector (important in time-critical software). It can even work without an allocator. As an example, it can be used for embedded development, from microcontrollers to larger CPUs. To that end, Rust code can easily import C functions and structures and use them natively, but the opposite works as well: you can expose functions and structures to be used by C (or other language) applications. This is a crucial aspect of rewriting C code: sometimes, we have to expose and manipulate the exact same types the target application is using.

Writing parsers manually in Rust is not enough. We can still find bugs, although they are often less critical than the ones you would find in C applications [7]. Parsing software correctly is hard, and anybody can make mistakes.

So we use nom [1], a parser combinators library written in Rust. Parser combinators are an interesting way to handle data. You assemble small functions, like one that recognizes "hello," or one that recognizes alphanumerical characters, and you combine them to make more complex parsers through the use of combinators. There are combinators for lots of cases, like "terminated," that would apply two parsers in a row, then return the result of the first if both are successful, or branching combinators that apply different parsers depending on the result of a first one.

Those parsers are always functions with the same signature, which means even complex parsers can be easily reused in other parsers. You end up writing a lot of small parsers, then you can test them separately, and reapply them in larger parsers as you see fit. An approach based on parsers generated from a grammar, on the other hand, tends to lack flexibility and is harder to test. Such parsers are also quite restrictive in what you can allow from the format you are trying to pass. But since nom parsers are just functions, you can perform whatever complex, ambiguous, dangerous tasks you need to, and as long as the interface is the same, you can plug that parser with other parsers. This is an important property, since most formats are badly designed and can require unsafe manipulations.

The nom library leverages Rust features for performance and safety: since the compiler always knows which part of the code owns which part of the memory, and tracks references properly, nom can work on slices of the original data instead of copying bytes around. In most cases, the parser will only allocate on the stack and be zero-copy [3].

nom has been available for some time now and has been used extensively for various formats and protocols in production software.

Armed with a safe, low-level language, and a parser library, we can now start rewriting core parts of our infrastructure.

How to Replace Part of a C Application

Not all existing applications will easily support a rewrite of their parser. If that part of the code is highly coupled with the rest, it will be problematic. Thankfully, as said earlier, we do not need to rewrite everything. Find a restricted subset of the parser, isolate it, rewrite it, then expand to other parts of the application.

The key is in defining the interface correctly. Deterministic functions are the easiest to replace, and structures are usually the hardest, since multiple parts of the code might use directly internal members of that structure (accessors are not a common practice in C). But there are a lot of tricks one can use to help in the task. As an example, commenting out a member of a structure and launching a build can expose all of the uses of that member, which makes it easier to measure how much work is needed.

When performing a rewrite, you will often need to import C code and expose your Rust code to C. You can write the Rust definitions and the C headers by hand, but Rust has tools to automate this. Rust-bindgen can import C structures and functions from C, and generate Rust bindings. While the generated code might be a bit complex at times, it is a great way to start a project and generate code that you can edit later. The opposite way works as well: you can employ rusty-cheddar to generate C headers.

The missing part for the integration is the linking phase: think of how you will link the Rust part to the C part. Do you make a static or dynamic library? Do you generate an object file that you feed to autotools? The Rust compiler can generate any of those, and they can then be handled by the build system, be it autotools and makefiles, CMake, scons, etc.

On the build-system side of things, Rust uses the cargo package manager to download libraries (called crates), build and link them, and publish new libraries and applications. That tool greatly increases the productivity of Rust developers. Unfortunately, the package management part requires an Internet connection to download packages, which might not be an option (do you expect your makefile to make network calls?). Fortunately, cargo is easy to extend with separate tooling. You can use cargo-vendor or cargo-local-registry to download crates in advance and store them in an archive somewhere. That way, you can freeze the dependency list of an application and make its compilation reproducible, while keeping a simple way to update dependencies when needed.

Start Integrating Some Rust

Once you have the build system set up, you can start actually writing Rust code. We would recommend that you develop the nom parser in a separate crate: that way, you can reuse it in other projects (Rust or other languages), and you can employ Rust's unit testing and fuzzing facilities. Any fuzzing result can then be reused as a test case for your parser.

nom parsers work well on byte slices, a Rust type that contains a pointer and a length. You can easily transform any C buffer to this. They never modify their input, and they don't even need to own it. This is important for integration in C applications: even if we know that Rust code could be stronger than the rest of the application, it is still a guest in someone else's house. If possible, let the host code handle allocations, opening files, etc. This is a really good tip to apply, because libraries with reentrant, deterministic functions without side effects are easy to integrate, and I/O is where most of the errors can happen. This is also a part that (hopefully) has been stabilized long ago in the host application.

The nom parser can return sub slices of the input without copying them and will guarantee that the data is within the bounds. In some cases, it does not even need to see the whole input. As an example, for media formats, you would read a block's header, let nom decide which type of block it is, and the parser would tell you how many bytes of the block you need to send to the decoder.

Here is the code of the TLS 1.3 ServerHello structure definition and message parsing:

```
rust
pub struct TlsServerHelloV13Contents<'a> {
    pub version: u16,
    pub random: &'a[u8],
    pub cipher: u16,

    pub ext: Option<&'a[u8]>,
}

pub fn parse_tls_server_hello_tlsv13draft18(i:&[u8])
    -> IResult<&[u8],TlsMessageHandshake>
{
    do_parse!(i,
        hv:     be_u16 >>
        random: take!(32) >>
        cipher: be_u16 >>
        ext:    opt!(length_bytes!(be_u16)) >>
        (
```

```
            TlsMessageHandshake::ServerHelloV13(
                TlsServerHelloV13Contents::new(hv,random,cipher,
                ext)
            )
        )
    )
}
```

This code generates a parser reading some simple fields, and an optional length-value field for the TLS extensions (not parsed in that example), and returns a structure. All error cases are properly handled, especially incomplete data.

One characteristic of TLS is that the parsing of messages is context-specific: the content of some messages cannot be decoded without having information about the previous messages. For example, the type of the Diffie-Hellman parameters, in the `ServerKeyExchange` message depends on the ciphersuite from the `ServerHello` message. Because of that, the context-specific part is separated from the parsing. A state is used to store the variables, and a state machine is implemented to check that transitions are correct, and also to choose the next parsing function when needed.

The state machine is implemented using pattern matching on the previous state, and the parsed incoming message, to select the new state.

```
rust
match (old_state,msg) {
    // Server certificate
    (ClientHello,         &ServerHello(_))         =>
    Ok(ServerHello),
    (ServerHello,         &Certificate(_))         =>
    Ok(Certificate),
    // Server certificate, no client certificate requested
    (Certificate,         &ServerKeyExchange(_)) =>
    Ok(ServerKeyExchange),
    (Certificate,         &CertificateStatus(_))   =>
    Ok(CertificateSt),
    (CertificateSt,       &ServerKeyExchange(_)) =>
    Ok(ServerKeyExchange),
    (ServerKeyExchange, &ServerDone(_))          =>
    Ok(ServerHelloDone),
    (ServerHelloDone    ,&ClientKeyExchange(_))  =>
    Ok(ClientKeyExchange),
    // ...

    // All other transitions are considered invalid
    _ => Err(InvalidTransition),
```

In some cases, the next state depends not only on the message type but also on content. In that case, the packet content is also used in the pattern matching to select the new state.

Finally, the combinator features of nom are especially useful for protocols like TLS: TLS certificates are based on X.509, which uses the DER encoding format. This makes writing an independent parser easier, as in the following code:

```
rust
use x509::parse_x509_certificate;

/// Read several certificates from the input buffer
/// and return them as a list.
pub fn parse_tls_certificate_list(i:&[u8])
    -> IResult<&[u8],Vec<X509Certificate>>
{
    many1!(i,parse_x509_certificate)
}
```

Parsing an X.509 certificate is done by combining the DER parsing functions:

```
rust
pub fn x509_parser(i:&[u8]) -> IResult<&[u8],X509Certificate> {
    map!(i,
        parse_der_defined!(
            0x10,
            parse_tbs_certificate,
            parse_algorithm_identifier,
            parse_der_bitstring
        ),
        |(_hdr,o)| X509Certificate::new(o)
    )
}
```

Be wary of the high coupling that can appear between the parser and the rest of the code in some C applications. This is where most of the work can happen and is usually the result of years of hacks upon hacks to add a feature "quick and easy."

We usually recommend that the parser has a clear interface with the rest of the code, in the form of a list of small, deterministic parsers and a reduced state machine above it: not a complete state machine intertwined with the parsing (as in this http parser [8]) since those are hard to debug and extend, nor a state machine informally implemented via calls from other parts of the code.

The state machine is the main interface for the rest of the code: you feed it data to parse, it decides which parser to apply depending on the current state, changes its state depending on the data that was parsed (if successful), then returns with info to drive the input consumption: how many bytes to consume (or how many more bytes are needed) or to stop consuming if there was an error. You can then query this state machine for the information you want and for data to write back to the network (in the case of a network protocol).

If the code is not highly coupled, you could even rewrite function by function, since the Rust code can expose C-compatible functions. Beware, though: take the time to write a correct internal API for Rust code, since at some point, you might stop exporting those functions and call the underlying functionality directly from Rust.

You could spend a large part of the work making the new parser bug compatible with the old one. This is often a bad approach, since both parsers will probably not recognize the exact same set of files. You only need to worry about recognizing the same representative set of samples. Most C parsers are not even really tested regularly anyway. If you still want to get close results to the original parser, you could employ a smart fuzzer to do the work of testing the difference. Write a program that wraps both the C parser and the new nom one, and that panics if both parsers do not return the same result.

Once the parser is written and in the source, be happy, for now the "interesting" part of the work will begin: getting it accepted in the tree and deciding how you will handle the software suddenly requiring a Rust compiler along with the old C toolchain.

Going Further

This approach of surgically rewriting parts of an application works well since it is designed to have a minimal impact on the original project. It can be used as a stepping stone to start replacing larger parts of the application once all the details of build systems and developer training are handled.

But some projects could never handle that kind of precise touch. Some libraries, still in active use today, have highly coupled spaghetti code, relying heavily on GOTO or setjmp, and are basically untested and unmaintained. This is one of the rare cases where we'd recommend rewriting the whole project in Rust. This is a place where this language can shine; you could write a whole new library, completely API-compatible with the old one, that you could drop into package managers as an alternative.

Think of how many parts of our infrastructure we could replace like this, bit by bit. It's a Herculean task, so we need to start now.

This work was presented in the 2017 LangSec Workshop [4], in the "Writing parsers like it is 2017" [2] paper. The parsers and tools are published in the Rusticata [5] and VLC module [6] GitHub projects.

References

[1] Rust parser combinator framework: https://github.com/Geal/nom.

[2] P. Chifflier and G. Couprie, "Writing parsers like it is 2017," IEEE LangSec Workshop '17: http://spw17.langsec.org/papers/chifflier-parsing-in-2017.pdf.

[3] G. Couprie, "Nom, a Byte-Oriented, Streaming, Zero-Copy Parser Combinators Library in Rust," IEEE LangSec Workshop '15: http://spw15.langsec.org/papers/couprie-nom.pdf.

[4] IEEE LangSec Workshop'17: http://spw17.langsec.org.

[5] Rusticata: Safe parsers community: https://github.com/rusticata.

[6] Helper library to write VLC modules in Rust: https://github.com/Geal/vlc_module.rs.

[7] List of Rust applications with bugs found by fuzzing: https://github.com/rust-fuzz/trophy-case.

[8] Node.js http parser: https://github.com/nodejs/http-parser.

Quick Testing

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

A small confession: when writing code, I don't usually write tests first. There, I've said it. Hate me. I suspect I'm not alone among Python developers. Yes, yes, testing is important, and for my major projects, tests still get written. However, for a lot of small things like little scripts, utilities, and personal projects, I just don't bother because I don't want to think about all of the extra steps and tooling that's usually involved. However, a recent conference experience may have changed some of my views. In this installment, I discuss a more lightweight approach to testing along with a brief introduction to some third-party testing libraries, including pytest [1] and Hypothesis [2].

A Revelation

Early this summer, I attended a talk by Aur Saraf at PyCon Israel in which he live-coded a simple interpreter from scratch in about 25 minutes [3]. Live coding in front of an audience is always a dicey affair, but what struck me about this particular talk is the fact that it was done entirely in a test-driven development style with no connection to any sort of testing tools, third-party libraries, or even standard library modules. I was both stunned and amazed.

