usenix ;login:

FEBRUARY 2013 VOL. 38, NO. 1

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

FAST '13: 11th USENIX Conference on File and Storage Technologies
February 12–15, 2013, San Jose, CA, USA
www.usenix.org/conference/fast13

TaPP '13: 5th USENIX Workshop on the Theory and Practice of Provenance
April 2–3, 2013, Lombard, IL, USA
www.usenix.org/conference/tapp13

NSDI '13: 10th USENIX Symposium on Networked Systems Design and Implementation
April 3–5, 2013, Lombard, IL, USA
www.usenix.org/conference/nsdi13

HotOS XIV: 14th Workshop on Hot Topics in Operating Systems
May 13–15, 2013, Santa Ana Pueblo, NM, USA
www.usenix.org/conference/hotos13

2013 USENIX Federated Conferences Week
June 24–28, 2013, San Jose, CA, USA
www.usenix.org/conference/fcw13

USENIX ATC '13: 2013 USENIX Annual Technical Conference
June 26–28, 2013
www.usenix.org/conference/atc13

ICAC '13: 10th International Conference on Autonomic Computing
June 26–28, 2013
www.usenix.org/conference/icac13
Submissions due: March 4, 2013

HotPar '13: 5th Workshop on Hot Topics in Parallelism
June 24–25, 2013
www.usenix.org/conference/hotpar13
Submissions due: March 7, 2013

ESOS '13: 2013 Workshop on Embedded Self-Organizing Systems
June 25, 2013
www.usenix.org/conference/esos13
Submissions due: March 4, 2013

8th International Workshop on Feedback Computing
June 25, 2013
https://www.usenix.org/conference/feedback13
Submissions due March 29, 2013

HotCloud '13: 5th USENIX Workshop on Hot Topics in Cloud Computing
June 25–26, 2013
www.usenix.org/conference/hotcloud13
Submissions due: March 7, 2013

WiAC '13: 2013 USENIX Women in Advanced Computing Summit
June 27, 2013
www.usenix.org/conference/wiac13
Submissions due: March 13, 2013

HotStorage '13: 5th USENIX Workshop on Hot Topics in Storage and File Systems
June 27–28, 2013
www.usenix.org/conference/hotstorage13
Submissions due: March 11, 2013

HotSWUp '13: 5th Workshop on Hot Topics in Software Upgrades
June 28, 2013
www.usenix.org/conference/hotswup13
Submissions due: March 7, 2013

USENIX Security '13: 22nd USENIX Security Symposium
August 14–16, 2013, Washington, DC, USA
www.usenix.org/conference/sec13
Submissions due: February 21, 2013

Workshops Co-located with USENIX Security '13

EVT/WOTE '13: 2013 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections
August 12–13, 2013

CSET '13: 6th Workshop on Cyber Security Experimentation and Test
August 12, 2013
www.usenix.org/conference/cset13
Submissions due: April 25, 2013

HealthTech '13: 2013 USENIX Workshop on Health Information Technologies
Safety, Security, Privacy, and Interoperability of Health Information Technologies
August 12, 2013

LEET '13: 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats
August 12, 2013

FOCI '13: 3rd USENIX Workshop on Free and Open Communications on the Internet
August 13, 2013

HotSec '13: 2013 USENIX Summit on Hot Topics in Security
August 13, 2013

WOOT '13: 7th USENIX Workshop on Offensive Technologies
August 13, 2013

LISA '13: 27th Large Installation System Administration Conference
November 3–8, 2013, Washington, D.C., USA

EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Rikki Endsley
rikki@usenix.org

COPY EDITOR
Steve Gilmartin
proofshop@usenix.org

PRODUCTION
Arnold Gatilao
Casey Henderson
Michele Nelson

TYPESETTER
Star Type
startype@comcast.net

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

www.usenix.org

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

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

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

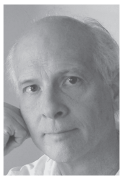
usenix ;login:

FEBRUARY 2013, VOL. 38, NO. 1

OPINION

Musings

RIK FARROW



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

It's a question of balance [1]. At least, that's how I see things as I look over the collection of articles in this month's issue.

We all need balance: balance in our lives, between work and play, between waking and sleeping, exercise and rest, eating what we like and staying healthy. Computer systems require balance too.

Leading off, Jeremy Elson and Ed Nightingale describe Flat Datacenter Storage, based on their OSDI 2012 paper. I liked Jeremy's presentation in Hollywood, and asked him to reprise that presentation as an article. But as much as I liked his presentation, I also liked the balance presented in the design of FDS.

At first glance, FDS is just another distributed storage system, like HDFS. But if you look closer, you can see evidence of the balance that is the theme of this issue. FDS uses hashes to distribute *tracts* (extents) evenly throughout servers. Metadata is also distributed evenly, via a similar mechanism. But the real balancing act of FDS has to do with the design of the network. Instead of hooking up servers to top-of-rack switches, and these switches to other switches or routers, the designers of FDS sought a balance between network bandwidth at both clients and servers and the capability of a client or server to consume or produce data. They spent extra money on network infrastructure so that there are no bottlenecks to slow down performance.

And did this level of balance make a difference? FDS can replace a lost drive containing a modest amount of data (92 GB) in 6.2 seconds if the cluster includes 1,000 disks. And using the MinuteSort benchmark [2], the FDS replaced Hadoop as the fastest at generic sorting and TritonSort at specialized (Formula 1) sorting.

I think there are lessons to be learned with FDS, which is really why I asked Jeremy and Ed to write for this issue.

How Hot Is Too Hot?

El-Sayed et al. take on a different type of balance. Most datacenters (DCs) choose a conservative setpoint for the cold side of the aisles. This is based on conservative information provided by server vendors. After all, the goal of a DC is to process information, not lose servers and disks because of too high temperatures.

Through the analysis of vast amounts of data, and their own tests using a heat chamber, these researchers found that the balance between too cold (spending more on cooling) and too hot (having to replace failed components) can shift toward warmer DCs. As cooling DCs represents a large proportion of their energy budgets, being able to adjust the setpoint higher saves energy, money, and perhaps the planet as well.

Along the way, El-Sayed et al. disprove some of the long-held assumptions about the effects of temperature on the reliability of disks and memory as temperature levels increase.

Michael Adam, a Samba developer, provides us with deep insight into where Samba is today and its path forward. The Samba project has gone from leading Microsoft, with the first version of clustered SMB storage, to playing catch-up today. Fitting this into my theme of balance is a bit more difficult, but when you consider that Samba had advanced the state of SMB storage, and is now playing catch up, I can see this as an attempt to recover the balance between Samba and SMB.

I interviewed Allen Wittenauer of LinkedIn, both a Hadoop committer and large scale cluster operator. I had been curious about how one goes about managing Hadoop clusters, and Allen was just the person to talk to. Allen provided information about both how to monitor a Hadoop cluster and how to manage and improve the performance of Hadoop tasks. So where's the balance? It turns out that there is a balance here too, between taking the macro overview and drilling down to see what is happening on individual systems.

Jeanvoine et al. present Kadeploy3, a system for installing OS images on grids. They have been developing this system for years, and have successfully tested it on grids with over a thousand nodes. For balance, they describe how they decided on the best way to utilize the network bandwidth for the quickest installation.

Thomas Barr and Scott Rixner present their Python development toolchain for ARM Cortex-M3 embedded controllers. With their open source toolkit, developing software for embedded systems becomes almost as easy as Python programming: you get to use Python, you have access to the special libraries provided by the provider of the system-on-chip, and you also get instant feedback. Barr and Rixner do mention balance, in the sense of the comparison of the expert embedded systems programmer versus the rest of us. With tools like Owl (their development toolchain), we don't need to be experts to get work done, while experts are still needed to create the libraries required by Owl.

Out of Balance

So much for balance. Or perhaps I could say that for balance, the rest of the issue doesn't follow the balance theme?

David Blank-Edelman continues his journey of showing us how to use various Web-based APIs, focusing this time on a service that can send and receive texts, make voice calls, and collect dial button presses as part of a survey, or automated call center.

David Beazley set out to scare his readers by diving deeper into the arcana of the Python import mechanism. In an earlier column, David explained how you can set the import path, but this time he shows us tricks for changing what actually happens when you attempt to import a module into your Python program. Not to worry, concludes David, as this will be rationalized with version 3.3.

Dave Josephsen continues his coverage of XI, the GUI and database extension to Nagios. Dave explains the various ways a group of sysadmins can hose each other's work when using different methods of administering Nagios when only one uses XI.

Robert Ferrell shares a blast from the past. Robert depicts a UNIX "expert" with advice that might send shivers, perhaps of annoyance, down your spine.

Elizabeth Zwicky reviewed five books for this issue, including a second edition of a book on regular expressions, two Python books (data analysis and Python for kids), a book on managing programmers, and finally another kids programming book. Coincidently, Mark Lamourine reviewed the same book, *Super Scratch Programming Adventure!*, and I'm including both of them, as the perspectives of the authors, and the ages of their testers, differ. Mark also reviewed a book on assembly language programming for ARM, helping to balance out the focus on Python. Trey Darley reviewed two books, one on insider threats, and the other on Internet protocols.

We also have three sets of summaries from OSDI in this issue: the conference itself, and two workshops, MAD and HotPower.

As the editor of *;login:*, I always strive for a balance between materials that I believe are of most interest to the greatest number of readers, and articles that have something to teach. To me, the process of research is one of learning, composed of new ideas, implementation, and trial and error, and that's something that we can all learn from.

References

[1] A Question of Balance: http://en.wikipedia.org/wiki/A_Question_of_Balance.

[2] Sort Benchmark: http://sortbenchmark.org/.

Flat Datacenter Storage

JEREMY ELSON AND EDMUND B. NIGHTINGALE

Jeremy Elson received his PhD from UCLA in 2003. He has worked in wireless sensor networks, time synchronization, online mapmaking, CAPTCHAs, and distributed storage. He also enjoys riding bicycles, flying airplanes, and DIY electronics.
jelson@microsoft.com

Ed Nightingale has worked at Microsoft since graduating with a PhD from the University of Michigan in 2007. Ed's favorite research areas include operating systems and distributed systems. Outside of work, Ed enjoys bicycling, reading, and attempting to keep up with his children.
ed.nightingale@microsoft.com

There's been an explosion of interest in Big Data—the tools and techniques for handling very large data sets. Flat Datacenter Storage (FDS) is a new storage project at Microsoft Research. We've built a blob store meant for Big Data, which scales to tens of thousands of disks, makes efficient use of hardware, and is fault tolerant, but still maintains the conceptual simplicity and flexibility of a small computer.

To make this idea concrete, consider the problem of "little data." From a systems perspective, little data is essentially a solved problem. The perfect little-data computer has been around for years: a single machine with multiple processors and disks interconnected by something like a RAID controller. For I/O-intensive workloads, such a computer is ideal. When applications write, the RAID controller splits the writes up and stripes them over all the disks. There might be a small number of writers writing a lot, or a large number of writers writing a little bit, or a mix of both. The lulls in one writer are filled in by the bursts in another, giving us good statistical multiplexing. All the disks stay busy, and high utilization means we're extracting all the performance we can from our hardware. Reads can also exploit the striped writes. Even if some processes consume data slowly and others consume it quickly, all the disks stay busy, which is what we want.

Writing software for this computer is easy, too. How many physical disks there are doesn't matter; programmers can pretend there's just one big one. Files written by any process can be read by any other without caring about locality. If we're trying to attack a large problem in parallel (for example, trying to parse a giant log file) the input doesn't need to be partitioned in advance. All the workers drain a global pool of work; when it's exhausted, they all finish at about the same time. This prevents stragglers and means the job finishes sooner. We call this *dynamic work allocation*.

Another benefit of the little-data computer is that it's easy to adjust the ratio of processors to disks by adding more of whichever is needed. An administrator can buy machine resources to match the expected workload, fully and efficiently making use of the hardware budget.

This machine has one major drawback: it doesn't scale. We can add a few dozen processors and disks, but not thousands. The limitation lies in the fact that such a system relies on a single, centralized I/O controller. Roughly, the controller is doing two things:

- It manages metadata. When a process writes, the controller decides how the write should be striped, and records enough state so that reads can find the data later.
- It physically routes the data between disks to processors—actually transporting the bits.

In FDS, we've built a blob store that fully distributes both of these tasks. This means we can build a cluster that has the essential properties of the ideal little-data machine, but can scale to the size of a datacenter. To maintain conceptual simplicity, computation and storage are logically separate. There is no affinity, meaning any processor can access all data in the system uniformly—that's why we call it "flat;" however, it still achieves very high I/O performance that has come to be expected only from systems that couple storage and computation together, such as MapReduce, Dryad, and Hadoop.

We've developed a novel way of distributing metadata. In fact, the common case read and write paths go through no centralized components at all. We get the bandwidth we need from full bisection bandwidth Clos networks, using novel techniques to schedule traffic.

With FDS, we've demonstrated very high read and write performance. In a single-replicated cluster, a single process in read or write loop can achieve more than 2 GBps all the way to the remote disk platters. In other words, FDS applications can write to remote disks faster than many systems can write locally to a RAID array.

Disks can also talk to each other at high speed, meaning FDS can recover from failed disks very quickly. For example, in one test with a 1,000-disk cluster, we killed a machine with seven disks holding a total of about two-thirds of a terabyte; FDS brought the lost data back to full replication in 34 seconds.

Finally, we've shown that FDS can make applications very fast. We wrote a straightforward sort application on top of FDS that beat the world record for disk-to-disk sorting in 2012. Our general-purpose remote blob store beat previous implementations that exploited local disks. We've also experimented with applications from other domains, including stock market analysis and serving an index of the Web.

The Basics

In FDS, all blobs are identified with a simple GUID. Each blob contains 0 or more allocation units we call *tracts*. Tracts are numbered sequentially, starting from 0 (Figure 1).

All tracts in a system are the same size. In most of our clusters, a tract is 8 MB; we'll see later why we picked that size. A tract is the basic unit of reading and writing in FDS.

The programming interface is simple; it has only about a dozen calls, such as *CreateBlob, ReadTract,* and *WriteTract*. The interface is designed to be asynchronous, meaning that the functions don't block, but rather call a callback when they're done. A typical high-throughput FDS application will start out by issuing a few dozen reads or writes in parallel, then issue more as the earlier ones complete. We call applications using the FDS API the *FDS clients*.

In addition to clients, there are two other types of actors in FDS. The first is the *tractserver*, lightweight software that sits between a raw disk and the network, accepting commands from the network such as "read a tract" and "write a tract."

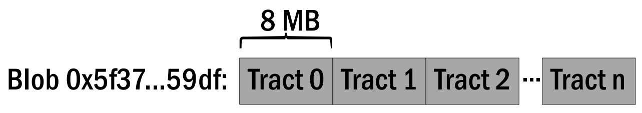


Figure 1: Blobs and tracts

There's also a special node called the *metadata server,* which coordinates the cluster and helps clients rendezvous with tractservers.

The existence of tractservers and the metadata server is invisible to programmers. The API just talks about blobs and tract numbers. Underneath, our library contacts the metadata server as necessary and sends read and write messages over the network to tractservers.

Metadata Management

To understand how FDS handles metadata, it's useful to consider the spectrum of solutions in other systems.

On one extreme, we have systems like GFS and Hadoop that manage metadata centrally. On essentially every read or write, clients consult a metadata server that has canonical information about the placement of all data in the system. This gives administrators excellent visibility and control; however, it is also a centralized bottleneck that has exerted pressure on these systems to increase the size of writes. For example, GFS uses 64 megabyte extents, nearly an order of magnitude larger than FDS tracts. This makes it harder to do fine-grained load balancing like the ideal little-data computer does.

On the other end of the spectrum are distributed hash tables. They're fully decentralized, but all reads and writes typically require multiple trips over the network before they find data. Additionally, failure recovery is relatively slow because recovery is a localized operation among nearby neighbors in the ring.

In FDS, we tried to find a spot in between that gives us some of the best properties of both extremes: one-hop access to data and fast failure recovery without any centralized bottlenecks in common-case paths.

FDS does have a centralized metadata server, but its role is limited. When a client first starts, the metadata server sends some state to the client. For now, think of this state as an oracle.

When a client wants to read or write a tract, the underlying FDS library has two pieces of information: the blob's GUID and the tract number. The client library feeds those into the oracle and gets out the IP addresses of the tractservers responsible for replicas of that tract. In a system with more than one replica, reads go to one replica at random, and writes go to all of them.

The oracle's mapping of tracts to tractservers needs two important properties. First, it needs to be *consistent*: a client reading a tract needs to get the same answer as the writer got when it wrote that tract. Second, it has spread load *uniformly*. To achieve high performance, FDS clients have lots of tract reads and writes outstanding simultaneously. The oracle needs to ensure (or, at least, make it likely) that all of those operations are being serviced by different tractservers. We don't want all the requests going to just one disk if we have ten of them.

Locator	Replica 1	Replica 2	Replica 3
0	A	B	C
1	A	D	F
2	A	C	G
3	D	E	G
4	B	C	F
...	...	...	...
1,526	LM	TH	JE

Figure 2: An example tract locator table. Each letter represents a disk.

Once a client has this oracle, reads and writes all happen without contacting the metadata server again. Because reads and writes don't generate metadata server traffic, we can afford to do a large number of small reads and writes that all go to different spindles, even in large-scale systems, giving us really good statistical multiplexing of the disks—just like the little-data computer.

This technique gives us the flexibility to make writes as small as we need to. For throughput-sensitive applications, we use 8 MB tracts: large enough to amortize seeks and make random reading and writing almost as fast as doing so sequentially. We have also experimented with *seek-bound* workloads, where we reduced the tract size all the way down to 64 KB. That's hard with a centralized metadata server but no problem with our oracle.

So, what is this oracle? Simply, it is a table of all the disks in the system, collected centrally by the metadata server. We call this table the *tract locator table*, or TLT. The table has as many columns as there are replicas; the example in Figure 2 shows a triple-replicated system. In single-replicated systems, the number of rows in this table grows linearly with the number of disks in the system. In multiply replicated systems, it grows as n^2; we'll see why a little later.

For each read or write operation, the client finds a row in this table by taking the blob GUID and tract number and deterministically transforming them into a row index:

```
Table_Index=(Hash(Blob_GUID) + Tract_Number) mod TLT_Length
```

As long as readers and writers are using consistent versions of the table, the mappings they get will also be consistent. (We describe how we achieve consistent table versioning in our full paper [3].) We hash the blob's GUID so that independent clients start at "random" places in the table, even if the GUIDs themselves are not randomly distributed.

A critical property of this table is that it only contains disks, not tracts. In other words, reads and writes don't change the table. This means clients can retrieve it from the metadata server once, then never contact the metadata server again. The TLT only changes when a disk fails or is added.

There's another clever thing we can do with the tract locator table: use it to fully distribute the *per-blob* metadata, such as each blob's length and permission bits. We store this in "tract -1." Clients find the metadata tract the same way that they find regular data, just by plugging -1 into the tract locator formula. This means that the metadata is spread pseudo-randomly across all tractservers in the system, just like the regular data.

Tractservers have support for consistent metadata updates. For example, imagine that several writers are trying to append to the same blob. In FDS, each executes an FDS function called *Extend Blob*. This is a request for a range of tract numbers that can be written without conflict. The tractserver serializes the requests and returns a unique range to each client. This is how FDS supports atomic append.

Unlike data writes, which go directly from the client to all replicas, metadata operations in multiply replicated systems go to only one tractserver—the one in the first column of the table. That server does a two-phase commit to the others before returning a result to the client.

Because we're using the tract locator table to determine which tractserver owns each blob's metadata, different blobs will most likely have their metadata operations served by different tractservers. The metadata traffic is spread across every server in the system; however, requests that need to be serialized because they refer to the same blob will always end up at the same tractserver, thus maintaining correctness.

Networking

So far, we've assumed that there was an uncongested path from tractservers to clients. We now turn to the question of how to build such a network.

Until recently, the standard way to build a datacenter was with significant oversubscription: a top-of-rack switch might have 40 Gbps of bandwidth down to servers in the rack, but only 2 or 4 Gbps going up to the network core. In other words, the link to the core was oversubscribed by a factor of 10 or 20. This, of course, was done to save money.

There has been a recent surge of research in the networking community in Clos networks [2]. Clos networks more or less do for networks what RAID did for disks: by connecting up a large number of low-cost, commodity routers and doing some clever routing, building full bisection bandwidth networks at the scale of a datacenter is now economical.

In FDS, we take the idea a step further. Even with Clos networks, many computers in today's datacenters still have a bottleneck between disks and the network. A typical disk can read or write at about a gigabit per second, but there are four, or 12, or even 25 disks in a typical machine, all stuck behind a single one-gigabit link. For applications that have to move data, such as a sort or a distributed join, this is a big problem.

In FDS, we make sure all machines with disks have as much network bandwidth as they have disk bandwidth. For example, a machine with 10 disks needs a 10-gigabit NIC, and a machine with 20 disks needs two of them. Of course, adding bandwidth has a cost; depending on the size of the network, we estimate about 30% more per machine. But as we'll explain a little later, we get a lot more than a 30% increase in performance for that investment.

We've gone through several generations of testbeds; the largest has 250 machines and about 1,500 disks. They're all connected using 14 top-of-rack routers and eight spine routers. Each router has 64 10-gigabit ports. The top-of-rack routers split their 64 ports into two halves: 32 ports connect to computers (clients or tractservers) and 32 connect to spine routers. There is a 40 Gbps connection between each top-of-rack and spine router—four 10 Gbps ports bonded together. In aggregate, this gives us more than 4.5 terabits of bisection bandwidth.

Unfortunately, just adding all this bandwidth doesn't automatically produce a storage system with good performance. Part of the problem is that in realistic conditions, datacenter Clos networks don't guarantee full bisection bandwidth. They only make it stochastically likely. This is an artifact of routing algorithms such as ECMP (equal-cost multipath routing) that select a single, persistent path for each TCP flow to prevent packet reordering. As a result, Clos networks have a well-known problem handling long, fat flows. In FDS, our data layout is designed to spread data uniformly across disks partly because of the network load that such an access pattern generates. FDS clients use a large number of very short-lived flows to a wide set of pseudo-random destinations, which is the ideal case for a Clos network.

A second problem is that even a perfect Clos network doesn't actually eliminate congestion; it just pushes the congestion out to the edges. Good traffic shaping is still necessary to prevent catastrophic collisions at receivers—a condition known as *incast* [6].

What's particularly unfortunate is that these two constraints are in tension. Clos networks need *short* flows to achieve good load balancing, but TCP needs *long* flows for its bandwidth allocation algorithm to find an equilibrium that prevents collisions.

In FDS, we ended up doing our own application-layer bandwidth allocation using a hybrid request-to-send/clear-to-send (RTS/CTS) scheme reminiscent of that found in wireless networks. Large messages are queued at the sender, and the receiver is notified with an RTS. The receiver limits the number of CTSes it allows outstanding, thus limiting the number of senders competing for its receive bandwidth. Small messages, such as control messages and RTS/CTS, are delivered over a different TCP flow from the large messages, reducing latency by enabling them to bypass long queues. FDS network message sizes are bimodal: large messages are almost all about 8 MB, and most other messages are 1 KB or smaller.

Microbenchmarks

Our full paper [3] has a more thorough evaluation of FDS. Here, we'll describe one set of microbenchmarks: testing the speed of simple test clients that read from or wrote to a fixed number of tractservers. We varied the number of clients and measured their aggregate bandwidth. The clients each had a single 10 Gbps Ethernet connection. The tractservers had either one or two, depending on how many disks were in the server.

Figure 3 shows results from a single-replicated cluster. Note the x-axis is logarithmic. The aggregate read and write bandwidth go up close to linearly with the number of clients, from 1 to 170. Read bandwidth goes up at about 950 MBps per client and write bandwidth goes up by 1,150 MBps per client. Writers saturated about 90% of their theoretical network bandwidth, and readers saturated about 74%.

Two different cluster configurations are depicted: one used 1,033 disks, and the other used about half that. In the 1,033 disk test, there was just as much disk bandwidth as there was client bandwidth, so performance kept going up as we added more clients. In the 516 disk test, there was much more client bandwidth available

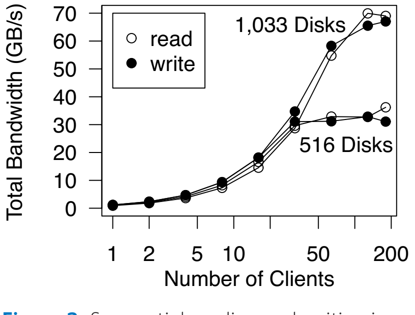


Figure 3: Sequential reading and writing in a single-replicated cluster

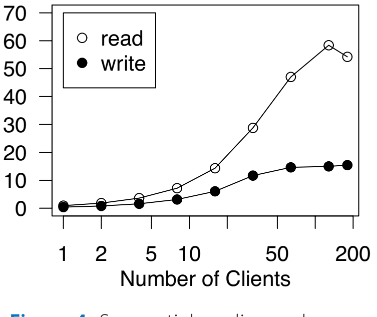


Figure 4: Sequential reading and writing in a triple-replicated cluster

than disk bandwidth. Because disks were the bottleneck, aggregate bandwidth kept going up until we'd saturated the disks, then leveled off.

We also tested clients that had 20 Gbps of network bandwidth instead of 10. These clients were able to read and write at over 2 GBps. In other words, writing remotely over the network all the way to disk platters, these FDS clients were faster than many systems can write to a local RAID. Decoupling storage and computation does not have to mean giving up performance.

Figure 4 shows a similar test against a triple-replicated cluster instead of a single-replicated cluster. Read bandwidth is about the same, but as expected, writes saturate the disks much sooner because clients have to write three times as much data (once for each replica). The aggregate write bandwidth is about one-third of the read bandwidth in all cases.

Failure Recovery

The way that data is organized in a blob store has a dramatic effect on recovery performance. The simplest method of replication is unfortunately also the slowest: mirroring. Disks can be organized into pairs or triples that are always kept identical. When a disk fails, an exact copy of the failed disk is created using an empty spare disk and a replica that's still alive. This is slow because it's constrained by the speed of a single disk. Filling a one terabyte disk takes at least several hours, and such slow recovery decreases durability because it lengthens the window of vulnerability to additional failures.

We can do better. In FDS, when a disk fails, our goal is not to reconstruct an exact duplicate of the failed disk. Instead, we ensure that *somewhere* in the system, extra copies of the lost data get made, returning us to the state where there are three copies of all data.

We exploit our fine-grained striping of data across disks, and lay out data so that when a disk fails, there isn't just a single disk that contains backup copies of that disk's data. Instead, the n disks that remain will each have about $1/n$th of the data lost. Every disk sends a copy of its small part of the lost data to some other disk that has some free space.

Because we have a full bisection bandwidth network, all the disks can do this in parallel, making failure recovery fast. In fact, because *every* disk is participating in recovery, FDS has a nice scaling property: as a cluster gets larger, recovery goes faster. This is just the opposite of systems that use simple mirroring, where larger volumes require *longer* recovery times.

Implementing this scheme using the tract locator table is relatively straightforward. We construct a table such that every possible *pair* of disks appears in a row of the table. This is why, in replicated clusters, the number of rows in the table grows as n^2. We can optionally add more columns for more durability, but to get the fastest recovery speed, we never need more than n^2 rows.

When a disk fails, the metadata server first selects a random disk to replace the failed disk in every row of the table. Then, it selects one of the remaining good disks in each row to transfer the lost data to the replacement disk. (Additional details are described in our paper [3].)

Disk Count	100		1,000		
Disks Failed	1	1	1	1	7
Total Stored (TB)	4.7	9.2	47	92	92
GB/disk	47	92	47	92	92
GB Recovered	47	92	47	92	655
Recovery Time (s)	19.2±0.7	50.5±16.0	3.3±0.6	6.2±0.4	33.7±1.5

Table 1: Mean and standard deviation of recovery time after disk failure in a triple-replicated cluster. The high variance in one experiment is due to a single 80 sec. run.

We tested failure recovery in a number of configurations, in clusters with both 100 and 1,000 disks, and killing both individual disks and all the disks in a single machine at the same time (Table 1).

In our largest test, we used a 1,000-disk cluster and killed a machine with seven disks holding a total of 655 GB. All the lost data was recovered in 34 seconds.

More interesting is that every time we made the cluster larger, we got about another 40 MBps per disk of aggregate recovery speed. That's less than half the speed of a disk, but keep in mind that's because every disk is simultaneously reading the data it's sending, and writing to its free space that some other disk is filling. Extrapolating these numbers out, we estimate that if we lost a 1 TB disk out of a 3,000-disk cluster, we'd recover all the data in less than 20 seconds.

MinuteSort

We've built several big-data applications on top of FDS from domains that include stock market analysis and serving an index of the Web. These applications are described in our recent paper on FDS [3]. In this article, we'll focus on just one: We set two world records in 2012 for disk-to-disk sorting using a small FDS application.

MinuteSort is a test devised by a group led by the late Jim Gray [1]. The question is: given 60 seconds, how much randomly distributed data can be shuffled into sorted order? Because the test was meant as an I/O test, the rules specify the data must start and end in stable storage. We competed in two divisions: one for general-purpose systems, and one for purpose-built systems that were allowed to exploit the specifics of the benchmark.

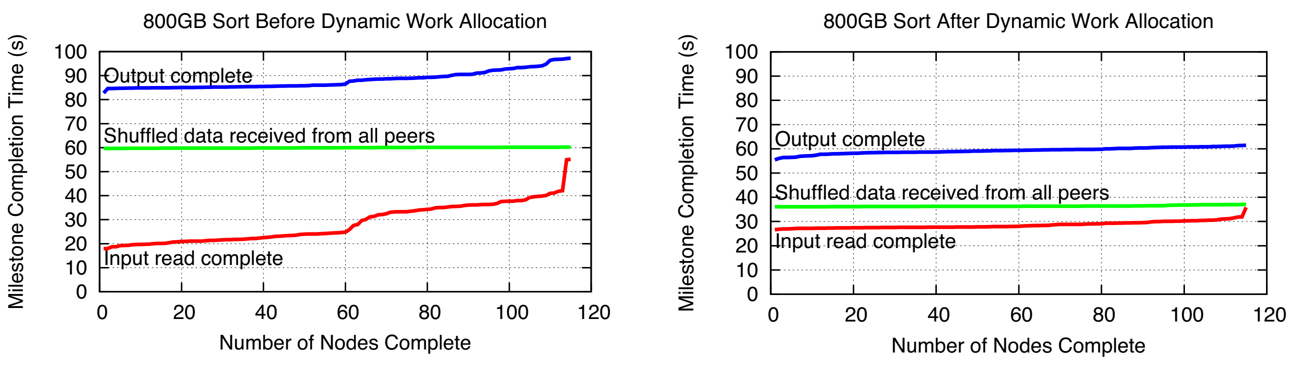


Figure 5: Visualization of the time to reach three milestones in the completion of a sort. The results are shown before (left) and after (right) implementation of dynamic work allocation. Both experiments depict 115 nodes sorting 800 GB.

In the general-purpose division, the previous record, which stood for three years, was set by Yahoo! using a large Hadoop cluster [4] consisting of about 1,400 machines, and about 5,600 disks. With FDS, using less than one-fifth of the computers and disks, we nearly tripled the amount of data sorted, which multiplies out to a 15x improvement in disk efficiency. The gain came from the fact that Yahoo!'s cluster, like most Hadoop-style clusters, had serious oversubscription both from disk to network, and from rack to network core. We attacked that bottleneck, by investing, on average, 30% more money per machine for more bandwidth, and harnessed that bandwidth using the techniques described earlier. The result is that instead of a cluster having mostly idle disks, we built a cluster with disks working continuously.

In the specially optimized class, the record was set last year by UCSD's Triton-Sort [5]. They wrote a tightly integrated and optimized sort application that did a beautiful job of squeezing everything they could out of their hardware. They used local storage, so they did beat us on CPU efficiency, but not on disk efficiency. In absolute terms, we set that record by about 8%. What distinguishes our sort is that it was just a small application sitting on top of FDS, a general-purpose blob store with no sort-specific optimizations.