The gist of the idea is simple. If you're going to write a function, you might as well first write an assertion or two for it. For example, suppose you were writing a function to split a URL into parts. You might start by writing this:

```
def split_url(url):
    pass

assert split_url('http://www.python.org') == ('http', 'www.python.org')
```

The `assert` statement serves as a kind of expectation for what you want to happen. Naturally, the code is going to fail immediately as you haven't actually written the function. However, the assertion gives you a target to aim for. So your next step is to implement the function and make the assertion pass.

```
def split_url(url):
    parts = url.split('://')
    return (parts[0], parts[1])

assert split_url('http://www.python.org') == ('http', 'www.python.org')
```

It passes. Very good. At first glance, this might seem too minimal and maybe even a bit crazy. However, there's a certain genius to it. First, it doesn't require any special knowledge of libraries or tools (e.g., the `unittest` standard library module): `assert` is a built-in statement of the core Python language. There are also no separate files to maintain or extra functions to write—the `assert` is just inlined right there in the code. It executes right after the function is defined. This means that the code won't even run or import unless the test passes. Thus, if you're working on some new thing and changing your code a lot, it can be useful to just leave it in there for the time being. It's a minimal test that doesn't require too much thought and doesn't really interfere with what you're doing.

Getting back to Aur's talk for a moment, he proceeded to write his entire interpreter in this style. Assertions first and then functions. As I watched, I kept thinking, "I bet I could use something like this." I also recognized that it could be a useful stepping stone to other more advanced testing tools. So let's explore that further.

Putting It into Practice

In one of my current projects, I'm faced with the problem of implementing a priority queue. A standard technique for creating such a queue is to use a heap data structure. In fact, Python provides a `heapq` standard library module that can be used to do it. However, my specific problem has the extra requirement of supporting cancellation (i.e., the ability to remove/cancel items anywhere in the queue). Sadly, the standard `heapq` module has no support for that. In fact, efficiently removing items from a priority queue is a rather tricky algorithmic problem. Thus, it seems that I'm probably going to have to roll my own class for it.

Let's start by sketching out a class:

```
class PriQueue:
    def __init__(self):
        pass

    def push(self, item):
        pass

    def pop(self):
        pass

    def remove(self, item):
        pass
```

It does nothing, but let's write some assertions that encode our expectations of how it should work:

```
class PriQueue:
    ...

# Test code (put right after the class)
q = PriQueue()
q.push(4)
q.push(3)
q.push(7)
q.push(10)
q.remove(4)
# Popping all items produces them in order
assert [ q.pop() for _ in range(3) ] == [ 3, 7, 10 ]
```

Running this code, it will fail because we haven't implemented anything. However, we can now fill in some details of the implementation:

```
# pqueue.py

import heapq

class PriQueue:
    def __init__(self):
        self.heap = []

    def push(self, item):
        heapq.heappush(self.heap, item)

    def pop(self):
        return heapq.heappop(self.heap)

    def remove(self, item):
        self.heap.remove(item)

q = PriQueue()
q.push(4)
q.push(3)
q.push(7)
q.push(10)
q.remove(4)
assert [ q.pop() for _ in range(3) ] == [3, 7, 10]
```

If you run this code, it passes its simple test and we're on our way.

From Asserts to Functions

Having assertions placed in the code is really only a starting point. As the code evolves, you can move the test into a more proper function. For example, maybe you do this:

```
# pqueue.py
...

def test_priqueue():
    q = PriQueue()
    q.push(4)
    q.push(3)
    q.push(7)
    q.push(10)
    q.remove(4)
    assert [ q.pop() for _ in range(3) ] == [3, 7, 10]

if __name__ == '__main__':
    test_priqueue()
```

Writing a function is an easy step—you don't even have to change your testing code (well, other than indenting it). However, if you do this, you'll open the doors to incorporating your tests with other testing tools.

For example, this code can be executed under a testing tool like pytest [1]. One of the nice things about pytest is that it works using standard Python `assert` statements. Assuming that you have it installed, drop into the shell and type this:

```
bash $ python3 -m pytest pqueue.py
================== test session starts ==================
platform darwin -- Python 3.6.1, pytest-3.0.2, py-1.4.31,
pluggy-0.3.1
rootdir: /Users/beazley/Desktop/UsenixLogin/beazley_fall_17,
inifile:
plugins: hypothesis-3.11.6
collected 1 items

pqueue.py .

=============== 1 passed in 0.00 seconds ===============
```

Excellent. Keep in mind it didn't take much to get here. No special imports or fooling around with the `unittest` module—just a function with an `assert` in it. Later on, you could move the testing function over to a more dedicated testing file. For now, it's fine where it is. After all, we're still working.

From a Function to Hypothesis

One of the problems with our code is that the test is fairly minimal. It tests just one case. How are we to know if our queue code actually works as intended across all inputs? We could generate more test cases by hand, but doing so is going to be rather painful and error-prone if it involves a bunch of cut-and-paste.

To better handle this, let's change our testing function so that it is parameterized with some inputs:

```
def test_priqueue(items, remove_item):
    q = PriQueue()
    for item in items:
        q.push(item)

    # Remove the given item
    q.remove(remove_item)
    items.remove(remove_item)

    # Verify that items come out in the proper order
    assert [ q.pop() for _ in range(len(items)) ] == sorted(items)
```

This change allows us to feed different inputs into the function. For example, we can do this:

```
...
if __name__ == '__main__':
    test_priqueue([4,3,7,10], 4)
    test_priqueue([9,2,1,8,5], 2)
    test_priqueue([4,1,6], 1)
```

Running this, you'll find that the code still seems to pass for those three test cases. Our confidence is building. However, how do we really know that we've covered all of our bases? It's hard to say.

One of the more interesting tools on the Python testing front is Hypothesis [2]. In a nutshell, Hypothesis can randomly generate test cases for you as long as you are able to describe the parameters to the test. Take the above test function exactly as you've written it and decorate it as follows:

```
# pqueue.py

...

from hypothesis import given
from hypothesis.strategies import lists, integers

@given(lists(integers(min_value=0, max_value=9),
             unique=True, min_size=10, max_size=10),
       integers(min_value=0, max_value=9))
def test_priqueue(items, remove_item):
    q = PriQueue()
    for item in items:
        q.push(item)

    # Remove the given item
    q.remove(remove_item)
    items.remove(remove_item)

    # Verify that items come out in the proper order
    assert [ q.pop() for _ in range(len(items)) ] == sorted(items)

if __name__ == '__main__':
    test_priqueue()
```

At first glance, this looks a bit scary, but the `@given` decorator is used to describe the arguments to the `test_priqueue()` function. In this case, the first argument (`items`) is going to be a 10-element list of unique integers with values in the range 0 to 9. The second argument (`remove_item`) is an integer with a value in the range 0 to 9.

Running the new code, you'll now find that it fails. Your output might vary from this, but it will look roughly like this:

```
$ python3 pqueue.py
Falsifying example: test_priqueue(items=[1, 2, 3, 4, 0, 5, 6, 7,
8, 9], remove_item=0)
Traceback (most recent call last):
  File "pqueue.py", line 35, in &lt;module&gt;
    test_priqueue()
  ...
  File "pqueue.py", line 32, in test_priqueue
    assert [ q.pop() for _ in range(len(items)) ] == sorted(items)
AssertionError
```

What's happened here is that Hypothesis has automatically found a test-case that fails and is reporting it. To better see what happens, put a print statement in your test code:

```
# pqueue.py

...
@given(lists(integers(min_value=0, max_value=9),
             unique=True, min_size=10, max_size=10),
       integers(min_value=0, max_value=9))
def test_priqueue(items, remove_item):
    print('TRYING:', items, remove_item)
    q = PriQueue()
    for item in items:
        q.push(item)

    # Remove the given item
    q.remove(remove_item)
    items.remove(remove_item)

    # Verify that items come out in the proper order
    assert [ q.pop() for _ in range(len(items)) ] == sorted(items)
```

Now, let's clear the environment and try running again:

```
bash $ rm -rf .hypothesis
bash $ python3 pqueue.py
TRYING: [3, 0, 1, 9, 8, 6, 4, 5, 2, 7] 5
TRYING: [9, 6, 4, 8, 3, 2, 0, 7, 1, 5] 1
TRYING: [9, 6, 4, 8, 3, 2, 0, 7, 1, 5] 1
TRYING: [9, 6, 4, 8, 3, 7, 1, 0, 2, 5] 1
TRYING: [9, 6, 4, 8, 3, 7, 1, 0, 2, 5] 1
TRYING: [9, 6, 4, 8, 3, 2, 0, 7, 1, 5] 1
TRYING: [9, 6, 4, 8, 3, 2, 0, 7, 1, 5] 1
TRYING: [9, 6, 4, 8, 3, 2, 0, 7, 1, 5] 1
TRYING: [9, 6, 4, 8, 3, 7, 0, 2, 1, 5] 1
TRYING: [9, 6, 4, 8, 3, 2, 0, 7, 1, 5] 1
...
TRYING: [0, 3, 4, 6, 1, 2, 8, 5, 7, 9] 0
Falsifying example: test_priqueue(items=[0, 3, 4, 6, 1, 8, 2, 5,
7, 9], remove_item=0)
TRYING: [0, 3, 4, 6, 1, 8, 2, 5, 7, 9] 0
Traceback (most recent call last):
  File "pqueue.py", line 36, in &lt;module&gt;
    test_priqueue()
  ...
  File "pqueue.py", line 33, in test_priqueue
    assert [ q.pop() for _ in range(len(items)) ] == sorted(items)
AssertionError
```

In this case, you'll see the test function invoked repeatedly with all sorts of inputs. Basically, Hypothesis is trying random inputs searching for a failure. Since our code is buggy, it will eventually find one although it might take some searching. That's pretty neat. It found a bad test case, and I really didn't have to do much work. Our testing code is still pretty small—just a single function.

Fixing the Bug

In the case of my example, there is a bug in item removal. When the item is removed, the underlying heap structure is not preserved properly. This can be fixed with a minor change.

```
# pqueue.py

import heapq

class PriQueue:
    def __init__(self):
        self.heap = []

    def push(self, item):
        heapq.heappush(self.heap, item)

    def pop(self):
        return heapq.heappop(self.heap)

    def remove(self, item):
        self.heap.remove(item)
        heapq.heapify(self.heap)    # <- Add this line
...
```

If you run the program again, you'll see Hypothesis fire 200 random inputs at the `test_priqueue()` function, but they'll all pass. In fact, each time you run the program, you'll get a different set of inputs as it searches for failing test cases. Should a failure be found, it will be recorded for inclusion in further tests. For now, we're safe though.

Final Thoughts

This whole approach to testing out new code and small libraries is interesting. When starting out, the inlined assertions provide a basic level of testing for implementing the initial code. Those tests can naturally evolve into a testing function that can be used with popular testing tools like pytest. Later, you can evolve that testing function into something for use with a package like Hypothesis, where hundreds of test cases can be generated for you automatically. The code is still small and it's allowing me to focus on the actual problem I'm trying to solve. For example, with just that one testing function, I can start experimenting with different queue implementations and have a reasonable expectation of finding bugs if I break anything. It's neat.

References

[1] pytest: http://pytest.org.

[2] Hypothesis: http://hypothesis.works.

[3] Aur Saraf, at PyCon, Israel: http://il.pycon.org/wwwpyconIL/agenda/174.

Practical Perl Tools
Come Fly With Me

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'. dnb@usenix.org

I occasionally read Quora for fun. I recently stumbled upon the following question:

What are the two closest airports to each other in the world?

The very first answer to the question I saw was from Kevin Lin who said:

> For fun, I wrote a Python script to do the following:
>
> (1) Take the list of airports from http://www.airportcodes.org/ and remove all the airports listed as "Bus service" or "Rail service" or "Van service" or "All airports".
>
> (2) Plug the remaining airports into http://www.gpsvisualizer.com/geo...to get their GPS coordinates.
>
> (3) Finally, compute the distances between pairs of airports by plugging their GPS coordinates into the haversine formula http://stackoverflow.com/questio...

You can find this question and answer here: https://www.quora.com/What-are-the-two-closest-airports-to-each-other-in-the-world/answer/Kevin-Lin.

Lin didn't include his Python code, but I was intrigued by the problem and thought I would take a swipe at doing this in Perl using roughly his method to see how hard it would be. Turns out it isn't that difficult, though there are a few tricky bits and some limitations we'll discuss later on. Let's take a walk through my implementation of Lin's solution and see what we can learn.

Oh, the Modules You Will Go

I don't know how hard Lin's implementation leans on existing extensions to Python, but since the availability of modules to do almost anything is one of Perl's strengths, I decided it would be fine to use them whenever they would make things easier for me. Here's the collection in play:

```
use HTTP::Tiny;
use HTML::Strip;
use Geo::Coder::Google;
use Algorithm::Combinatorics qw(combinations);
use GIS::Distance;   #::Fast
```

The first two will be used to grab the airport list Web page and remove all of the HTML from it. The second will be used to geolocate all of the airports. Algorithm::Combinatorics will make it easy to come up with all of the distances we will need to compute, and GIS::Distance will perform that calculation for us. The comment on GIS::Distance is meant to be a reminder that it would be advantageous to us to also install GIS::Distance::Fast in addition to GIS::Distance. The Fast module implements the distance calculations in C (versus the pure Perl implementations that ship with the main module). These much faster implementations will get used by GIS::Distance automatically if the Fast module has been previously installed.

Let's Get the Airports

Here's some code to fetch the contents of the page, strip off the HTML in the page, and then extract a list of all of the airports from the remaining text:

```
my $code_source = "http://www.airportcodes.org";
my $reply = HTTP::Tiny->new->get($code_source);
my $hs = HTML::Strip->new();
my @airports =
  grep ( /\w, [\w\s-]+\s?\(/,
    ( split( /\s?\n/, $hs->parse( $reply->{content} ) ) ) );
```

That last line is kinda gnarly (sorry), so let's take it apart piece by piece, working from the inside out.

First off, we need the contents of the page as returned by the HTTP GET operation:

```
$reply->{content}
```

Then we will want to strip out any of the HTML tags in the page:

```
$hs->parse( $reply->{content} )
```

Now that we have just the text, which largely consists of a string containing a bunch of lines (most of which contain an airport), we'll want to split the text into a list of lines:

```
split( /\s?\n/, $hs->parse( $reply->{content} ) )
```

With me so far?

As an aside, the use of `\s` in the `split()` takes care of an annoying property of the data where some of the airport listings have a trailing space. Mostly a cosmetic problem, but it was bugging me while I was writing the code. A few seconds ago I said "most of which contain an airport." The use of `grep()` here makes sure we only collect the lines that appear to contain an airport listing:

```
grep ( /\w, [\w\s-]+\s?\(/,
  ( split( /\s?\n/, $hs->parse( $reply->{content} ) ) ) );
```

I suspect there are more direct ways to extract only the airport data from this page using one of the HTML-parsing/extraction modules, but this method of tossing the HTML and grabbing only the lines we wanted seemed relatively straightforward.

Let's Geocode

We've dived into Geocoding in previous columns a couple of times, so I won't dwell too much on the process. One thing I do need to note is that in this example code, I'm using the Google Maps API Geocoding service, which is (after a certain number of calls) a paid service. More info on it here: https://developers.google.com/maps/documentation/geocoding/start (pricing can be found here: https://developers.google.com/maps/pricing-and-plans/).

Let's look at the code:

```
my $geo =
  Geo::Coder::Google->new( "key" => "{YOUR API CODE HERE}",
  );

my %airports;

foreach my $airport (@airports) {
   next if $airport =~ /[vV]an service/;
   next if $airport =~ /[bB]us service/;
   next if $airport =~ /[bB]us station/;
   next if $airport =~ /Park&Ride Bus/;
   next if $airport =~ /Van Galder Bus/;
   next if $airport =~ /[rR]ail service/;
   next if $airport =~ /[aA]ll airports/;
   next if $airport =~ /Heliport/;
   print STDERR "Locating $airport...";
   my $location = $geo->geocode( 'location' => $airport );

   if ( !defined $location ) {
      print STDERR "not found.\n";
      next;
   }

   $airports{$airport} = [
      $location->{geometry}{location}{lat},
      $location->{geometry}{location}{lng}
   ];
   print STDERR "done.\n";
}
```

I think the process is pretty straightforward. Once we initialize the geocoded object with our API key (see the doc I mentioned earlier for how to get one), we walk through the list of airports we scraped and attempt to geocode each one. As per Lin's solution, there are a number of bus and van service listings that aren't real airports, so we attempt to skip them.

As an aside, there's another thing I would probably do in the next version of this program to clean the data that Lin doesn't mention. There are (by my count) 59 duplicates in the data where the same airport code is listed in two places with slightly different descriptions—for instance:

```
Biloxi/Gulfport, MS (GPT) & Gulfport, MS (GPT)
Endicott, NY (BGM) & Binghamton, NY (BGM)
Leon, Mexico (BJX) & Guanajuato, Mexico (BJX)
Canton/Akron, OH (CAK) & Akron/Canton, OH (CAK)
```

It would be very simple to extract the airport code from each airport and store it in a hash after you attempt to geocode an airport. Then, before geocoding the rest, just skip any airports you previously have a hash entry for already. I leave this (and any other data cleanup you want to do) as an exercise for the reader.

Back to the action. For each airport, we store its latitude and longitude if the geocoder can find them. In my experience, it finds a very large percentage of the airports. If this were a setting where I really cared deeply about the results, I might choose to call a second geocoding service to attempt to find any that Google doesn't have listed.

The lookup (at least from my laptop and decent Internet connection) on average takes about a second or so to complete for each airport. If we wanted to speed this whole thing up, we could use one of the techniques we've discussed in past columns to make a number of queries in parallel. I have no doubt that Google can handle the multiple queries at once, so this would provide a dramatic speedup.

And while we are discussing optimizations, an even better addition would be code that could avoid doing the geolocation at all. It would be best to cache previous results we get back and drop them into some sort of persistent store (even just to a flat file). When we ran the program again, we could skip a query if we've already made it. This would save time, save you money from API calls, and speed things up tremendously on future runs. Given how seldom airports move locations, this is probably a safe thing to do in almost all cases.

Go the Distance

Okay, time to calculate the distance between every possible pair of airports. This process consists of determining all of those pairs and then computing the distance for each.

Figuring out the pairs is something we could do with some loops, but instead let's use this opportunity to learn about two of the easier modules for this process: Algorithm::Combinatorics and Math::Combinatorics. Both have an easy way to ask for all of the combinations of list elements. I choose the former because it uses some C extensions for speed, but if you need a pure Perl solution, Math::Combinatorics will work as well.

Algorithm::Combinatorics' combinations() subroutine will hand us back an iterator. We just call next() on that iterator each time we want a new pair of airports (when it runs out of pairs, it returns undef):

```
my $pairs = combinations( [ keys %airports ], 2 );

my %distances;

my $gis = GIS::Distance->new();

while ( my $pair = $pairs->next ) {

   my $trip = $pair->[0] . '-' . $pair->[1];
```

Above we snuck in the initialization of the GIS::Distance object, so let's talk about that next. There are a number of different ways to compute distance between two points, the most common is the haversine formula. So sayeth Wikipedia:

> The haversine formula determines the great-circle distance between two points on a sphere given their longitudes and latitudes.

(Be sure to check out the Wiki page on this for some other interesting trivia.)

By default, GIS::Distance uses this formula by default. Calculating the distance between the two airports becomes this easy:

```
   print STDERR "computing distance between $trip...\n";
   $distances{$trip} = $gis->distance(
      $airports{ $pair->[0] }->[0],
      $airports{ $pair->[0] }->[1] =>
      $airports{ $pair->[1] }->[0],
      $airports{ $pair->[1] }->[1]
   )->{values}->{kilometre};
```

We just ask the module to compute the distance between the pair of airports by feeding in the latitude and longitude of the first airport followed by the same for the second airport. GIS::Distance wants to hand us back a Class::Measure object (which could be handy later if we wanted to do conversions), but we immediately look up the actual value in kilometers and store it in the %distances hash instead.

Show Me the Distances

The last piece of code prints out the results (all 5,016,528 of them) sorted from shortest distance to longest distance. This was, by the way, the moment I realized that there were duplicate entries in the data as mentioned above. Finding two airports with 0 distance between them seemed mighty suspicious. Here's the code:

```
foreach my $trip (
  sort { $distances{$a} <=> $distances{$b} } keys %distances )
{
   print "$trip: $distances{$trip} kilometres\n";
}
```

And the Answer Is...

If you run the code, you get an answer. I find interesting that I got a slightly different answer from the one mentioned in Quora (though Lin's top answer is in the top five list). Here are the airports with the shortest distance between them:

```
Comox, BC (YQQ)-Vancouver, BC (YVR): 1.30501111815652
  kilometres
Vancouver, BC - Coal Harbour (CXH)-Comox, BC (YQQ):
  1.37633222675128 kilometres
Vancouver, BC - Coal Harbour (CXH)-Vancouver, BC (YVR):
  2.11243449299644 kilometres
Omsk, Russia (OMS)-Orsk, Russia (OSW): 2.28071127591897
  kilometres
```

```
Port Protection, AK (PPV)-Point Baker, AK (KPB):
  2.68865415751707 kilometres
Lebanon, NH (LEB)-White River, VT (LEB): 2.80395259148673
  kilometres
```

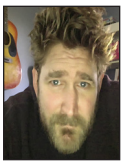
And just for the sake of completeness, here are the top five longest distances:

```
Rio Cuarto, CD, Argentina (RCU)-Fuyang, China (FUG):
  19993.286433724 kilometres
Padang, Indonesia (PDG)-Esmeraldas, Ecuador (ESM):
  19994.1381628879 kilometres
Ile Des Pins, New Caledonia (ILP)-Zouerate, Mauritania (OUZ):
  20000.9443793096 kilometres
Long Lellang, Malaysia (LGL)-Tefe, AM, Brazil (TFF):
  20002.7713227265 kilometres
Palembang, Indonesia (PLM)-Neiva, Colombia (NVA):
  20011.325933595 kilometres
```

If you do happen to fly any of these distances, do write me, I'd love to hear about it. And with that, take care, and I'll see you next time.

iVoyeur
Stacks and Piles

DAVE JOSEPHSEN



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

I keep having this conversation with my coworkers. Honestly, it's probably to be expected given my penchant for harping on about monitoring tools. Also, I was admittedly quite spoiled at my last job, Librato—a place whose singular mission in life is operational visibility, where everyone has unfettered access to a functionally infinite, free, world-class, metrics platform—where things were, of course, different.

Anyway, the conversation I'm talking about usually starts off with me suggesting some tool that we could use to measure something. "Well how many foos per second are actually happening in real life?" I'll ask, expecting a number rather than a shrug in response. Alas, no one will know, so I'll suggest that we count them. "Do we have a graphite instance up anywhere?" I'll ask.

"No," they'll answer slightly annoyed, knowing full damn-well that I know full damn-well by now that there is no graphite instance, "we use Monitoring Tool X."

"Ah hah," I reply delighted, having successfully baited them into my personal little Platonic dialog. "But I'm not talking about monitoring, I'm talking about *measuring*."

Yes, delight. It delights me every single time, which, I recognize maybe is a little pathetic, but I'm already too old to care. In fact, one of the things I'm genuinely enjoying about the aging process is a certain sort of selfish introspection. It's great. You'll be walking down the street and suddenly realize that you keep on offering to meet people for a beer when you don't particularly like beer. And it just goes on like that, realization after realization that you've been engaging in all these behaviors that you kind of despise, and then, best of all, you just *stop doing those things*—like pretending to know what DevOps means, or living in Texas.

Anyway, most people don't really catch my meaning when I say I'm talking about *measuring things as an activity distinct from monitoring things*, so this portion of the conversation usually involves a lot of skeptical sideways glances and eye-rolling. And, honestly, I hear myself. I sound like a pompous windbag who swallowed a know-it-all jerk. The words emerging from my lips sound like something a televangelist might say if televangelists were really opinionated on the subject of IT monitoring tools. Like, these sentences could only emerge from the lips of someone who doesn't live *here*, in the bloody trenches with you and me. Someone who will soon jet back to the money-laden consulting partnership from which he oozed. I get that. I do. So the first thing I do is remind them what they have to go through to measure the number of foos traversing the wire with Monitoring Tool X.

First you need to know Monitoring Tool X itself: its YAML/XML/JSON/whatever configuration DSL along with its questionable world-view and unique collection of pseudo-random assumptions that I'm sure totally made sense at the time. Then, these days, there's usually a code promotion and review process, so you'll have to traverse those as well as possibly a change control process. Those things only apply if you're lucky enough to be allowed to actually *change* Monitoring Tool X. I don't have numbers, but I'm willing to bet that most engineers in most places aren't. Most engineers in 2017 still need to traverse a gatekeeper to affect Monitoring Tool X, which means filling out something akin to a trouble-ticket.

And so nobody measures.

Of course they don't. What carpenter would measure if she had to submit paperwork in XML before she could use the tape measure? Maybe someone would, but I would not hire that person and neither should you. I mean, at this point I've been dealing with monitoring system configuration syntax for over 20 years, and *I wouldn't* bother to measure if that alone was the bar to entry. I'd monitor, sure. But 10–15 minutes config time per new metric? I'd never *measure*.

But what's the big deal? I mean, ultimately, what do I lose? Obviously, we can *get by* without measuring. We can make things that work. Yesterday I walked in to my living room and brushed against a stack of recently purchased books in want of a shelf, but I did not knock them over. They teetered off balance, and, eventually, they might fall over as a result of their imbalance, but for the time being that stack remained a stack rather than a pile.

That stack is *working*. It's getting by. Exactly like so many other well-monitored tech-stacks in the interclouds. And when they fall over...when the stack becomes a pile, our monitoring tells us so and we intervene. Like a fire-alarm. That's how monitoring works. You don't want the fire-alarm going off when stuff isn't on fire, and so you restrict access to it, to make sure nobody messes it up.