Dynamic work allocation was a key technique for making our sort fast. We noted earlier that one advantage of ignoring locality constraints—as in the little-data computer—is that all workers can draw work from a global pool, preventing stragglers. Early versions of our sort didn't use dynamic work allocation; we just divided the input file evenly among all the nodes.

As seen in the time diagram in Figure 5 (left), stragglers were a big problem. Each line represents one stage of the sort. A horizontal line would mean all nodes finished that stage at the same time, which would be ideal. Initially, the red (lowest line) stage was far from ideal. About half the nodes would finish the stage within 25 seconds and a few would straggle along for another 30. This was critical because there was a global barrier between the red stage and the green stage (middle line in graph).

We knew the problem did not lie in the hardware because different nodes were the stragglers in each experiment. We concluded that we had built a complex distributed system with a great deal of randomness; a few nodes would always get unlucky. We switched to using dynamic work allocation. In Figure 5 (right), each node would initially process a tiny part of the input. When it was almost done, it would ask the head sort node for more work. This dramatically reduced stragglers, making the whole job faster. A worker that finished early would get more work assigned and unlucky nodes would not. This was entirely enabled by the fact that FDS uses a global store; clients can read any part of the input they want, so shuffling the assignments around at the last second really has no cost.

Conclusion

FDS gives us the agility and conceptual simplicity of a global store, but without the usual performance penalty. We can write to remote storage just as fast as other systems can write to local storage, but we're able to discard the locality constraints.

This also means we can build clusters with very high utilization; we can buy as many disks as we need for I/O bandwidth, and as many CPUs as we need for processing power. Individual applications can use resources in whatever ratio they need. We do have to invest more money in the network. In exchange, we unlock the potential of all the other hardware we've paid for, both because we've opened the network bottleneck and because a global store gives us global statistical multiplexing.

Today, many data scientists have the mindset that certain kinds of high-bandwidth applications must fit into a rack if they're going to be fast, but a rack just isn't big enough for many big-data applications. With FDS, we've shown a path around that constraint. FDS doesn't just make today's applications faster. FDS may let us imagine new *kinds* of applications, too.

Acknowledgments

Dave Maltz built our first Clos networks and taught us how to build our own. Johnson Apacible, Rich Draves, and Reuben Olinsky were part of the sort record team. Trevor Eberl, Jamie Lee, Oleg Losinets, and Lucas Williamson provided systems support. Galen Hunt provided a continuous stream of optimism and general encouragement. We also thank Jim Larus for agreeing to fund our initial 14-machine cluster on nothing more than a whiteboard and a promise, allowing this work to flourish.

References

[1] D. Bitton, M. Brown, R. Catell, S. Ceri, T. Chou, D. DeWitt, D. Gawlick, H. Garcia-Molina, B. Good, J. Gray, P. Homan, B. Jolls, T. Lukes, E. Lazowska, J. Nauman, M. Pong, A. Spector, K. Trieber, H. Sammer, O. Serlin, M. Stonebraker, A. Reuter, and P. Weinberger, "A Measure of Transaction Processing Power," *Datamation*, vol. 31, no. 7 (April 1985), pp. 112–118.

[2] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Towards a Next Generation Data Center Architecture: Scalability and Commoditization," *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '08 (ACM, 2008), pp. 57–62.

[3] E.B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue, "Flat Datacenter Storage," Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12), October 2012.

[4] O. O'Malley and A.C. Murthy, "Winning a 60 Second Dash with a Yellow Elephant," 2009: http://sortbenchmark.org/Yahoo2009.pdf.

[5] A. Rasmussen, G. Porter, M. Conley, H. Madhyastha, R.N. Mysore, A. Pucher, and A. Vahdat, "TritonSort: A Balanced Large-Scale Sorting System," 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11), Boston, MA, April 2011.

[6] V. Vasudevan, H. Shah, A. Phanishayee, E. Krevat, D. Andersen, G. Ganger, and G. Gibson, "Solving TCP Incast in Cluster Storage Systems," 7th USENIX Conference on File and Storage Technologies (FAST '09), San Francisco, CA, February 2009.

Samba's Way Toward SMB 3.0

MICHAEL ADAM

Michael Adam is a software engineer with a university background in mathematics and computer science. He leads the Samba team of SerNet GmbH in Germany. As a developer of the Samba software, Michael currently concentrates on the new protocol aspects in the SMB file server and is especially interested in clustering Samba. obnox@samba.org

The well-known Samba software has been a pioneer in clustered Server Message Block (SMB) file sharing since 2007. With Windows 8 and Server 2012, Microsoft has released version 3.0 of the SMB protocol, introducing a whole set of new features for all-active SMB clustering and server workloads. This article describes the interesting challenges that Samba faces in implementing SMB 3 and the missing features of SMB 2—most notably durable file handles—and provides details about the techniques used to solve the issues.

The Samba software [1] has provided open source SMB file, print, and authentication services on UNIX systems since 1992. When version 3.6 was released in 2011, Samba added support for version 2.0 of the SMB protocol.

In late 2011, Microsoft presented what is now called SMB 3.0 under the name of SMB 2.2 at the SNIA Storage Developer Conference [2]. At that time, the Samba engineers had developed a rather good understanding of most of the tasks for implementing SMB 2.1 and had started to work on the one big feature of SMB 2.0 that was missing from Samba's SMB 2. 0 implementation in Samba 3.6: durable file handles.

The announcement of SMB 2.2 got the Samba developers started because this touched an area Samba had pioneered since 2007: all-active clustering of SMB itself. So in parallel to the design and development of durable file handles, the developers dove into the exploration of the new SMB version 3.0. It turned out that in order to implement durable handles, some of the internal design characteristics of Samba that had to be revised and changed were also obstacles for the implementation of the SMB 3 features as well as most parts of SMB 2.1. So the developers took as holistic an approach as possible when implementing durable handles in order to pave the way for moving toward SMB 3.

A year has passed with the outcome that Samba 4.0, which has just been released, features durable handles and support for SMB 2.1—complete except for leases—and basic support for SMB 3.0, not yet including any of the more advanced features. Before we dive into the Samba-specific details, here is an overview of the various versions of SMB.

SMB 1, 2, 3...

The SMB protocol has been around for many years and is the basis of Windows' network file and print services. There are many other implementations of the protocol—Samba being the most popular open source implementation—generally available in Linux and on most UNIX systems, and used in storage appliances, from small to very large, scale-out clustered systems.

With Windows Vista (released in 2007) and Windows Server 2008, Microsoft introduced version 2.0 of the SMB protocol. SMB 2.0 was essentially a cleanly re-designed variant in the long evolution of the SMB protocol. The number of calls was reduced, all file operations were handle-based, and chained requests were integrated into the protocol. One of the biggest additions of SMB 2.0 was the concept of durable file handles: durable handles can be reclaimed by the client after a short network outage with all lock and caching state, so as to allow for uninterrupted I/O.

SMB 2.1, introduced with Windows 7 and Windows Server 2008 R2 in late 2009, brought a couple of new features: leases, which are "oplocks done right," and handle caching introduced systematically with refined cache revocation rules as compared to the old batch oplocks; and applications were able to upgrade their caching mode without the need to break the mode in advance. Multi-credit (or large MTU) was a new mechanism that reduced the number of network round trips required for reading or writing large hunks of data by allowing for bigger data units to be transferred in a single read or write request.

Resilient file handles added certain guarantees to durable handles. Their usefulness was limited, though, because resiliency had to be requested by an aware application that used the published interface. Dynamic reauthentication allowed the client to reauthenticate a session proactively before the server signaled the session had expired. Branch cache enables machines in an office to cache file server contents locally after it has been fetched over a wide area network link. In summary, the major topics of SMB 2.1 were the reduction of network verbosity and increased reliability.

In September and October 2012, respectively, Windows Server 2012 and Windows 8 were published, along with version 3.0 of the SMB protocol. This version was called 2.2 until shortly before the official release, but was relabeled due to the scope of the additions.

The main topic of SMB 3 is all-active clustering, especially the scale-out and continuously available characteristic of file shares. With SMB 3, Microsoft for the first time specifically addressed server workloads, for Hyper-V and SQL. Apart from new signing and encryption mechanisms and the so-called secure negotiation, some of the major features of SMB 3 are: multi-channel as a mechanism for the client to bundle multiple transport connections (channels) into a single SMB session; persistent file handles are like durable handles with strong guarantees (in contrast to the resilient handles of SMB 2.1, these are transparent to the client); and support for RDMA-capable transports such as Infiniband or iWARP for reduced latency and CPU load imposed by network I/O operations.

Tasks for Samba

There are a lot of interesting features in the above list that Samba has not implemented or had not implemented in Samba 3.6. Specifically, durable handles of SMB 2.0, leases, multi-credit, dynamic reauthentication of SMB 2.1 and directory leases, the clustering concepts, persistent handles, multi-channel, and RDMA-support of SMB 3.0. To understand how the Samba developers have solved or are planning to solve the items of this voluminous list of tasks, one first must know some details about the design and internal functionalities of the Samba software.

Design of Samba's File Server

The main daemon of Samba's file server is the smbd process. It listens on the TCP ports 445 and 139 for SMB connections and, for each new TCP connection, forks one `smbd` child process that is afterward exclusively in charge of handling this connection. There is hence essentially a 1:1 correspondence between `smbd` child processes and SMB-client TCP connections to the Samba server.

A certain amount of interprocess communication is needed between the `smbd` processes, mostly regarding locking and other conflicting operations on files. Because the locking semantics in SMB differ widely from what POSIX provides, the obvious UNIX/POSIX locking mechanisms through the file system and kernel are not enough here. Therefore, Samba has introduced a file system abstraction layer that implements the required features of a file system with the required semantics. This layer maintains a set of databases to store information that is not available directly in the underlying POSIX file system. These are most importantly `locking.tdb`, which is the database for open file handles, including share modes, oplocks, and `brlock.tdb`, which stores the byte range locks. On the other hand, the SMB-level pieces of information were kept purely in memory in previous releases of Samba, including the latest version 3.6, which included support for SMB 2.0.

The mentioned databases use Samba's own database implementation, the trivial database (TDB) [3]. It is a simple Berkeley DB-style key-value database that also supports multiple concurrent writers through record locks and memory mapping, and can be used for fast interprocess communication. The TDB databases are used virtually everywhere inside Samba.

In addition to the databases, there is the so-called messaging as a means of interprocess communication. The smbd processes can register themselves as interested in a certain database record, for example, representing a lock on a file. And the process holding the lock will send a message to the waiting process when it releases the lock so that the waiting process can try again to get hold of the lock. The messages are implemented by signals to trigger the other process and TDB records that hold the actual message content.

Samba, the Clustered SMB Pioneer

This architecture of Samba's file server was the main reason that the implementation of a clustered Samba file server was initially relatively easy to achieve: Samba was multi-process, and the important data for interprocess communication was serialized into the TDB databases anyway. So when the developers first started to experiment seriously with clustered Samba in 2006, the main task quickly became making the TDB databases clustered, because serving different TCP connections to a single node by different daemons on that node is in principle not much different from serving TCP connections to different nodes by processes on those nodes.

Some clever design had to be invented to make TDB clustered in a fashion that scaled well. Using a conventional clustered database as a substitute, or even storing the TDB files in the clustered file system to be used for sharing files with Samba, turned out to scale negatively, but the intent was to create a system that would scale positively (if not linearly) with the number of nodes in the cluster with respect to the number of operations and the accumulated SMB throughput per time unit. In 2007, the clustered TDB software CTDB [4] was released and achieved this goal (see my paper about clustering Samba with CTDB [5]).

This clustering of course cannot be perfect, because it must be transparent to the client; the Samba server implementation cannot change the Windows SMB client software to be aware of the clustering. To the client, a Samba-CTDB cluster looks like a single SMB server with multiple network interfaces. The main issue is what happens when one node becomes unavailable. The IP addresses associated with that node are migrated to a different node, and the CTDB software sends so-called tickle-ACK packets to the clients previously connected to the now unavailable node to trigger a fast reconnect to the same IP address. But this reconnect means that the client loses all its open file handles with the associated caches (oplocks) and locks. Also, write or read operations that were in process when the node failure occurred are lost. Samba could not implement any retry or replay mechanisms without being able to modify the client.

This form of clustering was good enough, though, for the Samba-CTDB suite to become pretty popular quickly. Big installations started using it, and it was meanwhile used as a base technology in a couple of NAS appliances. The main point was that Windows at that time could not do all-active SMB clustering at all.

This has changed now with SMB 3.0, which has introduced all-active clustering of SMB into the protocol.

Step 0: Client Implementation and Tests

Despite the availability of protocol documentation from Microsoft in the MSDN library [6] since 2008, after the court decision in the European Commission competition case, the initial step in Samba's development process toward understanding and implementing new features in the server is usually still to write tests. The official source about SMB 2 and 3 is the [MS-SMB2] document from MSDN [7]. At the time Samba started to explore SMB 3, the SMB 3 content of this document was still nascent, so test cases were inevitable. The tests are usually written as part of the smbtorture program. These tests are run against Windows and are extended until a good understanding of the protocol aspect is achieved, and as a next step the server implementation is extended until the tests are passed. The prerequisite for writing these tests for SMB features is a client implementation of these features. At the beginning of the exploration of durable handles and SMB 3.0, there were effectively four SMB client libraries in Samba: implementations for SMB 1 and SMB 2 in `source3/` and `source4/`.

To understand this, one has to know that the Samba code had been split into two code bases with the release of Samba 3.0 and the start of the Samba4 project in 2003. These two code bases were then known as the Samba3 and the Samba4 projects and were, despite the original intent, developed in parallel until they were merged again in 2008. From this time on, the code that was originally from the Samba3 tree was found in the source3 directory, and the `Samba4` code in the source4 directory. Although the code has been increasingly reconciled since the merge, the client implementations and test tools were still completely separate until last year. None of these client implementations was complete and each had its own problems.

As a first step, the developers created a common low-level base library for SMB1 and SMB2, and the existing libraries were turned into wrappers around this new library. The new client library is located in the files

```
libcli/smb/smbXcli_base.h
libcli/smb/smbXcli_base.c
```

With this client library a whole new set of tests have been written for durable and persistent handles, leases, multi-credit, multi-channel, and many more aspects of SMB 1 and 2. The test tool even uncovered a couple of bugs in the Windows Server 2012 prereleases. It still remains to unify the higher level libs into a single SMB client library that is used in all tests and in the smbclient command line client tool.

Implementing Durable Handles

The basic support for SMB 2.0 in Samba 3.6 could be added while keeping the original overall design paradigms explained above, namely the principle that the SMB-level pieces of information about sessions, share connections ("tree connects" in SMB speak), and open files were previously kept in memory and not marshaled into databases, because there was no need for interprocess communication at that level.

Durable file handles of SMB 2.0 created a new situation, in that they are a purely SMB-level concept. When the client is disconnected, the SMB server effectively keeps the file behind the durable handle open and gives it back to the client when it reconnects and reclaims the durable handle. More concretely, when a client reconnects after a short network outage, it uses a special form of the SMB2 session setup, the so-called session reconnect, which is characterized by the presence of the `PreviousSessionId` field. The server is to delete all tree connects associated to the session and close associated file handles except for the durable ones before replying to the session setup request. Thus, Samba needs to be able to look up SMB sessions, especially disconnected ones, by session ID, and tree connects by the tree ID. After establishing a new tree connect, the client requests to reconnect its durable handles by specifying the `FileId` in the durable handle reconnect create context. Hence, Samba needs to be able to look up open file handles by their file ID.

Furthermore, a reconnecting client will talk to a newly created `smbd` process, and hence the file handle in Samba can no longer be tied to a single `smbd` process. This shows that in order to implement durable handles, Samba needed a way to access session, tree connect, and file handle information from different processes, i.e., to create new SMB-level databases for sessions, tree connects, and open file handles.

After some initial thoughts about really keeping the files open until a client reconnects and tries to reclaim its durable handles, and then passing the open file to the new `smbd` via fd-passing, the developers chose a different initial approach: At disconnect, Samba closes the file and marks the entry in the `locking.tdb` database as disconnected. When the client reconnects the durable handle, Samba looks up the handle information in the corresponding databases, reopens the file, and reestablishes all modes and locks. This approach has the advantages of being easier to implement and of having a chance of succeeding when an SMB process gets killed, or when a reconnect happens to a different node in a Samba-CTDB cluster. One disadvantage is that it is not interoperable; between the server closing the file and the client reconnecting there is a possibility that a different application (such as NFS) could access the file without Samba noticing.

In addition to the requirement for new databases, the assumption that the entries in the databases always refer to an existing `smbd` process had to be given up. This point is quite subtly important, since the lazy cleanup of the VFS-level databases relied on each entry being valid if and only if the process referred to by the entry existed. The new disconnected state enters a special new server ID token into the entries that lets the cleanup mechanism skip the pruning of the corresponding entry.

The introduction of the new SMB-level databases was in fact much more involved than one might guess, because the old structures mixed elements from the SMB/SMB2 layer, the FSA layer (see [MS-FSA] referred to by the [MS-SMB2], [MS-SMB], and [MS-CIFS] documents of the MSDN documentation [6]), and Samba's POSIX-VFS layer. The structures were also used across the various layers, so it was rather difficult to change a behavior in just one layer. This situation arose because the Samba code was not cleanly designed in a greenfield environment but was production code, the roots of which began at a time of limited understanding of the protocol and which grew over many years. Hence the introduction of the databases also required a cleanup and reworking of the SMB server.

The result was the `smbXsrv` system, a set of structures, databases, and attached code to form the core code of the protocol side of the SMB server. The data structures are defined in the idl (interface definition language) file

```
source3/librpc/idl/smbXsrv.idl
```

and these are the corresponding new databases:

```
smbXsrv_session_global.tdb
smbXsrv_tcon_global.tdb
smbXsrv_open_global.tdb
```

Now the separation between the SMB layer and the VFS layer is clearer. A couple of the old structures such as `connection_struct` and `user_struct` have been unburdened and moved to the VFS layer, and the old `sessionid.tdb` and `connections.tdb` databases are gone. With a lot of work required on the nitty-gritty details, this was the basis for the implementation of durable handles.

Low-Hanging Fruit

Based on the introduction of the smbXsrv system, a few tasks have become relatively easy and straightforward to implement.

1. *Session reconnect.* This is actually part of the durable handle implementation, but more on the client behavior side and not strictly part of the durable handle negotiation.
2. *Dynamic re-authentication.* This item of SMB 2.1 permits a client to proactively reauthenticate its session.
3. *Multi-Credit.* This feature of SMB 2.1 allows the client to consume multiple data units (so-called credits) in a single SMB request, resulting in a reduced number of network round trips required for the same high-level copy operations.

SMB in Samba 4.0

Samba 4.0.0 has been released on December 11, 2012. This is the first version of Samba shipping with the long-awaited Active Directory domain controller. That is, Samba 4.0 provides all the components required for an Active Directory server, most prominently LDAP, Kerberos, and DNS, along with a whole set of RPC (remote procedure call) services. In contrast to the original plans of the Samba4 project, this release is made possible by the combination of the Active Directory server part with the advancement of the file server of the Samba3 releases. The original plans of the Samba4 project were to complete the file server of the Samba4 code base, but with the limited developer resources concentrated on the directory features while the production-proven Samba3 file server grew and matured. So the 4.0 release is also the direct continuation of the Samba 3.x file server releases. And for pure file-serving purposes, one can configure and run 4.0 in exactly the way familiar from version 3, omitting the Active Directory part.

Although 4.0 is clearly the Active Directory server release of Samba in the public perception, the changes discussed in this article also make it a big and important file server release.

With the support for durable file handles, Samba 4.0 now ships with full SMB 2.0 support. SMB 2.1 is supported, with the omission of leases, resilient handles, and branch cache. Basic support for SMB 3.0 is also provided, and this includes the new cryptographic algorithms, secure negotiation, and the new versions of the durable handle requests. All missing features are negotiable capabilities that a server need not offer, and hence Samba 4.0 is a correct SMB 3.0 server. Furthermore, these changes lay the foundation for further SMB 3 development currently in preparation.

The maximum SMB version that the server will offer can be controlled with the configuration parameter `max protocol`. The default is set to `SMB3`, but older protocol versions can be chosen with values such as `SMB2`, which is a synonym of `SMB2_10` (i.e., SMB 2.1), `SMB2_02` (i.e., SMB 2.0), and `NT1` (i.e., SMB 1). The full details can be found in the `smb.conf(5)` manual page.

The durable handle feature can be turned on or off in the configuration via the parameter `durable handles`. The default is turned on, but this will only be effective if Samba's means for interoperability have been disabled. The reason for this is that with the current implementation, external access to a disconnected durable handle (e.g. with NFS or AFP protocol) would not be noticed, hence activating durable handles in that multi-protocol situation would open the door for data corruption. More concretely, the following settings in the configuration file `smb.conf` activate durable handles in Samba 4.0:

```
[global]
        durable handles = yes
        kernel oplocks = no
        kernel share modes = no
        posix locking = no
```

The remaining sections of this article describe the plans for the further development of SMB 2.1 and 3.0 features that are currently in preparation.

Leases

One of the interesting features of SMB 2.1 is leasing. As mentioned above, leases can be seen as SMB oplocks done right. Leases and oplocks are modes for caching data operations, opens, and closes on files. In principle, there are three primitives of caching: *read caching* allows caching of data read operations, *write caching* allows one to cache data writes and byte range locks, and *handle caching* allows caching of open and close operations.

The traditional SMB oplocks know three combination of these: *level 2 oplocks* provide read caching, *exclusive oplocks* provide read and write caching, and *batch oplocks* provide read, write, and handle caching. Leases come in four flavors: read leases, read handle leases, read/write leases, and read/write handle leases.

One important change is that by virtue of a so-called *lease key* that identifies the lease, clients can, in contrast to the case of oplocks, upgrade their caching mode without revoking their original lease. Furthermore, the maximum sharable caching mode is changed from read to read and handle, which is implemented by the new read handle leases.

Finally, SMB 3.0 introduces a new concept of *directory leases*: a lease on a directory allows the client to cache metadata operations on files in that directory in contrast to the data operations cached by leases on files.

Leases, including directory leases, will be implemented in Samba relatively soon. Based on the preparatory work on the `smbXsrv` system, the necessary steps are rather clearly arranged. The additions on the SMB protocol level are mostly obvious. The more subtle changes are required at the FSA layer. The developers have designed extensions to the format of the `locking.tdb` entries to cope with the additional caching modes on that level, so that both the semantics for oplocks and leases can be covered. The main work will lie in the extension of the existing cache handling routines, to reflect the broader cache revocation matrix. Although there is a certain level of protocol interoperability for SMB oplocks due to the so-called Linux kernel leases, this interoperability is already far from perfect, and for leases the offered semantics are simply too different. Therefore, as in the case of durable handles, Samba will probably not start off being able to offer leases in an interoperable environment, and definitely not for directory leases.

More SMB 3

The Samba developers are currently planning and designing the implementation of the large list of features of SMB 3. The remaining part of this paper describes the current plans to implement some of the most compelling features.

Clustering

From Samba's perspective, Windows finally embracing active-active clustering is very exciting. The most interesting question to start with is how well this can be integrated with Samba's CTDB clustering. Fortunately, initial investigations indicate that the concepts introduced by Windows are either orthogonal to Samba's clustering or capable of being integrated quite nicely. SMB 3.0 offers three clustering capabilities that can be attached to a share:

1. *Cluster:* The availability of the share can be monitored with the Witness service, an RPC service described in the [MS-SWN] document at [8]. The Witness service also allows the client to be notified of network interface changes. This is to speed up failovers in a highly available server. With the current state of research and testing, the Witness service will integrate cleanly with the CTDB clustering.
2. *Continuous availability:* This share allows for transparent failover of SMB clients. That is, in case of planned or unplanned outage of a cluster node, the SMB client is guaranteed to be able to reconnect to a different node without interruption of I/O, so the applications using the SMB file share will not notice. This

is based on certain retry and replay concepts, and it is the foundation and the prerequisite for persistent file handles.

3. *Scaleout:* This is the all-active nature of shares, that is the share is available on all nodes of the cluster simultaneously. It increases the available accumulated bandwidth for a share. Load can be balanced across the cluster nodes, in particular administratively triggered using the transparent SMB failover. The all-active nature also enables faster recovery of durable handles. Because the scale-out characteristic is the basis of Samba's CTDB clustering model, the understanding is that CTDB will enable building scale-out SMB3 clusters with Samba. The details of this are currently being worked out.

The bottom line is that SMB 3.0 adds clustering capabilities that CTDB clusters also have, but without client awareness. With SMB 3.0, the server introduces clustering infrastructure, while the main logic for failover and retry is in the client. This fact is the main reason that this will integrate with CTDB, so Samba developers will not need to invent a completely new clustering model.

Multi-Channel

As mentioned above, multi-channel is a mechanism for the client to bundle multiple transport connections (channels) into a single SMB session. The I/O load is spread across the channels, and the session is more robust against network failures: a session is intact as long as there is at least one intact channel on the session. Multi-channel is also the basis for RDMA support. The prerequisite for channel bundling is the new interface discovery control that the server has to offer. Based on the information this control delivers, the client decides which interfaces it uses.

The starting point for implementing multi-channel in Samba is to give up the assumption that smbd processes correspond bijectively to TCP connections to the SMB ports. The plan is to transfer TCP connections that belong to the same client to a common process by applying a technique called fd-passing to the TCP-socket. This transfer will be done at protocol negotiation time based on the client identifier called Client GUID. There are concrete plans for most of the details, but the complete implementation of multi-channel in Samba will be a rather large task, since the infrastructure for finding a server based on the Client GUID and passing a connection from one server to another needs to be created. Furthermore, the assumption that one smbd serves only one TCP connection is currently rooted firmly in the server code, and this needs to be carefully removed.

Persistent Handles

Persistent file handles are like durable file handles but with guarantees, whereas durable handles are a best-effort concept. This kind of file handle is only offered on continuously available shares. The SMB protocol mechanisms for obtaining and reconnecting persistent handles are in principle already available in Samba 4.0, although not activated: they are treated by special flags in the SMB3 version of the durable request. The difficult part of the implementation are the additional guarantees attached to persistent handles. In contrast to the durable handles, whose pieces of information are stored in the usual volatile databases, information for persistent handles will need to be stored persistently, so that a disconnected persistent handle will, for instance, survive a server restart. The details of the implementation are not designed yet, but this will definitely come with a performance penalty regarding opening files. But this is expected and accepted for persistent handles; these are targeted at server workloads such as Hyper-V and SQL, which don't open and close files at a high frequency but rather work on a small number of open-end files for a long time, the important thing being that the open cannot be lost.

SMB over RDMA

The final SMB3 topic to be touched here is *SMB Direct,* the variant of SMB 3.0 that can use RDMA-capable transports such as Infiniband or iWARP-capable 10G Ethernet adapters for the data transfers. SMB direct uses a normal TCP connection as the initial connection, and this is always used for the protocol head. Then multi-channel is used to bind an RDMA-capable channel to the session for the payload data in reads and writes. This is a topic where much research is still needed, because it requires integration with special RDMA-capable network hardware since Windows does not offer a pure software iWARP implementation as Linux does. The transport abstraction is already largely designed and prototyped, but the work does not stop there; existing iWARP client libraries `libibverbs` or `librdmaca` must be integrated. They are currently not fork-safe, and they can't be used with fd-passing, which is the proposed mechanism for multi-channel session binds. So Samba needs some changes in such libraries.

Conclusion

Samba 4.0 will be an exciting release, not only because of the new Active Directory server component, but also as a file server release. The new release features basic SMB 3.0 support, and comes with support for durable file handles as the big new feature. The developers are currently busy designing and even starting the implementation in Samba of leases and many features of SMB 3, most notably support for scale-out and continuously available shares, multi-channel, and persistent handles. The main goal is to enable full support for SMB 3.0 clustering, e.g., running Hyper-V on a Samba-CTDB cluster with one of the next releases (4.1 or 4.2) of Samba.

References

[1] Samba, the open source SMB server for UNIX: http://www.samba.org/.

[2] The SNIA Storage Developer Conference 2011: http://www.snia.org/events/storage-developer2011.

[3] TDB, Samba's trivial database: http://tdb.samba.org/.

[4] CTDB, the clustered TDB project: http://ctdb.samba.org/.

[5] Michael Adam, "Clustered NAS for Everybody: Clustering Samba with CTDB," http://www.samba.org/~obnox/presentations/sambaXP-2009/samba-and-ctdb.pdf.

[6] MSDN Library, Open Specifications: Windows Protocols: http://msdn.microsoft.com/en-us/library/ms123401.aspx.

[7] MSDN Library, [MS-SMB2]: Server Message Block (SMB) Protocol Versions 2 and 3: http://msdn.microsoft.com/en-us/library/cc246482%28v=prot.20%29.aspx.

[8] MSDN Library, [MS-SWN]: Service Witness Protocol: http://msdn.microsoft.com/en-us/library/hh536748%28v=prot.20%29.aspx.

Temperature Management in Datacenters

Cranking Up the Thermostat Without Feeling the Heat

NOSAYBA EL-SAYED, IOAN STEFANOVICI, GEORGE AMVROSIADIS, ANDY A. HWANG, AND BIANCA SCHROEDER

Nosayba El-Sayed is a PhD student in the Department of Computer Science at the University of Toronto, working under the supervision of Professor Bianca Schroeder. Her research focuses on improving the reliability and energy-efficiency of large-scale systems. Nosayba received her BS and MS in computer engineering from the University of Kuwait. nosayba@cs.toronto.edu

Ioan Stefanovici is a PhD student in the Computer Systems and Networks Group at the University of Toronto under the supervision of Professor Bianca Schroeder. His research deals primarily with improving the reliability and performance of large-scale computer systems. He also has industry experience working at Microsoft, Google, and IBM Research. ioan@cs.toronto.edu

George Amvrosiadis is a PhD student in computer science at the University of Toronto. His research interests include storage reliability and idleness characterization, detection, and utilization. He completed his BS at the University of Ioannina in Greece, with a thesis on namespace management of federated file systems. gamvrosi@cs.toronto.edu

Andy A. Hwang is a PhD student at the University of Toronto under the supervision of Professor Bianca Schroeder. He received his BAS from the University of Waterloo and MS from the University of Toronto. Andy works on enhancing the reliability and performance of computer systems, both as a whole and for specific components such as DRAM. hwang@cs.toronto.edu

Bianca Schroeder is an assistant professor in the Department of Computer Science at the University of Toronto. Before coming to Toronto, Bianca completed her PhD and a two-year post-doc at Carnegie Mellon University. Her research focuses on computer systems, in particular the reliability of large-scale systems and the empirical analysis of computer systems. bianca@cs.toronto.edu

Datacenters have developed into major energy hogs, and more than a third of this energy is spent on cooling. While estimates suggest that increasing datacenter temperature by just one degree could reduce energy consumption by 2–5%, the effects of increased temperature on server components are not well understood. In this article, we present results from a large-scale field study, which demonstrate that there is ample potential for increasing datacenter temperatures without sacrificing system reliability or performance.

Datacenters currently account for nearly 3% of the world's electricity consumption, and the annual cost for a single cooling system can be as high as $8 million. Not surprisingly, a large body of research has been devoted to reducing cooling cost. Interestingly, one key aspect in the thermal management of a datacenter is still not well understood: controlling the setpoint temperature at which to run a datacenter's cooling system. While Google and Facebook have begun increasing temperatures in some of their datacenters, most organizations are typically more conservative, operating their datacenters in the 20°C to 22°C range, some as cold as 13°C degrees [2, 6]. Setting datacenter temperatures is more of a black art than a science, and many operators rely on anecdotal evidence or manufacturers' (conservative) suggestions, as there are no field studies detailing the effects of temperature on hardware components. In this article, we present results from our recent work [4] on the impact of temperature on component reliability and performance based on large-scale field data from multiple organizations and on experiments we conduct in a lab using a heat chamber. We also discuss several other concerns related to increased datacenter temperatures.