That's not measuring. Measuring is what we do when we want to understand the things we build. How many queries is my service actually putting on the wire? How many threads does it spawn with real-life users? What's actually faster, the new parsing function or the old? Is round-robin actually round-robining (Hint: No)? Measuring invites us to answer these questions for ourselves. No paperwork. No fuss. Like a tape measure in our pocket, this is self-service. Nobody is worried about you breaking your tape measure.

When we measure, we can communicate actual, real-life systems behavior to one another, rather than hunches and estimates. Its output is truth. Not Warning, not Critical, just Truth. Measurement, therefore, gives us a common basis of understanding. It reaches across disciplines like application-development and ops (or SRE or whathaveyou) and provides a common comprehension of operational reality. Measurement gives us the ability to have objective conversations about the best way to fix things, and as your operational visibility improves, you begin to formulate a tangible sense of normality, and inversely, abnormality. You move from alerting on problems to detecting imbalance. You stop saying holy shit and start saying huh, that's weird, and seemingly overnight, you find yourself intervening before the stack falls over rather than scrambling to clean up piles.

Most importantly, measuring things changes *you*. It's one thing to read about the process versus thread model in Web servers, but it's quite another thing to see it for yourself. Measuring things, it turns out, removes the political subtext from our technology discussions. You no longer have to invest belief in the solutions for which you advocate. You are free to question and to formulate hypotheses and test them. It's habit forming, and it's a *really good* habit for an engineer.

From Logs to Sprites

A few days ago I participated in my first Hackathon at Sparkpost, and since I kept having this conversation, I thought I'd try to make something that celebrated the act of measuring as opposed to monitoring. Coincidentally, I've also been playing around lately with Phaser.io [1], a videogame development framework for HTML5-enabled browsers, so I thought I'd try to make a little traffic visualization toy.

DNS and SMTP are the lifeblood of Sparkpost, yet no second-scale metrics systems currently exist to visualize this traffic. Given this, I figured it would be impossible to render this traffic and not learn something in the process. I wanted to *show* everyone what our mail flow actually looked like, so I settled on SMTP and got coding.

Some 24ish hours later, Sparkviz was born, and I was super happy with how it came out. Here's a video of it in action [2].

On the far left, you see two Amazon ELBs: one balances inbound REST traffic from our customers and the other SMTP. This traffic is represented by green balls. The next tier inwards is our MTA tier. These servers relay mail outward to various proxies (the third tier), which in turn deliver to the Internet (represented as a large orange ball on the right). You'll notice the right-hand side of the screen is metered from 10 to 256. These obviously form a scale of first octets. Email successfully delivered appear as blue dots, which hit the far right-hand side of the screen at the point matching their destination IP's first octet.

The yellow balls represent transient bounces, and the red balls that impact the floor are permanent delivery errors. As the project took shape I noticed that heavy traffic often obscured patterns, so I used phaser's "enableDrag()" method on each of the sprites to make them draggable, as you can see in the video. When this wasn't quite enough I added a toggle to squelch out the errors entirely.

The project totaled 407 lines of code: 161 lines of JavaScript and 246 lines of Go. Unfortunately, I can't share it, but there's no reason it couldn't be open-sourced eventually.

It's implemented as a daemon designed to run on our internal log aggregation boxes. It listens on a UNIX domain socket for log-lines, which it parses and extracts into JSON blobs. You can see my highly technical architectural design document for the daemon in Figure 1.

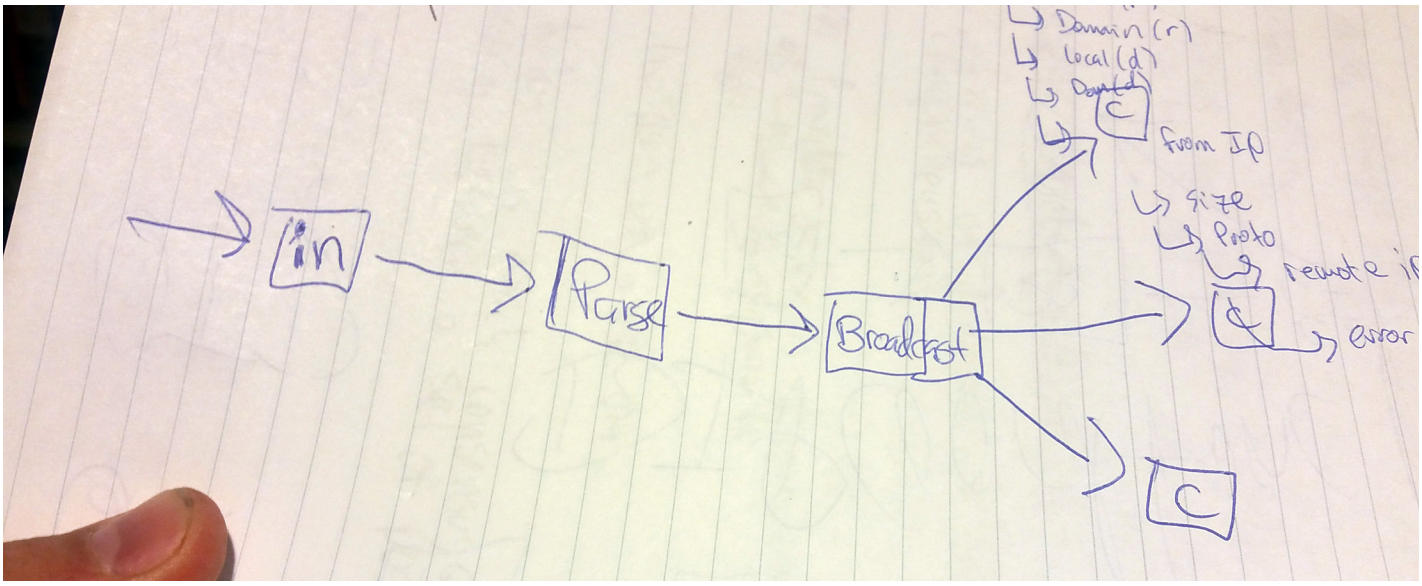


Figure 1: Highly technical architectural design document

The daemon also listens on port 8000 for HTTP clients, to whom it delivers the phaser-based JavaScript UI. The UI, running in the browser on the client, turns around and creates a WebSocket connection back to the server. The daemon keeps a globally scoped slice of these connected WebSockets and broadcasts each parsed log line to every connected client as a JSON blob (using a millisecond sleep function inside each client's broadcast go-routine to throttle the outbound traffic to 1000 blobs per client per second).

Differentiation of traffic type happens client-side, where the JavaScript UI uses a series of handler functions to parse out the event-type from each inbound JSON blob, pushing them on to another queue with the appropriate sprite value for phaser to render and tween. The tl;dr is that I created a firehose between the MTA logs and the end-user's browser. As always with hack-day projects, there's plenty of room for improvement, but as you can see, it gets the job done.

As I suspected, we all learned quite a bit from the exercise. It's kind of impossible for humans to avoid pattern-parsing data like this, and you don't need to look at it very long to recognize that we have a distribution imbalance in this environment. Certain MTAs clearly prefer certain proxies. Like the books in my living room, this stack works despite its imbalance. I, for one, am really looking forward to smugly pointing back to Sparkviz when I curmudgeonly lecture my contemporaries on the importance of operational telemetry, a process from which I'm sure I will extract far more than 407 lines of delight.

Take it easy.

References

[1] Phaser.io: http://phaser.io/.

[2] My traffic visualization tool in action: https://www.youtube.com/watch?v=htidm6DWq2s.

Golang
Creating and Using Certificates with TLS

CHRIS MCENIRY

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

In this article, we're going to extend Kelsey's original work from Spring 2016 *;login:* on the gls service [1]. To recap, gls is a distributed ls tool, which calls out to a listening service to perform a directory listing. One of the open items left from that article is the concern around authentication and authorization. To extend that, we're going to add secured authentication to both sides of the gls tool and with this we're going to gain a minimal amount of authorization.

The ubiquitous Internet connection security protocol is currently Transport Layer Security (TLS). TLS is used to encrypt, authenticate, and authorize (to a degree) connections. The defaults handle encryption for us well enough, so in this article, we're going to examine authentication and authorization. Authentication is based on the names on exchanged certificates that have been signed by third party certificate authorities. Once identity has been established, the service can then incorporate a base level of authorization based on the names (e.g., parsing user=$username so it will get access to items specific to $username) on the certificates or on the certificate chain (e.g., this was signed by the "users" CA, so it will get access to common user items).

In our example, we want to ensure four items: encrypted communication, successful identification of the glsd server (that the one gls connects to is the proper one), successful identification of the gls client (that the one that connects to the glsd server is the proper one), and restricted access of the gls client as appropriate. To accomplish this, we're going to add TLS between the client and the server, enable verification on both server and client, and compare the certificate identity to a good list. In order to support all of this, we need to first generate some private keys and certificates for gls and glsd to use.

NOTE: We've cut some corners to simplify the example in this article. Several additional areas should be considered in a full production PKI infrastructure, including, but not limited to, use of intermediate CAs, revocation lists, full subjects, selection of hash, key properties, private key encryption with a passphrase, etc.

Certificates

In terms of authentication, TLS is a form of public key cryptography. If you're not familiar with it, you can read Radia Perlman's *;login:* article about Bitcoin [2]. The issue with plain public key cryptography is that you have to distribute the public keys. Instead of having to distribute every certificate for every service to every potential user of that service, TLS builds a chain of trust in the same way that a Web browser authenticates a Web site like a bank or hospital.

When I use a browser to connect to a Web site, the site sends my browser a certificate. This certificate has the Web site's public key and a subject name that identifies the Web site, and it is signed by a trusted third party called the certificate authority (CA). My browser has a bundle of certificate authorities, and it looks for a match for the signature in that bundle. If there isn't a match, the browser will alert about an untrusted certificate. With a matching

signature, the browser can verify that the Web site's certificate has been issued by the CA, and so the browser trusts it. In this way, the browser doesn't have to have the certificate for the Web site ahead of time but only needs to have a much smaller set of certificate authorities to use to verify.

After the chain of trust has been used to verify that the Web site's certificate is valid, the browser does another check. This time, it takes the subject name on the certificate and compares that to the DNS name that the browser used to connect. If the certificate name does not match the DNS name, the browser will alert to a name mismatch. If it does match, the browser trusts the Web site and proceeds.

This chain of trust can be used to authenticate the client side as well, with one caveat. The Web server can require that my browser supplies a certificate as well, and it can compare the signature on that certificate to its bundle of trusted certificate authorities. In most cases, this is for an internal or private situation, so there's only one certificate authority to check against, but uses can vary. However, a DNS check of the client is unlikely to work in many cases: multiple clients behind a Network Address Translation, residential networks, or networks behind dynamic addressing are all unlikely to be able to issue certificates appropriately to match the actual end client. Therefore, the server is very unlikely to check the name on the certificate in the same way as the client does to authenticate the server. The server uses the certificate in two ways: the name on the certificate can be used to identify the user or provide a group or role; and the fact that the certificate is signed is often used to provide a base level of authorization ("if it's signed, it's allowed in").

Since this is a private service, we can consider that our certificate authority handling and chain handling is working together. That allows us to only produce three certificates: a common certificate authority, a server certificate, and a client certificate. The server certificate will get the localhost name since that is what is being used to connect to; and we're going to encode a username, glss Client A into the client certificate to show a stronger authentication approach than just verifying the certificate.

Building on the gls Package with the glss Package

Before we start, we need a place to work that isn't conflicting with previous work. We want to use the existing work of the RPC mechanisms in the gls package and only add the pieces that we need. We're going to use the built-in package manager `go get` to pull in Hightower's work, and augment this with our own working path. For article space, the full code is not in this article, but it is available on GitHub [3]. You can pull in the final source code for this exercise along with the original source code. If you want to assemble the code yourself, this article steps through that, but you will have to fill in some of the gaps. To get started down that path:

```
$ go get github.com/kelseyhightower/gls
$ mkdir -p $GOPATH/src/github.com/cmceniry/login-glss
$ cd $GOPATH/src/github.com/cmceniry/login-glss
$ mkdir -p certs server client
```

Otherwise, you can pull in the new code along with the original:

```
$ go get github.com/kelseyhightower/gls
$ go get github.com/cmceniry/login-glss
```

`go get` will place the gls package at $GOPATH/src/github.com/kelseyhightower/gls. We will be referencing it in our import statements much as we do for the standard library utilities:

```
import (
   "fmt"
   "github.com/kelseyhightower/gls"
)
```

Instead of using the utilities gls and glsd in the existing gls package, we're going to create three new utilities in the login-glss package: `client/main.go` and `server/main.go`, to hold the service like before but with TLS encryption, and a new command, `certs/main.go`, which we'll next use to generate our keys and certificates.