Temperature and Reliability

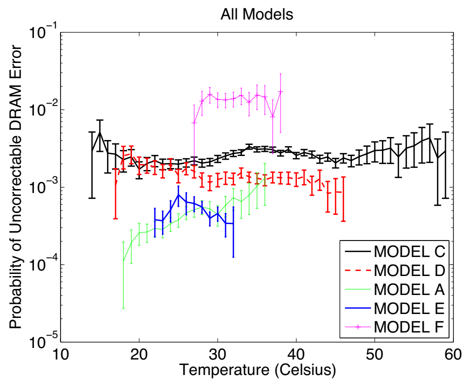


Figure 1: Probability of uncorrectable DRAM errors at Google as a function of temperature

Our study of the effect of temperature on system reliability focuses on two specific hardware components, hard disk drives and DRAM, as these are among the most frequently replaced components in today's datacenters. We also study the impact of temperature on the overall reliability of a node.

Errors in DRAM

We study the effect of temperature on two different DRAM error modes: correctable errors (CEs), where bits on a DRAM chip are flipped but can be corrected with error correcting codes (ECC), and uncorrectable errors (UEs), where the number of erroneous bits is too large for the ECC to correct, causing an application or machine crash. In many environments, UEs affect component lifetimes, as a single UE is often considered serious enough to replace the component.

Our largest data source comes from datacenters at Google, covering five different hardware platforms, and includes per DIMM counts of the occurrence of uncorrectable errors, as well as periodic temperature measurements based on sensors on the motherboard. Figure 1 shows the monthly probability of an uncorrectable DRAM error per DIMM as a function of the average monthly temperature.

Interestingly, we find that error rates are mostly flat with temperature, with the exception of one model (model A). We find that when further breaking down the data for model A and looking at the trends for individual datacenters, error rates are flat with temperature except for one outlier datacenter that exhibited a slightly increasing trend.

We performed a similar analysis on data collected at Los Alamos National Lab (LANL) on node outages that were due to DRAM problems and on data collected at the SciNet Consortium (from the largest supercomputing cluster in Canada) on DIMM replacements. Again, we found no correlation between higher temperatures and increased DRAM failure rates.

We had also looked at the impact of temperature on *correctable* errors in an earlier study [7] based on Google data and, again, found no evidence of a correlation between temperature and correctable errors in DRAM.

Observation: We do not observe evidence for increasing rates of uncorrectable DRAM errors, correctable DRAM errors, DRAM DIMM replacements, or node outages caused by DRAM problems as a function of temperature (within the range of temperatures our data comprises).

Latent Sector Errors in Hard Disks

We concentrate our study of hard disk reliability on two common disk failure modes: latent sector errors (LSEs), where individual sectors on a disk become inaccessible, and complete disk failures. Both failure modes occur at a significant rate in the field, posing threats to data safety.

Our data on latent sector errors was collected from January 2007 to May 2009 from seven different datacenters at Google, covering three disk models and a total of 70,000 disks. For each disk, we have monthly reports of the average internal disk temperature (collected from SMART) and the temperature variance.

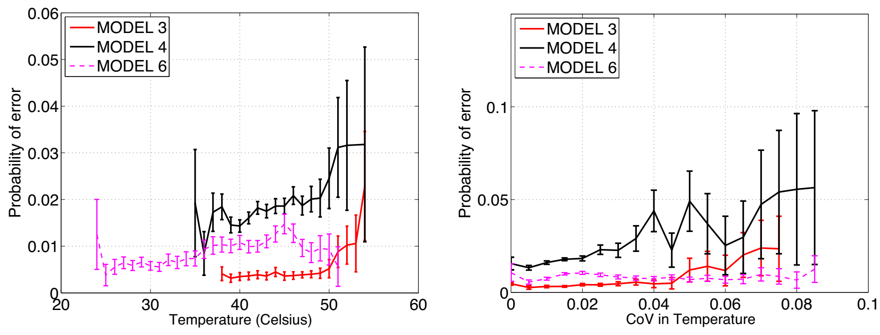


Figure 2: The monthly probability of LSEs by temperature (left) and by coefficient of variation (right)

Figure 2 (left) shows the monthly probability of a disk developing an LSE as a function of temperature for each of the three disk models. We observe a trend of increasing LSE rates as temperature rises; however, an interesting observation is that the magnitude of increase is much smaller than expected based on common models (e.g., the Arrhenius model), which predict exponential growth of hardware failures with temperature. Using curve-fitting techniques, we found that a linear fit was comparable and in many cases even better than an exponential fit to our data.

Observation: The prevalence of latent sector errors increases much more slowly with temperature than reliability models suggest. Half of our model/datacenter pairs show no evidence of an increase, while for the others the increase is linear rather than exponential.

We also study the effect of variability in temperature on LSEs. Figure 2 (right) shows the monthly probability of LSEs as a function of coefficient of variation (CoV: the coefficient of variation is defined as the standard deviation divided by the mean) in temperature. When further breaking down the data by datacenter, we observe the following:

Observation: When further breaking down the data by datacenter, we find that the variability in temperature tends to have a more pronounced and consistent effect on LSE rates than mere average temperature.

Hard Disk Replacements

We consider a hard disk failure as any kind of disk problem that is serious enough to replace the disk in question. We have obtained data on disk replacements and disk temperatures collected from January 2007 to May 2009 at 19 different datacenters at Google, covering five different disk models and 200,000 disks.

When analyzing the monthly probability that a disk will fail as a function of disk temperature, we observe that only three out of the five disk models show any increase in failure rates as temperature increases. Moreover, for these three models, visual inspection as well as results from statistical curve-fitting indicate that the increase in failure rates tends to be linear rather than exponential.

Observation: Only some disk models experience increased failure rates with temperature, and for those models the increase is weaker than what existing reliability models predict.

Other Datacenter-Specific Factors

There are many datacenter-specific factors beyond temperature that might affect reliability (workload, humidity, handling procedures, etc.). We make an interesting observation when separating the data on LSEs by datacenter: LSE rates for the same disk model can vary widely, by a factor of two, for disks in different datacenters. We considered differences in age or usage as possible reasons, but found that neither has a significant correlation with temperature.

Observation: There are other datacenter-specific factors that have a stronger effect on disk reliability than temperature. These could, for example, include humidity, vibration, or handling procedures.

Node Outages

Rather than focusing on a particular hardware component, we also considered overall system reliability as a function of temperature. We study a data set covering all node outages in 13 clusters (a total of 4384 nodes) at LANL recorded between 2001 and 2005, and event logs containing periodic temperature measurements provided by sensors on the motherboard.

The two graphs in Figure 3 show the monthly probability of a node outage for LANL system 20 as a function of the average temperature and the coefficient of variation (CoV) in temperature, respectively. The left graph compares the node outage probability of the coldest 50% of the nodes (left bar) with the node outage probability of the hottest 50% of the nodes (right bar). The right graph compares the node outage probability for the top 50% of nodes with the highest CoV and the bottom 50% of nodes with lowest CoV.

Observation: We observe no evidence that hotter nodes have a higher rate of node outages or node downtime (node downtime graph omitted for space).

Observation: We find that high variability in temperature seems to have a stronger effect on node reliability than average temperature.

We have expanded our analysis to node outages at 12 other clusters at LANL and hardware replacements at a cluster at SciNet and make similar observations. Full results are included in [4].

Other Concerns with Increased Temperatures

While it is widely known that higher temperatures might negatively affect the reliability and lifetime of hardware devices, less attention is paid to other concerns with increased temperatures. The first is the fact that high temperatures can also

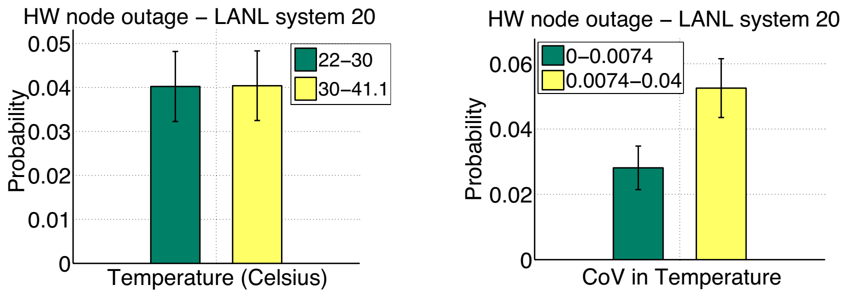


Figure 3: Probability of node outages by temperature (left) and by coefficient of variation (right)

negatively affect the *performance* of systems. Many server components employ a variety of mechanisms that will activate at high temperatures to protect themselves against temperature-induced errors and failures. These mechanisms can introduce overheads, potentially affecting server performance. Other concerns include the effect of higher temperatures on a server's power consumption and potential hot spots in a datacenter. We discuss each of these below.

Performance of Hard Disk Drives

We begin with a study of the effect of temperature on hard disk drives. It has been suggested that in order to protect themselves against a possibly increasing rate of LSEs, some hard disk models enable read-after-write (RAW) when a certain temperature threshold is reached. Under RAW, every write command sent to the disk is followed by a verify operation, which will read back the sector that has just been written and verify its contents. Unfortunately, features such as RAW are often considered trade secrets and their associated parameters (or even existence) are not well documented. In fact, even within a company manufacturing hardware these features are regarded as confidential and not shared outside product groups.

As a result, we decided to investigate experimentally how the performance of different components changes with increasing temperatures using a testbed based on a heat chamber. We equip a Dell PowerEdge R710 server, a model that is commonly used in datacenter server racks, with a variety of hard disk drives, including SAS as well as SATA drives, covering all major manufacturers. We run a wide range of workloads, including synthetic benchmarks and a set of macro-benchmarks, while using the heat chamber to vary the ambient temperature within a range of 10°C to 55°C.

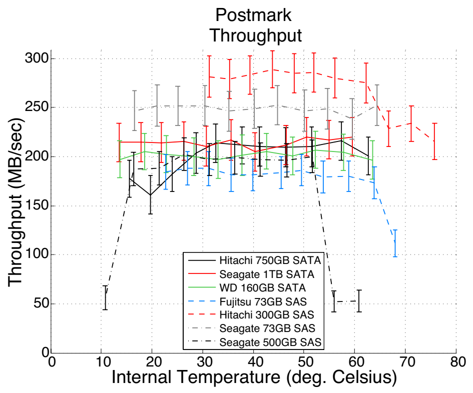


Figure 4: Throughput of an I/O-intensive workload (Postmark) as a function of disk internal temperature.

Figure 4 shows the throughput (in MBps) for one sample workload, the Postmark file system benchmark, as a function of the drive internal temperature as reported by the drive's SMART statistics; the observations below, however, summarize our findings across all workloads (see paper [4] for all results).

Observation: All SAS drives and one out of the three SATA drives experience some drop in throughput for high temperatures. The drops are typically in the 5–20% range, sometimes as high as 40–80% for certain models running disk-intensive workloads.

Observation: Because for a particular drive model the throughput drop happens consistently at the same temperature (between 50°C and 60°C disk-internal temperature, depending on the drive model), and none of the drives report any errors, we speculate that the drop in throughput is due to protective mechanisms enabled by the drive.

Observation: We also observe throughput drops for read-only workloads, suggesting the presence of other non-publicized protection mechanisms besides RAW.

Observation: When translating the drive-specific internal temperatures into ambient temperatures (inside the heat chamber), we observe a drop in throughput for temperatures ranging from 40°C to 55°C, depending on the drive model. Although datacenters will rarely run at an average inlet temperature above 40°C, most datacenters have hot spots, which might routinely reach such temperatures.

Performance of Other Server Components

Most enterprise-class servers include features to protect the CPU and memory subsystems from damage or excessive error rates due to high temperatures. These include scaling of the CPU frequency, reducing the speed of the memory bus (e.g., from 1066 MHz to 800 MHz), and employing protection mechanisms for DRAM (e.g., SEC-DED ECC, Chipkill ECC, memory mirroring). During experiments where we placed the server in a heat chamber, we did not observe that our server model enabled any of these features automatically. When manually enabling varying protective features, we find that they can introduce reductions in throughput by as much as 50% for microbenchmarks that stress the memory system, and 3–4% for macrobenchmarks modeling real-world applications.

Server Power Consumption

Increasing the air intake temperature of IT equipment can have an impact on the equipment's power dissipation. Leakage power of a processor increases with higher temperatures. Additionally, most IT manufacturers start to increase the speed of internal cooling fans once inlet air temperatures reach a certain threshold, to offset the increased ambient air temperature. Together, these can make up a significant fraction of a server's total power consumption. To study the effect of increasing ambient temperatures on a server's power consumption, we attached a power meter to our server (results shown in Figure 5 [left]) and monitored fan speeds (see Figure 5 [right]) while placing the server in our heat chamber and running a variety of different workloads.

Observation: The server's total power consumption increases dramatically (by more than 50%) in ambient temperatures over 40°C.

Observation: Based on our measurements of server fan speeds during the experiments and after consulting manufacturer's specifications for the server's fans, we can attribute the majority of the server's additional power consumption to the increased fan speeds, rather than leakage power.

Observation: Interestingly, we find that the server fan speeds seemed to increase solely as a function of ambient temperature, irrespective of the server's internal temperature, suggesting the need for smarter fan controllers.

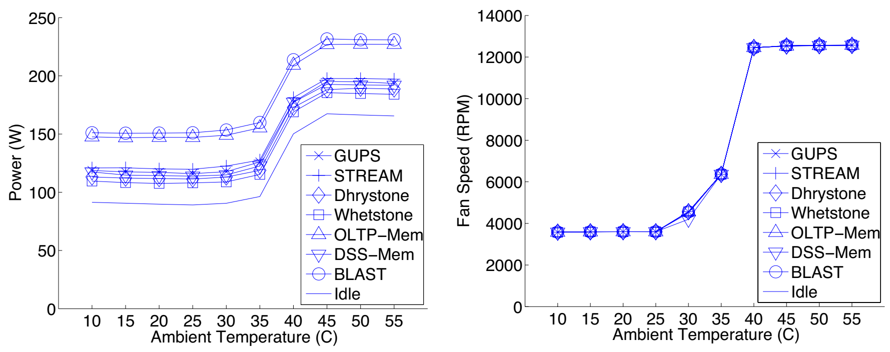


Figure 5: The effect of ambient temperature on power consumption (left) and server fan speeds (right)

Reduced Safety Margins

One final concern with increasing datacenter temperatures are hot spots: portions of a datacenter that are significantly hotter than the average room temperature. The concern is that as the average temperature in a datacenter increases, the hot spots will approach critical temperature thresholds at which servers are configured to shut down in order to avoid equipment damage. This reduces the amount of time available to shut down a server cleanly, in case of an event such as AC or fan failure. We used data from seven datacenters at Google and a 256-node cluster at LANL to study variations in temperature between different servers in the same datacenter.

Observation: Interestingly, the trends for temperature imbalances are very similar across datacenters and organizations. The node/disk in the 95th percentile is typically around 5°C hotter than the median node/disk, and the 99th percentile is around 8–10°C hotter than the median node/disk.

Lessons Learned

Based on our study of data spanning more than a dozen datacenters at three different organizations, and covering a broad range of reliability issues, we find that the effect of high datacenter temperatures on system reliability is smaller than often assumed. For some of the reliability issues we study, namely DRAM failures and node outages, we do not find any evidence for a correlation with higher temperatures (within the range of temperatures in our data sets). For those error conditions that show a correlation (latent sector errors in disks and disk failures), the correlation is much weaker than expected. These observations imply that there is ample room for increasing datacenter temperatures without sacrificing system reliability.

Rather than average temperature, the variability in temperature might be the more important factor. Even failure conditions, such as node outages, that do not show a correlation with temperature, do show a clear correlation with the variability in temperature. Efforts in controlling such factors might be more important than low average temperature in keeping hardware failure rates low.

We find evidence that other datacenter-specific factors (such as humidity, vibration, handling procedures) are likely to have a stronger, or at least an equally strong, effect as temperature. Although we do not have sufficient data for a detailed study of these factors, anecdotal evidence from discussions with datacenter operators suggests, for example, that poor handling procedures for equipment are major factors in the field. Our observations demonstrate the need for more work in this area.

Do our results mean that common models (e.g., the Arrhenius model) that predict exponential growth in failure rates with temperature are wrong? We think no. Instead, it is likely that in practice (and for realistic temperatures) the effects of other factors dominate failure rates. The Arrhenius model solely tries to capture the effect of heat on hardware components without taking into account other possible factors that impact hardware reliability in the field. Our results indicate that, when all real-world factors are considered, the effect of temperature on hardware reliability is actually weaker than commonly thought.

The error mode that was most strongly correlated with high temperatures is latent sector errors (LSEs) in hard disk drives. In experiments with our testbed based on a heat chamber, we observe that not all hard disks employ mechanisms, such as read-after-write, to protect against increases in LSEs under high temperature.

For those that do (mostly enterprise-class drives), we find that they tend to kick in only at very high temperatures and are associated with significant performance penalties. Operators concerned about LSEs might want to implement independent protection mechanisms under high temperatures, such as “scrubbing” their data at increased rates.

We find that higher temperatures can raise a server’s power consumption by more than 50%, and attribute the additional dissipation to increased fan speeds through our experiments. In many cases, this could likely be avoided by employing more sophisticated fan controller algorithms. This suggests that smarter fan controllers are needed to run datacenters hotter.

We find that in a typical datacenter, the top 5% of nodes are 5°C hotter than the median temperature, whereas the top 1% of nodes are 8–10°C hotter than the median. This is important to keep in mind when raising datacenter temperatures, as it will bring these hot spots even closer to critical thresholds when thermal shutdown becomes necessary. Operating at higher temperatures will therefore require mechanisms (including, for example, a detailed temperature monitoring infrastructure) to detect and react quickly to unforeseen events, such as AC or fan failures.

References

[1] C. Belady, A. Rawson, J. Pfleuger, and T. Cader, “The Green Grid Datacenter Power Efficiency Metrics: PUE & DCiE,” technical report, Green Grid, 2008.

[2] J. Brandon, “Going Green in the Datacenter: Practical Steps for Your SME to Become More Environmentally Friendly,” *Processor,* vol. 29, Sept. 2007.

[3] California Energy Commission, “Summertime Energy-Saving Tips for Businesses”: consumerenergycenter.org/tips/business_summer.html.

[4] N. El-Sayed, I.A. Stefanovici, G. Amvrosiadis, A.A. Hwang, and B. Schroeder, “Temperature Management in Datacenters: Why Some (Might) Like It Hot,” in *Proceedings of the Fourteenth International Joint Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS ‘12), ACM, 2012, pp. 163–174.

[5] Lawrence Berkeley National Labs, Benchmarking—Datacenters: http://hightech.lbl.gov/benchmarking-dc.html, December 2007.

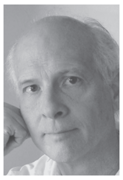
[6] Rich Miller, “Google: Raise Your Datacenter Temperature”: http://www.datacenterknowledge.com/archives/2008/10/14/google-raise-your-data-center-temperature/, 2008.

[7] B. Schroeder, E. Pinheiro, and W.-D. Weber, “DRAM Errors in the Wild: A Large-Scale Field Study,” in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS ’09), ACM, 2009, pp. 193–204.

Allen Wittenauer on Hadoop

An Interview

RIK FARROW



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

Allen Wittenauer is currently the Senior Grid Computing Architect at LinkedIn, Inc. He has been working with Hadoop with an eye toward operability for almost six years. aw@apache.org

I have wanted to run an article about Hadoop performance for a while, and my search for an expert finally paid off. Justin Sheehy, CEO of Basho, suggested that I interview Allen Wittenauer of LinkedIn.

I had naively thought that the way to improve, or at least maintain, Hadoop performance was to monitor the performance of the cluster, but Allen, with experience as both a Hadoop developer and large cluster operator, has very different thoughts on the subject.

I started out by reading the slides of a old presentation that Allen wrote called "Hadoop 24/7". Among his suggestions were to run the checks `hadoop dfsadmin -fsck` and <t>`hadoop dfsadmin -report` nightly, and to use an NFS backup of the NameNode file system image. You can find an updated version of his presentation at https://www.usenix.org/sites/default/files/login-1302-wittenauer-slides.pdf.

Rik: Let's start with some information about your background as it pertains to Hadoop. Looking at your LinkedIn references, I can see you worked for Sun, then Yahoo!, and now LinkedIn. Also, you are a Hadoop committer, which implies a lot about your ability to talk about Hadoop. Can you tell me more?

Allen: After a few years working at a software company that supported hospitals, I wanted to get involved with bigger scale problems. A friend of mine said they were hiring at Sun and would I be interested. "The network is the computer"… yes! I was lucky enough to get hired and moved up the ranks. As a result, I had the privilege of bringing order to the chaos that organic growth brings at larger and larger scales. Three such problems were building a single NIS domain to handle 100,000+ hosts spread across the entire SF Bay Area, OS provisioning on an intercontinental scale, and building a world-wide, single realm Kerberos deployment.

A layoff and a short stint at a startup later, I found myself talking to Yahoo! about Hadoop. They were interested in the same sorts of problems, but with a view toward sharing the solution with the world at large. Many companies solve these problems using home-grown tools; however, when one is trying to build a community around what was mostly (at the time) unproven technology, it is vital that you show a solution that they themselves can use. I was brought in to "rethink" Yahoo!'s operational infrastructure with an eye toward open source, what I would call "burning the house down." It was very "startup"-like, so while that's what I was hired for, I ended up doing a lot more than that and had the honor of directly impacting some of the key components in Hadoop, especially as it relates to operations.

Rik: In the earlier versions of your slides, you mentioned using `hadoop` commands nightly to check the health of the underlying HDFS. Have these suggestions changed over time?

Allen: The basics are still the same, but Hadoop has greatly matured in the past four years since this presentation was completed. While doing at least a nightly fsck to check for health is still important, there is a lot more information now available to use for basic monitoring and metrics collection via the normal Hadoop metrics plugins, JMX and JSON. In addition to CPU usage, network usage, etc., we collect a lot of that extra Hadoop data plus some additional ones such as disk service times to build a macro view of the overall system's health. While the metrics Hadoop provides tend to be extremely low level, just keeping track of values such as "tasks completed" can be useful to see if the overall grid is getting faster/slower.

One of the key points about monitoring is that you want to check the health of the service. I recommend having a tiny job that runs every 15 minutes as a canary to verify everything is working. In addition, alerting on percentage of down nodes vs. on individual node failures becomes increasingly important as you scale up. That's a common mistake when people first start working with Hadoop. We're all trained from the beginning that every machine is important because most services are fragile. If the compute nodes are properly provisioned, losing a few shouldn't matter. It's a hard concept to unlearn.

Rik: You also mentioned issues with the NameNode, which has always (for me) been the scariest part of HDFS.

Allen: From an Enterprise view of the world, the fact that single points of failure exist in the system at all is shocking. From an HPC/scientific view, not as much. I think people forget that for a long time Hadoop was being built in a "two guys in a garage" fashion to solve fairly specific problems. There wasn't a priority placed on five-9s of uptime for a system that was geared toward batch computation. Scaling up to petabytes of data and fixing performance issues related to processing that large of a data set were the priorities. It was acceptable to have, say, an hour of downtime in the case of a failure. Now that many more people are interested in using HDFS for real-time access with technologies like HBase and the ever increasing commercial interest, the focus has shifted. The 2.x branch of Apache Hadoop will include high availability of the NameNode.

Something else to consider: using something like Xen's Live Migration or another HA solution—Linux HA, SunCluster, whatever—is also an option. Administrators shouldn't be afraid of trying to apply other methods.

That said, I think too much focus is placed on this limitation. In the almost six years, 30+ grids, 30k hosts, and two companies where I've run Hadoop, the number of times where a highly available NameNode would have saved me from service downtime can be counted on one hand. Almost all cases of NameNode failures are configuration problems or bad user behavior, none of which high availability will actually help prevent from a downtime perspective. Additionally, I find it somewhat odd that no one seems to be too concerned about the lack of availability for the JobTracker. A file system with nothing running on it isn't very useful.

Rik: You have suggestions in your slides [1] for setting up the NameNode (mostly lots of memory, as that is key to performance), and recovering a NameNode using an NFS replica of the image. As I want to focus on troubleshooting performance issues, I am more interested in maintaining the health of the NameNode than recovering it. But writing the change log to an NFS mounted file might be a performance issue. What other performance issues should people running a NameNode be aware of? Do you still recommend using NFS for the backup change log?

Allen: Until the 2.x branch is stabilized, yes, I do. At LinkedIn, we actually store our primary NFS copy on the secondary NameNode. At Yahoo!, we had some older NetApp boxes that were too small for another project that we repurposed for this task. The number of IOPS obviously scale to how busy a particular HDFS might be, but in most cases this is relatively light. Even so, it is important to make sure that the NFS server is nearby on the same network core since you don't want too much latency. Another key point is that the NameNode can work off of only one fsimage and edits file. If the primary machine has a file system failure where the image is stored, it will continue to run off of the NFS copy.

Rik: How does a sysadmin go about uncovering performance problems in a Hadoop cluster? You've said that maintaining the health of HDFS is one key, and the NameNode is another. But there may be other causes of big (noticeable) drops in performance when executing a job that gets done every day.

Allen: One of the most important actions that users can do is take a holistic view of the work they are trying to accomplish. Many, many times users will tune individual jobs, making them as fast as possible so that they can iterate during development quickly. In the process, they hit an anti-pattern in the production phase where the parallelization is too high to the point that the entire workflow is performing worse. For example, increasing reducers to the point that many small files are generated has an extremely negative network impact on the next phase of the MapReduce job that is going to read those files, either during the read of the input or in the shuffle phase.

I'm also a big fan of getting your hands dirty at the micro level. When dealing with large, scalable systems, the temptation is high to look at your metrics for the entire system and say everything looks fine. What you miss out on is that you might have one job or an internal framework or something else that has bad behavior. Averaged out, that's easily missed. But dropping down to the shell and just getting a feel for what is happening on a per-node basis is extremely helpful. Years ago, we shaved hours of processing from some very important workflows at LinkedIn by discovering a hidden bug in a commonly used internal framework. It wasn't properly caching data from an external hint file and in turn was triggering an extra I/O on one of the six disks for every record read. From the macro level, it was averaged away but dropping down, and using tools like truss, dtrace, iostat, and sar made it really stick out.

Rik: What about tunables? I hear that there are a ton of them and they almost all have an impact on performance.

Allen: Yes, there are an amazing assortment of settings that can be applied to a job. The running joke in the early days was that we'd hit a particular corner case and have to add something to the configuration to tackle that problem. Unfortunately, there is no magic bullet to know which particular tunable is the correct one to modify. This is made worse by the fact that some tunables, such as io.sort.mb, have a direct impact on how other tunables will operate. Today, the best bet is to use the job and system metrics to determine whether the setting you've changed made a difference. Usually one of the biggest mistakes people make here is to just throw more heap at the problem. In most cases, this is the wrong thing to do. So that should be the last thing that is changed.

I'm excited about tools like Duke University's Starfish project (http://www.cs.duke.edu/starfish/). We've been doing cooperative research in the hopes that exposure to "real world" conditions will improve its suggestions. In the future, this means that we will have some utilities to automatically tune workflows for nearly maximum performance.

Of course, bad algorithms will still be bad no matter how much tuning one does.

Rik: In your slides, you mention there is no real security in Hadoop, and I've seen postings to this effect. I've also read a posting by you mentioning adding Kerberos support. That means adding another dependency to Hadoop, although one that most orgs can easily support if they have AD or Samba 4, so perhaps this is not really a performance issue. But maybe I am wrong, and having an overutilized AD server or congested network link to it could slow down a Hadoop cluster..

Allen: There was a great debate about how do we secure Hadoop. For quite a while, Hadoop didn't even have the concept of permissions or even of different users. Those were only added to prevent users from accidentally deleting each others' data. There was always the thought that "we should secure the system," but other priorities prevented that from happening. Eventually, business realities forced the issue and we had the challenge of how do you secure a highly scalable service while also making it perform? We knew we didn't want the system to prompt for a password and then pass that around. Clearly we needed something that was single sign-on, the holy grail. There was a big debate around using x.509 certificates/PKI vs. Kerberos. Ultimately, the decision came down to implement SASL so that people could build their own solution if necessary, but we were going to go with built-in support for GSSAPI with Kerberos.

A big chunk of that decision was exactly as you stated: most places have Kerberos in the form of Active Directory, even if they don't know it. That takes care of the authentication portion, but now what about scale? It was decided to use a token mechanism so that individual daemons wouldn't be an inadvertent DoS against the KDCs. In this way, your Kerberos credential is used by your job client against individual services, that client gets a token, and then that token is used throughout the rest of your job's lifecycle to access other parts of the system without impacting the KDC. That takes care of overextending a potentially overutilized AD server.

Reality, however, is a bit different. Internal politics for most organizations likely make this less cut and dried. If user accounts, and therefore their AD credentials, are controlled by IT, does that same organization really want to hand out Kerberos keytab files for potentially thousands of machines that they don't control? Probably not. So what ends up happening is that companies do a one-way trust so that IT still owns the corporate infrastructure and the "other" organization (Web operations, DBAs, development, whoever) can tie their Hadoop systems to a local Kerberos infrastructure. Users still have one password (which makes them happy), the Hadoop operations team has control over their boxes (which makes them happy), and all of your data is nicely secured (which makes everyone happy). As an added bonus, this also helps balance out the performance implications as the Hadoop systems will mostly be hitting the local KDCs.

Rik: This is great information. Is there anything else you'd like to add?

Allen: Just a big thank you to the wonderful services and information that USENIX and LISA have provided over the years. I'm truly honored to be able to contribute back to the technical excellence that these organizations represent.

Kadeploy3

Efficient and Scalable Operating System Provisioning for Clusters

EMMANUEL JEANVOINE, LUC SARZYNIEC, AND LUCAS NUSSBAUM

Emmanuel Jeanvoine is a Research Engineer at Inria Nancy Grand Est. He specializes in distributed systems and high performance computing. In 2007, he obtained a PhD in computer sciences at Université de Rennes 1. emmanuel.jeanvoine@inria.fr

Luc Sarzyniec is a Junior Engineer at Inria Nancy Grand Est. He started working on distributed systems and high performance computing after he obtained his master of research in computing sciences in 2011. luc.sarzyniec@inria.fr

Lucas Nussbaum is an Assistant Professor at Université de Lorraine. His research focuses on high performance computing and distributed systems. He is also involved in a professional curriculum focusing on system administration. lucas.nussbaum@loria.fr

Installing an operating system can be tedious when it must be reproduced on many computers, on large scale clusters, for instance. Because installing the nodes independently is not realistic, disk cloning or imaging with tools such as Clonezilla [1], Rocks [5], SystemImager [6], or xCAT [8] is a common approach. In those cases, the administrator must keep updated just one node (sometimes called the golden node) that will be replicated to other nodes. In this article, we present Kadeploy3, a tool designed to perform operating system provisioning using disk imaging and cloning. Thanks to its efficiency, scalability, and reliability, this tool is particularly suited for large scale clusters.

Reliable Deployment Process with Kadeploy3

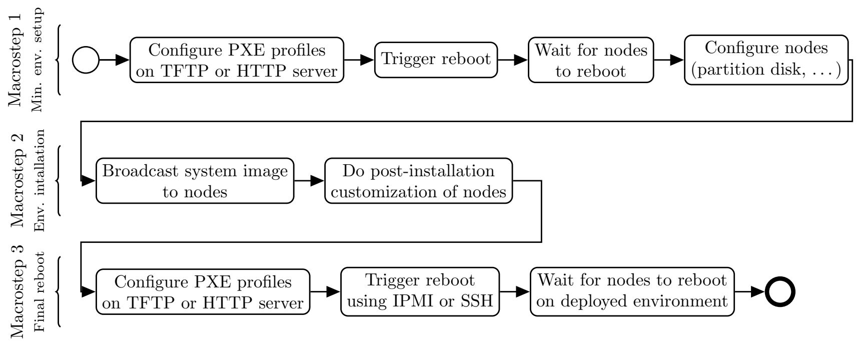
Kadeploy3 belongs to the family of disk imaging and cloning tools. It takes as input an archive containing the operating system to deploy, called an environment, and copies it on the target nodes. As a consequence, Kadeploy3 does not install an operating system following the classical installation procedure, and the user must provide an archive of the environment (as a tarball, for Linux environments).