Generating Keys and Certificates

As a private service, we're going to handle all of the certificate and certificate authority management internally. In a production case, this may work, or you may want to use a commercial vendor or Let's Encrypt [4]—the process for obtaining certificates and keys is slightly different, but we'll end up with the same resulting items. In addition, since this is again internal, we're going to use one certificate authority for the client and the server certificate signing. Since this exercise is on Go, we're going to generate these using Go itself. Let's start this by opening a new file:

```
certs/generate_certs.go
```

The Go standard crypto library has all of the functions needed to generate certificate/key pairs. We'll want to import these libraries and some other ones that we'll be using into our file:

```
package main

import (
  "crypto/rand"
  "crypto/rsa"
  "crypto/x509"
  "crypto/x509/pkix"
  "encoding/pem"
  "io/ioutil"
  "math/big"
  "time"
)
```

Since we have three keys and certificates to generate, we're going to wrap this process up into a single function, `generateKeyAndCert`. This function takes in a subject name and the certificate and key of a certificate authority. We can use the same function for our certificate authority, and in that case, nil can be passed for the `signer` and `signerkey`.

```
func generateKeyAndCert(
  name string,
  signer *x509.Certificate,
  signerkey *rsa.PrivateKey,
) (
  *rsa.PrivateKey,
  *x509.Certificate,
) {
```

`generateKeyAndCert`'s body has four parts to it. First, we have to generate the private key/public key pair. As mentioned, a key is a set of cryptographic numbers, in this case represented as an `rsa.PrivateKey` [5] struct. The inputs to it are limited—a random number source, which we're using as the default, and a key length. Later, we'll be using one of the fields of the `key`, the paired `PublicKey`, to generate the certificate.

```
key, _ := rsa.GenerateKey(rand.Reader, 2048)
```

Second, we must generate a template x509.Certificate [6]. It might be a bit confusing, but the template is of type `x509.Certificate`, which is the same type that we'll receive at the end. The template is used by the standard library function to generate certificates for where to source all of the information that we'll need. There are a few required fields: SerialNumber (unique distinguisher), Subject (which is where we're going to push CommonName), NotBefore/NotAfter (which determine the lifetime of this certificate), and KeyUsage (the intended purpose of this certificate).

```
template := &x509.Certificate{
    SerialNumber: big.NewInt(1),
    Subject:      pkix.Name{CommonName: name},
    NotBefore:    time.Now().Truncate(24 * time.Hour),
    NotAfter:     time.Now().Truncate(24 * time.Hour).
      Add(365 * 24 * time.Hour),
    KeyUsage:     x509.KeyUsageKeyEncipherment |
      x509.KeyUsageDigitalSignature,
  }
```

Since this is a dual purpose function, we might be generating a certificate authority. In those cases, we need to set a couple of additional fields: `IsCA` must be true, and `KeyUsage` must be extended for this additional purpose. Additionally, we also need to set our currently nil-valued `signer` and `signerkeys`. As a root CA, we're going to set these to themselves.

```
if signer == nil || signerkey == nil {
  template.IsCA = true
  template.KeyUsage |= x509.KeyUsageCertSign
  signer = template
  signerkey = key
}
```

Next, we're ready to generate our certificate using the standard library function: `x509.CreateCertificate`. In addition to the default source for random numbers, it uses the template, the signer, our newly generated public key, and the signer's private key to create a binary blob representing the signed certificate.

```
der, _ := x509.CreateCertificate(
  rand.Reader,
  template,
  signer,
  &key.PublicKey,
  signerkey,
)
```

And, finally, we need to make this binary blob useful. This binary blob is DER encoded [7]. While this is useful to functions handling binary data, we want to force the structure and type consistency of the language and turn this into a full certificate datatype.

```
cert, _ := x509.ParseCertificate(der)
```

We now have the actual key and cert, so we can pass those back:

```
  return key, cert
}
```

Once we generate these, we'll need to be able to save them to disk to be used by our client and server utilities. The standard format for handling key and certificate files is called privacy-enhanced electronic mail (PEM; https://en.wikipedia.org/wiki/Privacy-enhanced_Electronic_Mail) encoding. The PEM is an ASCII form generated from the binary data, held as an array of bytes in Go, of the keys and certificates. Extracting the binary data is slightly different for keys and certificates, but both need to be converted over to this PEM format, and there are standard library functions available for this. Once we get the PEM form in memory, we can dump this to disk using the convenient `ioutil.WriteFile` function.

```
func saveKeyAndCert(
  prefix string,
  key *rsa.PrivateKey,
  cert *x509.Certificate,
) {
  keyBytes := x509.MarshalPKCS1PrivateKey(key)
  keyPem := pem.EncodeToMemory(
    &pem.Block{Type: "RSA PRIVATE KEY", Bytes: keyBytes})
```

```
  ioutil.WriteFile(prefix+".key", keyPem, 0444)
  certPem := pem.EncodeToMemory(
    &pem.Block{Type: "CERTIFICATE", Bytes: cert.Raw})
  ioutil.WriteFile(prefix+".crt", certPem, 0444)
}
```

With our wrapping and save-to-disk functions, we can put together our main function. Note the use of the CA keys and certificates to generate the actual end keys and certificates:

```
func main() {
  caKey, caCert := generateKeyAndCert(
    "glss Root CA",
    nil, nil)
  saveKeyAndCert(
    "certs/CA", caKey, caCert)
  serverKey, serverCert := generateKeyAndCert(
    "localhost",
    caCert, caKey)
  saveKeyAndCert(
    "certs/server", serverKey, serverCert)
  clientKey, clientCert := generateKeyAndCert(
    "glss Client A",
    caCert, caKey)
  saveKeyAndCert(
    "certs/client", clientKey, clientCert)
}
```

With this utility written, we're now ready to execute it. Since this is a one-time tool for this exercise, let's just run it:

```
$ go run certs/generate_certs.go
```

You should see several certificate and key files in the certs directory:

```
CA.crt
CA.key
client.crt
client.key
server.crt
server.key
```

Now that we have all of the certificates, we can proceed into encryption and authenticating our communications.

Server Changes

Part of what makes this powerful in Go is that we won't have to change much code to wrap the calls in TLS. We can change some pieces of the setup to include TLS setup, and the rest of the application is unchanged. Part of this is because we're able to swap out different types that satisfy the same Go interface—in particular `net.Conn` on the server side.

Start by copying the original server and client utilities from the gls package.

```
$ cp \
  $GOPATH/src/github.com/kelseyhightower/gls/server/main
  go \
  ./server/main.go
```

We're going to start by updating the import list. We have to add specific crypto libraries that we're going to be using as well as add back in the reference to the original gls library.

```
import (
  ...
  "crypto/tls"
  "crypto/x509"
  "io/ioutil"

  "github.com/kelseyhightower/gls"
)
```

Next, we need to initialize the TLS settings for the server. This involves three parts: loading the server key pair, loading the certificate authority certificate to verify against, and then using those to set the TLS configuration. To load the key pair, we will use the `tls.LoadX509KeyPair` function.

```
func main() {
  cert, err := tls.LoadX509KeyPair("certs/server.crt",
    "certs/server.key")
  if err != nil {
    log.Println(err)
    return
  }
```

TLS connections are verified against a `CertPool`, which is a list of certificate authorities used to check for signatures. In the case of verifying against a wide range of certificate authorities, like a browser would do, you can keep adding certificate authorities to the pool. In this case, we only have our internal certificate, so we can add only it to the `CertPool`. Since the certificate authority is a bare certificate (i.e., it doesn't include a private key), we can't use `tls.LoadX509KeyPair` to get the certificate; we have to load it separately and then add it bare to the `CertPool`.

```
caCert, err := ioutil.ReadFile("certs/CA.crt")
if err != nil {
  log.Fatal(err)
}
caCertPool := x509.NewCertPool()
caCertPool.AppendCertsFromPEM(caCert)
```

Now with the server certificate and the certificate authority, we can set the TLS configuration. In addition to the certificates, we want to require that we authenticate the client using TLS.

```
config := &tls.Config{
  Certificates: []tls.Certificate{cer},
  ClientCAs:    caCertPool,
  ClientAuth:   tls.RequireAndVerifyClientCert,
}
```

As we'll see in the client, Go has a convenience function inside of TLS for connections; for the server, `tls.Listen` can replace `net.Listen`. However, we need to be able to access the peer information, so we have to set up TLS directly and can't use this. Luckily, this only requires a couple of lines (plus error checking): one to create the TLS connection object, and one to perform the TLS handshake.

```
for {
  conn, err := l.Accept()
  if err != nil {
    log.Println(err)
  }
  tlsconn := tls.Server(conn, config)
  err = tlsconn.Handshake()
  if err != nil {
    log.Fatal(err)
  }
```

Once the TLS handshake is successful, we can inspect the connection for the client information and confirm it is correct. Note that we may get multiple certificates on the connection. A client may send its full certificate chain or a partial certificate chain over the connection if it needs to connect intermediate certificates to a root. The key here is that first certificate (index 0) will be the leaf certificate for *this* client, so it will be the one we check against. In our particular case, we're going to compare the subject's `CommonName`, but other situations could use other fields of the certificate.

```
  tlsclient := tlsconn.ConnectionState().PeerCertificates[0]
  if tlsclient.Subject.CommonName != "glss Client A" {
    log.Fatal("Invalid client")
  }
  log.Printf("user=\"%s\" connect",
    tlsclient.Subject.CommonName)
```

Now that we've verified the certificate chain (via the `ClientAuth` setting on `tls.Config`) and checked that the `CommonName` is correct, we can proceed with the `net/rpc` call. Special Note: since this is providing a wrapper layer, we're going to insert this between the `Accepted` connection and `rpc.ServConn`. `Accept` and `tls.Server` both return `net.Conn`, and `rpc.ServConn` takes in a `net.Conn`. `rpc.ServConn` isn't aware that the data is being encrypted underneath it.

```
  rpc.ServConn(tlsconn)
  conn.Close()
}
```

You can confirm everything by building the server the same as before:

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

At this point, we've added TLS to the server side without having to change any of the underlying `net/rpc` items. Now we need to do the same on the client side.

Client Changes

The client changes are the same as on the server side except that we don't have to check anything additional on the certificate's `CommonName`—this is handled by default when TLS authenticates servers. As before, start by copying the existing gls client over to our new working directory:

```
$ cp \
  $GOPATH/src/github.com/kelseyhightower/gls/client/main.
go \
  ./client/main.go
```

Then update the imports the same as before.

```
import (
  ...
  "crypto/tls"
  "crypto/x509"
  "io/ioutil"

  "github.com/kelseyhightower/gls"
)
```

Next, load the client certificate and private key, the certificate authority certificate, and configure TLS. The main differences are to flip from authentication of the clients to authentication of the server in the `tls.Config`: we're not specifying `ClientAuth`, since that's a server side optional setting, and we're specifying the `RootCAs` instead of `ClientCAs` to indicate that we're connecting out and authenticating the server instead of being connected to and authenticating the client.

```
cert, err := tls.LoadX509KeyPair("certs/client.crt",
  "certs/client.key")
if err != nil {
  log.Fatal(err)
}
caCert, err := ioutil.ReadFile("certs/CA.crt")
if err != nil {
  log.Fatal(err)
}
caCertPool := x509.NewCertPool()
caCertPool.AppendCertsFromPEM(caCert)
conf := &tls.Config{
  Certificates: []tls.Certificate{cert},
  RootCAs:      caCertPool,
}
```

Next, we connect to the server with the convenience function `tls.Dial`, and pass the returned `net.Conn` to `rpc.NewClient`. In the same way as encryption and authentication are transparent on the server, this is transparent to `net/rpc` on the client.

```
conn, err := tls.Dial("tcp", "localhost:8080", conf)
if err != nil {
  log.Fatal(err)
}
client := rpc.NewClient(conn)
```

Build the client, and you should now have a fully encrypted and authenticated gls client:

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

Start up the server and, separately in another terminal, start up the client:

```
$ ./glssd
# In another terminal
$ ./glss ~
```

Conclusion

At the end of this, we have protected the gls connection with mutual TLS authentication. In addition, we've relied on the power of the golang interface to only make minimal changes to the original program to enable secure communication.

References

[1] K. Hightower, "Modern System Administration with Go and Remote Procedure Calls (RPC)," *;login:*, vol. 41, no. 1 (Spring 2016), pp. 63–67: https://www.usenix.org/publications/login/spring2016/hightower

[2] R. Perlman, "Blockchain: Hype or Hope?" *;login:*, vol. 42, no. 2 (Summer 2017): https://www.usenix.org/publications/login/summer2017/perlman.

[3] Source code for glss: https://github.com/cmceniry/login-glss.

[4] Let's Encrypt: https://letsencrypt.org.

[5] Go Doc on PrivateKey: https://godoc.org/crypto/rsa#PrivateKey.

[6] Go Doc on x509: https://godoc.org/crypto/x509#Certificate.

[7] DER encoding: https://en.wikipedia.org/wiki/X.690#DER_encoding.

Flipping Out in Computer Science

MARGO SELTZER

Margo Seltzer is the Herchel Smith Professor of Computer Science and the Faculty Director for the Center for Research on Computation and Society in Harvard's John A. Paulson School of Engineering and Applied Sciences. Her research interests are in systems, construed quite broadly: systems for capturing and accessing provenance, file systems, databases, transaction processing systems, storage and analysis of graph-structure data, new architectures for parallelizing execution, and systems that apply technology to problems in health care. She was a co-founder and CTO of Sleepycat Software, the makers of Berkeley DB, and is now an Architect at Oracle Corporation. She is a past President of the USENIX Board of Directors. She is recognized as an outstanding teacher and mentor, having received the Phi Beta Kappa teaching award in 1996, the Abrahmson Teaching Award in 1999, and the Capers and Marion McDonald Award for Excellence in Mentoring and Advising in 2010. Dr. Seltzer received an AB degree in applied mathematics from Harvard/Radcliffe College in 1983 and a PhD in computer science from the University of California, Berkeley, in 1992. margo@eecs.harvard.edu

For a while, every conversation about education seemed to lead to the term MOOC (massive open online course). The hype around such courses seems to have died down to some extent, but MOOCs still exist and are largely good things, even if they have not fulfilled the promise of educating the world. However, there has been an unanticipated side effect to the (forgive me here) MOOC-ification of courses. We suddenly find ourselves in possession of some really high-quality teaching materials. What else might we do with such assets? I'd like to make the point that the wealth of online material opens up the possibility that those of us in the education business can undertake experiments in education that lead to deeper learning. In this article, I'll focus on the flipped classroom.

In 2013, I began revising all my undergraduate courses so that I could teach them in a flipped style (my graduate courses are typically research seminars, so in some sense, they are already flipped). But what is flipping? The high-level idea is that rather than spending class time absorbing information and then practicing use of the information at home, we flip those two activities around. Students use prepared materials at home for first exposure to new concepts and then come to class and work in small groups to practice applying those concepts.

I had been intrigued by the idea of flipping for a long time but hadn't quite figured out how to apply it to my own courses. My problem sets are large monolithic projects, not something on which one can make meaningful progress in a class period. So while I could easily imagine preparing materials for them to review at home, what would I have them do in class?

By pondering that question, I realized that one of the biggest challenges students face in programming courses is connecting new concepts to the programming tasks we give them. Maybe I could use in-class time to more effectively connect conceptual material to programming pragmatics, so students would not have to struggle with the question of how to get started.

My first experience flipping a course was with my (insanely time-consuming) operating systems course. Students report spending 30 hours per week completing the long but rewarding problem sets—students start with a simple operating system kernel and build user-level processes, a virtual memory system, and a journaling file system. I blogged my first experience flipping it here: http://mis-misinformation.blogspot.com/2013/08/an-index-to-my-flipping-blog-postings.html.

I ended up using three different styles of in-class exercises: gaining familiarity with the course software, completing problems that demonstrate mastery of the material presented, and engaging with open-ended design problems. I'll give short examples of each of these approaches.

Infrastructure

Traditionally, the first assignment in the course includes instructions on how students acquire the course software, install a virtual machine, configure and build a kernel, attach the debugger to a running kernel, etc. Small glitches in this process can result in students wasting a lot of time without learning much. Instead, I had them get their hypervisor licenses and install the course VM as pre-class work and then used class time to let them config and build their first kernel and complete some debugging exercises.

There were a number of positive outcomes from this structure. First, if students encountered any problems, we fixed them within a few minutes rather than having students beat their heads against the wall for hours. Second, it's actually pretty exciting to build your first kernel and watch it run. We got to all experience that together, so by the end of class there was a shared sense of accomplishment. Third, while we always encourage students to read code (and we assign them code-reading questions), as we wandered around the room interacting with the groups, we could ask questions that required that they look at code and could then gently walk them through how to approach a new code base.

Problem Solving with Virtual Memory

It's pretty easy to assume that once you've explained the four-level page table structure of the x86, students would then understand how address translation works. You would, however, be wrong.

Historically, when I taught VM, I would have the class "play MMU" and perform address translation one step at a time, having each student contribute something. This wasn't bad, but a lot of things fall through the cracks. With flipping, after introducing students to the concept of virtual memory and the x86 VM system, it was easy to create short problems that let small groups of students "play MMU" and translate addresses, draw page tables, populate the page tables, deduce what page faults really are, experience a segmentation violation from the point of the MMU, etc. Instead of each student contributing a tiny piece (and sleeping through the rest of the discussion), every student was exposed to every operation; by the end of class it was pretty clear that there was a much more uniform and deep understanding of what was going on.

Design Exercises

As the semester progresses in my operating systems classes, more of the conceptual material involves helping students develop the intuition and skills to design software and make tradeoffs. Prior to flipping, I would always present alternatives and let the class come up with the advantages and disadvantages of the different approaches. Of course, the five students who knew exactly what was going on were the ones who would pretty much answer all the questions no matter how much I cajoled the rest of the class and tried not to call on the frequent contributors. I converted these to small design exercises, requiring groups of two, three, or four students to assess tradeoffs, and then we'd come together as a class to compare answers.

As a result, everyone felt they could contribute. Even if they hadn't been entirely comfortable with the material, after discussing it with their peers for 10 or 15 minutes, they usually could effectively compare their conclusions with those of other groups. I've done a large variety of different activities around this theme ranging from peripatetic design reviews (when the class was small), to design debates, to collaborative analyses. One former student reports that she uses the skills learned in these exercises every day in her job.

I'm completely hooked on flipping at this point. I distilled the advantages I see in the approach into the following 10 bullet points:

1. It's good for an old dog to learn new tricks. This is really about making sure your teaching doesn't get stale. It's way too easy to keep teaching the same thing over and over again. Whether you use new pedagogy, new technological breakthroughs, or just good self-discipline, it's important to keep classes fresh.
2. Flipping lets me spend time with those students for whom the material is most challenging. This is so obvious in retrospect, but so exhilarating in practice. I have always run a relatively interactive class, but for the most part, the students who ask and answer questions in class are the ones who need you least—they are typically the most confident and are not struggling to understand the material. The silent ones, meanwhile, are frequently struggling, and the time spent helping these students in small groups during class time is incredibly useful.
3. Learning takes place by doing, not by listening to me. There are a lot of different styles of hands-on learning, but I think this point cannot be emphasized enough. Learning is not just the process of transferring information from the teacher to students; learning is about gaining new information and knowing how to use it, and the latter requires practice.
4. Teaching assistant engagement is critical. We call our teaching assistants "teaching fellows," or TFs for short. Flipping effectively requires a good staff that is comfortable engaging with students, walking them through problems, and posing the right questions. I am extraordinarily fortunate to have a truly amazing and dedicated teaching staff.
5. It takes a lot of effort to come up with effective in-class work. It's important that the in-class exercises or problems relate both to the concepts the students are learning and to the homework or problem sets they will be doing. Designing these exercises so they can be completed in the time allotted and add real value to the course is demanding.

6. Pre-class Web forms are AWESOME. They allow me to engage with students in an entirely different way and to gather lots of interesting data. This is perhaps the best surprise of all! I used Google Forms to have students submit answers to the pre-class questions. This created a mechanism I could use to obtain all sorts of useful information, including how things were going in partnerships, how much time people were spending on various parts of the assignment, what was working for students, what wasn't working, etc. Once you have students regularly filling out forms, they will answer anything you put there, and you can use that to make the class better. Score!
7. My operating systems course, CS161, is even more time intensive than I thought. I had been saying 20 hours per week for decades; when the going gets rough, students were regularly reporting 30-hour weeks. Oops.
8. It would be useful to help students learn what it really means to design something. Software design is really hard! We spend a lot of time in class doing small group design exercises—I could imagine developing an entire course around this idea.
9. Flipping is a great equalizer when students enter with different experience levels or exposure to different topics. It's relatively easy to provide supplementary material as pre-class work, so that students who have gaps in their background can catch up.
10. Fully integrated and coordinated materials take real effort but pay off tremendously. This should be a no-brainer, but thinking deeply about the relationship between the videos I prepared, the exercises we completed in class, and the problem sets was time well spent.

For Good Measure
When Opinion Is Data

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

Obvious to all, the sea of data is rising. It's a remarkable thing really. Even if all you can remember is 10 years back, the comparison of "then" with "now" is pretty startling. No, that does not qualify as news, but to reparse Orwell's "Who controls the past, controls the future: who controls the present, controls the past," the data "we" collect now is what will soon enough become the past for a data-driven world. If that data past comes to exert a force in some sort of proportion to its volume, is there, or will there be, any room for mere human opinion?

Cybersecurity has long had a measurement problem. Progress has certainly been made, both in the pages of this publication and elsewhere. Defenses now include mass data collection and tools whose main job is to reduce data volume to something that is straightforwardly actionable. In the Orwell sense, the algorithms that collect and reduce the instrumentation data are coming to control if not the present itself then our understanding of the present. In due course, the "actionable" becomes the automatically acted upon, that is to say that algorithms are trusted to do what we seem unable to do—to protect us from other algorithms. Such is progress.

Yet the nuance here is that the algorithms are, by and large, uninterrogatable—they cannot be meaningfully asked why they made such and such a decision. The outcome of action, not the reason for action, becomes the only check and balance that we humans have at all. This may be a tradeoff that is not just inevitable but welcome, welcome in the sense of freeing front-line cybersecurity staff from having to juggle a million balls all at once. At the same time, if you/we cannot examine the reasoning behind an automatic action but only react to the outcome of it, what then do we know about the present? What kind of past will the accumulating data create? Behaviorally oriented cybersecurity is entirely crafted along these lines, the line of learning enough about the recent past to be able to tell that the present is diverging from that past and, ipso facto, algorithmically control the future. What then is the role of the human in the loop?

The Index of Cyber Security (ICS) was created six years and a little more ago on the premise that we didn't know enough about the details of cybersecurity to make prediction and planning really possible—that "the present" was (is) a bit of a miasma and, as such, the best and only trustable prediction of the future was to be found in the pooled opinions of front-line cybersecurity practitioners. As with the oft-noted "wisdom of crowds," ours was not a search for the single smartest oracle but rather a pooling of opinion from a body of experts whose views were tempered by the heat of daily practice. Speaking for myself and my colleague in this project, we think that the need for pooled expert opinion is greater than ever, both between practitioners (as with the ICS) and inside each firm that is itself large or connected enough to be a constant target.

A developed muscle that is not exercised will atrophy. A developed skill that is not exercised will atrophy. If we humans are to remain the ultimate decision makers regarding our fate,

then our ability to form strong opinions must not be left unexercised, it must not be left to atrophy. The desire for automaticity runs toward setting the stage for an atrophy of some skills; choosing what to let go may require the greatest of wisdom. Perhaps, then, the state of our wisdom is worth close attention. To illustrate that point, consider this ICS question:

Your organization is likely more reliant on the cloud than you think. According to Symantec's Internet Security Threat Report, the average enterprise organization was using 928 cloud apps, up from 841 earlier in the year. However, most CIOs think their organization only uses around 30 or 40 cloud apps. Reliance on the cloud goes beyond the traditional infrastructure hosting arrangement. Unknown to IT, the "business" will often sign up for SAAS services on the cloud where data (or metadata at least) gets out on the cloud.

What is your assessment of your security organization's handle on cloud engagement:

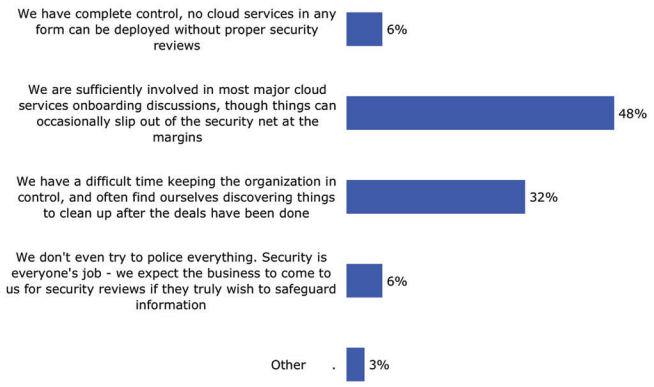


Figure 1

That question above and its answers by a pool of front-line cybersecurity people is illustrative—both of the spread of opinion and its logic. That we can ask practitioners such a question is the interrogatability part. That some entities centralize control while others delegate responsibility is no real surprise but is still worth noting insofar as it says pretty clearly that no single "right" answer has come along.