Kadeploy3 does not directly take control of the nodes because doing so requires some specific and uncommon hardware support. Instead, it uses common network boot capabilities based on the PXE protocol [4], and it manages the associated PXE profiles.

Using such a mechanism, combined with the capability to update the PXE profiles of the nodes dynamically and to reboot the nodes in a reliable way (thanks to out-of-band control interfaces, such as Baseboard Management Controller, Remote Supervisor Adapter, or the power distribution unit's capabilities), taking control of the nodes and specifying what they are booting is possible.

As shown in Figure 1, a typical deployment with Kadeploy3 is composed of three major steps, called macro steps.

1. Minimal environment setup: the nodes reboot into a trusted minimal environment that contains all the tools required for the deployment (partitioning tools, archive management, etc.), and the required partitioning is performed.
2. Environment installation: the environment is broadcast to all nodes and extracted on the disks. Some post-installation operations also can be performed.
3. Reboot using the newly deployed environment.



Macrostep 1
Min. env. setup
Configure PXE profiles on TFTP or HTTP server
Trigger reboot
Wait for nodes to reboot
Configure nodes (partition disk, . . .)
Macrostep 2
Env. intallation
Broadcast system image to nodes
Do post-installation customization of nodes
Macrostep 3
Final reboot
Configure PXE profiles on TFTP or HTTP server
Trigger reboot using IPMI or SSH
Wait for nodes to reboot on deployed environment

Figure 1: Kadeploy deployment process, composed of three macro-steps

Each macro step can be executed via several different mechanisms to optimize the deployment process depending on required parameters and the specific infrastructure. For instance, the reboot using the newly deployed environment step can perform a traditional reboot or it might instead rely on a call to kexec(8) for a shorter reboot.

Reconfiguring a set of nodes involves several low-level operations that can lead to failures for various reasons, e.g., temporary loss of network connectivity, reboot taking longer than planned, etc. Kadeploy3 reliability is achieved because (1) the deployment process has powerful error management and (2) critical reboot operations required for the node control are based on reboot commands escalation in order to be able to take control of the nodes in any situation.

Reliability of the Deployment

Kadeploy3 is designed to detect failures as quickly as possible and improve deployment reliability by providing a macro-step replay mechanism on the nodes of interest. To illustrate that, let's consider the last deployment macro step that aims at rebooting using the deployed environment. Kadeploy3 implements, among others, the following strategies:

1. Directly load the kernel inside the deployed environment thanks to kexec.
2. Perform a hard reboot using out-of-band management hardware without checking the state of the node.

Thus it is possible to describe strategies such as: try the first strategy; if some nodes fail, try the second strategy, several times if required.

Because all the steps involved in the deployment process rely on system calls (hard disk operations, network communications, specific hardware management), special attention has been paid to error handling. Kadeploy3 collects the result of every operation (exit status, stdout, stderr), even when it is performed on remote nodes. As a consequence, some steps can be replayed on nodes where a problem occurs.

Furthermore, some operations may last too long (e.g., network boot, file-system creation, etc.), but Kadeploy3 provides administrators with the capability of defining specific timeouts for some operations in order to adapt the deployment process to the infrastructure. That allows identifying some problems quickly and replaying some operations on the related nodes.

Reliability of Reboot Operations

Because reboot operations are essential to control the cluster nodes, and ultimately the entire deployment process itself, they must behave correctly and reliably. Several methods can be used to reboot nodes, for instance:

1. Directly execute the /sbin/reboot command.
2. Use out-of-band management hardware with protocols such as IPMI. Various kinds of reboots can be executed: reset, power cycle, etc.
3. Use the power management capability of the power distribution unit (PDU).

Performing an /sbin/reboot is the best solution with regards to speed and cleanliness; however, it may not be an option if the target node is unreachable via in-band methods such as SSH (e.g., the node is already down, the OS has crashed, an unfriendly operating system is installed, etc.). In this scenario, we would use IPMI-like features if available. Also, because it bypasses the power-on self test, it might be better for speed to perform a reset rather than a power cycle, but sometimes this is not sufficient. Finally, if onboard management hardware is unreachable, we may be required to use the capabilities of a remotely manageable PDU.

Kadeploy3 provides administrators with a way to specify several levels of commands in order to perform escalation if required. This allows them to perform highly reliable deployments if the clusters have the appropriate hardware. Unfortunately, depending on the methods chosen, reboot escalation comes at a cost, and a balance must be struck between desired reliability and the time to deployment.

Scalability

In addition to having a reliable node-control mechanism, deploying large scale clusters in a reasonable time requires being able to execute several commands efficiently and to send large files on a large number of nodes.

Parallel Commands

The deployment workflow contains several operations that reduce to executing a command on a large set of nodes.

Thanks to SSH, one can execute commands remotely and retrieve their outputs, but launching SSH commands on a large number of nodes in sequence does not scale at all. Furthermore, launching all commands simultaneously can impose an extreme load on the server and can consume all of its file descriptors.

Several tools have been built to overcome these limitations. For instance, Pdsh [3] and ClusterShell [2] are designed to execute SSH commands on many nodes in parallel. Both tools use windowed execution to limit the number of concurrent SSH commands, and both also allow retrieval of command outputs on each node.

We choose to leverage TakTuk [9] as our mechanism for parallel command execution and reporting. TakTuk is based on a model of hierarchical connection. This allows TakTuk to distribute the execution load on all the nodes in a tree and to perform commands with low latency. Using such a hierarchical mechanism would normally require the tool to be installed on all nodes. Fortunately, TakTuk includes a convenient auto-propagation feature that ensures the tool's existence on all necessary nodes. The tool also uses an adaptive work-stealing algorithm to improve performance, even on heterogeneous infrastructures.

File Broadcast

The broadcast of the system image to all nodes is a critical part of the deployment. In cluster environments where the most important network for applications is using Infiniband or Myrinet, the Ethernet network is often composed of a hierarchy of switches (e.g., one switch per rack) that is hard to leverage for a high-performance broadcast. File distribution to a large number of nodes via any sequential push or pull method is not scalable. Kadeploy3 provides system administrators with three scalable file distribution approaches during the *Environment installation* macro step to minimize deployment time.

With tree-based broadcast, a file is sent from the server to a subset of nodes, which in turn send the file to other subsets until all the nodes have received the file. The size of the subsets, called tree arity, can be specified in the configuration. A large arity can reduce the latency to reach all nodes, but transfer times might increase because global bandwidth is equal to the bandwidth of a network link divided by the tree arity. The opposite effect occurs when the arity is small. In general, this broadcast method does not maximize bandwidth and should be used primarily for the distribution of small files. This method is also inefficient when used in hierarchical networks. We implement tree-based broadcast using TakTuk.

Chain-based broadcast facilitates the transfer of files with high bandwidth. A classical chain-based broadcast suffers from the establishment time of the chain in large-scale clusters. Indeed, because each node must connect to the next node in the chain (usually via SSH), a sequential initialization would drastically increase the entire broadcast period. Thus we perform the initialization of the chain with a tree-based parallel command. This kind of broadcast is near-optimal in a hierarchical network if the chain is well ordered because, as shown in Figure 2, all the full-duplex network links can be saturated in both directions, and the performance bottleneck becomes the backplane bandwidth of the network switches. For this method, we implement chain initialization using TakTuk and perform transfers using other custom mechanisms.

BitTorrent-based broadcast is able to send files at large scale without making any assumptions about the quality of the network. Furthermore, BitTorrent is able to handle churn efficiently, an important property in large scale systems such as petascale and future exascale clusters. Currently, our experiments show that there are two scenarios in which the performance of this broadcast method is inferior to the other methods. The first pathological case is one in which we are broadcasting

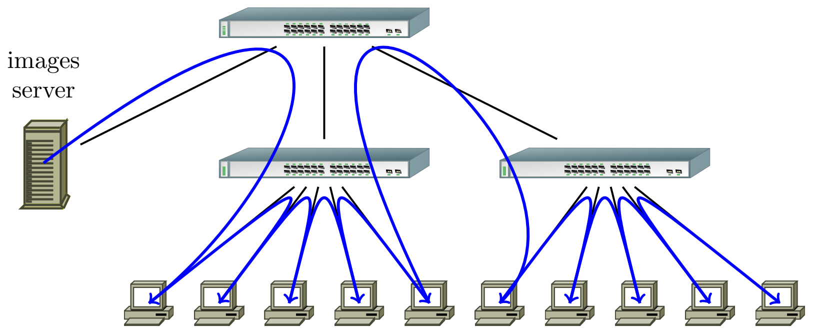


Figure 2: Topology-aware chained broadcast. Data is pipelined between all nodes. When correctly ordered, this ensures that inter-switch links are only used once in both directions.

on a small-scale cluster with a high-speed network, and the second is one in which we are broadcasting small files. In both cases, BitTorrent exhibits high latency, and the overhead of the protocol dominates the time to broadcast. The large number of established connections between nodes induced by the protocol can lead to bottlenecks depending on the network topology.

In a default configuration, Kadeploy3 uses tree-based broadcast for the files used in the deployment process (e.g., disk partition map) and chain-based method for the environment broadcast that is usually a large file; however, this behavior can be modified in the configuration.

Other Advanced Features

In addition to being reliable and scalable, Kadeploy has many useful features.

Multi-Cluster Support

Kadeploy3 can be configured to manage several clusters at the same time through a hierarchical set of YAML configuration files. In a grid-like environment, initiating and controlling deployments on several Kadeploy servers from a unique Kadeploy client is also possible.

Hardware and Software Compatibility

Kadeploy3 does not generally rely on vendor-specific mechanisms. Vendor-specific remote control systems used to trigger node reboots can be used easily, even if they do not support the IPMI protocol. Environments can be stored either as tarballs (for Linux environments) or as raw partitions, which enables the deployment of Windows or BSD-based systems.

Rights Management and Environments Library

Kadeploy3 can be used to provide users with a cloud-like experience with bare-metal system reservation. It can integrate with a cluster batch scheduler used to manage reservations in order to delegate system deployment rights to specific users for the duration of a job. Kadeploy3 can also manage a set of environments and their visibility (public, private) in order to provide default environments, on which users can base their work to create and register custom environments.

Statistics Collection

Identifying defunct nodes in a cluster is often hard, especially when failures are transient. Kadeploy3 integrates a statistics-collection mechanism that enables the detection of nodes that often fail during the deployment process.

Performance Evaluation

Grid'5000 Experimental Testbed

Kadeploy3 has been used intensively on the Grid'5000 testbed (http://www.grid5000.fr) since the end of 2009 (and previous versions of Kadeploy were used since 2004). In that time, approximately 620 different users have performed 117,000 deployments. On average, each deployment has involved 10.3 nodes. The largest deployment involved 496 nodes. To our knowledge, the deployed operating systems are mostly based on Linux (all flavors) with a sprinkling of FreeBSD.

Although the Grid'5000 use case does not exercise all the goals targeted by Kadeploy3 (e.g., scalability), it shows the tool's adequacy with regard to most characteristics, such as reliability.

Curie Petascale Supercomputer

We had the opportunity to evaluate Kadeploy3 on the Curie [7] supercomputer owned by GENCI (http://www.genci.fr/) and operated by CEA (http://www.cea.fr): 2088 nodes were available to perform the test and the goal was to deploy the production environment. After a single administrative cycle, 2015 nodes were successfully deployed. This proved the efficiency and the reliability of Kadeploy3 in a large-scale production infrastructure.

Virtual Testbed

Validating scalability on large physical infrastructures can become complex because it requires privileged rights on many components (e.g., access to management cards, modification of PXE profiles, etc.). For example, because the Curie supercomputer is used for production purposes, we only had access to it for several hours. Thus we chose to build our own large-scale virtual testbed on Grid'5000, leveraging important features such as link-layer isolation, and Kadeploy3 of course.

We performed an experiment in which we used 635 physical nodes of the Grid'5000 testbed. Depending on the nodes capabilities, we launched a variable number of KVM virtual machines. In total, 3,999 virtual machines were launched and participated in a single virtual network (despite that the physical nodes were located on four different sites). Then we installed all the required servers: DHCP, TFTP, MySQL, HTTP server, Kadeploy3. Once the testbed was launched, we were able to perform deployments within a single cluster of 3,999 nodes. During the largest run, a 430 MB environment was installed on 3,838 virtual machines in less than an hour; 161 virtual nodes were lost due to network or KVM issues. A significant amount of time was also wasted because of the high latency between geographically distant sites (10–20 ms), which affected some infrastructure services such as DHCP and the PXE protocol.

Wrapping Up

We think that Kadeploy3 can help system administrators of large-scale clusters save precious time by reducing OS provisioning time. The best way to be convinced is to try it. Kadeploy3 is free software (CeCill 2 license) written in Ruby and available from http://kadeploy3.gforge.inria.fr/. Source code, as well as Debian and RPM packages, can be downloaded. Kadeploy3 is configured thanks to few YAML files. To help administrators, a complete guide describes the entire installation and configuration process [10].

References

[1] Clonezilla: http://clonezilla.org.

[2] ClusterShell: http://cea-hpc.github.com/clustershell.

[3] Parallel Distributed Shell: http://sourceforge.net/projects/pdsh.

[4] Preboot Execution Environment (PXE) Specification: http://download.intel.com/design/archives/wfm/downloads/pxespec.pdf.

[5] Rocks: Open-Source toolkit for real and virtual clusters: http://www.rocksclusters.org.

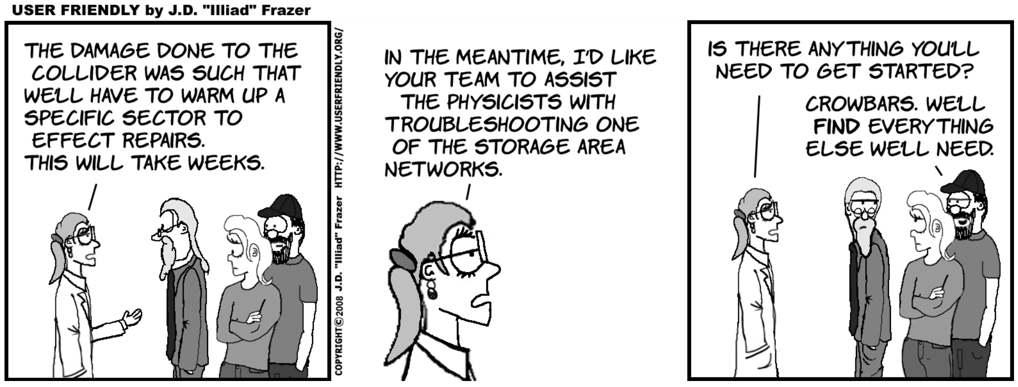
[6] SystemImager: http://systemimager.org.

[7] The Curie supercomputer: http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm.

[8] xCAT: Extreme Cloud Administration Toolkit: http://xcat.sourceforge.net.

[9] Benoit Claudel, Guillaume Huard, and Olivier Richard, "TakTuk: Adaptive Deployment of Remote Executions," Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC), ACM 2009, pp. 91–100.

[10] Kadeploy3: https://gforge.inria.fr/frs/download.php/27606/kadeploy-3.1-3.pdf.


USER FRIENDLY by J.D. "Illiad" Frazer
THE DAMAGE DONE TO THE COLLIDER WAS SUCH THAT WE'LL HAVE TO WARM UP A SPECIFIC SECTOR TO EFFECT REPAIRS. THIS WILL TAKE WEEKS.
COPYRIGHT © 2008 J.D. "Illiad" Frazer HTTP://WWW.USERFRIENDLY.ORG/
IN THE MEANTIME, I'D LIKE YOUR TEAM TO ASSIST THE PHYSICISTS WITH TROUBLESHOOTING ONE OF THE STORAGE AREA NETWORKS.
IS THERE ANYTHING YOU'LL NEED TO GET STARTED?
CROWBARS. WE'LL FIND EVERYTHING ELSE WE'LL NEED.

The Owl Embedded Python Environment

Microcontroller Development for the Modern World

THOMAS W. BARR AND SCOTT RIXNER

Thomas W. Barr is a fifth-year PhD student at Rice University in the Department of Computer Science. He received his BS degree in engineering and music from Harvey Mudd College in 2008. He has published research in computer architecture, embedded systems software, and high-performance computing. Outside of graduate school, he has worked as an expert witness and in litigation support for intellectual property.
twb@rice.edu

Scott Rixner is an Associate Professor of Computer Science at Rice University. His research focuses on the interaction between operating systems, runtime systems, and computer architectures; memory controller architectures; and hardware and software architectures for networking. He works with both large server-class systems and small embedded systems. Prior to joining Rice, he received his PhD from MIT.
rixner@rice.edu

Imagine my typical day. My alarm clock goes off and I immediately check my email on my iPhone. I stumble out of bed, make myself a cup of coffee and watch the morning news that my TiVo kindly recorded for me. I unlock my car and am presented with a map that shows the traffic on my route to work.

While there were three obvious computers in this little story, there are dozens more unsung heroes you probably didn't even think about. My alarm clock, my coffee maker, my TiVo remote, and my car key all are built around a microcontroller. My car is built around dozens of them. This article is about programming these very real, very complex computer systems.

Modern microcontrollers are almost always programmed in C. Applications run at a very low level without a real operating system. They are painfully difficult to debug, analyze, and maintain. At best, a simple real-time operating system (RTOS) is used for thread scheduling, synchronization, and communication [3]. These systems provide primitive, low-level mechanisms that require expert knowledge to use and do very little to simplify programming. At worst, they are programmed on the bare metal, perhaps even without a C standard library. As the electronic devices of the world become more and more complex, we absolutely have to do something to make embedded development easier.

We believe that the best way to do this is to run embedded software on top of a managed runtime system. We have developed and released as open source an efficient embedded Python programming environment named Owl. Owl is a complete Python development toolchain and runtime system for microcontrollers. Specifically, Owl targets systems that lack the resources to run a traditional operating system, but are still capable of running sophisticated software systems. Our work focuses on the ARM Cortex-M3 class of devices. These microcontrollers typically have 64–128 KB of SRAM, and have up to 1 MB of on-chip flash. Surprisingly, though, they are quite fast, executing at up to 100 MHz. This makes them more than fast enough to run an interpreter. These devices are absolutely everywhere; by 2015, ARM Cortex-M3-based systems are estimated to outsell x86 systems by a factor of 40.

Owl is a complete system that includes an interpreter, a programmer, an IDE, and a set of profilers and memory analyzers. Owl is derived from portions of several open-source projects, including CPython and Baobab. Most notably, the core runtime system for Owl is based on Dean Hall's Python-on-a-Chip (p14p) [2]. We support it on Texas Instruments LM3S9x9x Cortex-M3 microcontrollers as well as STM ST32F4 Cortex-M4 microcontrollers.

Owl demonstrates that it is possible to develop complex embedded systems using a high-level programming language. Many software applications have been developed within the Owl system, including a GPS tracker, a Web server, a read/write FAT32 file system, and an artificial horizon display. Furthermore, Owl is capable of running soft real-time systems; we've built an autonomous RC car and a pan-and-tilt laser pointer mount. These applications were written entirely in Python by programmers with no previous embedded systems experience, showing that programming microcontrollers with a managed runtime system is not only possible but easy. Additionally, Owl is used as the software platform for Rice University's r-one educational robot [5]. A class using this robot is now being taught for the third time, and groups of first-semester college students have been able to program their robots in Python successfully without any problems from the virtual machine. Moreover, at a demo, we had children as young as seven years old programming robots.

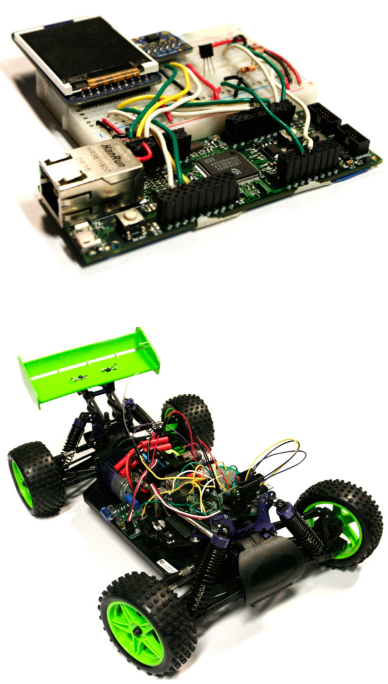


Figure 1: An artificial horizon (a) and an autonomous car (b) built with Owl

The cornerstone of this productivity is the interactive development process. A user can connect to the microcontroller and type statements to be executed immediately. This allows easy experimentation with peripherals and other functionality, making incremental program development for microcontrollers almost trivial. In a traditional development environment, the programmer has to go through a tedious compile/link/flash/run cycle repeatedly as code is written and debugged. Alternatively, in the Owl system a user can try one thing at the interactive prompt and then immediately try something else after simply hitting "return." The cost of experimentation is almost nothing.

This sort of capability is invaluable to both the novice and expert embedded programmer. While there are certainly many people in the world who are skilled in the art of low-level microcontroller programming, the process is and always will be expensive, slow, and error-prone. By raising the level of abstraction, we can allow expert embedded developers to spend their limited time making more interesting and complex systems, not debugging simple ones.

Finally, Owl is fun to use. Microcontrollers help put a lot of the joy back into programming because they make it possible to build real, physical systems. Go out and build a super-intelligent barbeque, a home automation controller, or a fearsome battle robot. We'll make sure that register maps and funky memory layouts don't get in your way.

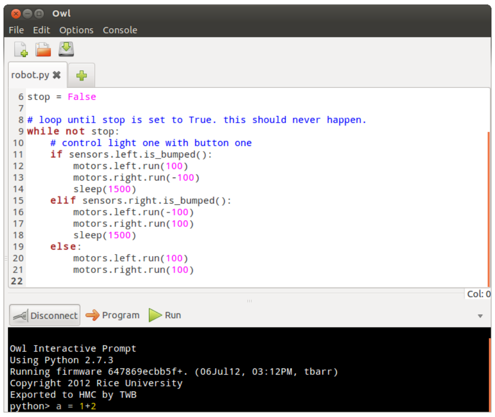
The Owl System

Modern 32-bit ARM-based microcontrollers now have enough performance to run a relatively sophisticated runtime system. Such systems include eLua [1], the Python-on-a-Chip project [2], and Owl, our project. These systems execute modern high-level languages and provide system support for everything from object-oriented code to multithreading to networking.

The Owl runtime natively executes a large subset of the standard Python bytecodes. This allows us to use the standard Python compiler and even to execute many existing programs. Owl natively supports multithreading and includes a novel feature to call C functions. This is critical on microcontrollers because programmers must call into a C driver library to control peripherals. Our system allows Python programmers to call functions in these libraries exactly as if they were standard Python functions.

The high-level design of Owl allows a user to build embedded systems without having low-level knowledge about microcontrollers. Users start by connecting to the microcontroller from a standard desktop computer over USB. Owl then shows a Python prompt, just like regular Python. When the user types a statement, the host computer compiles the statement into bytecodes, sends it to the controller where it is executed. Any resulting output is sent back for display and the process repeats.

Along the way, Owl automatically manages resources on the controller. The prompt is a built-in feature, as is the thread scheduler, the memory manager and countless others. A user doesn't need to write code on the microcontroller to connect over USB; it just works. The user doesn't need to allocate memory for a variable, nor remember to free it later. These automatic features are nothing particularly new in large computer systems, but they are nearly unheard of in the embedded space. Higher level languages including Python are heavily used in everything from package managers to cell phones to scientific computing. We believe that the time has come to use these ideas to make microcontroller programming easier.



```
stop = False

# loop until stop is set to True. this should never happen.
while not stop:
    # control light one with button one
    if sensors.left.is_bumped():
        motors.left.run(100)
        motors.right.run(-100)
        sleep(1500)
    elif sensors.right.is_bumped():
        motors.left.run(-100)
        motors.right.run(100)
        sleep(1500)
    else:
        motors.left.run(100)
        motors.right.run(100)
```

```
Owl Interactive Prompt
Using Python 2.7.3
Running firmware 647869ecbb5f+. (06Jul12, 03:12PM, tbarr)
Copyright 2012 Rice University
Exported to HMC by TWB
python> a = 1+2
```

Figure 2: Owl can be programmed from the command line, or using our cross-platform, Arduino-like IDE

Does It Work?

Often, people tell us that building a high-level language interpreter for microcontrollers must be "impossible." There's simply not enough RAM or enough flash or enough speed! We've found this to be false and refer you to our research paper on the Owl system [4] for a more in-depth look at these issues. We would like to take this opportunity to address some specific questions people have asked us.

Question: Surely a complete virtual machine takes up a lot of flash?

Indeed, flash memory that stores programs and data is a precious resource on a microcontroller; however, it turns out that Owl doesn't need much more flash than a traditional RTOS does.

The Owl VM itself is actually quite small, around 35 KB, and it contains all of the code necessary for manipulating objects, interpreting bytecodes, managing threads, and calling C functions. When compared to the 256 KB or more available on a microcontroller, this is not much larger than the so-called "light weight" FreeRTOS, which requires 22 KB.

The largest fraction of this space is used by C libraries, such as a network stack, a USB library or specialized math routines. The size of the standard Owl distribution is on the order of 150 KB, the majority of which are compiled C libraries. Any C application that uses these libraries would have to include them, just like Owl. Therefore, the overhead incurred by Owl for any complex application that utilizes a large set of peripherals and C libraries will be quite low.

Question: Okay, but won't it be very, very slow?

This depends greatly on what you're doing. At one extreme, the bytecode to add two numbers together takes about 10 μs. This is 500 times slower than the single cycle 32-bit add that the Cortex-M3 is theoretically capable of; however, the interpreter supports much more complicated operations, including function calls into native C code, which incur far lower overhead.

Overall, the proof is in what you can build. We've implemented many applications using Owl, and the performance has always been sufficient. We've connected our controller to a GPS receiver, three-axis accelerometer, three-axis MEMS gyroscope, digital compass, LCD display, microSD card reader, ultrasonic range finder, steering servo, and motor controller. We then built an artificial horizon display (using the display and accelerometer), a GPS tracker (using the GPS, compass, microSD, and display), and an autonomous RC car (using the gyroscope, GPS, range finder, steering servo, and motor controller). All of these applications work just fine.

Question: Aren't all embedded systems real-time? You've said nothing about building real-time systems.

There is some truth to this. Owl is not a hard real-time system. Owl provides no guarantees about when code will run; however, most real-time systems only need soft real-time guarantees. There is no loss of life if a thermostat takes an extra few milliseconds to switch on.

Our autonomous car is one such system. The car is based on an off-the-shelf remote controlled car that has had the R/C receiver disabled. Instead, a microcontroller running Owl drives the outputs. The car senses its position with GPS, navigating to waypoints. Meanwhile, it monitors a rangefinder to detect obstacles and uses a gyroscope to drive straight. All of these functions are "real-time systems," and they all work to form a functional autonomous car.

Question: Well, okay, but what about garbage collection? Doesn't that ruin everything?

The impact of garbage collection on embedded workloads is much smaller than even we expected it to be.

Owl's garbage collector (GC) is a simple mark-and-sweep collector that occasionally stops execution for a variable period of time. We found that the garbage collector has the largest impact on applications that use complex data structures, such as CPU benchmarks that we ported to Owl. These structures take a long time to traverse during the mark phase of collection. Additionally, there are a large number of objects in total, slowing the sweep phase. Overall, garbage collection can take up to 65 ms or 41% of execution time on these types of programs.

The embedded systems we have examined use much simpler data structures. This means that GC runs more rarely, and for shorter periods of time. For the worst-case embedded workload we tested, this takes 8 ms on average, only 11% of the application's running time.

Further reducing the impact of GC on embedded workloads, our virtual machine runs the collector when the system is otherwise idle. In an event-driven system, there are often idle times waiting for events. We take advantage of those idle times to run the garbage collector preemptively. In practice, this works quite well. For example, when we tested our autonomous car, all GC happened during sleep times. In other words, the garbage collector never interrupted or slowed useful work.

Sounds great for a beginner. What about me, though? I've been writing low-level assembler and C for decades! I can already build microcontroller applications. Why do we need yet another development system?

Of course it is possible for one skilled in the art to build a complex embedded system using low-level programming tools. Owl itself is an example of such a system;

however, just because it's possible to build a system using these tools doesn't mean we can't do better!

The time of an expert embedded systems programmer is a precious commodity. A higher level language makes building complicated algorithms and data structures easier. Tools such as profilers and interactive prompts make exploring the performance and behavior of a system possible. These tools mean that a programmer has to spend less time debugging and can spend more time building products. In the time that it might take an expert engineer to build a programmable thermostat in C, the same expert engineer might be able build a machine learning thermostat in Python. The fact that an expert programmer is capable of repeatedly writing low-level code doesn't mean that it should be necessary.

Perhaps more critically, though, raising the level of abstraction can make programs more reliable. Programs are simpler, so they are less error-prone. Owl can detect internal errors, such as stack overflow, turning a catastrophic memory corruption bug into a properly detected and reported error condition. Owl can detect programming bugs, such as array bounds violations, reporting a sensible error before a device is deployed into the field.

Finally, Owl allows users to reprogram part or all of their devices easily, sometimes without even needing to restart the controller. This means that deployed devices can be tested, modified, fixed, and upgraded without having to take critical systems offline. This would be extremely difficult to accomplish with normal embedded toolchains and is rarely, if ever, done.

We don't see the Owl system as a replacement for skilled embedded systems programmers. Rather, we see it as a productivity multiplier. We are skilled embedded systems programmers—we built the Owl system—yet we can accomplish a lot more a lot faster using the Owl system itself!

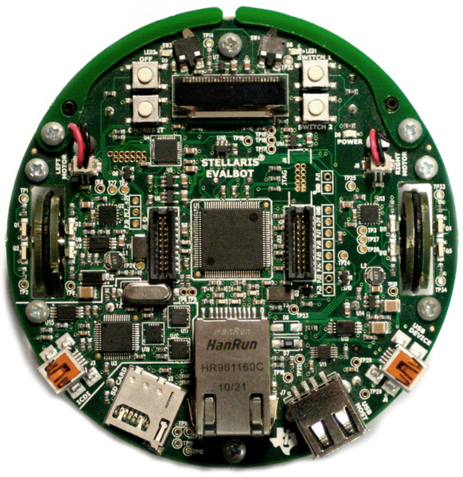


Figure 3: The Texas Instruments Evalbot is a commercially available robot that can be programmed using Owl

Using Owl

Getting started with Owl is easy. Here, we show a simple robotic example using an off-the-shelf Texas Instruments "Evalbot." The Evalbot is a simple, two-motor turtle, or Roomba-like, robot. You can download all the software we use here and find links to the hardware from our Web site. We demonstrate how to use the interactive prompt to control the hardware using both prepackaged libraries as well as through low-level driver library calls. Finally, we show how to flash a program onto the robot and run it while disconnected from the host computer.

First, we assume that the Owl tools distribution is installed onto a UNIX-like system and that the robot is connected over USB. From a prompt, type:

```
$ mcu interactive
Owl Interactive Prompt
Using Python 2.7.3
Running release firmware 0.01. (05Sep12, 01:37AM, twb)
mcu> 1+1
2
mcu>
```

This prompt looks and works just like any other Python prompt. You can assign values to variables, evaluate expressions, call functions, and even define functions and classes. Of course, this would be a very boring robot if we didn't dig into controlling the hardware. The Owl distribution for the TI Evalbot contains prebuilt modules to control some of the robot peripherals. As a simple example, let's play with the motors module. After each statement, the robot responds immediately. First, we run the left motor 100% forward, then 100% backward, then we stop it:

```
mcu> import motors
mcu> motors.left.run(100)
mcu> motors.left.run(-100)
mcu> motors.left.run(0)
mcu>
```

Suppose, however, that you were designing your own device. You won't have access to high-level peripheral libraries for it, so you'll need to make calls directly into the low-level driver library. Owl makes this relatively easy by making those libraries appear just like any other Python module. In fact, the conversion from C to Python is so transparent that you can use the original C documentation provided by the microcontroller vendor.

Suppose we are trying to read the current value of the bump sensor, which is just a simple push button attached to a general purpose I/O pin. We will first need to enable the GPIO module. In C, we would do this with the line `SysCtlPeripheralEnable (SYSCTL_PERIPH_GPIOE)`. In Python, this translates very simply. In fact, we can call this function from the prompt:

```
mcu> import sysctl
mcu> sysctl.PeripheralEnable(sysctl.PERIPH_GPIOE)
mcu>
```

Similarly, we will call `gpio.PinTypeGPIOInput` and `gpio.PadConfigSet` to configure the correct pin. Finally, we will read the value by calling `gpio.PinRead`. Putting all of this together, we can write a simple program to emulate the "bounce against walls" behavior of a Roomba:

```
# robot.py
import motors, sysctl, gpio, sys

# initialize the bump sensors
sysctl.PeripheralEnable(sysctl.PERIPH_GPIOE)

gpio.PinTypeGPIOInput(gpio.PORTE_BASE, gpio.PIN_0) # right bumper
gpio.PinTypeGPIOInput(gpio.PORTE_BASE, gpio.PIN_1) # left bumper

gpio.PadConfigSet(gpio.PORTE_BASE, gpio.PIN_0, gpio.GPIO_STRENGTH_2MA,
                  gpio.GPIO_PIN_TYPE_STD_WPU)
gpio.PadConfigSet(gpio.PORTE_BASE, gpio.PIN_1, gpio.GPIO_STRENGTH_2MA,
                  gpio.GPIO_PIN_TYPE_STD_WPU)
# loop forever
while True:
    if gpio.PinRead(gpio.PORTE_BASE, gpio.PIN_1): # bumped!
        motors.left.run(100) # full forward
        motors.right.run(-100) # full backwards
        sys.sleep(1500) # wait 1500 ms
    elif gpio.PinRead(gpio.PORTE_BASE, gpio.PIN_0):
        motors.left.run(-100)
        motors.right.run(100)
        sys.sleep(1500)
    else: # go straight ahead.
        motors.left.run(100)
        motors.right.run(100)
```

We can now flash this program as a module onto the robot. This process erases all user-programmed modules from the device and programs one or more new files. In this case, we only program one module, robot.py, by resetting the robot and calling mcu robot.py at the UNIX prompt. This module could be imported (and therefore executed) from the Python prompt, or it can be run in stand-alone mode. When the microcontroller starts up, it checks to see if it is connected to USB. If it is not, it automatically runs the primary module, which was the first module listed when the device was programmed.

Now, our robot is free to explore the world!

Next Steps

There are unfathomable numbers of microcontrollers in the world, but for some reason, we don't think of them as "real" computer systems. While we've developed incredible programming environments for cell phones and Web apps, we still program most embedded systems as if they were PDP-8s. As a result, we have countless lines of unportable, unreliable, and unsafe code that we use every day. Programmers have very little visibility over what their software is doing and must debug software using multimeters and oscilloscopes. This is expensive, painful, and error-prone. Furthermore, the process is not really all that fun. Let's start thinking of these tiny devices as the fully fledged computer systems that they are. We think Owl and the other open-source projects are a great start. They enable interactive software development that's high-level, safe, and easy as opposed to the current approach that is more akin to flipping front-panel switches on an Altair.

Go try Owl out! Microcontroller development boards are cheap nowadays. Numerous boards are available for less than $100 and some for as low as $15. Companies such as SparkFun Electronics and AdaFruit Industries sell a lifetime worth of peripherals that are easy to work with. Check out our Web site at http://embeddedpython.org/ for links to these products, buy some of them, download Owl, and get out there and build something. We promise that you'll have a lot of fun!

References

[1] eLua: http://www.eluaproject.net/.

[2] Python-on-a-Chip: http://code.google.com/p/python-on-a-chip/.

[3] T.N.B. Anh and S.-L. Tan, "Real-Time Operating Systems for Small Microcontrollers," *IEEE Micro,* vol. 29, no. 5), 2009.

[4] T.W. Barr, R. Smith, and S. Rixner, "Design and Implementation of an Embedded Python Runtime System," USENIX ATC, 2012.

[5] J. McLurkin, A. Lynch, S. Rixner, T. Barr, A. Chou, K. Foster, and S. Bilstein. A Low-Cost Multi-Robot System for Research, Teaching, and Outreach. Distributed Autonomous Robotic Systems, pages 597–609, 2010.

Practical Perl Tools

I Just Called to Say $_

DAVID N. BLANK-EDELMAN

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010. dnb@ccs.neu.edu

If you've noticed a spate of "here's how to use Perl to talk to X Web service" topics in this column lately, it probably isn't a coincidence. I have to confess, I'm a sucker for a Web service that gives you super powers just by using the simple API they provide. For example, in our last column we looked at how to easily translate text in and out of a large number of the world's major languages using Google Translate's API. For this column, we're going to do all sorts of fun things with phones from Perl. If you've ever wanted a way to send and receive SMS and voice messages, retrieve input from a caller, and stuff like that, have I got a column for you.

Like last time, in this column, we're going to be using an API from a commercial vendor. I am not a shill for that vendor. They are not paying me to promote their product. Like last time, I'm actually paying them to use the service. There are other vendors offering similar services. I'm choosing this one because their API is easy to use and the cost for small volumes of use is sufficiently low that it doesn't cost very much to play around (and, in fact, they offer a free account should you want to pay nothing during your playtime). Their API has the added benefit of using things that we've seen in past columns such as a REST and XML. You won't need to reference past columns, but if you're an avid reader of this column (hi mom!), a number of things we'll be looking at should be comfortably familiar.

So who is the lucky vendor this time that gets to take my money? In this column we're going to work with the Twilio API. As we've done in the past, I'm going to hold off on talking about the Twilio-specific Perl modules available for a bit just so we can get a good handle on the basics of what is going on before we let someone else's code do the driving. In this case looking at the underlying stuff is doubly important because Twilio's API and the Perl modules that interact with it assume you understand TwiML, their little mini-XML dialect for command and control. And that's just where we are going to start.

Twinkle, Twinkle, Little TwiML

We're starting a Perl-themed column with an XML dialect because in order to use their service, you'll be slinging TwiML around lots. In the past I've praised XML because it can be superbly readable as long as you don't take pains to thwart this quality (I'm looking at you Microsoft Office). TwiML is no exception; it basically consists of a set of "verb" tags that instruct the service to do something. For example, if we wanted to ask it to send an SMS message, we could write:

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
   <Sms from="+14105551234" to="+14105556789">
           The king stay the king.</Sms>
</Response>
```

That's a direct quote from their API docs at https://www.twilio.com/docs/api (which I just had to quote because of the embedded reference). If we wanted to call our special Twilio number (more on this shortly) and have it speak to us, we could write:

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Twinkle, Twinkle, little TwiML</Say>
</Response>
```

The first line is the standard XML declaration. The commands we write will live in a response tag (you'll see why it is called this when we get to talking about REST stuff). Twilio will perform the commands, and if the context is a phone call as in the second example, it will hang up at that point.

There are other verbs available with names that all make sense such as Dial to dial a call, Play to play a sound file on to the call from a specified URL, Record to record sound from the call, and so on. The only one that may not be obvious at first blush is Gather. Gather is used to receive input from a caller (i.e., "Press 1 to speak to Larry Wall…"). We'll see an example of that later in this column.

The REST of the Story

To actually use this stuff, we need to see how to set up conversations with the Twilio's server. This is where the REST stuff we mentioned in the beginning comes in. Let's dive right into some useful examples to see how this works. The TwiML part won't show up until our second example, so we'll set that aside for a brief moment as we look at some code for sending an SMS message:

```
use HTTP::Request::Common qw(POST);
use LWP::UserAgent; # can't use LWP::Simple to POST
use strict;

my $tw_serverURL = 'https://api.twilio.com';
my $tw_APIver    = '2010-04-01';

my $tw_acctsid   = '{YOUR ACCT SID HERE}';
my $tw_authtoken = '{YOUR AUTH TOKEN HERE}';

my $tw_number   = '{YOUR TWILIO NUMBER}';
my $test_number = '{A VALIDATED NUMBER}';

# create the user agent and give it credentials to use
my $ua = LWP::UserAgent->new;
$ua->credentials( 'api.twilio.com:443', 'Twilio API',
           $tw_acctsid, $tw_authtoken );
```

```
# create a request
my $req = POST "$tw_serverURL/$tw_APIver/Accounts/
               $tw_acctsid/SMS/Messages",
   [
   'From' => "$tw_number",
   'To'   => "$test_number",
   'Body' => 'Just a spiffy message!',
   ];

  # ...and send it
my $response = $ua->request($req);

if ( $response->is_success ) {
   print $response->decoded_content;
}
else {
   die $response->status_line;
}
```

Let's walk through this fairly generic LWP::UserAgent code together. After loading the modules we'll need, we define some variables that include the URL we're going to contact (their server plus API version), the API SID and token (user name and password we get when we sign up), and the numbers we'll be using. When you sign up for Twilio, you are given the opportunity to choose a number from which your text and voice messages will be sent and received. Once you pay for the service, you are also able to purchase additional numbers. At demo signup time you are also prompted to validate a number (i.e., prove you own it)—their system calls you and asks you to provide a passcode the Web site shows you—to act as a test number. You will need to use this test number as the source/destination number to call or be called by your Twilio number until you become a paying customer.

With these things defined, we can create a UserAgent object (the thing that is going to pretend to be a browser) and give it the credentials it will need to access that REST API URL. We construct the request by specifying the URL and the parameters we'll want to pass in when we do the POST. We send it off, and then print the response we get back. Here's a sample response that I've pretty printed so it is easier to read:

```
<?xml version="1.0"?>
  <TwilioResponse>
     <SMSMessage>
        <Sid>SMb70e55827ff00117fd88060902ffddddd</Sid>
        <DateCreated>Mon, 26 Nov 2012 02:56:44 +0000</DateCreated>
        <DateUpdated>Mon, 26 Nov 2012 02:56:44 +0000</DateUpdated>
        <DateSent/>
        <AccountSid>{MY ACCOUNT SID}</AccountSid>
        <To>{THE VALIDATED NUMBER}</To>
        <From>{MY TWILIP NUMBER}</From>
        <Body>Just a spiffy message!</Body>
        <Status>queued</Status>
        <Direction>outbound-api</Direction>
        <ApiVersion>2010-04-01</ApiVersion>
        <Price/>
```

```
            <Uri>/2010-04-01/Accounts/{MY_ACCOUNT_SID}/SMS/Messages/
            SMb70e55827ff00117fd88060902ffddddd</Uri>
        </SMSMessage>
    </TwilioResponse>
```

It is basically an echo of the message we sent, but I want to draw your attention to one of the elements:

```
        <Status>queued</Status>
```

When you send a message, it gets queued to be sent. Unlike some services, you do not stay connected to the server until the message is actually sent. This means you don't get a definitive response code back from the request that indicates success or failure on the sending. How you get the response back brings us to the REST stuff…

In my request, I didn't include the optional StatusCallback parameter. If I were to include that in my request, e.g.,

```
 my $req = POST "$tw_serverURL/$tw_APIver/Accounts/$tw_acctsid/SMS/
Messages",
       [
       'From'          => "$tw_number",
       'To'            => "$test_number",
       'Body'          => 'Just a spiffy message!',
       'StatusCallback' => 'http://your.Web.server.com/messagestat.pl',
       ];
```

Twilio would fire off a POST request to that URL once the message has gone through (or not). This callback style of programming shows up throughout the API, so if you plan to do much with it you'll need a Web server where you can place code that will receive messages from their server. Here's a very simple example we could use as messagestat.pl to receive a message from them:

```
use CGI;
use Data::Dumper;

my $q = CGI->new;
my $params = $q->Vars;

open my $OUTPUT, '>>', 'twilio.out' or die "Can't write output: $!";
print $OUTPUT Dumper \$params;
close $OUTPUT;
```

This receives the POST from Twilio's servers (and anyone who can contact that URL) and writes the parameters of the request to a file. If we look at the contents of the file, we see it says:

```
$VAR1 = \{
            'AccountSid' => '{MY_ACCOUNT_SID}',
            'SmsStatus' => 'sent',
            'Body' => 'Just a spiffy message!',
            'SmsSid' => 'SMe1620214513ccee199851ad9f13ffff',
            'To' => '{THE VALIDATED NUMBER}',
            'From' => '{MY TWILIP NUMBER}',
            'ApiVersion' => '2010-04-01'
          };
```

Here we can see that the SmsStatus was 'sent', so the message went out. If for some reason it couldn't be sent successfully, that would have been reflected in the status posted to our CGI script.

Early in this section I mentioned "conversations with their server," but the last example didn't offer anything particularly scintillating in this regard. Let's do something more sophisticated, this time with a voice call instead of an SMS message. Let's do a two-question telephone poll from Perl using Twilio. This will allow us to bring the TwiML we learned earlier back into the picture.

The first step is to tell Twilio's servers to initiate a voice call. I'm going to leave out all of the initialization code from the example below to save space because it is exactly the same as the previous example. Here's the one part of the code that changes:

```
my $req = POST "$tw_serverURL/$tw_APIver/Accounts/$tw_acctsid/Calls",
   [
   'From'          => "$tw_number",
   'To'            => "$test_number",
   'Url'           => 'http://your.Web.server.com/voicepoll.pl',
   'StatusCallback' => 'http://your.Web.server.com/messagestat.pl',
   ];
```

The first change is we're requesting a different kind of REST object; we're asking for Calls instead of SMS/Messages. The second change is we've told Twilio that once it initiates a call, it should contact the voicepoll.pl script for further instructions to follow once the call has connected. And this is where TwiML becomes important.

The URL pointed to by the Url parameter is expected to provide Twilio's server with a TwiML document it should process. Here's our voicepoll.pl script that will provide this document:

```
use CGI qw(:standard);
use strict;

my $q      = CGI->new;
my $params = $q->Vars;

print $q->header('text/xml');

if ( not $params->{'Digits'} ) {
   print <<POLL;
<?xml version="1.0" encoding="UTF-8"?>
  <Response>
   <Gather numDigits="1" action="/voicepoll.pl">
      <Say>Welcome to the login poll</Say>
      <Say>Press 1 if you are happy and you know it</Say>
      <Say>Press 2 if you really want to show it</Say>
   </Gather>
   <Say>No input, toodles!</Say>
  </Response>
POLL
}
```

```
else {
   if ( $params->{'Digits'} ne "3" ) {

      open my $RESPONSE, '>>', 'twresp.out'
         or die "Can't write to twrestp.out";
      print $RESPONSE "Received $params->{'Digits'}\n";
      close $RESPONSE;

      print <<POLL;
<?xml version="1.0" encoding="UTF-8"?>
     <Response>
       <Gather numDigits="1" action="/voicepoll.pl">
         <Say>Next Question</Say>
         <Say>Press 1 if you are happy and you know it</Say>
         <Say>Press 2 if you really want to show it</Say>
         <Say>Press 3 if you are suffering from ennui</Say>
       </Gather>
         <Say>No input, toodles!</Say>
     </Response>
POLL
}
else {
      open my $RESPONSE, '>>', 'twresp.out'
         or die "Can't write to twrestp.out";
      print $RESPONSE "END POLL\n";
      close $RESPONSE;

      print <<BYEBYE;
<?xml version="1.0" encoding="UTF-8"?>
         <Response>
             <Say>End of Poll, thanks!</Say>
         </Response>
BYEBYE
   }
}
```

Here are two small caveats before we look at what the script and its embedded TwiML is doing. First, this script is going to spit out TwiML in the most straightforward but uncouth way possible. It's just a bunch of print statements using HEREDOC syntax (<<). If you were doing this for real, you'd want to use some sort of XML generator or one of the custom Twilio modules we'll get to in a moment. Second, all TwiML fetched from the servers is coming from this one script with a bunch of dumb control logic. In real life it might make more sense to have different responses to different queries from their servers handled by different scripts ("Press 1 for Sales" then points to the sales.pl script and so on).

Let's walk through what is going on one step at a time. We used a modified version of our SMS script to ask Twilio to initiate a call and then have it fetch the URL for the CGI script above. This CGI script checks to see whether it has received a parameter called 'Digits' for reasons you'll see in just a second. If that parameter isn't defined yet (true because this will be the first time it has been accessed by Twilio for this call), it prints the following TwiML back to their server:

```
<?xml version="1.0" encoding="UTF-8"?>
 <Response>
   <Gather numDigits="1" action="/voicepoll.pl">
       <Say>Welcome to the login poll</Say>
       <Say>Press 1 if you are happy and you know it</Say>
       <Say>Press 2 if you really want to show it</Say>
   </Gather>
   <Say>No input, toodles!</Say>
 </Response>
```

This TwiML uses "Gather," a verb we haven't seen before. Gather will attempt to read keypad input from the call (i.e., the caller pressed the phone's number buttons). In the TwiML, there are two attributes being passed in for Gather: "numDigits" for the number of digits we hope to get back (one) and "action" for the URL that will be called if Gather is successful.

As I mentioned above, the TwiML makes another call to the voicepoll.pl script in <Gather>, but it could easily have been told to fetch some other CGI script for its next batch of TwiML. Embedded in this Gather call are a number of <Say> elements used to speak the menu for the person on the line. Because they are listed as sub-elements of Gather, this means that the Gather is active while they are speaking. The caller doesn't have to wait to press a button. She or he can interrupt the <Say> directives, and the Gather will complete and immediately pass the results to the URL specified in the action attribute (bypassing anything else in this TwiML file). The action URL is used as the source of the next TwiML directive, and the control flow continues using whatever TwiML it provides. Should the <Gather> fail, e.g., time out if it doesn't get input, the directives outside of the <Gather> are run. In this case, a final <Say> command will bid the caller adieu before hanging up.

Let's assume the <Gather> was able to retrieve a choice from the caller and see what happens next. The CGI script specified in its action attribute gets called with the results from the Gather command being passed as a parameter called 'Digits'. This is why the script looks to see whether it has received that parameter. If it hasn't, it knows it is the first time it is being called. If it gets the number 3 in that parameter, it will immediately print the "BYEBYE" TwiML code. Any other number tells the script to print the Next Question TwiML. All along the way, we collect the results we received to a response file that accumulates lines like:

```
Received 2
Received 5
Received 1
END POLL
```

This is just to show off the input we received. If we really cared about it we'd want to store it in a more considered way, like a database. At the very least, we'd want a way to store things so multiple polls going at once record their results properly. And speaking of results, in the script that originated the call, we kept the

```
'StatusCallback' => 'http://your.Web.server.com/messagestat.pl',
```

line. Just like with an SMS message, this URL gets called after the operation has been completed (successfully or unsuccessfully). In the case of a voice message, we get a cool set of parameters posted to the messagestat.pl URL:

```
'AccountSid' => '{MY_ACCOUNT_SID}',
        'ToZip' => '02283',
        'FromState' => 'MA',
        'Called' => '{THE VALIDATED NUMBER}',
        'FromCountry' => 'US',
        'CallerCountry' => 'US',
        'CalledZip' => '02283',
        'Direction' => 'outbound-api',
        'FromCity' => 'CAMBRIDGE',
        'CalledCountry' => 'US',
        'Duration' => '1',
        'CallerState' => 'MA',
        'CallSid' => 'CA4fdd2c584cd58230f9e7413c452fffff',
        'CalledState' => 'MA',
        'From' => '{MY TWILIO NUMBER}',
        'CallerZip' => '02139',
        'FromZip' => '02139',
        'CallStatus' => 'completed',
        'ToCity' => 'BOSTON',
        'ToState' => 'MA',
        'To' => '{THE VALIDATED NUMBER}',
        'CallDuration' => '17',
        'ToCountry' => 'US',
        'CallerCity' => 'CAMBRIDGE',
        'ApiVersion' => '2010-04-01',
        'Caller' => '{MY TWILIO NUMBER}',
        'CalledCity' => 'BOSTON'
```

Yup, a little bit of geolocation is thrown in for free.

There's one last category of operations I want to mention (but not demonstrate for space reasons). So far we haven't seen any code for the case where someone calls in to your Twilio number or sends an SMS message to it. When you set a Twilio number up, you associate two URLs with it: one for voice, the other for SMS. When a call or an SMS message comes in to that number, Twilio attempts to post information about the incoming call/message to the appropriate URL (as parameters, same as we've seen before) and expects to be handed back some TwiML telling it what to do. That CGI script can do whatever you need with the incoming information (e.g., log the parameters) and direct Twilio to do something (like start a phone poll or take an order for a pizza).

WWW::Twilio::API and WWW::Twilio::TwiML

I'd like to end with a quick look at the two special purpose Perl modules for interacting with Twilio. The first lets you make API calls without having to trouble your pretty little head with all of the LWP::UserAgent details. Instead of our first code example, we could write:

```
use WWW::Twilio::API;

  my $twilio = WWW::Twilio::API->new(
     AccountSid => '{MY ACCOUNT SID}',
     AuthToken  => '{MY ACCOUNT TOKEN}',
  );
```

```
 my $response = $twilio->POST(
    'SMS/Messages',
    'From' => '{MY TWILIO NUMBER}',
    'To'   => '{THE VALIDATED NUMBER}',
    'Body' => 'Just a spiffy message!'
);
```

WWW::Twilio::API lets you use all of the other API calls we've seen before. For example, in the WWW::Twilio::API doc, we see an example of making a call:

```
$twilio->POST(
    'Calls',
    To   => '5558675309',
    From => '4158675309',
    Url  => 'http://www.myapp.com/myhandler'
);
```

The other special purpose Twilio module courtesy of the same author is WWW::Twilio::TwiML. It is designed to make authoring TwiML easier, but I'll say up front that I'm not entirely clear it is much easier to use than any of the other XML authoring modules that are available. I think it holds the most promise for people who enjoy writing chained method expressions (i.e., code with lots of thing->thing->thing statements). For example, if we wanted to output the first set of TwiML we printed in voicepoll.pl above, we would write:

```
use WWW::Twilio::TwiML;

my $twiml = WWW::Twilio::TwiML->new;

$twiml
    ->Response
      ->Gather( { action => '/voicepool.pl' } )
    ->Say('Welcome to the login poll')
    ->parent->Say('Press 1 if you are happy and you know it')
    ->parent->Say('Press 2 if you really want to show it')
    ->parent->parent->Say('No input, toodles!');

print $twiml->to_string;
```

The chained statement can be read something like this: "Create a <Response> element. In this element create a <Gather> element. In the <Gather> element, create a <Say> element. Now, instead of creating the next <Say> within the current <Say> element, put it in the parent (the <Gather>). Do that again for the next <Say>. Then, go to that element's grandparent (the <Response> element) and place a final <Say> element in it." If your brain has no problems mapping the chained steps to the process of building our little XML tree structure, great, this might be the module for you. If not, seek another solution.

And with that, we now have a good start on how to use Twilio's API from Perl to do all sort of fun phone-related stuff. Take care and I'll see you next time.

Python: Import Anything

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also a co-author of the forthcoming *Python Cookbook* (3rd Edition, O'Reilly & Associates, 2013). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

In the August 2012 issue of *;login:*, I explored some of the inner workings of Python's import statement. Much of that article explored the mechanism that's used to set up the module search path found in sys.path as well as the structure of a typical Python installation. At the end of that article, I promised that there is even more going on with `import` than meets the eye. So, without further delay, that's the topic of this month's article.

Just as a note, this article assumes the use of Python 2.7. Also, because of the advanced nature of the material, I encourage you to follow along with the interactive examples as they nicely illustrate the mechanics of it all.

Import Revisited

Just to revisit a few basics, each Python source file that you create is a module that can be loaded with the import statement. To make the import work, you simply need to make sure that your code can be found on the module search path `sys.path`. Typically, `sys.path` looks something like this:

```
>>> import sys
>>> sys.path
['',
 '/usr/local/lib/python2.7/site-packages/setuptools-0.6c11-py2.7.egg',
 '/usr/local/lib/python2.7/site-packages/pip-1.1-py2.7.egg',
 '/usr/local/lib/python2.7/site-packages/python_dateutil-1.5-py2.7.egg',
 '/usr/local/lib/python2.7/site-packages/pandas-0.7.3-py2.7-macosx-
10.4-x86_64.egg',
 '/usr/local/lib/python2.7/site-packages/tornado-2.1-py2.7.egg',
 '/usr/local/lib/python27.zip',
 '/usr/local/lib/python2.7',
 '/usr/local/lib/python2.7/plat-darwin',
 '/usr/local/lib/python2.7/plat-mac',
 '/usr/local/lib/python2.7/plat-mac/lib-scriptpackages',
 '/usr/local/lib/python2.7/lib-tk',
 '/usr/local/lib/python2.7/lib-old',
 '/usr/local/lib/python2.7/lib-dynload',
 '/Users/beazley/.local/lib/python2.7/site-packages',
 '/usr/local/lib/python2.7/site-packages']
>>>
```

For most Python programmers (including myself until recently), knowledge of the `import` statement doesn't extend far beyond knowing about the path and the fact that it sometimes needs to be tweaked if code is placed in an unusual location.

Making Modules Yourself

Although most modules are loaded via `import`, you can actually create module objects yourself. Here is a simple interactive example you can try just to illustrate:

```
>>> import imp
>>> mod = imp.new_module("mycode")
>>> mod.__file__ = 'interactive'
>>> code = '''
... def hello(name):
...     print "Hello", name
...
... def add(x,y):
...     return x+y
... '''
>>> exec(code, mod.__dict__)
>>> mod
<module 'mycode' from 'interactive'>
>>> dir(mod)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'add',
'hello']
>>> mod.hello('Dave')
Hello Dave
>>> mod.add(10,20)
30
>>>
```

Essentially, if you want to make a module you simply use the `imp.new_module()` function. To populate it, use the `exec` statement to execute the code you want in the module.

As a practical matter, the fact that you can make modules from scratch (bypassing import) may be nothing more than a curiosity; however, it opens a new line of thought. Perhaps you could create modules in an entirely different manner than a normal import statement, such as grabbing code from databases, from remote machines, or different kinds of archive formats. What's more, if all of this is possible, perhaps there is some way to customize the behavior of import directly.

Creating an Import Hook

Starting around Python 2.6 or so, the sys module acquired a mysterious new variable `sys.meta_path`. Initially, it is set to an empty list:

```
>>> import sys
>>> sys.meta_path
[]
>>>
```

What purpose could this possibly serve? To find out, try the following experiment:

```
>>> class Finder(object):
...     def find_module(self, fullname, path=None):
...         print "Looking for", fullname, path
...         return None
...
>>> import sys
>>> sys.meta_path.append(Finder())
>>> import math
Looking for math None
>>> import xml.etree.ElementTree
Looking for xml None
Looking for xml._xmlplus ['/usr/local/lib/python2.7/xml']
Looking for _xmlplus None
Looking for xml.etree ['/usr/local/lib/python2.7/xml']
Looking for xml.etree.ElementTree ['/usr/local/lib/python2.7/xml/etree']
Looking for xml.etree.sys ['/usr/local/lib/python2.7/xml/etree']
Looking for xml.etree.re ['/usr/local/lib/python2.7/xml/etree']
Looking for xml.etree.warnings ['/usr/local/lib/python2.7/xml/etree']
Looking for xml.etree.ElementPath ['/usr/local/lib/python2.7/xml/etree']
Looking for xml.etree.ElementC14N ['/usr/local/lib/python2.7/xml/etree']
Looking for ElementC14N None
>>>
```

Wow, look at that! The `find_module()` method of the Finder class you just wrote is suddenly being triggered on every single import statement. As input, it receives the fully qualified name of the module being imported. If the module is part of a package, the path argument is set to the package's `__path__` variable, which is typically a list of subdirectories that contain the package subcomponents. With packages, there are also a few unexpected oddities. For example, notice the attempted imports of `xml.etree.sys` and `xml.etree.re`. These are actually imports of `sys` and `re` occurring inside the `xml.etree` package. (Later these are tested for a relative and then absolute import.)

As output, the `find_module()` either returns `None` to indicate that the module isn't known or returns an instance of a loader object that will carry out the process of loading the module and creating a module object. A loader is simply some object that defines a `load_module` method that returns a module object created in a manner as shown earlier. Here is an example that mirrors the creation of the module that was used earlier:

```
>>> import imp
>>> import sys
>>> class Loader(object):
...     def load_module(self, fullname):
...             mod = sys.modules.setdefault(fullname, imp.new_module(fullname))
...             code = '''
... def hello(name):
...     print "Hello", name
...
... def add(x,y):
...     return x+y
... '''
...             exec(code, mod.__dict__)
...             return mod
```

```
...
>>> class Finder(object):
...     def find_module(self, fullname, path):
...         if fullname == 'mycode':
...             return Loader()
...         else:
...             return None
...
>>> sys.meta_path.append(Finder())
>>> import mycode
>>> mycode.hello('Dave')
Hello Dave
>>> mycode.add(2,3)
5
>>>
```

In this example, the code is mostly straightforward. The `Finder` class creates a `Loader` instance. The loader, in turn, is responsible for creating the module object and executing the underlying source code. The only part that warrants some discussion is the use of `sys.modules.setdefault()`. The `sys.modules` variable is a cache of already loaded modules. Updating this cache as appropriate during import is the responsibility of the loader. The `setdefault()` method makes sure that this happens cleanly by either returning the module already present or a new module created by `imp.new_module()` if needed.

Using Import Hooks

Defining an import hook opens up a variety of new programming techniques. For instance, here is a finder that forbids imports of certain modules:

```
# forbidden.py

import sys

class ForbiddenFinder(object):
    def __init__(self, blacklist):
        self._blacklist = blacklist

    def find_module(self, fullname, path):
        if fullname in self._blacklist:
            raise ImportError()

def no_import(module_names):
    sys.meta_path.append(ForbiddenFinder(module_names))
```

Try it out:

```
>>> import forbidden
>>> forbidden.no_import(['xml','threading','socket'])
>>> import xml
Traceback (most recent call last):
  File "<stdin>", line 1, in
ImportError: No module named xml
>>> import threading
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in
ImportError: No module named threading
>>>
```

Here is a more advanced example that allows callback functions to be attached to the import of user-specified modules:

```
# postimport.py

import importlib
import sys
from collections import defaultdict

_post_import_hooks = defaultdict(list)

class PostImportFinder:
    def __init__(self):
        self._skip = set()

    def find_module(self, fullname, path):
        print "Finding", fullname, path
        if fullname in self._skip:
            return None
        self._skip.add(fullname)
        return PostImportLoader(self)

class PostImportLoader:
    def __init__(self, finder):
        self._finder = finder

    def load_module(self, fullname):
        try:
            importlib.import_module(fullname)
            modname = fullname
        except ImportError:
            package, _, modname = fullname.rpartition('.')
            if package:
                try:
                    importlib.import_module(modname)
                except ImportError:
                    return None
            else:
                return None
        module = sys.modules[modname]
        for func in _post_import_hooks[modname]:
            func(module)
        _post_import_hooks[modname] = []
        self._finder._skip.remove(fullname)
        return module
```

```
def on_import(modname, callback):
    if modname in sys.modules:
        callback(sys.modules[modname])
    else:
        _post_import_hooks[modname].append(callback)

sys.meta_path.insert(0, PostImportFinder())
```

The idea on this hook is that it gets triggered on each import; however, immediately upon firing, it disables itself from further use. The `load_module()` method in the `PostImportLoader` class then carries out the regular import and triggers the registered callback functions. There is a bit of a mess concerning attempts to import the requested module manually. If an attempt to import the fully qualified name doesn't work, a second attempt is made to import just the base name.

To see it in action, try the following:

```
>>> from postimport import on_import
>>> def loaded(mod):
...     print "Loaded", mod
...
>>> on_import('math', loaded)
>>> on_import('threading', loaded)
>>> import math
Loaded <module 'math' from '/usr/local/lib/python2.7/lib-dynload/math.so'>
>>> import threading
Loaded <module 'threading' from '/usr/local/lib/python2.7/threading.pyc'>
>>>
```

Although a simple example has been shown, you could certainly do something more advanced such as patch the module contents. Consider this additional code that adds logging to selected functions:

```
def add_logging(func):
    'Decorator that adds logging to a function'
    def wrapper(*args, **kwargs):
        print("Calling %s.%s" % (func.__module__, func.__name__))
        return func(*args, **kwargs)
    return wrapper

def log_on_import(qualified_name):
    'Apply logging decorator to a function upon import`
    modname, _, symbol = qualified_name.rpartition('.')
    def patch_module(mod):
        setattr(mod, symbol, add_logging(getattr(mod, symbol)))
    on_import(modname, patch_module)
```

Here is an example:

```
>>> from postimport import log_on_import
>>> log_on_import('math.tan')
>>>
>>> import math
>>> math.tan(2)
Calling math.tan
```

```
-2.185039863261519
>>>
```

You might look at something like this with horror; however, you could also view it as a way to manipulate a large code base without ever touching its source code directly. For example, you could use an import hook to insert probes, selectively rewrite part of the code, or perform other actions on the side.

Path-Based Hooks

Manipulation of `sys.meta_path` is not the only way to hook into the import statement. As it turns out, there is another variable `sys.path_hooks` that can be manipulated. Take a look at it:

```
>>> import sys
>>> sys.path_hooks
[<type 'zipimport.zipimporter'>]
>>>
```

The items on `sys.path_hooks` are callables that process individual items in the `sys.path` list, and it either responds with an ImportError or it returns a finder object that is used to load modules from that path component. Try this experiment:

```
>>> import sys
>>> def check_path(name):
...     print "Checking", repr(name)
...     raise ImportError()
...
>>> sys.path_hooks.insert(0, check_path)
>>> # Clear the cache to have all path entries rechecked
>>> sys.path_importer_cache.clear()
>>> import foo
Checking ''
Checking '/usr/local/lib/python27.zip'
Checking '/usr/local/lib/python2.7'
Checking '/usr/local/lib/python2.7/plat-darwin'
Checking '/usr/local/lib/python2.7/plat-mac'
Checking '/usr/local/lib/python2.7/plat-mac/lib-scriptpackages'
Checking '/usr/local/lib/python2.7/lib-tk'
Checking '/usr/local/lib/python2.7/lib-old'
Checking '/usr/local/lib/python2.7/lib-dynload'
Checking '/usr/local/lib/python2.7/site-packages'
Traceback (most recent call last):
  File "<stdin>", line 1, in
ImportError: No module named foo
>>>
```

Notice how every entry on `sys.path` is checked by our function. To expand this code, you would make the `check_path()` function look for a specific pathname pattern. If found, it returns a special finder object that's similar to before. Try this:

```
>>> class Finder(object):
...     def find_module(self, name, path=None):
...         print "Looking for", name, path
...         return None
```

```
...
>>> def check_path(name):
...     if name.endswith('.spam'):
...         return Finder()
...     else:
...         raise ImportError()
...
>>> import sys
>>> sys.path_hooks.append(check_path)
>>> import foo
Traceback (most recent call last):
  File "<stdin>", line 1, in
ImportError: No module named foo
>>> sys.path.append('code.spam')
>>> import foo
Looking for foo None              # Notice Finder output here
Traceback (most recent call last):
  File "<stdin>", line 1, in
ImportError: No module named foo
>>>
```

This technique of hooking into sys.path is how Python has been expanded to import from .zip files and other formats.

Final Words and the Big Picture

Hacking of Python's import statement has been around for quite some time, but it's often shrouded in magic and mystery. Frameworks and software development tools will sometimes do it to carry out advanced operations across an entire code base; however, the whole process is poorly documented and underspecified. For instance, internally, Python 2.7 doesn't use the same machinery as extensions to the import statement. Frankly, it's a huge mess.

One of the most significant changes in the recent Python 3.3 release is an almost complete rewrite and formalization of the import machinery described here. Internally, it now uses `sys.meta_path` and path hooks for all stages of the import process. As a result, it's much more customizable (and understandable) than previous versions.

Having seen of all of this, should you now start hacking on `import`? Probably not; however, if you want to have a deep understanding of how Python is put together and how to figure things out when they break, knowing a bit about it is useful. For more information about import hooks, see PEP 302, http://www.python.org/dev/peps/pep-0302/.

iVoyeur

Nagios XI (cont.)

DAVE JOSEPHSEN

Dave Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and is Senior Systems Engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his co-authored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project. dave-usenix@skeptech.org

Before I begin in earnest, I should point out that this article is the second in a series of articles on Nagios XI, which is the commercial version of Nagios. Herein I assume you've read the previous article [1] and/or have a working understanding of the general XI architecture, which is different from Open Source Nagios, or "Nagios Core." So now that we have that out of the way...

Quick, what do you think of when I say "wizard"?

I'll risk being a bit presumptions in my hope that the image of Milamber, Gandalf, Dallben, Merlin, or etc. is what probably occurs to the type of person who might happen to accidentally read this article. Or if you're of a certain disposition, perhaps it was Sidi, Sauron, Arawn, or etc. (I'm not judging). If that's what you thought, then I'm with you. Those guys and the ideas connected with them are certainly the first thing that pops into my head, and although a few alternatives occur to me, the absolute last wizard on my mind, a wizard worse than the absolute darkest of the wizards of lore, a wizard so utterly corrupt and vile that I hesitate to mention it much less write an entire article about it, is the configuration wizard.

Was ever there a thing less wizardly? The configuration wizard is like a wizard in the same way Facebook is like a book (or dare I say for you Colorado readers: in the same way the flower pot is... well never mind). So I admit, I'm not looking forward to writing this particular article. And I've put it off, as long as absolutely possible (as my editor may attest), but it must be done. This of course is no slight to Nagios XI, which is awesome, and although the Nagios crew have done a top-notch job implementing a feature that will help a ton of people and fling wide for them the heavy, spiked portcullis that bars the entrance to corporate America, you'll forgive me, I'm sure, for feeling a bit reluctant in the documenting of it.

As I write this in the twilight of the year two thousand and twelve, there are system administrators who, while mostly competent and sane in other respects, have managed to carry out their entire careers using nothing but graphical configuration tools. As I related in the previous article, one of the major, oft-repeated gripes these admin have with Nagios Core is its reliance on configuration files and the accompanying assumption that you will edit them when you want the configuration to change.

To address this—perhaps the largest barrier to adoption for many corporate shops who need to simplify the configuration process—Nagios XI comes complete with all of the plugins in the standard plugins package, as well as NRPE, NSCA, and NRDP pre-installed. Additionally, the XI developers have provided a plethora of semi-automated configuration wizards, which, given the bare-minimum information about a host, take care of the initial setup as well as adding and modifying services on already-configured hosts.

Pay No Attention to the Files Behind the Curtain

If you consult the official XI documentation at http://library.nagios.com/library/products/nagiosxi/documentation, you'll quickly form the impression that the wizards are the only method for host and service configuration. The configuration files themselves are rarely if ever mentioned, as if they don't exist. With names such as "Exchange Server," "Website," and "Windows Workstation," the wizards make setting up new hosts and services easy enough that these tasks can be delegated to 1st-level support techs, or even end-users. The auto-discovery wizard is capable of bootstrapping an environment given only a CIDR net-block to start with, and in my experience does a good job of initial setup. To add NRPE-based host checks, or other services after the fact, just run the appropriate wizard on the preexisting host.

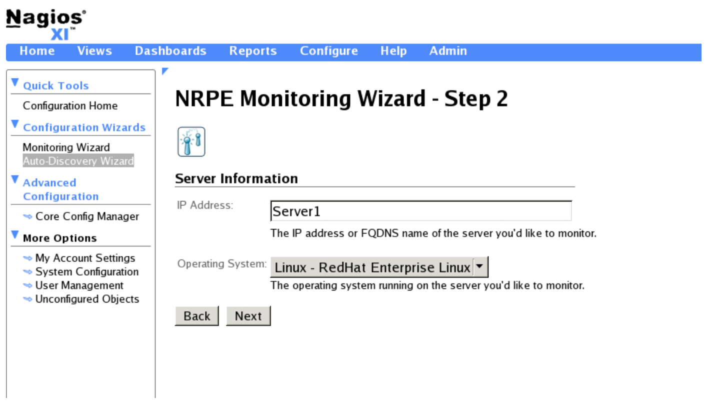
For example, if Server1 was created with the auto-discovery wizard, and you now want to add NRPE checks to get CPU, Memory, and Disk information from the host, you must first install NRPE on Server1. If Server1 doesn't already have NRPE on it, and is one of several common server types, such as a Windows server, Red Hat, or Ubuntu, the XI developers have an agent package designed to work with XI specifically at:

```
http://assets.nagios.com/downloads/nagiosxi/wizards
```

Once the agent is installed on Server1, simply run the NRPE wizard on the server from the configuration tab of the XI user interface, as shown in Figure 1, entering the IP or FQDN of the server, and choosing the type from the drop-down list. The wizard will then display a pre-configured subset of available check commands relevant to your server type, and provide text-entry fields for you to specify custom settings or additional commands if you wish.

Auto-Configuration Gotchas

Static configuration files may still be maintained in etc/nagios/static. So it's entirely possible to run your own scripts, or auto-generation tools such as those



NRPE Monitoring Wizard - Step 2

Server Information

IP Address: Server1
The IP address or FQDNS name of the server you'd like to monitor.

Operating System: Linux - RedHat Enterprise Linux
The operating system running on the server you'd like to monitor.

Figure 1: The Nagios XI NRPE wizard

included with Check_MK, provided you configure them to write their configuration to the static directory. I can't deny that the automated configuration features in XI have, ironically, complicated things a bit for those of us who have reason to maintain the configuration manually. While in the Nagios Core universe, there is a single way to configure Nagios (text files), there are three ways to configure Nagios in the XI universe (text files, NagiosQL, and XI wizards), and although the three co-exist as well as I think it's possible, it can become burdensome to ensure uniform parameters if the administrators mix-and-match their configuration methodologies in XI. I'll give you an example.

Larry, his brother Darryl, and his other brother Darryl all work at bloody stump lumber mill, where they recently purchased a Nagios XI server to monitor their growing sales Web-application server farm. Larry was a UNIX admin in college, so he prefers to edit the config files; Darryl likes to have fine-grained control over the config, but isn't very good in vim, so he uses the XI advanced configuration section; and other Darryl would rather be watching football (an American sport, similar to rugby but with armor), so he just runs the wizard for everything. Each of the brothers has a server running SSHD that he wants to configure in XI.

When other Darryl runs the auto-discovery wizard on his server's IP, XI scans the host and automatically configures a host check and a check_tcp service check for the SSH port. It then pushes the config to NagiosQL, which commits it to the DB, writes out the configuration, and restarts the daemon.

Darryl, meanwhile, sets up his host using the NagiosQL forms directly, but instead of choosing check_tcp, he chooses the check_ssh service, which does pretty much the same thing, but returns slightly different output. He also names the service "ssh" instead of "SSH" like the wizard does.

Larry, meanwhile, has really done his homework. He already has a service group for ssh servers in the static config files he created, so rather than doing all the typing and clicking that his brothers do, he simply adds his server to the ssh_servers service group, and the rest takes care of itself. The problem is, his service group inherits a different set of templates than NagiosQL, so although his service check uses the same name and check command as the wizard, his polling interval is different, and he has a different notification target for service warnings.

In this way the brothers end up with three different definitions for the same service, which might not be a problem immediately, but will cause all manner of headaches if and when they want to integrate Nagios with another tool, or generally try to do any sort of automation using their monitoring server.

I admit these sorts of disconnects are possible with text configuration files, but my point is the text configuration encourages administrators to use templates to normalize the configuration, as Larry did in the example above. The automated tools by comparison encourage isolating the configuration at the host level, because it's easier for the automated tools to parse them that way. Thus in Larry's configuration, we find a single services.cfg wherein every service is defined and assigned a host group, while in NagiosQL's configuration we find a services directory with a single file for each host. The former makes it pretty easy to verify that all the service checks for every host are implemented in the same way. The latter makes it much more difficult.

Further, in my experience, the disdain that people like Larry naturally feel for people like other Darryl generally discourages them from paying close attention to what people like other Darryl are doing. In fact, merely inviting other Darryl to configure the monitoring server with wizards might trigger a tendency in Larry to go off on his own and "do it the right way" using well-written static config files, which only exacerbates the problem by more widely diverging the configuration paths.

Whether this will be a problem in your shop will depend on how many hands are stirring the pot, and the extent to which the more clueful users are aware of the potential problem. The idea of delegating the configs is certainly tempting, and I'm not saying you shouldn't. If you do, my advice would be to use either the wizards or static config for service and host creation, and avoid using NagiosQL directly if you can avoid it (you could still safely use it to modify objects, just not to create them). That way, you can carefully set up the static config to ensure it references the wizard templates, or simply copy definitions from the NagiosQL files, and everything should remain pretty much uniform.

Automated Configuration for Passive Checks

One cool bit of functionality that is related to automated configuration in Nagios XI is the "Unconfigured Objects" feature. In the event that XI receives a passive check result for a host or service that it doesn't know about, it automatically generates an inert configuration for that host or service, and places it in the "Unconfigured Objects" section of the "Configure" tab. Administrators may then approve the inert objects, and they will become part of the running configuration. This is a welcome addition that I can imagine myself becoming reliant on, and it wouldn't be possible without the other wizards in place.

Auto-Discovery Is Dead, Long Live Auto-Discovery

Four or five years ago, a monitoring system's ability to perform auto-discovery seemed to be the feature that enabled forum trolls to distinguish the "cool" monitoring systems from the insipid wanna-be toys, and Nagios, being bereft in this respect, was in the latter group. At the time, it seemed like I couldn't read a monitoring-related Slashdot post without being bombarded with comments from the adherents for various commercial products who were forever chanting this strange "auto-discovery or death" rhetoric.

Why they chose that particular feature I can't guess. I've rarely in my professional career found myself in want of such a tool for Nagios, which is not to imply that options were lacking. On the contrary, the whining in the forums begat an explosion of these add-ons for Nagios in every language at every level of complexity. So numerous were they that groups of them would loiter in the parks at night, and in the morning they would flock outside the Best Buy entrance, hoping for work. As a group I think most of us found them unwieldy; they made strange assumptions and were overly enamored of XML.

Today the various auto-discovery add-ons for Nagios have either disappeared or have become abandonware. Yes, all of them, 100%. Some light googling retrieves only ancient blog posts from bygone tool-writers announcing or justifying the creation of their now-abandoned hot new auto-discovery tool for Nagios (now with extra XML!). Given the firestorm of controversy that once surrounded this topic, I find it disorienting that not only the tools, but even the trolls have utterly vanished. It's a vexing turn of events but not, I think, an unhealthy one for the Nagios community, and I suspect two things account for it.

The first is a plugin written by Mathias Kettner called Check_MK, which I covered at length in [2] and [3]. The second is Nagios XI, which has everything the trolls would expect to see in a "cool kid" monitoring system and more, especially configuration wizards. I can't prove it, but my suspicion is that real administrators with real problems to solve discovered Check_MK and never looked back, or convinced their managers to pony up and buy them XI (or both); at the same time, one look at the XI screenshots caused a massive spontaneous troll migration away from the monitoring forums and toward dpreview.com or perhaps YouTube, where they all live happily trolling it up to this day (sorry about that, YouTube).

I jest, but truly, I think my hypothesis has some merit. If you're the kind of sysadmin who likes to get hacky with Nagios Core, you're going to write a one-liner for auto-discovery and be done. (The old auto-discovery tools wouldn't have given you enough control, anyway.) If you're the type who just wants to install something without getting too involved, you'll install Check_MK and be done. And if you're in the market for an effective, established, polished commercial product with support behind it, then you'll buy Nagios XI and be done. Even if it is an untestable assertion, I think I've decided to believe it on the grounds that it's also poetic; the wizards, after all, appear to have conquered the trolls.

Take it easy.

[1] https://www.usenix.org/publications/login/december-2012-volume-37-number-6/ivoyeur-nagios-xi.

[2] https://www.usenix.org/publications/login/june-2012-volume-37-number-3/ivoyeur-changing-game-part-4.

[3] https://www.usenix.org/publications/login/august-2012-volume-37-number-4/ivoyeur-gift-fire.

/dev/random
What is UNIX?

ROBERT G. FERRELL

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

I stumbled across this piece I wrote as a writing sample in 1997 in a forgotten directory on a seldom-accessed backup drive. I can't remember what I was auditioning for; whatever it was, I probably triggered some sort of mental aberration in the editor and she had to go to a sanitarium or monster truck rally. That happens with my writing on a fairly regular basis. Whatever the case, now that this vintage nonsense has had 16 years to ferment, I thought I'd pop it open and give y'all a hearty slurp. Remember: 1997.

* * * * *

I encounter an uncomfortably large number of people who, for one reason or another, want desperately to convince some unsuspecting slice of the population that they are technically knowledgeable, if not downright expert, at All Things UNIX. Most of them, however, have neither the aptitude nor the patience to endure the years of dedication it takes to get this sort of experience on their own. It is a bewildering fact that virtually all of these hapless souls somehow end up in my office seeking advice, training, or consolation (usually in that order). So that I might save my vocal cords untold wear and get in at least a minimal amount of work for the agency that pays my salary, I have come up with a short guide for all those who want to speak UNIX without actually knowing anything about it. A little time invested memorizing this and you can add at least $25 an hour onto your consulting fee. That will help to offset the psychiatric bills.

Q: What, exactly, is UNIX?

A: UNIX is an operating system that has been around since 1970, almost as long as that shrimp cocktail in the back of your fridge. When you push down those little buttons on the keyboard, magic pixies carry the scan code for each button to the keyboard controller fairy, who puts them all in an envelope, licks it, and shoots it up to the processor in one of those vacuum-powered hamster tube things, only digital. The processor steams open the envelope and arranges the scan codes like building blocks, creating a surprisingly realistic model of the Taj Mahal before reluctantly knocking it over and getting on with the business at hand. The scan codes are instructions to the program currently running in the foreground, which may be GUI or just slightly sticky.

Q: How does UNIX work?

A: UNIX runs as a series of processes. These processes can be started at boot time or later on by a user or another process. When one process starts a second one it is said to **fork**. The result of forking is, not surprisingly, a **child**, and the original is then a **parent**, which should be apparent. If the parent process dies, the child is orphaned and Social Services has to be called in. Processes that are supposed to be controlled by other processes but aren't are defunct and become eligible for a government bailout. Each process has an identifying number, the PID. The parent of a child process has another number, called a PPID. To understand why this is, you need a PhD. Users have a number, too, known as a UID (which should never under any circumstances be confused with an IUD). The most *useful* number for you to know is probably the one for the help desk.

Q: How about UNIX and the Internet?

A: UNIX and the Internet are like two socks in a shoe. While all other operating systems have to be clever and sweet talk their way onto the Internet, UNIX just strolls right in without even breaking a sweat. When UNIX systems communicate with one another over the Internet, they talk TCP/IP or UDP, through little revolving doors called ports. There are thousands of ports available, but a lot of them have been reserved for visiting diplomats or taken over by applications no one except other software developers and family members will ever use because the hinges squeak. This is called progress, and it gets people into trouble. Ports are identified by simple decimal numbers, such as 25. This is surprising and a little misleading, since virtually everything else about UNIX is in hex, octal, binary, or worse.

Q: What the heck are daemons, anyway?

A: There are special processes in UNIX that slink around in the background listening for requests for service. These are called daemons, because they can turn on you when you least expect it. Some of them peek out through those ports we just talked about, waiting for the odd packet to stray too close and slurp it up like a frog snatching a dragonfly. These processes are controlled by the all-powerful, all-knowing INETD, without which ARPANET would have been just (tremendously expensive and highly classified) cans with strings running between them. INETD is really a whole suite of listening programs started at the same time, including TCP/IP, FTP, UUCP, Telnet, RCP, and more. As a result, if your INETD is DOA, you can't even send out an SOS. LOL.

I could go on ad infinitum, as many of my friends will readily attest, but in the interests of brevity I will now turn to a list of the essential terminology you absolutely must be able to bandy about to impress and, if necessary, confuse your clients.

cat: Reads a file and prints it to the screen, or combines files, or appends to a file. Likes milk. Opposite of dog().

cc: Compile a C program. Or copy a message to someone. CC is a cool command to toss casually into conversation because it has so darn many options, like -dalign, -fnonstd, -qp, -W[p02abl], -xsbfast, and -xstrconst.

chmod: Make files inaccessible or render them non-executable. Then magically fix them for your awestruck and deeply grateful clients. Example: *chmod 000 | find / * -name -print*

cmp: This one is good just to impress people with its voluminous and highly cryptic output. Example: *cmp -l /etc/disktab /usr/adm/messages*.

cof2elf: I don't want to explain what this one does. I just like the way it sounds, like something you might hear in Elrond's infirmary.

cpio: Archives files onto or off of a disk. Use in conjunction with r2d2. Also has a plethora of options. Example: *cpio -i bBcCdEfHIkmMrRsStuvV*. Gesundheit.

crontab: Run a command or shell script at a set time. Not to be taken internally.

crypt: Bury a file so deep in gobbledygook that its meaning can't ever get out. A favorite of legislators and instruction book authors worldwide.

df: See how many devices are attached to the system, and how much can be deleted from them.

du: Report how much disk space is being wasted by useless fluff such as /vmUNIX.

egrep: An old version of *grep*. Or a long-legged water bird if your mouf is foo ob peanub bubba.

find: A career-enhancing command with more options than Georgia has peaches. A well-written find command can approach a Perl script for unreadability:

```
find . \? -mtime -4 | xargs grep [Oo]bfuscate -o -prune -perm 444
-exec lp {} \;
```

ftp: Share those unwanted files and programs. The Internet equivalent of dumping stuff out on the curb.

grep: Feel around in a file for something. Not legal in all states. Be careful you don't prick your finger and get AIDS.

head: Print the first ten lines or so. A useful command, mostly for its puerile suggestive value.

hostid: Spits back a mysterious-looking hexadecimal number for no apparent reason.

kill: The ultimate aggressive sysadmin tool. Looks bad, is worse.

ln: A great way to confuse any file system hopelessly. Example: *ln -f pwd kill -9 1*.

nohup: Disallow current or former drill instructors from logging onto the system.

od: A command that is as descriptive as it is functional. Example: *od -bv kernel.o. > / etc/inittab*.

pack: A good, well-rounded command. Example: *pack -f - **.

pg: A great way to ensure that important material is read. Find a nice, long text file and then try *pg -r -1* file.

red: Print files with all characters far to the left of center (warning: process obsoletes itself after a while).

rksh: The most versatile and useful shell. Add the line SHELL=/bin/rksh to /.profile right now.

stty: One of the all-time great UNIX commands. Virtually any combination of options and modes is a veritable work of poetic `art`. Example: *stty -a cstopb parodd -ixoff -olcuc flusho stappl ctrl-char dsusp cooked -dtrxoff x cibrg r setctbrg*.

talk: A clever little utility that might get you committed. Example: *talk / whoami*.

tar: Rolls up your files into a small, viscous black ball for storage. Surprisingly, it is not GUI.

timex: A version of **rollx** that uses a plainer font and a lot fewer system resources.

touch: Update and otherwise control files. Leave yourself open to various lawsuits and possible criminal indictment in the process. See **grep**.

tee: Balance the ball on a little wooden pegleg. Then hit that sucker hard. Yelling "fore!" optional.

truss: Trace system parameters during program execution. Primarily for customer support, especially of the lower abdomen and pelvic region. Example: */usr/bin/hernia > truss*.

uuglist: Stops the system clock abruptly.

vc: Destroys the target file, then disappears into the operating system before you can kill it.

wall: A collection of bricks. Or Just Another Perl Hacker.

wc: A little chickadee that really doesn't do much of anything. Example: *wc | find / * -print*.

who: Checks your system for owls.

yacc: Checks your system for yaks.

xargs: Checks your system for, um, zargs.

Book Reviews

ELIZABETH ZWICKY, WITH MARK LAMOURINE AND TREY DARLEY

Regular Expressions Cookbook, Second Edition
Jan Goyvaerts and Steven Levithan
O'Reilly and Associates, 2012. 575 pp.

ISBN 978-1-449-31943-4

This is an excellent reference work, which will some day—perhaps many days—save you untold effort. Yes, this book goes over how regular expressions work, but where it shines is in providing practical recipes that take into account not only the details of regular expressions but also the details of the world. For instance, in processing ZIP codes it notes that there is one ZIP+4 (and only one) that contains letters, but then notes that your mail to Saks' shoe department will deliver just fine without it anyway, and recommends you just ignore it. *Regular Expressions Cookbook* is happy to suggest combining regular expressions and code for readability and performance.

The book is admirably agnostic, bearing in mind the possibility that you will want to deal with phone numbers and postal codes from outside the US, use non-ASCII character sets, and parse Windows-specific values. Although it is impossible to cover all the languages and situations where you may want to use regular expressions, it covers a good wide variety, including uses in text editors, and provides references to useful testing tools. I might not have picked up this title had I not been looking at books to review (after all, I already own two books on regular expressions), and that would have been a real loss. Even if you're already a pro with regular expressions, this book will point out details and save thought; if you're not, it will help you without making you too terribly dangerous.

Python for Data Analysis
Wes McKinney
O'Reilly and Associates, 2012. 432 pp.

ISBN 978-1-449-31979-3

This is a specialist's book. If you read the title and think, "Wow, how handy; I have this data I know how to analyze, and I know some Python, and learning all of R seems a bit unwieldy when I could do all my processing in Python," then you really want this book. If you are fully confident in your skills in one thing or another, either Python or data analysis, and you're interested in teaching yourself the other with a bit of assistance from a reference work, this title would still be a good choice.

If you need hand-holding, move on. This is the kind of the book that says airily that there are many ways to get a random sample of items, with different performance implications, and then provides an example of exactly one of them. You are expected to already know what performance implications it has and to think of the rest for yourself. (It's hardly an unusual problem, after all.) The book also, in the Macintosh installation instructions, tells you to download a software package without specifying where you would download it from. (For one thing, the answer is easily findable in search engines, and for another, it already told you in the Windows instructions—why you would read the Windows installation instructions in order to do a Macintosh installation, I do not know.)

I'm probably going to use my copy, if I can pry it out of the hands of the Python guy at work who has been asking me wistfully for months whether I know anything about pandas (the Python library, not the bamboo-eating animals).

Managing the Unmanageable: Rules, Tools, and Insights for Managing Software People and Teams
Mickey W. Mantle and Ron Lichty
Addison Wesley, 2012. 406 pp.

ISBN 978-0-321-82203-1

There are some good insights in this book and some pithy rules of thumb; it's an approachable book about managing programmers, which will probably help many managers, especially those who manage groups composed entirely of programmers turning out new projects. All the same, I couldn't love it. Some of the problem was the authors' style, which doesn't work for me (and that's a highly personal thing, so you should check out the book to see how you feel about it yourself). Some of that was the laser-like focus on traditional programming. The authors are quite condescending about people who program in scripting languages or, worse yet, use GUI tools, and they don't care about non-programmers—including QA, system administrators, designers, and technical writers—at all. Apparently in their world, programming managers don't deal with such people. Also programming managers only manage development groups, not support groups.

The book does a much better job than most on the nitty-gritty of interviewing and hiring programmers, and the rules of thumb it presents get a nice wide range of perspectives represented. If the style and the tight focus work for you, this book is a good place to start in the programming management game; the content strikes me as mostly right, if occasionally over-opinionated.

Python for Kids

Jason R. Briggs

No Starch Press, 2012. 313 pp.

ISBN 978-1-59327-407-8

Super Scratch Programming Adventure!

The LEAD Project

No Starch Press, 2011. 158 pp.

ISBN 978-1-59327-409-2

These two books take superficially similar approaches; both of them use video game development to motivate kids to learn to program. *Python for Kids* is aimed at kids age 10 and up, whereas *Super Scratch* is geared toward kids 8 and older; however, even apart from the age difference, these books will suit radically different children.

Super Scratch is in a comic book format, and it focuses on a language designed for children. *Python for Kids* is a standard introduction to Python, gently modified for children. For a lot of kids, particularly kids on the younger end of the age range, *Super Scratch* is going to be the more attractive option. *Super Scratch* starts off with an intergalactic adventure, and gets to making a cat move on page 21 (and that includes 10 pages kids can skip and a couple in which you're already looking at the cat on your screen). *Python for Kids* gets halfway through before it starts covering a game, and it begins by adding numbers. If your kid wants to program for programming's sake and is likely to be offended by having things dressed up with irrelevant space-going comic strips, *Python for Kids* is the better choice.

Of the two, I think *Super Scratch* does a better job of bringing programming to kids because it talks about debugging, for example, and does a better job of providing questions kids are likely to be interested in answering. Of course, *Super Scratch* also starts with a programming language designed for kids, which is a major leg up.

Python for Kids has to work within the limitations of Python, which requires a certain amount of typing and discussing integers and the like. On the whole, I think *Python for Kids* copes pretty well, although my head exploded at the paragraph "Why is a giraffe like a sidewalk? Because both a giraffe and a sidewalk are things, known in the English language as nouns, and in Python as objects." OK, first of all, a giraffe is not a noun. The word "giraffe" is a noun. Second, there is no guarantee that nouns are things or things are describable with nouns. "Beauty" is a noun, but beauty is not a thing, and a pregnant giraffe is a thing, but only describable with a noun phrase. Third, objects, nouns, and things have very different characteristics. A giraffe is even less like a Python object than it is like a noun. Fourth, while giraffes and sidewalks are like each other in their degree of dissimilarity from both nouns and Python objects, this totally fails to illuminate me about Python objects and doesn't come up again. Presumably, if I were 10 years old this would bother me less, but I still don't think it would do much to help me understand Python objects.

My test child is 8; she has encountered *Python for Kids* in its previous online existence, and by all reports was unimpressed. (Like me, she is not interested in programming for programming's sake, so she's pretty much out of its target audience in several directions.) She was quite taken with both *Super Scratch* and the Scratch programming language, and although she required a little help to make the connection between the book and the screen, she was enthused about working with it. At which point, using only the instructions she could not proceed without in *Super Scratch*, she carefully recreated in Scratch…the first turtle drawing exercise in *Python for Kids*, which she ran into at least six months ago. Go figure.

Meanwhile, these experiences seem to have communicated only some of what they were trying to. Days later, we looked at the screen saver on my computer, drawing fancy flowers, and I said to her, "You know that's a computer program, right? People write programs that draw flowers." "Really?" she said. "Huh. I've written three programs, you know." Score a point for empowerment; take it away for not having connected that experience to the things computers do that she loves.

—Elizabeth Zwicky

Assembly Language Programming: ARM Cortex M3
Vincent Mahout
Wiley-ISTE, 2012. 246 pp.

ISBN 978-1-84821-329-6

I'm one of those people who thinks that software developers should be aware of the workings at least one and probably two levels below where they are working. That would be reason enough to want to read up on assembly language. The recent growth in consumer and hobbyist ARM systems makes that a good selection.

Modern compiled and scripted languages plaster over so much of the arcana that goes on at the machine level that there's no good place to just jump in and get coding. Mahout takes about five chapters to get to some working code. Those chapters cover the ARM architecture and elements of assembly syntax.

The final four chapters are where this book earns its keep. Chapter 6 demonstrates how to implement logical constructs such as looping and branching blocks that in a high-level language might be represented with a single keyword and a couple of curly braces. Chapter 7 covers modularity and constructing procedures and functions, including detailing the ARM-calling convention. Chapter 8 is about handling hardware- and software-generated exceptions. Chapter 9 walks through the creation of a complete simple program, detailing each of the steps required to assemble, link, load, and run the program. Remember, in assembly you're responsible for initializing the stack and all of the memory you've allocated before branching to your program.

Aside from the long exposition that must happen before getting to the meat, this book has several other quirks that effect the reading experience. The contrast of the graphics and code typesetting detract somewhat from the otherwise clean layout. The code boxes use an unnecessarily dark background that makes the black text hard on the eyes. Many of the graphics appear to be color images converted to gray-scale without any additional touch up.