Let's try another:

After years of study, we still do not seem to be able to agree on the question of vulnerabilities and, in particular, matters of their discovery, use, retention, and disclosure. Policy constraints vary across countries like night and day. These are strategic issues or, should we say, Strategic Issues that fully prove that cybersecurity and the future of humanity are conjoined now. Allowing for ambiguity, which of these directions should free-world governments favor:

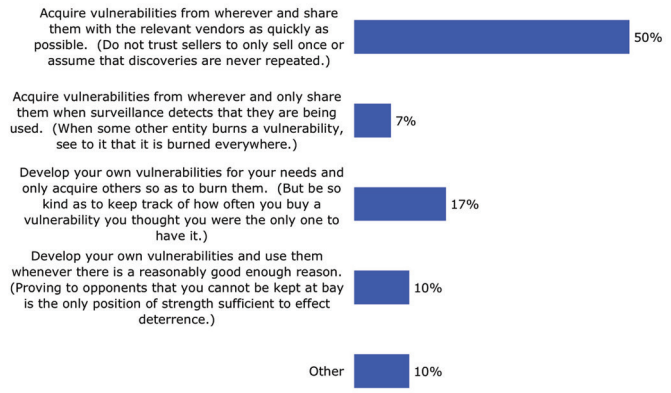


Figure 2

As with the first example, the spread of opinion is valuable in and of itself. Does not the preponderance of the first option, to acquire vulnerabilities from wherever and share them with the relevant vendors post-haste reflect a strong prediction on the part of the respondents about what they expect the vulnerability situation to be in future? Would an algorithm fed by a sensor network come to the same conclusion?

Let's try a third:

Newly discovered vulnerabilities create workload for defenders that is immediate—in the form of security updates and patches to apply—and workload that is deferred—as everything built and deployed from that point on has to be inoculated against the continuously accretive database of known weaknesses. Yet this work cannot be perfectly sufficient, as Mirai has shown; the capabilities of the attackers can increase even if the defense is doing everything right for their organization.

How have you been seeing your workload fluctuate over the past year:

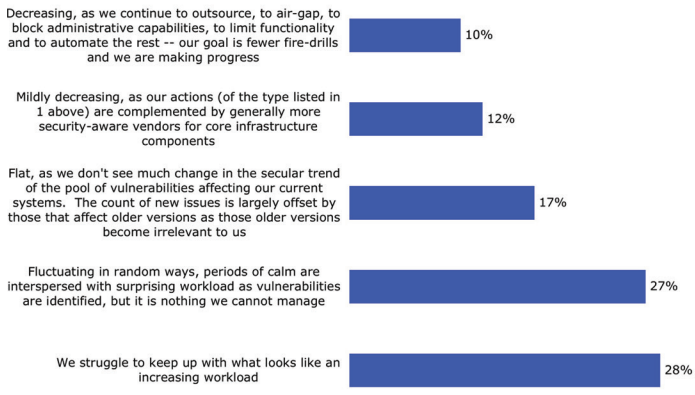


Figure 3

Here, the respondents' opinions are certainly predictive about the future of their own practice, and, from that, one can make broader statements about the cybersecurity situation in general. This human judgment seems better than any sensor network-driven machine learning could be expected to deliver. Of course, sometimes it is not a question of data but rather of the handling of data, such as this fourth example:

Information sharing with the government, even after large incidents, is an activity fraught with anxiety and stress. Differential reporting by the victim targets means the data that public authorities have is not useful for rational planning. Some target entities will report; some will not. Has the time come to have an escalation rule for sharing of information about attacks?

We do this with different rationales in some contexts, such as when we require prompt and detailed attack information from defense contractors to Pentagon authorities, when state laws force disclosure if a customer's credit card or other personal information is exposed, and when the SEC requires the announcement of security breaches that materially impair corporate operations. Has the time come for a mandatory reporting regime for all events that are above some threshold of severity?

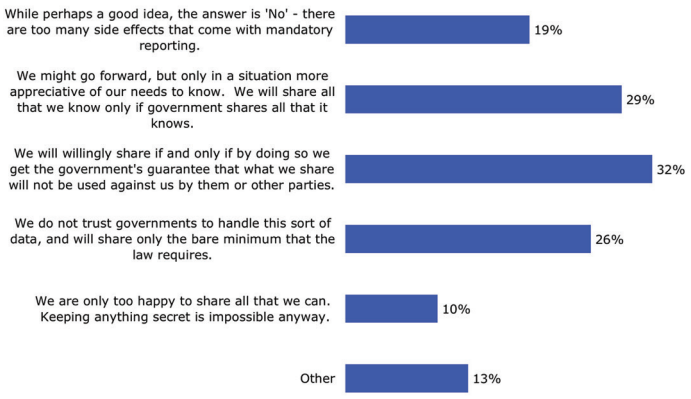


Figure 4

Collectively, these questions illustrate what shared, expert opinion can mean, and it seems unlikely that algorithms would take over these areas of informed choice, but 10 years ago we would not have guessed what algorithms have taken over today either. While we can (and will) ask the ICS respondents about the role of automation in the near-term future, our imagination may not be up to the task of asking the right questions. By all means, make suggestions as to what questions we should ask. If you are, yourself, a front-line security practitioner, then please consider becoming one of our respondents (it will cost you 10 minutes a month, and you will see a lot of analysis that we reserve for our respondents—though we'll happily provide a sample to help you make a decision).

Nevertheless, at the end of the day, the biggest question is whether a human in the loop is a failsafe or a liability. We favor the "failsafe" view, but to keep and maintain that a human in the loop is a failsafe, they have to actually be in the loop. Being an observer of algorithms that don't ask (permission) and don't tell (what it is they are doing) won't keep the practitioner in fighting trim. There's no such thing as a free lunch…

/dev/random
Offensive Computing

ROBERT G. FERRELL

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

Now that state-sponsored retaliatory computer operations are apparently a thing, this seems like a great time to jump on that self-driving full-auto bandwagon. But forget boring stuff like reconnoitering port scans and penetration probes prior to attacking; let's use the nuke option on those nefarious puppers from the get-go and move on with our lives, what do you say? I stock an entire arsenal of potent ordnance for taking down the cyber bad guys, be they corporate, governmental, or just private mercenaries with a yen for easy money.

Most of the "best defense is a strong offense" proposals I've seen involve fighting fire with fire. Tedious and predictable, my young apprentice. The way to fight fire is to bury it under a deluge of sloppy wet stuff. As a longtime purveyor of same, here are some of my suggested tactical instruments of vengeance, both offensive and defensive, along with the philosophical statement you'll be making with each. All are dedicated to the proposition that protecting one's information assets can also count as entertainment.

Chaos in the Middle: intercept network traffic heading to and from your enemy and attach random headers and payloads derived from Pinterest or /r/SubredditSimulator. Then sit back and watch their logs fill up.

Message it sends: Hr r yr lulz.

Matrix Honeypot: divert hostile traffic into a honeypot universe where all of the attackers' initial strategic goals seem to be met perfectly. Once they're hooked, create increasingly more complex and comprehensive layers of alternate reality until they no longer have any objective means by which to differentiate that virtual world from the real one. They will now be trapped forever. Not recommended for teams on a budget, as the necessary pecuniary outlay can approach infinity over time.

Message it sends: Take two blue pills and WhatsApp me in the morning.

Grade School Playground: a bot that replies to every email, text message, or other enemy communication with, "I know you are, but what am I?" Attach optional raspberry.mp3, nyah-nyah.mp3 to complete the experience.

Message it sends: It's always recess somewhere.

Reverse Ransomware: threaten to break the enemy's encryption with your supercomputer and supply the key to their victims for free unless the crooks pay half the ransoms to you. Not so much an attack as a business model.

Message it sends: Thank you for your patronage.

Mirror, Mirror: automatically reflect every packet sent by an attacker in FILO order. Essentially a variation on the *Grade School Playground* method (cf.). Economical because it only requires a modified network appliance. Will not make you popular with your upstream neighbors, though, and renders the node pretty much useless for getting any real work done.

Message it sends: We're sorry, the number you have reached is not in service.

Blast Phishing: Perform both passive and active phishing-based reconnaissance on the target to establish patterns, methods, and locations. Once all the necessary intelligence has been gathered, launch absolutely everything in your malware database simultaneously along all mapped hostile vectors. Messy, but effective if you don't want any survivors. A healthy chunk of bandwidth is a must here.

Message it sends: Today is a good day to die().

Hydra Hail: Invest in sufficient infrastructure to spawn virtually endless numbers of cloned virtual machines on isolated VLANs. Every time the enemy attacks, take the affected virtual presence down instantly and plop another clone in its place. Rinse and repeat ad infinitum until the attacker gives up in frustration. Have a beer to celebrate your victory.

Message it sends: Sticks and stones may break my bones, but I have a heck of a lot of bones.

Catatonia: Trace the IP address of the attacker and forward every known cat video to it, effectively purr-alyzing the hostile network under a dense blanket of furry cuteness.

Message it sends: Get some of this meow up in your grill, evildoer.

And finally,

Utter Acquiescence: powers down the entire network and reverts everyone to slide rules and typewriters. See also RFC 1149.

Message it sends: We have a constitutionally mandated Postal Service for a reason.

If you don't know how to work a slide rule, I'll be happy to teach you, although admittedly I mostly used mine as a straightedge for drawing castles on my notebooks. I still have my Pickett N902-ES from high school, along with my grad school-era Brother Professional CX-90 daisy wheel electric typewriter. They've never been compromised, although I did misplace my italic daisy wheel once.

As for me, I do all of my mission-critical computing on my trusty Osborne 1 these days and thus I'm not too vulnerable to attack unless you're into crafting exploits for CP/M 2.2. And mailing them to me on a 5.25" disk.

Book Reviews

MARK LAMOURINE

REST API Design Rulebook

Mark Massé
O'Reilly Media Inc., 2012, 94 pages
ISBN 978-1-449-31050-9

You would be right in assuming that any book with the word "rulebook" in the title would express an opinion. Massé certainly doesn't hold back, but that seems to be a trait of REST advocates in general.

At under 100 pages, Massé's book packs quite a lot into a little space. It is really presented as a list of one-line rule statements with a matching brief explanation. Each is meant to address one of the common questions raised when designing a REST protocol.

The first three sections treat the interactions between the client and the server, detailing how each uses the HTTP protocol features to communicate and interpret the intent of the other. The fourth section describes how to add metadata that allows the self-discovery that is characteristic of REST protocols.

I was struck by how little the rules had to do with the formatting of the content. The only rules that deal directly with content are those that state that the payload must use a standard structured data format such as JSON or XML. The rest of the rules describe how to make use of the simple CRUD (Create, Read, Update, Delete) operations that HTTP offers to define the more complex interactions that a rich application protocol needs.

Massé notes that the contents of these first four sections are based largely on consensus reached over time among the developer community. In the final two sections, he discusses rules for data representation and for client-side concerns like authentication and applications with multi-origin data sources. The wording of the rules here changes from "must" to "should." Massé indicates that these are his answers to the questions that remain open, based on his experience.

This book was written in 2011, more than a decade after the publication of Roy Fielding's PhD dissertation in 2000. Since then REST has come to be the preeminent model for client-server communications, replacing proprietary binary models and earlier Web standards like SOAP and XML-RPC. While many services claim to conform to the REST conventions, a close read of this book will show that few really meet the full criteria.

When thinking about REST, people often focus on representing the payload content using a structured data format. Many forget that a major tenet of REST is that the relationships between the different data objects must be included in the query responses. Links and relationships must be discoverable by the client without the need to code assumptions into the client-side logic. Defining and presenting these relationships in the metadata of a REST response requires a lot of thought and work on the part of the server writer. Many applications that claim to be RESTful take shortcuts on the protocol design, coding the relationships into the client.

Massé correctly focuses on how to define and present these relationships. He understands that simply representing the content as structured data is the easy part. He gives very little space to how to write the code, though he does include a simple app example in the final chapter.

In the end it may not matter if developers strictly adhere to the REST guidelines, so long as the code works, but I suspect much code could be improved after a few minutes spent with the *Rulebook*.

CoreOS in Action

Matt Bailey
Manning Publications Inc., 2017, 178 pages
ISBN 978-1-61729-374-0

In the grand migration to software containers, there is a largely overlooked component that I think deserves more attention: the container host. The conventional OS distribution design is based on old assumptions about how applications work and how they will use OS underneath. Container hosts are designed with the containerized application in mind: a minimal Linux install on a read-only file system.

CoreOS began as a kind of customizable single-application host distribution. Originally, CoreOS was designed for building an image for each service as if it were an embedded system or unikernel. The build system is based on Gentoo, and the code base began as a variant of ChromeOS.

CoreOS itself didn't get much attention until the advent of Docker and the growth of containers. Creating custom images with embedded applications required skill and specialized knowledge, and there was little incentive for developers to focus on those skills. Docker changed that by creating an easy, consistent model for creating single-purpose images, with the advantage of portability and a distribution infrastructure, the container registry. Once CoreOS included the Docker runtime, it became an ideal place to create distributed container services.