There is, throughout the book, an odd use of language, at least to my American English ear. When describing the sample project used to illustrate the use of the assembler/linker/loader tool chain, Mahout begins, "This entire project is of restricted algorithmic interest." I probably would have chosen "limited." The word choice doesn't confuse the meaning but can stand out as you read. If this issue had happened once I would have passed it off as a quirk, but it occurs repeatedly. Mahout is a native French speaker. The book is published and printed in the UK. I would have thought that an English-speaking editor would have spent a bit more time polishing simple word choices.

The number of ARM family variations and the fact that ARM SOC (System on a Chip) are manufacturer-specific mean that Mahout can't talk about things outside the core spec itself. He chose a fairly recent mobile core, the Cortext M3, as his working model.

In the same way that there are different flavors of compiler for high-level languages, there are multiple assembler environments for a given processor family. Mahout based his book on the Keil ARM-MDK (Microcontroller Development Kit). Kiel has been purchased by ARM, and the "Lite" version is available from the arm.com Web site for free and is capable of demonstrating all of the work in the book. Appendix D of the book details how the GNU-GCC assembler (specifically the assembler from the Sourcery G++ suite) differs from the ARM-MDK.

This is certainly not a book for a novice programmer. If you need proper ARM references, the ARM site itself has those for each of the processor flavors, and for a specific SOC you will need the manufacturer references. I don't want to recommend against this book for an experienced coder who wants to taste assembly language or get a look under the hood of an ARM system, but I will warn that reading it will take some dedication. This might be a good book for the classroom, but I would hope that the teacher would re-organize or gloss the early chapters and somehow get the students straight into some hands-on work. I'm still looking for the K&R or Stevens of modern assembly.

Super Scratch Programming Adventure!
The LEAD Project
No Starch Press, 2011. 158 pp.

ISBN 978-1-59327-409-2

Since the invention of Logo and the turtle in 1967, people have been trying to create languages and environments that invite kids to learn and explore programming. The Scratch programming environment was created at the MIT Media Lab's Lifelong Kindergarten project in 2006. An environment like Scratch still has to be presented to kids in a way which helps them engage.

Super Scratch Programming Adventure is published in North America by No Starch Press, but was developed and written by The Lead Project, a collaboration between the Hong Kong Federation of Youth Groups and the MIT Media Lab.

When I got this book in the mail, the first thing I did was set up Scratch on my 13-year-old daughter's computer. After supper I handed her the book and walked away. My daughters have both been resistant to learning programming from me and I generally don't push except occasionally to offer some new toy to try, like this. Several hours later she was still playing with Scratch. I'll call that a win. She continued to play with it on and off for several days.

When I asked her what she thought of the book she said she liked it in general. She thought the comic book presentation was a bit young for her, but that it didn't detract once she got into it. She played with each of the games and explored some of the variations, but she didn't follow the progression of the book faithfully and she didn't formally complete any of the "lessons" in the way the authors intended. She said that a big part of Scratch is creating the artwork for the stories. She doesn't consider herself an artist so she stopped when she ran out of things to do with the (large) provided set of "avatars."

With the experiment over I started working through the book myself.

Scratch is a programmable storytelling environment. The user can draw characters (avatars) and backgrounds or use some from the provided library. The stories are programmed by dragging and dropping a set of tool bar objects, representing logic constructs and methods, on various other objects, such as avatars or drawing pens. A loop or code block actually wraps around the contained steps so the nesting and scope are visually clear. Method parameters are text boxes whose contents the user can change. Types of programming objects (logic, avatars, drawing tools) are color coded. Scratch and the programming examples for the book are available online from the URLs provided.

Super Scratch Programming Adventure! has the typical cast of characters: the human, for the reader to identify with, and a collection of animals and aliens to play the roles of helpers and villains. The adventure is presented as a series of crises to be overcome. Each crisis has a program that starts out working, but not in the desired way. The text guides the reader through the process of changing the program to solve the problem. The end of each chapter suggests some other ways to experiment to see the effect of different changes.

The chapters present the typical concepts of variables, code blocks, looping, and procedures in a purely practical and experimental way without any attempt at theory. The students get a visceral understanding through their play. In a classroom setting a teacher might have a discussion session to get the students to talk about the implications of what they've done, but that's not part of the text. By the end the students have played with 2D motion, sound, color, and user interaction.

As I mentioned, Scratch is a storytelling environment and *Super Scratch Programming Adventure !* is a storybook. Storytelling isn't much fun without an audience. Scratch provides a means to upload stories to a public Web site, and the book encourages the student both to do that and to explore the stories there for additional ideas.

My experiences with recent middle and high school "computer" classes have been disappointing, and I expect it's not uncommon. Recent activities in the UK [1, 2] and this book from Hong Kong (not to neglect any US efforts I'm not aware of) give me hope that middle and secondary computer education may yet grow beyond teaching proprietary word processing software. This book is probably best suited to a middle school environment. It's going to require creative and enthusiastic teachers to foster the sense of expressive freedom needed so that the students never know they're "programming." I'd certainly recommend this book and Scratch to an involved parent whose child has expressed an interest in using computers for something more than viewing videos and playing games. This book will stay on my daughter's shelf, and it may yet call her back to play.

[1] http://www.guardian.co.uk/politics/2012/jan/11/michael-gove-boring-it-lessons.

[2] http://www.raspberrypi.org/about.

—Mark Lamourine

The CERT Guide to Insider Threats: How to Prevent, Detect, and Respond to Information Technology Crimes (Theft, Sabotage, Fraud)

Dawn Cappelli, Andrew Moore, and Randall Trzeciak

Addison-Wesley Professional, 2012. 432 pp.

ISBN: 978-0-321-81257-5

Carnegie Mellon's CERT Insider Threat Center has (in collaboration with various law enforcement agencies) amassed a substantial data set of criminal cases involving malicious trusted insiders. Through analysis of this database the authors (all of whom work for the Insider Threat Center, by the way) have identified distinct profiles associated with fraud, IP theft, and sabotage. The authors use these case histories to great effect throughout the book to drive their points home.

They won my heart early with this line in the book's overview: "If you learn only one thing from this book, let it be this: Insider threats cannot be prevented and detected with technology alone." For managers, faced with a difficult and a subtle problem, the temptation to throw an expensive black box at it, put a tick in the box, and assume that it does what it says on the tin can be irresistible. Couple that with the trend of outsourcing critical functions and you've got a recipe for danger.

The first four chapters provide a fairly high-level overview of case histories, profiles, motivations, and mitigation strategies. The rest of the book is devoted to issues specific to the software development life cycle, best practices for prevention and detection, suggested technical controls, and in-depth examination of selected cases. Technical types can glean useful insights from this book, but to get the maximum benefit, try organizing a reading group with the folks over in HR.

Advanced Internet Protocols, Services, and Applications

Eiji Oki, Roberto Rojas-Cessa, Mallikarjun Tatipamula, and Christian Vogt

Wiley, 2012. 260 pp.

ISBN: 978-0-470-49903-0

I marvel that such a slender volume can pack such a wallop of disappointment. Based on the publisher's description, this book sounded like it would pair nicely with the new edition of *TCP/IP Illustrated, Volume 1*. I hoped it would fill in the gap on topics that Kevin Fall omitted for brevity's sake (i.e., dynamic routing protocols, traffic shaping, QoS, and so forth). Sadly, this book contains so many errors (both linguistic and technical) that I cannot imagine an editor was ever even in the same room with the manuscript. This is a rambling 260-page paraphrasing of RFCs that somehow manages to be less readable than the RFCs themselves. This book lists for $US 99.95. For that amount of money you can buy two copies of Fall's opus. Do yourself a favor and skip this one. Hopefully, Fall is hard at work updating *TCP/IP Illustrated, Volume 2*.

—Trey Darley

Conference Reports

Complete conference reports from HotPower '12, MAD '12, and OSDI '12 are available online at www.usenix.org/publications/login

10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)

Hollywood, CA
October 8–10, 2012

Opening Remarks

Summarized by Rik Farrow (rik@usenix.org)

Program Co-chair Amin Vahdat opened the conference, telling the audience that this year's attendance was high, close to but not greater than the record for OSDI. Vahdat explained the review process: 25 out of 215 papers were accepted, producing 1079 written reviews. Authors were sent reviewers' comments so that they could address any concerns about the research before they submitted their final versions of their papers.

Co-chair Chandu Thekkath presented the two Jay Lepreau Best Paper awards to the 26 authors of "Spanner: Google's Globally-Distributed Database" for Best Paper, and to Mona Attariyan, Michael Chow, and Jason Flinn for "X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software," the Best Student Paper.

The UCSC Cancer Genomics Hub

David Haussler, University of California, Santa Cruz (adjunct at UCSF & Stanford)

Summarized by David Terei (davidt@scs.stanford.edu)

Twelve years ago, two teams raced to sequence the human genome. Today, DNA sequencing technology is outpacing Moore's Law; it is the dawn of personal genomics. The first human genome cost more than $100 million to sequence, while in 2013, sequencing will cost $1,000 per personal genome.

Instead of simply looking at one reference genome to understand diseases, imagine looking at millions—this is where a lot of medical research will head. Cancer will be the main target as the disease is caused by changes to the human genome and is highly individual. There is also a willingness to try new ideas; the rest of the medical community is generally slower moving, said Haussler.

Thus, genomes are the key to the future of cancer treatment. A patient's genome will be compared to a database of other genomes to devise a treatment. Drugs targeted to the individual are needed and feasible.

The Cancer Genome Atlas is currently the largest such database, containing around 10,000 entries. Each entry consists of a biopsy of the tumor and normal tissue, with both sequenced and the differences analyzed and mapped to the first human genome reference to account for normal and abnormal variations between individuals. Generally speaking, there are 3 to 4 million normal differences between individuals; this variation among us is referred to as a persons "germ line." For sequenced genomes, a variety of mutations between the tumor and healthy tissue can be detected, including point mutations, deletions, duplications, inversions, and shifts.

The analysis of these mutations and the sequenced genome data itself are now being stored in the UCSC Cancer Genomics Hub (CGHub). CGHub provides a secure platform to host the confidential data and manage access to it for authorized researchers. The platform also supports an app-like model for authorized software that provides analysis and visualization tools to researchers. Each entry (patient) is around 100 GB when compressed; CGHub is currently designed for 50,000 genomes and holds 24,000 files from 5,500 cases. No collocated computing power is provided for now.

The analysis of these files is an interesting and challenging field for the systems and machine learning fields. Ideally, we would have large-scale, automated discovery of diagnostic signatures. (Think of how the prediction of treatment outcome would be improved if based on genetic data!) Here is where the systems community can help, by providing information on (1) how to handle big data, (2) how to analyze the data, (3) how to predict outcomes based on this analysis/data, and (4) how to make treatment recommendations from all of this.

Jeanna Matthews (Clarkson) asked if the benchmark challenge was available yet. Haussler replied that they will be announcing the benchmark challenge soon: read in the raw DNA and output how it was mutated. Ken Yocum (UCSD) asked Haussler to comment on consent from patients and privacy concerns for getting data into the database. Haussler said they were about to release some data without restrictions. In general, you need to apply for permission with NIH to gain access and abide by its privacy policy/expectations (prove you will do valuable research). Amin Vahdat (UCSD) asked about the cost for analyzing DNA data. Haussler replied that if the cost of sequencing is as low as $500 and the cost of analysis is $2000, there will be enormous pressure to also drive down the computational cost. They need better algorithms and novel techniques to accomplish this. For example, to compare every piece of DNA with 50 others currently takes a few weeks. Bryan Ford (Yale) asked whether Haussler could describe the major computational and storage roadblocks. Haussler said, in a word, I/O. Currently, individuals are writing small, isolated tools that create lots of intermediate files.

Big Data

Summarized by Jim Cadden (jmcadden@bu.edu)

Flat Datacenter Storage

Edmund B. Nightingale, Jeremy Elson, and Jinliang Fan, Microsoft Research; Owen Hofmann, University of Texas at Austin; Jon Howell and Yutaka Suzue, Microsoft Research

Jeremy Elson presented the Flat Datacenter Storage (FDS), a datacenter-scale blob store that has the agility and conceptual simplicity of a global store without the usual performance penalty. The novel approach taken by FDS to alleviate the network bottleneck is to multiplex the application's I/O across the available throughput and latency budget of the disks within the system.

Jeremy began the talk with a conceptual introduction to a "little data" platform—a highly utilized, tightly coupled multicore machine. This commonplace example illustrated the inherent problem with big data computation in that our traditional machine architectures do not scale. FDS attempts to provide the essential properties of little data platforms with the scale and performance necessary for big data application. This is realized through a novel combination of three attributes: a simple scalable blog store, decentralized metadata management, and a full bisection bandwidth CLOS network with novel distributed traffic scheduling. The performance gains of FDS come at the cost of the requirement of additional underlying hardware, in specifically the 1:1 matching between I/O and network bandwidth.

In the experimentation results, FDS showed read/writes to remote disks at up to 2 GBps—faster than most systems write locally. In addition, intra-disk high-speed communication of FDS allows for impressive data recovery benchmarks, with over 600 GB of data being recovered in 34 seconds within a 1000 node cluster. Applications built atop FDS are able to achieve world-record-breaking performance. MSR trumped Yahoo!'s 2009 record at Minute Sort benchmark by sorting data at a 15x efficiency improvement over the existing record.

Geoff Kuenning (Harvey Mudd) asked if the replication communication of the coherence protocol would present a potential bottleneck. Jeremy agreed that a replicated cluster would incur some necessary cost, but the applications flexibility to extend blobs and lazy disk space allocation will help alleviate this cost. Someone asked whether the authors had compared the performance of FDS against other commercial high-end storage systems (e.g., EMC, Hatachi, etc.) and how they expected FDS to scale further. Jeremy explained that they do not have the opportunity to do a 1:1 datacenter-scale comparison and, in addition, the linear scaling characteristics of FDS should allow for a scale of up to tens of thousands of nodes.

PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs

Joseph E. Gonzalez, Yucheng Low, Haijie Gu, and Danny Bickson, Carnegie Mellon University; Carlos Guestrin, University of Washington

Joseph Gonzalez presented PowerGraph, a framework for graph-parallel computation on natural graphs. PowerGraph was shown to produce order-of-magnitude computation improvement on natural graphs over the existing graph-parallel abstraction frameworks such as Pregel and GraphLab version 1. PowerGraph is now integrated into GraphLab 2.1 and released under the Apache license.

Joseph began by introducing natural graphs, graphs that, by definition, are derived from real-world phenomena, like a Twitter connection graph. Natural graphs are commonplace in machine learning and data mining problems throughout science. A distinct trait of natural graphs is their highly skewed power-law degree distributions that create a star-like graph motif. An example illustrating this was that of President Obama's twitter account and his many followers.

PowerGraph changes the model for structuring the parallel computation on a graph by splitting up a high-degree vertex and distributing it across machines. PowerGraph's gather, apply, and scatter (GAS) technique further enables the work to be divided, computed in parallel, and sent to a "master" machine where changes are applied on the master and synced to mirror nodes. The GAS method was shown to be applicable to all existing graph processing systems by a new theorem that states any edge cut can be reconstructed as a vertex-cut. Preprocessing and greedy-cut techniques can further increase the performance of PowerGraph.

In the experimental results, PowerGraph was shown to provide an order-of-magnitude performance increase over previous frameworks in both throughput and runtime. The scalability of PowerGraph was illustrated through PageRank iterations on the billion-node Yahoo! Altavista Web Graph, spending only seconds running across on 64 Amazon EC2 HPC nodes.

Terence Kelly (HP Labs) noticed that the graphs used to gather experimental data were small enough to fit into main memory, and proposed that a comparison between PowerGraph and a simple straightforward serial process would be interesting. Joseph agreed that PowerGraph can be beaten by a single core to a point, but it makes great strides on larger graphs and in the cloud. Aapo Kyrola (CMU) inquired about the difference between the approach of PowerGraph and previous vertex partitioning techniques. Joseph explained that the objectives of PowerGraph include an asynchronous operation and the ability to transform edge data. Aapo also made the same point as Terence Kelly, that it is often possible to compute graph problems on a single host, without partitioning.

GraphChi: Large-Scale Graph Computation on Just a PC

Aapo Kyrola and Guy Blelloch, Carnegie Mellon University; Carlos Guestrin, University of Washington

Aapo Kyrola introduced GraphChi, a natural graphic processing framework designed to compute on a single desktop PC via appropriate use of data structure and algorithms. GraphChi was written in 8000 lines of C code and is also available as a Java implementation.

Aapo began by setting the assumption that most large-scale natural graphs (e.g., Facebook social connections) have billions of edges yet can be reasonably stored on a single hard drive and, therefore, may not need the added overhead and cost required by the cloud if the computation can be handled on a single machine. The GraphChi model, similar to that of PowerGraph, is designed to exploit the inherent characteristics of natural graphs. Perhaps more of a fan of music than politics, Aapo used Lady Gaga's Twitter followers to illustrate the high-degree vertices trait of natural graphs.

One way in which GraphChi enables sequential scalable graph computation on a single machine is through increased I/O performance gained through optimizations made to alleviate random-access reads on disk. The Parallel Sliding Window (PSW) works by loading subgraphs into memory, one at a time, running computation, and returning that subgraph to disk. Heavy preprocessing is required on the graph to sort the vertex and edges in a way that both in and out edges of a directed subgraph can be extracted per load interval.

In the experimental results, Aapo showed that GraphChi performs reasonably well with large scale Big Data graph computations on a single Mac Mini machine. For problems dealing with complex computational issues, a 4x speedup was recorded by running GraphChi across four cores. The same speedup was observed with problems involving high I/O as GraphChi would saturate the I/O lines with only two concurrent threads. Graphs with billions of edges required less than an hour of preprocessing.

Mohit Saxena (Wisconsin) asked if their research compared PWS with OS techniques for memory mapping or SSD as a cache. Aapo explained that since the graph is passed over once in its entirety (and then flushed), OS caching techniques don't really apply.

Privacy

Summarized by Edmund Wong (elwong@cs.utexas.edu)

Hails: Protecting Data Privacy in Untrusted Web Applications

Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, and John C. Mitchell, Stanford University; Alejandro Russo, Chalmers University

Web platforms, such as Facebook, currently host many third-party apps that access private user data. It is hard to place trust in third-party app code; developers of such apps, due to malice or ignorance, may leak private data. However, even if developers are well-meaning, it is hard to trust their app, as building secure Web apps is difficult. Typically, developers implement security policy in an error-prone fashion, by injecting if-statements in application logic. A typical platform protects user data by allowing users to decide whether an app gets access to data, but the platform does not control how the app uses said data.

To address this problem, Deian Stefan from Stanford University presented Hails, a Web platform framework that enables security policies to be explicitly specified alongside data. Hails aims to be deployable today, usable by non-security developers, and suitable for building extensible Web platforms. In Hails, the trusted platform provider hosts untrusted apps and enforces security through language-level information-flow control (IFC) techniques. Stefan argued that, unlike Hails, previous IFC systems, such as Aeolus, HiStar, Nexus, or Jif, provide no guide for structuring apps, require policies that are hard to write, are not appropriate for dynamic systems such as the Web, and/or require modifications to the entire app stack.

Hails introduces a new design pattern, Model-Policy-View-Controller (MPVC), which is an extension of Model-View-Controller. In MPVC, the model-policy consists of the data model and the policy associated with the data. The policy follows the data as it flows through the system and specifies where data can flow. The view-controller components provide application logic and user-interface elements;

they do not implement security policy and are not trusted by users. View-controller components invoke model-policy components to store/fetch user data, and a Hails runtime enforces that the policy is followed end-to-end.

Hails is implemented as a Haskell library that enables quick turnaround on API design and allows developers to use their existing tools and libraries. Part of the library is the Hails runtime which provides an HTTP server that runs the view-controller. To demonstrate Hails in action, Stefan presented GitStar, a Web site for hosting source code much like GitHub, except the platform is provided by Hails and untrusted apps run atop this platform, unlike the monolithic structure of GitHub. In GitStar, the model-policy consists of data on projects and users, and third-party developers can build apps (view-controllers) to implement functionality, such as a code viewer or a wiki. Hails ensures that these apps cannot leak data even if the apps access project and user data. Stefan evaluated the usability of Hails by asking five developers to implement apps using Hails. These developers thought that Hails greatly simplified the process of writing security policies and securing their apps. Stefan also showed that Hails was faster than Sinatra, another Ruby Web application framework, for small Ruby apps but slower than Apache running PHP.

The first questioner asked how Stefan evaluated the usability of Hails for users. Stefan reiterated that the five developers (including one high-school student) who were asked to develop on Hails were successful in doing so; these developers found that Hails greatly simplified the process. Stefan was particularly excited by this result because he felt that developers using other IFC systems were typically experts in IPC, not Web developers. Jonas Wagner (EPFL) asked whether there were obstacles to applying the techniques used in Hails to a framework in Ruby on Rails or Python and whether policy can be enforced without modifying the language runtime. Stefan replied that other languages were considered, but Haskell was chosen due to control over side effects. Stefan said that for any other language, the compiler must be modified to support Hails. However, Hails currently allows developers to write apps that call untrusted (Linux) executables and referred to the paper for a further discussion on the use of untrusted executables within Hails. Peter Goodman (U of Toronto) asked which compiler was used with Hails; Stefan replied that GHC was used.

Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels

Alan M. Dunn, Michael Z. Lee, Suman Jana, Sangman Kim, Mark Silberstein, Yuanzhong Xu, Vitaly Shmatikov, and Emmett Witchel, The University of Texas at Austin

Alan Dunn presented Lacuna, whose goal is to provide forensic deniability: no evidence is left for a non-concurrent attacker once the program has terminated. Dunn argued that current approaches, such as private browsing and secure deallocation, still leave traces of private data because these approaches are hindered by the lack of proper system support. Lacuna's goals are to protect a user's privacy even under extreme circumstances—even if the machine is compromised at the root level or is physically seized—while maintaining usability. Lacuna supports running private applications (i.e., those that preserve a user's privacy under Lacuna) alongside non-private applications; supports a wide variety of private applications; and has reasonable overhead that is only incurred on private applications.

Lacuna achieves these goals by running applications inside erasable program containers and providing privacy-preserving I/O channels from these containers to hardware. The erasable program containers are virtual machines (VMs), where I/O can be intercepted by the virtual machine monitor (VMM) as necessary. Lacuna provides several I/O channels: disk I/O is encrypted before leaving the VMM and decrypted within the VMM upon being read back. For hardware that supports hardware virtualization (e.g., USB, network), Lacuna allows the driver running within the erasable program container to control and communicate directly with the hardware, bypassing the host OS in the process. This approach requires no modifications to host drivers, and any code running outside the erasable program container never sees unencrypted data. For hardware that does not support hardware virtualization (e.g., the graphics card), Lacuna provides software proxies that are placed close to where data is pushed or pulled from hardware, in host drivers or even on a graphics card. When a contained application performs an I/O operation with this type of hardware, the VMM passes the data for the operation to/from that hardware's associated software proxy via a cryptographically secured channel. This approach requires no modification to guest applications, and any residual data that may remain in the host OS is cryptographically erased by deleting the encryption/decryption keys when the channel is no longer in use.

Dunn then described how he evaluated Lacuna, which was implemented as a modified version of QEMU-KVM (a virtual machine monitor) running atop a modified version of Linux as the host OS, to show that Lacuna met its privacy and usability goals. In one experiment, Dunn injected random tokens into peripheral I/O paths and scanned memory to see whether these tokens could be located in various applications and in the OS in order to gauge what code holds on to sensitive data. Without Lacuna, these tokens are almost always found; with Lacuna, the tokens are never found. While the latency incurred when switching between private and non-private applications is low, Lacuna incurs higher overhead for performing USB I/O operations due to interactions between the guest and host OSes; Lacuna reduces this overhead to some degree by eliminating extra disconnections that occur when the guest OS performs USB initialization. Finally, Dunn showed that while Lacuna incurs higher CPU utilization, applications experience minimal slowdown, partly thanks to the availability of hardware AES support.

Fitz Nolan wondered whether external parties could potentially force the user to decrypt sensitive data and whether usage of Lacuna would imply guilt. Dunn said that Lacuna prevents users from incriminating themselves since all traces of private applications are removed (at least cryptographically), and it would be up to the courts whether users could get in trouble for not being able to decrypt their data. Dunn also disagreed that users would only use Lacuna if they had something to hide. Someone asked how unencrypted data in the device buffers were handled. Dunn replied that Lacuna uses public APIs to clear out as much data as possible, but it is possible that device buffers keep some data around. Peter Desnoyers (Northeastern) asked whether the graphics benchmark unfavorably favors Lacuna because Lacuna's implementation could potentially avoid some steps that would otherwise have to be taken. Dunn said they did not exploit this shortcut in their evaluation. Finally, someone from Microsoft asked how difficult it was to modify device drivers for Lacuna and about the complexity of supporting 3D APIs. Dunn responded the difficulty varies per subsystem and that often one can capture a lot of devices with a single modification; at the moment, Lacuna does not support 3D acceleration.

CleanOS: Limiting Mobile Data Exposure with Idle Eviction

Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda, Columbia University

Yang Tang began by saying that CleanOS provides new OS abstractions for protecting sensitive data on mobile devices. The mobile nature of these devices results in their being easily stolen, seized, or lost. Because the OSes running on these devices are not designed to protect sensitive data, mobile devices accumulate large amounts of sensitive information that can be accessed by anyone who has physical access to the device. Tang showed that 13 out of the 14 Gingerbread apps his research group studied kept sensitive data in cleartext either in memory or persistent storage. Moreover, Tang cited that many users do not lock their devices or use poor passwords.

Tang proposed CleanOS, a mobile Android-based OS that rigorously protects sensitive data in anticipation of device theft/loss and allows the user to know exactly what data is exposed. CleanOS leverages three critical insights: (1) sensitive data is often rarely used (e.g., an email password is only needed when sending or fetching new messages); (2) mobile apps often already contain cloud components that store data; and (3) mobile devices are almost always connected via WiFi and cellular connections. In CleanOS, sensitive data is encrypted and stored in a sensitive data object, or SDO. When access to a SDO is needed, CleanOS contacts a trusted cloud component to retrieve the decryption key needed to access the SDO. CleanOS uses taint-tracking on the local device to track accesses to sensitive data located in RAM and stable storage. When an SDO has not been accessed in a while, CleanOS will automatically cryptographically evict the SDO by securely deleting the decryption key off the local device. By minimizing the amount of sensitive data on a user's local device and shifting the protection of sensitive data to the cloud, CleanOS offers the ability to audit or limit the amount of exposure or access to said data; access to sensitive data can be completely revoked if the user's device is stolen.

CleanOS is implemented as a modified version of Android/TaintDroid that uses a CleanOS cloud service on Google App Engine. Tang described how an email app that his research group implemented within CleanOS reduced exposure of sensitive data by roughly 90% without modification (CleanOS automatically puts SSL-related state, passwords, and user input into SDOs). Modifying the app to use SDOs reduced content exposure to 0.3% that of the unmodified app. Moreover, Tang showed that auditing is very precise when the app is modified to support SDOs and can still be precise with specific types of data (e.g., passwords) even when the app is not. Finally, Tang showed that the overheads of CleanOS are largely unnoticeable over WiFi. On 3G, the overheads are more significant but can be made reasonable through a series of optimizations that Tang proposed, including batching evictions and retrievals of decryption keys.

Mark Silverstein (UT Austin) asked what the power consumption overhead of using CleanOS was. Tang responded by stating that while CleanOS adds some overhead (less than 9% overall), this overhead is largely dwarfed by the power consumed by the screen. Jason Flinn (Michigan) asked about the fundamental tradeoff between the performance benefits associated with caching and the security and granularity of caching. Tang replied that this is a policy decision that the user can configure. Stefan Bucur (EPFL) asked whether eviction continues when the device is taken offline. Tang responded that in the case of short-term disconnections, CleanOS can delay the eviction of SDOs by a bounded amount of time; for long-term disconnections, CleanOS can be configured to hoard keys before being disconnected. Finally, Bryan Ford (Yale) asked whether taint explosion is a problem. Tang said that in his experience it was not, and that running their email app for 24 hours resulted in only about 1.8% objects being tainted.

Mobility

Summarized by William Jannen (wjannen@cs.stonybrook.edu)

COMET: Code Offload by Migrating Execution Transparently

Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, and Z. Morley Mao, University of Michigan; Xu Chen, AT&T Labs—Research

Mark Gordon began with a discussion of offloading. He observed that mobile device resources are limited in terms of computation, energy, and memory; yet mobile devices are often well connected to the network. COMET explores the question of how to add network resources to mobile device computations transparently. Previous projects have explored the capture-and-migrate paradigm, such as CloneCloud and MAUI. COMET distinguishes itself from these approaches by enhancing support for multithreaded environments and synchronization primitives. Mark also noted that COMET's fine granularity of offloading efficiently handles functions with work loops, since resources can be offloaded in the middle of a method. At a high level, COMET merges the motivation of offloading—to bridge the computation disparity among nodes in a network—with the mechanism of distributed shared memory, which provides a logically shared address space.

Mark explained that distributed shared memory (DSM) is traditionally applied in cluster environments, which have low latency and high throughput. However, COMET relies on wireless communication between two endpoints. COMET implements a simple field-based DSM scheme, in which dirty fields are tracked locally, and only dirty fields are transmitted. COMET DSM leverages the Java memory model, which is field based and which specifies a happens-before partial ordering among all memory accesses.

VM synchronization is used to establish the happens-before relationship between two endpoints. VM synchronization is a directed operation between a pusher and a puller, and is responsible for synchronizing thebytecode sources, Java thread stacks, and Java heap. Thus, thread migration is implemented as a push VM synchronization operation. Mark noted that VM synchronization is designed to be recovery safe; the client is always left with enough state to resume operation in the event that the server thread is lost.

Mark was asked about the user input component of most Android apps. He replied that certain application types will not benefit from offloading, including applications with a lot of user interactions. However, there are applications, such as turn-based games, and applications with some type of kernel computation, that would benefit from using COMET. COMET may also open up new types of applications. Mark was also asked to restate the differences between COMET and CloneCloud. Mark noted that COMET provides complete support for offloading multiple threads—threads never have to block to wait for remote state. He also noted that COMET can offload within a method. Arjun Roy (UCSD) asked about I/O requests, and Mark responded that I/O requests translate to native functions. They did not have time in this paper to try to virtualize the FS like CloneCloud.

AppInsight: Mobile App Performance Monitoring in the Wild

Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh, Microsoft Research

Lenin Ravindranath exclaimed that developers are interested in two things: where user-perceived delays crop up, and, when they do, what is the bottleneck? To answer these questions, developers must manually instrument apps, which poses a significant barrier for the average developer. AppInsight significantly reduces the barrier for monitoring performance in the hands of users. It is completely automatic, requiring no effort from developers; AppInsight does not require source code, runtime, or OS modifications. A developer simply writes an app, AppInsight performs binary instrumentation, and the developer submits the instrumented app to the app store.

Lenin explained that a fundamental problem for automatic app instrumentation is that modern apps are highly interactive, UI-centric programs, which are written using very asynchronous programming patterns. With synchronous code, the user-perceived delay can be calculated by observing the beginning and end of functions. With asynchronous code, background threads process individual tasks. The user-perceived delay includes the time for the entire execution, and to measure this, the monitor must track time across thread boundaries. However, AppInsight does not modify the runtime, so it has no context for executing threads. It must have a way to know which asynchronous call is responsible for invoking each thread. Lenin defined a user transaction as beginning with a UI manipulation, and ending with the completion of all synchronous and asynchronous threads triggered by that manipulation. The critical path is the bottleneck path through a user transaction, where speeding up the path will reduce the user-perceived delay. AppInsight automatically instruments apps to track user transactions and the critical path.

In additional to performing critical path analysis for each transaction, AppInsight provides aggregate analysis. Aggregate analysis can give developers additional insight into what factors cause delay. AppInsight can group transactions and use statistical analysis to identify the root causes of variability, identify group outliers, and highlight common critical paths. AppInsight can also be used to understand app failures in the wild—since entire transaction graphs are tracked, developers can walk backwards through the path to figure out which user event triggered the exception. All the analysis is made available to developers through a Web-based tool.

Frank Lih asked if AppInsight can be used to identify performance problems caused by other applications, perhaps due to competition for resources. Mark replied that AppInsight can be imagined as the first step in finding a performance problem. He was next asked about their evaluation, and how experiences might change when applications with thousands of users are monitored. Lenin responded that more interesting problems will be identified as more diverse device configurations and environmental conditions are encountered.

OSDI 2012 Poster Session 1

Summarized by Peter Gilbert (petergilbert@gmail.com)

Be Conservative: Enhancing Failure Diagnosis with Proactive Logging

Ding Yuan, University of Illinois at Urbana-Champaign and University of California, San Diego; Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, Stefan Savage, University of California, San Diego

Diagnosing production failures is difficult because the execution environment and user inputs often cannot be reproduced exactly. Log messages are often the best available resource, reducing diagnosis times by 1.4x–3.0x for failures in software such as Apache, PostgreSQL, and Squid. However, the authors found that log messages were printed in only 43% of failures in a survey of real-world errors. Further examination revealed that 77% of failures had an explicit and generic error condition. They present ErrLog, an automated tool that analyzes and instruments source code to insert logging for error conditions. They are able to reduce the diagnosis time by 60% for real-world failures while adding a modest 1.8% runtime overhead.

ϖBox: A Platform for Privacy-Preserving Apps

Sangmin Lee, Edmund L. Wong, Deepak Goel, Mike Dahlin, and Vitaly Shmatikov, The University of Texas at Austin

There is growing concern about smartphone apps mishandling users' privacy-sensitive data. Instead of relying on untrustworthy apps to properly handle personal data, the authors propose shifting the responsibility to a platform that isolates apps from user data. ϖBox provides a sandbox that spans the device and the cloud and controls storage and communication of sensitive data. Privacy policies are configured based on an app's functionality: for example, an app that uses location information for localization is prevented from releasing that data, while usage statistics and advertising data are allowed to be released only through an aggregate channel that respects differential privacy.

Diagnosis-Friendly Cloud Management Stack

Xiaoen Ju and Kang G. Shin, University of Michigan; Livio Soares, Kyung Dong Ryu, and Dilma Da Silva, IBM T.J. Watson Research Center

The authors argue that it is important for a cloud management layer to be both easy to use and reliable, easy to diagnose and debug. To address this need, they propose logging message flows and building diagnostic tools to analyze this information. Examples of proposed tools include a tool for detecting anomalous message flows, a testing tool that can inject faults to explore recovery logic, and a replay tool to run tests offline.

Processing Widely-Distributed Data with JetStream

Matvey Arye, Ariel Rabkin, Siddhartha Sen, Michael J. Freedman, and Vivek Pai, Princeton University

This work aims to enable queries over data sets that are distributed over wide areas, such as smart grid monitoring data or traffic statistics from Web services. Goals include moving computation to data when possible, adapting to network variation, using approximations when bandwidth limitations prohibit exact answers, and allowing users to configure algorithms. To support these features, they present JetStream, which combines data cubes from online analytical processing (OLAP) with techniques from streaming databases. Advantages of their approach include efficient per-cube durability, easy specification of approximations through dimension hierarchies, explicit data movement via streams, and compatibility with open-ended data like logs.

C3A: Client/Server Co-Verification of Cloud Applications

Stefan Bucur, Johannes Kinder, George Candea, EPFL

Cloud applications are increasingly (1) split among multiple administrative domains and (2) heterogeneous, consisting of components built using different programming languages. These characteristics make cloud applications difficult to test and verify. To address this problem, the authors propose a technique called federated symbolic execution, in which specialized symbolic execution engines for different languages share a common symbolic data representation. Symbolic execution is driven by request specifications provided by the developer using an API in the target language. Testing runs as a cloud service, and properties defined in request specifications are verified along execution paths.

Hails: Protecting Data Privacy in Untrusted Web Applications

Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, and John C. Mitchell, Stanford University; Alejandro Russo, Chalmers University

This work addresses the problem of protecting users' private data spread across inter-connected Web applications. The authors argue that existing APIs force users to resort to coarse-grained access control and to choose between privacy and features. For example, a user must decide between granting an application access to her Facebook data, at which point she forfeits control of the data, or not using the application at all. The authors present the Hails multi-application Web platform as an alternative. Hails executes each Haskell application inside a jail and controls whether private data can be released by the application. Language-level information flow control is used to track private data as an application executes. Policies specifying how private data can be shared are stored alongside the data itself and enforced globally across all applications.

ContextJob: Runtime System for Elastic Cloud Applications

Wei-Chiu Chuang, Bo Sang, Sunghwan Yoo, Charles Killian, and Milind Kulkarni, Purdue University

A key advantage offered by the cloud for applications is elasticity, or the ability to scale dynamically with demand to take advantage of additional resources. However, the authors argue that it is difficult for programmers to reason about application semantics and ensure correctness when designing elastic applications. To alleviate this problem, they propose a programming model in which developers write seemingly inelastic code and elasticity is handled by the runtime system. The programmer works with the simple abstraction of an event queue.

What Does Distributed Computing Look Like on a Multicore Machine?

Stefan Kaestle and Timothy Roscoe, ETH Zürich

This work explores how to achieve high performance for parallel applications running on complex multicore machines. This can be difficult due to the challenges of ensuring efficient access to global state such as database tables. Both hardware characteristics and software requirements must be considered. The authors advocate an approach that (1) abstracts global state to support agile placement and access, and (2) chooses among distributed algorithms dynamically. In ongoing work, they plan to explore how to best abstract global state and to quantify the cost of automating these choices compared to hand-tuned implementations.

X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software

Mona Attariyan, University of Michigan and Google, Inc.; Michael Chow and Jason Flinn, University of Michigan

This work focuses on troubleshooting performance problems in complex production software. While profiling and logging can reveal which events occurred, determining how and why the events affected performance is often a challenging manual task. The authors present X-ray, a tool for automating this process by attributing performance to specific root causes. X-ray does so by assigning costs to operations such as system calls and applying taint tracking to connect these operations to a root cause. The time-consuming analysis is completed offline using deterministic replay, adding an online overhead of only 1–5%. In an evaluation using real performance issues in software such as Apache and PostgreSQL, X-ray correctly ranked the actual root cause first or tied for first in 16 out of 17 cases.

Toward Emulating Large-Scale Software Defined Networks (SDN)

Arjun Roy, Danny Yuxing Huang, Kenneth Yocum, and Alex Snoeren, University of California, San Diego

To facilitate developing emerging software defined network (SDN) technologies, testing platforms are needed for experimenting with large-scale SDNs. The authors propose an architecture consisting of multiple ModelNet emulator instances to increase bandwidth. Challenges include how to maximize bandwidth for each emulator host and how to account for the effects of different OpenFlow implementations from different vendors. They propose profiling real OpenFlow switches to quantify idiosyncrasies and then replicating them in emulated switches.

Rearchitecting System Software for the Cloud

Muli Ben-Yehuda and Dan Tsafrir, Technion—Israel Institute of Technology

The authors observe that traditional operating systems are poorly suited for cloud environments where users pay per-use for a number of reasons: applications are constrained by kernel abstractions and implementation choices that are hidden by design, and applications share a single I/O stack and device drivers. The authors present an alternative platform called nom that takes advantage of architectural support for virtualization to provide each application direct and secure access to its own I/O device. This enables the use of I/O stacks and device drivers optimized for specific applications. Applications can also change behavior to adapt to changing resource availability and pricing.

Who is Going to Program This?

Marcus Völp, Michael Roitzsch, and Hermann Härtig, Technische Universität Dresden

This work anticipates challenges programmers will face when developing for future heterogeneous manycore machines. Potential components include powerful cores "fused" from multiple smaller cores, redundant cores to handle specific hardware errors, and specialized accelerator cores. The authors argue that there is a mismatch between new properties (two-way mediation between hardware and applications, adaptive software parallelism, reconfigurable hardware, and spatial data placement) and current application characteristics (hardcoded threads, ad hoc use of accelerators, and opaque data use). They propose an "Elastic Manycore Architecture" that uses lambdas not threads for parallelism, queues for asynchronous work, runtime profiling, decentralized scheduling decisions, and an "execution stretch" metric to quantify utilization and efficiency. Cross-layer information is shared at interfaces between the OS, runtime, and applications.

Performance Isolation and Fairness for Multi-Tenant Cloud Storage

David Shue and Michael J. Freedman, Princeton University; Anees Shaikh, IBM T.J. Watson Research Center

The authors argue that existing cloud storage services, while implementing pay-per-use service on shared infrastructure, either offer no fairness or isolation, or assume uniform demand and are non work-conserving. They present a system for predictable shared cloud storage called Pisces. Pisces provides per-tenant weighted fair shares of system resources without sacrificing high utilization. The approach comprises four mechanisms: placement of partitions by fairness constraints, allocation of local weights by tenant demand, selection of replicas using local weights, and weighted fair queuing. Evaluation results show that Pisces achieves nearly ideal fair sharing, performance isolation, and robustness to changes in demand, while imposing an overhead of less than 3%.

Devirtualization: I/O Virtualization Based on Device Files

Ardalan Amiri Sani, Rice University; Sreekumar Nair, Nokia Research Center; Kevin A. Boos and Lin Zhong, Rice University; Quinn Jacobson, Nokia Research Center

Devirtualization is an approach for allowing the OS running in a guest virtual machine (VM) to directly use a device driver running in the host OS through a simple virtual device driver. The approach leverages the widely used device file interface, which provides a narrow and stable boundary for device drivers and already supports multiplexing among processes. Devirtualization can support many GPUs and input devices in guest VMs with minimal device-specific changes while imposing no user-perceptible latency. A virtual device file in a guest VM corresponds to the actual device file in the host, and file operations performed by the guest are forwarded to the host and performed by the actual device driver. Because guest file operations use separate guest virtual addresses, a hybrid address space is provided to bridge guest user space memory and host kernel memory.

Dune: Safe User-Level Access to Privileged CPU Features

Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis, Stanford University

The authors argue that many applications could benefit from a safe, efficient way to directly access privileged CPU features such as page tables, tagged TLBs, and ring protection. Standard mechanisms like ptrace and mprotect are too slow and clumsy, while modifying the kernel for every application is risky and does not scale. Simply running each application in its own virtual machine (VM) is dismissed due to poor integration with the host OS and unacceptable overhead. Instead, Dune leverages architectural support for hardware-assisted virtualization but exports a process abstraction rather than a VM abstraction. A minimal kernel module manages virtualization hardware and interactions with the kernel, while the libDune library helps applications manage privileged CPU features. The authors demonstrate Dune's value by implementing a privilege separation facility, a sandbox for untrusted code, and a garbage collector.

Mercurial Caches: OS Support for Energy-Proportional DRAM

Asim Kadav, Rathijit Sen, and Michael M. Swift, University of Wisconsin—Madison

While DRAM is a significant contributor to power consumption, techniques that take advantage of unused DRAM to save power are limited. The authors propose mercurial caches to provide OS abstractions for low-power DRAM. Mercurial caches occupy portions of DRAM to put them in a low-power state. Challenges include how to dynamically resize mercurial caches without affecting application performance, how to ensure that mercurial caches do not appear as missing memory, and how to utilize mercurial caches when they are not completely turned off. Energy-aware page migration is needed for mercurial caches to be effective despite fragmentation of the physical address space. A preliminary evaluation using an analytical model shows that mercurial caches can achieve energy usage proportional to DRAM usage.

Herding the Masses—Improving Data/Task Locality in Hadoop

Bingyi Cao and Daniel Abadi, Yale University

The authors demonstrate that the default scheduler for map tasks in Hadoop, which greedily selects tasks with data nearby in FIFO order, can result in poor locality. They propose an alternative two-level sort algorithm consisting of a coarse-grained sort followed by a fine-grained sort, using only information about the local node and tasks. High efficiency is possible because only a few tasks need be sorted for each node.

Optimizing Shared Resource Contention in HPC Clusters

Sergey Blagodurov and Alexandra Fedorova, Simon Fraser University

This work focuses on performance problems in HPC clusters due to contention for shared multicore resources such as caches and memory controllers. The authors argue that HPC clusters must be made contention-aware to remedy this problem. They present Clavis-HPC, a contention-aware virtualized HPC framework. Tasks are classified as devils if they are memory-intensive with a high last-level cache miss rate, or turtles otherwise. The scheduling algorithm minimizes the number of devils on each node, maximizes the number of communicating processes on each node, and minimizes the number of powered-up nodes in the cluster. The schedule is enforced using low-overhead live migration.

OSDI Poster Session 1

Summarized by Lisa Glendenning (lglenden@cs.washington.edu)

Towards a Data Analysis Recommendation System

Sara Alspaugh, University of California, Berkeley; Archana Ganapathi, Splunk, Inc.; Randy Katz, University of California, Berkeley

This project proposes building a tool for semi-automated, user-guided exploration of arbitrary data sets. The proposed approach is to build a tool on top of Splunk, a platform for indexing and searching semi-structured time series data. Existing packages built on top of Splunk provide front-end analyses such as reports and dashboards, but each package is tailored to a specific class of data. The proposed tool would create and recommend front-end analyses for arbitrary data sets and iterate based on user feedback.

Nested Virtual Machines and Proxies for Easily Implementable Rollback of Secure Communication

Kuniyasu Suzaki, Kengo Iijima, Akira Tanaka, and Yutaka Oiwa, AIST: National Institute of Advanced Industrial Science and Technology; Etsuya Shibayama, The University of Tokyo

The objective of this work is to use fuzz testing to verify implementations of protocols for secure communication; the current target is TLS/SSL. This project's approach for verification depends on a rollback capability that the authors have implemented using nested virtual machines, proxies, and a control protocol. Work in progress includes developing a protocol fuzzing generator as a client component.

MiniStack: Operating System Support for Fast User-Space Network Protocols

Michio Honda and Felipe Huici, NEC Europe Ltd.; Luigi Rizzo, Universita di Pisa

The motivation of this work is to move the network stack into user space while approaching the high performance, isolation, and security of kernel- and hardware-based networking implementations. MiniStack builds on the high performance of netmap while addressing some of netmap's limitations: applications can snoop and overwrite each other's packets; applications can spoof the packet source address; and there is no mechanism to demultiplex received packets to the appropriate application. MiniStack extends VALE, a virtual Ethernet switch implemented as a Linux and BSD kernel module.

POD: Performance-Oriented I/O Deduplication for Primary Storage Systems

Bo Mao and Hong Jiang, University of Nebraska—Lincoln; Suzhen Wu, Xiamen University

Deduplication reduces I/O redundancy in primary storage systems but can lead to data fragmentation and resource contention. POD mitigates these problems by selectively deduplicating write requests and by dynamically adjusting the memory space between the index cache and the read cache based on the I/O accesses.

A Verified Kernel and Commodity Hardware

Yanyan Shen and Kevin Elphinstone, University of New South Wales, NICTA

Formally verified microkernels such as seL4 provide a strong foundation on which to build trustworthy systems, but commodity hardware is susceptible to transient faults such as silent memory corruption and bus errors. This project proposes using techniques such as redundant execution on multiple cores to detect and recover from hardware faults in the trusted software layer.

Closing the Gap Between Driver Synthesis and Verification

Alexander Legg and Leonid Ryzhyk, NICTA; Adam Walker, NICTA and University of New South Wales

Driver synthesis automatically generates a device driver implementation from a device and OS specification. This work proposes using a code template for the OS specification rather than a state machine, which is susceptible to state explosion and is hard to maintain. A code template requires some manually written code, but these code snippets undergo verification during the synthesis process.

Collaborative Verification with Privacy Guarantees

Mingchen Zhao, University of Pennsylvania; Wenchao Zhou, Georgetown University; Alexander Gurney and Andreas Haeberlen, University of Pennsylvania; Micah Sherr, Georgetown University; Boon Thau Loo, University of Pennsylvania

In distributed systems, nodes fail in a number of ways including misbehavior—e.g., violating protocol specifications. The goal of this work is to verify whether a node is behaving as expected without revealing sensitive information about non-faulty nodes. The approach is to generalize the collaborative verification techniques that were used in the SPIDeR system to verify the interdomain routing decisions of BGP systems. Ongoing work includes automatic generation of verification protocols via a new programming language.

The Ethos Project: Security Through Simplification

W. Michael Petullo and Jon A. Solworth, University of Illinois at Chicago

Existing operating systems have very high complexity, and this complexity limits the level of assurance possible. Ethos is an experimental, clean-slate OS with security as a primary goal. Ethos is designed to support the development and deployment of secure systems for both application developers and system administrators; for instance, Ethos provides a small set of carefully chosen system calls with high-level semantics and compulsory security protections. Ethos currently supports applications written in Go.

GReplay: A Programming Model for Kernel-Space GPU Applications

Xinya Zhang, Jin Zhao, and Xin Wang, Fudan University

The goal of this project is a GPU programming model for kernel-space applications with a high-level API, high portability, and low overhead. GReplay applications are developed with OpenCL and compiled in user space, but GPUs are invoked from a kernel module. The kernel-space component implements an abstraction layer over GPU drivers. Evaluation of four applications shows high throughput compared to Mesa 3D and Catalyst, an official driver.

Malleable Flow for Time-Bounded Replica Consistency Control

Yuqing Zhu and Jianmin Wang, Tsinghua University; Philip S. Yu, University of Illinois at Chicago

Replication schemes across datacenters typically trade off consistency with availability and operation latency. A system with malleable flow (M-flow) supports latency-bounded operations that maximize replica consistency within the given time. The key idea for malleable flow is to decompose the replication process into stoppable stages and then into a directed graph of ordered steps. Given a time constraint, the system will reform an execution flow into a path through the graph that guarantees fault-tolerance and maximizes consistency. An M-flow system has been implemented over Cassandra.

Rebasable File Systems for Enhanced Virtual Machine Management

Jinglei Ren, Bo Wang, Weichao Guo, Yongwei Wu, Kang Chen, and Weimin Zheng, Tsinghua University

Fast virtual machine cloning creates a VM by linking it to a base with block-level mapping. Two drawbacks of current approaches for fast cloning for VDI and cloud environments are that (1) updates to the base cannot propagate to derived images, and (2) derived images cannot seamlessly roll back to a previous base. Cinquain is a file-based storage that addresses these problems by providing a file system view for each VM. Cinquain can rebase operations for VMs by seamlessly changing the parent of a child view.

Experiences with Hardware Prototyping Solid-State Cache

Mohit Saxena and Michael M. Swift, University of Wisconsin—Madison

High-speed solid-state drives (SSDs) composed of NAND flash are often deployed as a block cache in front of high capacity disk storage. This work prototypes FlashTier, a lightweight, consistent and durable storage cache, on the OpenSSD evaluation board. To compensate for the low baseline performance of OpenSSD, the prototype implements a number of techniques within the OpenSSD device firmware to efficiently manage large flash block sizes and increased channel and plane parallelism. These techniques include: (1) merge buffer for aligned random writes, (2) read buffer for efficient random reads, (3) perfect page striping to maximize flash bank parallelism, and (4) minimized read-modify-write cycles for partial overwrites.

User-Mode Storage Systems for Storage-Class Memory

Haris Volos, Sankaralingam Panneerselvam, and Michael M. Swift, University of Wisconsin—Madison

Storage class memory (SCM) technology is byte-addressable, non-volatile, and has a low access time. This work redesigns the storage architecture of operating systems to take advantage of this new technology. The approach is to build a memory file system with high flexibility and performance by enabling direct access of SCM from user-mode applications. The system has three main components: (1) a user-mode library file system that implements naming and mapping, (2) a trusted file system service that enforces concurrency control and maintains the integrity of metadata, and (3) an SCM manager that securely records and enforces resource usage.

Distributed Systems and Networking

Summarized by Jim Cadden (jmcadden@bu.edu)

Spotting Code Optimizations in Data-Parallel Pipelines through PeriSCOPE

Zhenyu Guo, Microsoft Research Asia; Xuepeng Fan, Microsoft Research Asia and Huazhong University of Science and Technology; Rishan Chen, Microsoft Research Asia and Peking University; Jiaxing Zhang, Hucheng Zhou, and Sean McDirmid, Microsoft Research Asia; Chang Liu, Microsoft Research Asia and Shanghai Jiao Tong University; Wei Lin and Jingren Zhou, Microsoft Bing; Lidong Zhou, Microsoft Research Asia

Zhenyu Guo presented PeriScore, a procedural optimization technique for improving performance of data-parallel computation systems. This is achieved through pipeline-aware holistic code-optimization techniques.

Zhenyu began by defining network I/O as the bottleneck in distributed parallel pipeline jobs (such as MapReduce). One way to alleviate a network bottleneck is to reduce the data shuffling between computation procedures though optimizations. Traditional compilers do not optimize the procedure code in relation to the pipelining, and this can be a painstakingly process when done manually. PeriScore allows for automatic optimization of a distributed programs procedure code in the context of data flow.

By optimizing the procedure code directly, PeriScore removes unnecessary data, relocates operations, and calculates early predicates, which results in less data being shared across the network overall. The optimization process of PeriScore is to (1) construct an inter-procedural flow graph, (2) add safety constraints for skipping or shuffling code, and (3) transform code for the reduction of shuffling I/O.

In the question and answer session, Jason Flinn asked if the optimization based on static analysis can be improved with the addition of profiling. Zhenyu agreed that profiling would be sure to improve the overall performance increase as conservative approximations are currently done in cases where data sizes are unknown (e.g., streams).

MegaPipe: A New Programming Interface for Scalable Network I/O

Sangjin Han and Scott Marshall, University of California, Berkeley; Byung-Gon Chun, Yahoo! Research; Sylvia Ratnasamy, University of California, Berkeley

Sangjin Han presented MegaPipe, a new programming interface for scalable network I/O, designed to replace the standard BSD socket I/O. Sangjin began with the observation that message-oriented I/O (HTTP, RPC, key-value stores) with the BSD socket API can be very CPU intensive; small message sizes and short duration of connections lead to undesired performance, and adding additional cores does not alleviate the problem.

MegaPipe works through the combination of three system optimizations: (1) I/O batch processing assisted by a kernel library, (2) a per-core channel abstraction for a listening socket that allows for per-channel accept queues (avoiding contention), and (3) lightweight sockets that skip the file abstraction layers. MegaPipe assumes that file abstractions are no longer appropriate for sockets since sockets are short-lived and rarely shared.

In the evaluation section, MegaPipe was shown to improve throughput by up to 100% on an eight-core machine for messages of one kilobyte and smaller (smaller improvements were short for larger packet sizes). In addition, MegaPipe provided a near 15% throughput improvement for memcached and 75% throughput improvement for nginx for short connections. MegaPipe enjoys near linear scalability with evaluation run on up to 128 cores.

Mendel Rosenblum (Stanford) pointed out that people had just given up on using sockets and asked if their lightweight sockets are good enough to quit using other solutions. Sanjin answered that in most cases you don't need the full generality of sockets. When you really need the generality of sockets, you want to convert to our sockets. A researcher from MSR asked about the delay costs of the I/O batching. Sangjin responded that the batch size is small enough (~32 operations) so that it does not affect latency much. Ali Mashtizadeh (Stanford) asked if their research takes system scheduling into account. Sangjin explained that a basic assumption with MegaPipe is that there is one thread per core and that the MegaPipe process is scheduled like any other user-level application.

DJoin: Differentially Private Join Queries over Distributed Databases

Arjun Narayan and Andreas Haeberlen, University of Pennsylvania

Arjun Narayan presented DJoin, a technique for processing differentially private queries (including, but not limited to, 'JOIN') across distributed databases. Differential privacy is a process to control the amount of sensitive information that a database can release about its protected data set. Differential privacy software exists for central databases, as does non-private distributed join queries. DJoin applies both these techniques to allow for differentially private joins across distributed databases.

Arjun began with a motivating example of a scientist researching a recent outbreak of malaria in Albania. Ideally, this scientist would want to directly compare the records of who recently contracted malaria with those who had traveled to Albania (data that exists across databases of airlines and hospitals). However, free and open access to this information would be a giant violation of individual privacy and, in many cases, against the law.

Differential privacy works by factoring noise into the result of the queries made on a database. Every query on the database has a privacy cost attached, and the amount of noise added depends on the balance "spent" on that particular query. Once a user has spent their allotted resource they can no longer query the database. DJoin introduced two novel primitives: BN-PSI-CA, a differentially private form of private set intersection cardinality, and DCR, a multi-party combination operator that can aggregate noised cardinalities without compounding the individual noise terms.

In the evaluation section, DJoin was shown to incur non-trial computation costs, e.g., over an hour of computational overhead per query for databases greater than 30,000 rows. However, this work was described as "embarrassingly" scalable, shown by a nearly 4x increase through parallelizing across four cores. Arjun concluded by defining DJoin to be not fast enough for interactive use and more appropriate for offline analysis.

Being that this is a system's venue, it was no surprise that most questions involved curiosity surrounding the workings of differential privacy. Anunjun (Yale) asked if it is possible to support differentially private sum operations. Arjun replied that, yes, an extension of the existing count mechanism would be trivial. Henry Gibbs (Yale) inquired about a possible timing attack involving the size of the database and the time of the computation. Arjun explained that the set intersection is padded to the size of the entire database and noise is added to the polynomial size. Mike Freedman (Princeton) asked about the theoretical limitation of differential privacy, which requires a database to be "retired" after a certain amount of queries/cost have been processed. Arjun acknowledged that this characteristic of differential privacy was unfortunate and suggested that the lifespan of a sensitive database can be extended by carefully vetting access and queries.

Security

Summarized by Amit A. Levy (amit@amitlevy.com)

Improving Integer Security for Systems with KINT

Xi Wang and Haogang Chen, MIT CSAIL; Zhihao Jia, Tsinghua University IIIS; Nickolai Zeldovich and M. Frans Kaashoek, MIT CSAIL

Xi Wang from MIT presented several vulnerabilities resulting from integer overflow bugs. For example, such bugs allow an attacker to mount buffer overflow attacks or a malicious process to over-consume resources. Wang and his collaborators developed a static analysis tool, KINT, for identifying such bugs. They used KINT to find 114 bugs in the Linux kernel. They also propose two extensions to the C language and standard library that help mitigate integer overflow bugs.

Their case study yielded several interesting observations. First, of the 114 Linux kernel bugs they found, 79% caused buffer overflows or logical bugs. Second, two-thirds of the bugs had existing checks in the code, but the checks were wrong! Wang argued that this is evidence of how hard it is to reason about integer overflow bugs manually.

KINT is a tool for detecting integer overflows by statically analyzing LLVM IR (intermediate representation). KINT runs three separate passes on the program, one analyzing each function independently, another that performs whole-program analysis to reduce false positives from the first pass, and a third pass employing taint analysis using user annotations. Finally, KINT combines the results of all three passes and generates a bug report.

Wang discussed two mechanisms for mitigating the integer overflow bugs KINT finds. The authors added kmalloc_array, a new library call to the Linux kernel as of version 3.4. kmalloc_array implements the integer overflow check rather than relying on the caller to do so. Wang argued that while this fix targets a very specific bug, the bug is so common and potentially harmful that this simple library extension would have a large effect. The authors also propose adding the NaN value to the C language, which holds a special value for the results of computations that overflow an integer value. They implemented NaN in the clang C compiler.

Cristian Zamfir asked how Wang determined that the 125,172 possible bugs in KINT found in the Linux kernel were false positives. Wang responded that actually he's not sure whether they are false positives or actual bugs. Rather, the remaining 741 were bugs he was able to manually verify. Luis Pedrosa from USC asked how KINT deals with cases that are too hard for the solver to solve. Wang responded that they mark them explicitly as unsolved in the final report.

Dissent in Numbers: Making Strong Anonymity Scale

David Isaac Wolinsky, Henry Corrigan-Gibbs, and Bryan Ford, Yale University; Aaron Johnson, US Naval Research Laboratory

"Anonymity enables communication of sensitive ideas without fear of reprisal from government, organizations or peers." While, traditionally, anonymous systems must trade strong anonymity for scale, and users end up choosing weaker systems with more users, Daniel Wolinsky presented the argument that anonymous systems actually depend on large numbers of users—that weak anonymous systems with many numbers are often stronger than strong anonymous systems with few users. He introduced Dissent, a system that is able to provide strong anonymity while scaling to 1000s of active participants.

The key insight in Dissent is to combine the peer-to-peer architecture of DC-nets and Mix-nets to achieve strong anonymity and the client-server model of Tor to achieve scalability. For example, while each peer in DC-nets requires O(N^2) (N is the total number of participants) random number generations while in Dissent, clients need a random number generation per server, and each server needs one for each client—i.e., O(M*N) random number generations for the whole system. Similarly, communication cost in DC-nets is O(N^2) ciphertext transmissions, while Dissent uses multicast trees to achieve linear communication costs. Finally, Wolinsky presented an evaluation of Dissent that shows it can scale to thousands of participants.

Finally, Wolinsky presented an evaluation of Dissent that shows it can scale up to 1000s of participants. Mike Walfish from UT-Austin asked whether 2000 people are "really too many to throw in jail" since with Dissent it is very clear when someone is a member. He followed up, asking whether there was a way to hide membership. Wolinsky replied that the Tor project has made some progress in that regard, but that in the end he believes this comes down to an arms race with the adversary. Saurabh Bagchi (Purdue) asked whether there was a tradeoff between privacy and the number of honest servers that can be deployed. Wolinsky responded that the security of Dissent relies on the existence of at least one honest server as well as a number of honest clients.

Efficient Patch-Based Auditing for Web Application Vulnerabilities

Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich, MIT CSAIL

Buggy Web applications can lead to security vulnerabilities that may only be discovered long after the applications have been introduced. If a vulnerability has been around for months or years, has it been exploited? By whom? Taesoo Kim presented a technique for using security patches to identify past attacks.

The key insight is that since security patches render previous attacks harmless, comparing the execution of each past request with the patch and without it will expose differences. For example, the same request may result in different SQL queries to the database with and without the patch. If the request in fact behaves differently, that request may be part of an attack. However, a naive implementation would execute each request twice—so auditing one month of traffic would require two months.

Kim identifies three opportunities he and his colleagues used to improve performance in their auditing system. First, a patch may not affect all requests, so not all requests need to be re-executed. They used control-flow filtering to determine which requests could not be affected by the patch. Second, much of the pre-patch and post-patch execution runs are identical, so there is no need to run both, identical, versions. Finally, multiple requests may execute similar code, so further redundancies can be eliminated. The authors memoized requests throughout the re-execution to avoid re-running similar requests.

Using all of these techniques, the authors were able to audit requests 12–51 times faster than the original execution. They evaluated the effectiveness of their system on patched vulnerabilities from MediaWiki (using Wikipedia traces) and HotCRP (using synthetic workloads). Finally, they found that the overhead of their system during normal execution was about 14% higher latency and 15% lower throughput.

Matvey Arye (Princeton) asked about the overhead of storing all of the information along with requests. Kim replied that in their Wikipedia traces the overhead averaged 5.4 KB per request. Jonas Wagner (EPFL) asked how they can re-execute requests without access to state that might only be available, such as specific data, in the production database. Kim clarified that instead of making actual database calls, their system records responses from external sources like the database and replays those responses during re-execution.

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

SAVE THE DATE!

APRIL 3–5, 2013 • LOMBARD, IL

nsdi '13

10th USENIX Symposium on Networked Systems Design and Implementation

NSDI '13 focuses on the design principles, implementation, and practical evaluation of large-scale networked and distributed systems. Systems as diverse as data centers, Internet routing, peer-to-peer and overlay networks, storage clusters, sensor networks, wireless and mobile systems, Web-based systems, and measurement infrastructures share a set of common challenges. NSDI '13 will bring together researchers from across the networking and systems community to foster a broad approach to addressing our common research challenges.

Full program information and registration details are available on the conference Web site:
www.usenix.org/nsdi13

Sponsored by USENIX in cooperation with ACM SIGCOMM and ACM SIGOPS