Bailey packs a lot of information and many examples into a slim book. In some ways this reflects what CoreOS is good at: minimizing complexity (at least in some realms). The whole idea of container hosts is that you don't administer them in the same way that you would a conventional host: you can't install packages. Persistent storage must come from a shared resource. This doesn't mean that you don't need to manage them or that your applications will magically appear and work. For a sysadmin, using container hosts means unlearning and relearning a lot.

The examples include short bits to create the container images, to deploy CoreOS itself, and to configure the services that bind the individual hosts into a cluster. I wish Bailey had spent a little more time on the theory and internals of these services: etcd, fleet, flannel. The code fragments and the callouts that explain these services are clear and well presented, but a bit more on how they work might make these samples easier to adapt to the reader's own purposes.

Bailey asks a lot of his readers because adopting CoreOS requires thinking about applications in new ways. Only the first third of the book is given to actually installing the OS and configuring the clustering services. In the second section, Bailey shows how to build applications that will be suited to the container environment. He does address legacy applications, but leaves it implicit that they must be decomposed and migrated, not "forklifted" into containers.

In the final section, Bailey talks about aspects of using CoreOS in production. He shows a CoreOS deployment in AWS using Cloud formation to describe the configuration and topology. He closes with a brief discussion of what might be a taboo subject: a container designed to allow the sysadmin access to the tools they are used to having on a conventional host.

Container hosts are still in the shadows of the containers themselves, but I think they should be given more light. *CoreOS in Action* shines a light on the foundation. This might even be a good path for introducing containers themselves.

SAVE THE DATES!

usenix

SRECON®

AMERICAS

SANTA CLARA, CA, USA

MARCH 27–29, 2018

ASIA/AUSTRALIA

SINGAPORE

JUNE 6–8, 2018

EUROPE/MIDDLE EAST/AFRICA

DUSSELDORF, GERMANY

AUGUST 29–31, 2018

srecon.usenix.org

2018 USENIX Annual Technical Conference

July 11–13, 2018 • Boston, MA, USA

Sponsored by USENIX, the Advanced Computing Systems Association

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Important Dates

- Complete paper submissions due: **Tuesday, February 6, 2018**
- Notification to authors: **Wednesday, April 18, 2018**
- Final papers due: **Thursday, May 31, 2018**

Conference Organizers

Program Co-Chairs

Haryadi Gunawi, *University of Chicago*
Benjamin Reed, *Facebook*

Program Committee

TBA

Overview

Authors are invited to submit original and innovative papers to the Refereed Papers Track of the 2018 USENIX Annual Technical Conference. We seek high-quality submissions that further the knowledge and understanding of modern computing systems with an emphasis on implementations and experimental results. We encourage papers that break new ground, present insightful results based on practical experience with computer systems, or are important, independent reproductions/refutations of the experimental results of prior work. USENIX ATC '18 has a broad scope, and specific areas of interest include (but are not limited to):

- Architectural interaction
- Big data infrastructure
- Cloud and edge computing
- Distributed and parallel systems
- Embedded systems
- Energy/power management
- File and storage systems
- Internet of Things
- Machine learning and systems interactions
- Mobile and wireless
- Networking (WAN, LAN, and datacenter) and network services
- Operating systems
- Reliability, availability, and scalability
- Security, privacy, and trust
- System and network management and troubleshooting
- Usage studies and workload characterization
- Virtualization

USENIX ATC '18 is especially interested in papers broadly focusing on practical techniques for building better software systems: ideas or approaches that provide practical solutions to significant issues facing practitioners. This includes all aspects of system development: techniques for developing systems software; analyzing programs and finding bugs; making systems more efficient, secure, and reliable; and deploying systems and auditing their security.

Reports of deployment experience and operations-oriented studies, as well as other work that studies software artifacts, introduces new data sets of practical interest, or impacts the implementation of software components in areas of active interest to the community are well-suited for the conference.

The conference seeks both long-format papers consisting of 11 pages and short-format papers of 5 pages, including footnotes, appendices, figures, and tables, but not including references. Short papers will be included in the proceedings and will be presented as normal but in sessions with slightly shorter time limits.

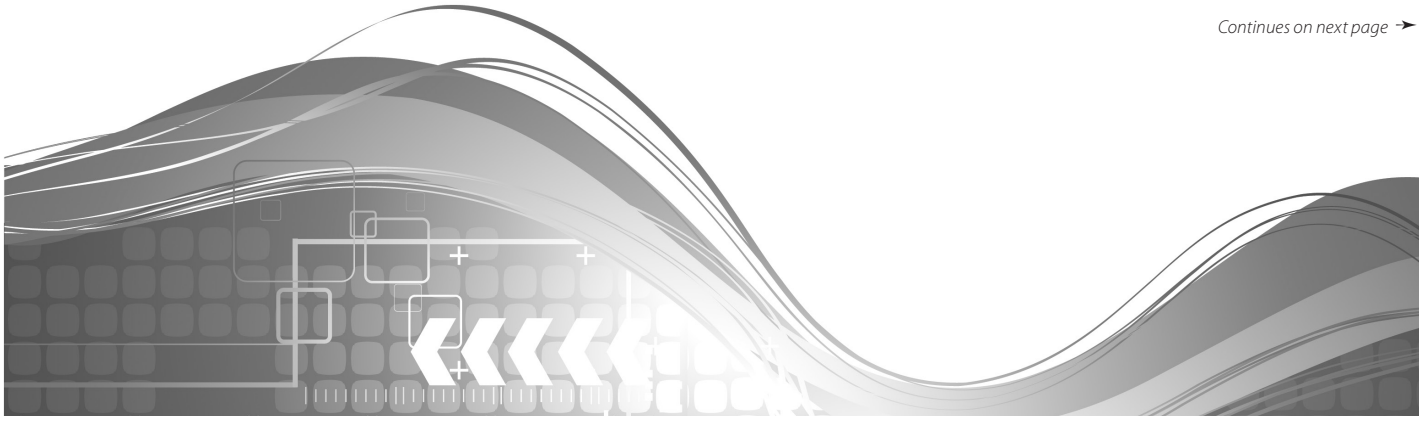
Best Paper Awards

Cash prizes will be awarded to the best papers at the conference. Please see the USENIX proceedings library for Best Paper winners from previous years at https://www.usenix.org/conferences/best-papers.

Best of the Rest Track

The USENIX Annual Technical Conference is the senior USENIX forum covering the full range of technical research in systems software. Over the past two decades, USENIX has added a range of more specialized conferences. ATC is proud of the content being published by its sibling USENIX conferences and will be bringing a track of encore presentations to ATC '18. This "Best of the Rest" track will allow attendees to sample the full range of systems software research in one forum, offering both novel ATC presentations and encore presentations from recent offerings of ATC's sibling conferences.

Continues on next page ➔



What to Submit

Authors are required to submit full papers by the paper submission deadline. *It is a hard deadline; no extensions will be given.* All submissions for USENIX ATC '18 will be electronic, in PDF format, via the Web submission form on the Call for Papers Web site, www.usenix.org/atc18/cfp.

USENIX ATC '18 will accept two types of papers:

Full papers: Submitted papers must be no longer than 11 single-spaced 8.5" x 11" pages, including figures and tables, but not including references. You may include any number of pages for references. Papers should be formatted in 2 columns, using 10-point type on 12-point leading, in a 6.5" x 9" text block. Figures and tables must be large enough to be legible when printed on 8.5" x 11" paper. Color may be used, but the paper should remain readable when printed in monochrome. The first page of the paper should include the paper title and author name(s); reviewing is single blind. Papers longer than 11 pages including appendices, but *excluding references,* or violating formatting specifications will not be reviewed. In a good paper, the authors will have:

- Addressed a significant problem
- Devised an interesting and practical solution or provided an important, independent, and experimental reproduction/refutation of prior solutions
- Clearly described what they have and have not implemented
- Demonstrated the benefits of their solution
- Articulated the advances beyond previous work
- Drawn appropriate conclusions

Short papers: Authors with a contribution for which a full paper is not appropriate may submit short papers of at most 5 pages, not including references, with the same formatting guidelines as full papers. You may include any number of pages for references. Examples of short paper contributions include:

- Original or unconventional ideas at a preliminary stage of development
- The presentation of interesting results that do not require a full-length paper, such as negative results or experimental validation
- Advocacy of a controversial position or fresh approach

For more details on the submission process and for templates to use with LaTeX and Word, authors should consult the detailed submission requirements at https://www.usenix.org/conference/atc18/requirements-authors. Specific questions about submissions may be sent to atc18chairs@usenix.org.

By default, all papers will be made available online to registered attendees before the conference. If your accepted paper should not be published prior to the event, please notify production@usenix.org. In any case, the papers will be available online to everyone beginning on the first day of the conference, July 11, 2018.

Papers accompanied by nondisclosure agreement forms will not be considered. Accepted submissions will be treated as confidential prior to publication on the USENIX ATC '18 Web site; rejected submissions will be permanently treated as confidential.

Simultaneous submission of the same work to multiple venues, submission of previously published work, or plagiarism constitutes dishonesty or fraud. USENIX, like other scientific and technical conferences and journals, prohibits these practices and may take action against authors who have committed them. See the USENIX Conference Submissions Policy at www.usenix.org/conferences/submissions-policy for details.

Note that the above does not preclude the submission of a regular full paper that overlaps with a previous short paper or workshop paper. However, any submission that derives from an earlier paper must provide a significant new contribution (for example, by providing a more complete evaluation), and must explicitly mention the contributions of the submission over the earlier paper. If you have questions, contact your program co-chairs, atc18chairs@usenix.org, or the USENIX office, submissionspolicy@usenix.org.

Authors will be notified of paper acceptance or rejection by April 21, 2018. Acceptance will typically be conditional, subject to shepherding by a program committee member.

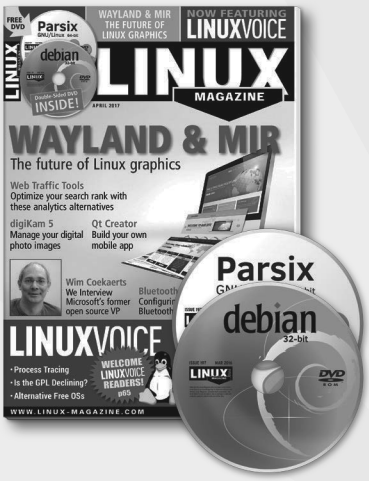
Poster Session

The poster session is an excellent forum to discuss ideas and get useful feedback from the community. Posters and demos for the poster session will be selected from all the full paper and short paper submissions by the poster session chair. If you do not want your submissions to be considered for the poster session, please specify on the submission Web site.

Program and Registration Information

Complete program and registration information will be available in April 2018 on the conference Web site.

RISK-FREE TRIAL!



3 issues + 3 DVDs for only $15

PRACTICAL. PROFESSIONAL. ELEGANT.

Enjoy a rich blend of tutorials, reviews, international news, and practical solutions for the technical reader.

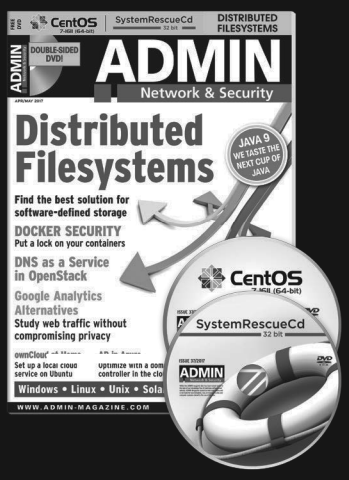
ORDER YOUR TRIAL NOW!

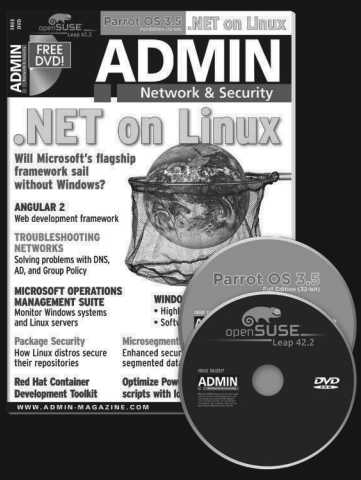
shop.linuxnewmedia.com

2 ISSUES ONLY $15.99

ALSO AVAILABLE

ADMIN: REAL SOLUTIONS FOR REAL NETWORKS





Technical solutions to the real-world problems you face every day.

Learn the latest techniques for better:

- network security
- performance tuning
- system management
- virtualization
- troubleshooting
- cloud computing

on Windows, Linux, Solaris, and popular varieties of Unix.

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

POSTMASTER
Send Address Changes to *;login:*
2560 Ninth Street, Suite 215
Berkeley, CA 94710

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES