;login:

JUNE 2014 VOL. 39, NO. 3

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

File Systems and Storage

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

2014 USENIX Federated Conferences Week
June 17–20, 2014, Philadelphia, PA, USA

USENIX ATC '14: 2014 USENIX Annual Technical Conference
June 19–20, 2014
www.usenix.org/atc14

ICAC '14: 11th International Conference on Autonomic Computing
June 18–20, 2014
www.usenix.org/icac14

HotCloud '14: 6th USENIX Workshop on Hot Topics in Cloud Computing
June 17–18, 2014
www.usenix.org/hotcloud14

HotStorage '14: 6th USENIX Workshop on Hot Topics in Storage and File Systems
June 17–18, 2014
www.usenix.org/hotstorage14

9th International Workshop on Feedback Computing
June 17, 2014
www.usenix.org/feedbackcomputing14

WiAC '14: 2014 USENIX Women in Advanced Computing Summit
June 18, 2014
www.usenix.org/wiac14

UCMS '14: 2014 USENIX Configuration Management Summit
June 19, 2014
www.usenix.org/ucms14

URES '14: 2014 USENIX Release Engineering Summit
June 20, 2014
www.usenix.org/ures14

23rd USENIX Security Symposium
August 20–22, 2014, San Diego, CA, USA
www.usenix.org/sec14

Workshops Co-located with USENIX Security '14

EVT/WOTE '14: 2014 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections
August 18–19, 2014
www.usenix.org/evtwote14

USENIX Journal of Election Technology and Systems (JETS)
Published in conjunction with EVT/WOTE
www.usenix.org/jets

CSET '14: 7th Workshop on Cyber Security Experimentation and Test
August 18, 2014
www.usenix.org/cset14

3GSE '14: 2014 USENIX Summit on Gaming, Games, and Gamification in Security Education
August 18, 2014
www.usenix.org/3gse14

FOCI '14: 4th USENIX Workshop on Free and Open Communications on the Internet
August 18, 2014
www.usenix.org/foci14

HotSec '14: 2014 USENIX Summit on Hot Topics in Security
August 19, 2014
www.usenix.org/hotsec14

HealthTech '14: 2014 USENIX Summit on Health Information Technologies
Safety, Security, Privacy, and Interoperability of Health Information Technologies
August 19, 2014
www.usenix.org/healthtech14

WOOT '14: 8th USENIX Workshop on Offensive Technologies
August 19, 2014
www.usenix.org/woot14

OSDI '14: 11th USENIX Symposium on Operating Systems Design and Implementation
October 6–8, 2014, Broomfield, CO, USA
www.usenix.org/osdi14

Co-located with OSDI '14 and taking place October 5, 2014

Diversity '14: 2014 Workshop on Supporting Diversity in Systems Research
www.usenix.org/diversity14

HotDep '14: 10th Workshop on Hot Topics in Dependable Systems
www.usenix.org/hotdep14
Submissions due: July 10, 2014

HotPower '14: 6th Workshop on Power-Aware Computing and Systems
www.usenix.org/hotpower14

INFLOW '14: 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads
www.usenix.org/inflow14
Submissions due: July 1, 2014

TRIOS '14: 2014 Conference on Timely Results in Operating Systems
www.usenix.org/trios14

LISA '14
November 9–14, 2014, Seattle, WA, USA
www.usenix.org/lisa14

Co-located with LISA '14:

SESA '14: 2014 USENIX Summit for Educators in System Administration
November 11, 2014

Stay Connected...

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

;login:

JUNE 2014 VOL. 39, NO. 3

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

EDITOR
Rik Farrow
rik@usenix.org

COPY EDITORS
Steve Gilmartin
Amber Ankerholz

PRODUCTION MANAGER
Michele Nelson

PRODUCTION
Arnold Gatilao
Casey Henderson

TYPESETTER
Star Type
startype@comcast.net

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

www.usenix.org

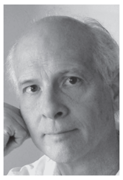
;login: is the official magazine of the USENIX Association. *;login:* (ISSN 1044-6397) is published bi-monthly by the USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

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

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

Musings

RIK FARROW



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

After writing this column for more than 15 years, I'm a bit stuck with what I should talk about. But I do have a couple of things on my mind: Krste Asanović's FAST '14 keynote [1] and something Brendan Gregg wrote in his *Systems Performance* book [2].

Several years ago, I compared computer systems architecture to an assembly line in a factory [3], where not having a part ready (some data) held up the entire assembly line. Brendan expressed this differently, in Table 2.2 of his book, where he compares the speed of a 3.3 GHz processor to other system components by using human timescales. If a single CPU cycle is represented by one second, instead of .3 nanoseconds, then fetching data from Level 1 cache takes three seconds, from Level 3 cache takes 43 seconds, and having to go to DRAM takes six minutes. Having to wait for a disk read takes months, and a fetch from a remote datacenter can take years. It's amazing that anything gets done—but then you remember the scaling of 3.3 billion to one. Events occurring in less than a few tens of milliseconds apart appear simultaneous to us humans.

Krste Asanović spoke about the ASPIRE Lab, where they are examining the shift from the performance increases we've seen in silicon to a post-scaling world, so we need to consider the entire hardware and software stack. You can read the summary of his talk in this issue of *;login:* or go online and watch the video of his presentation.

A lot of what Asanović talked about seemed familiar to me, because I had heard some of these ideas from the UCB Par Lab, in a paper in 2009 [4]. Some things were new, such as using photonic switching and message passing (read David Blank-Edelman's column) instead of the typical CPU bus interconnects. Asanović also suggested that data be encrypted until it reached the core where it would be processed, an idea I had after hearing that Par Lab paper, and one that I'm glad someone else will actually do something about. Still other things remained the same, such as having many homogeneous cores for the bulk of data processing, with a handful of specialized cores for things like vector processing. Asanović, in the question and answer that followed his speech, said that only .01% of computing requires specialized software and hardware.

Having properly anticipated the need for encryption of data while at rest and even while being transferred by the photonic message passing system, I thought I'd take a shot at imagining the rest of Warehouse Scale Computing (WSC), but without borrowing from the ASPIRE FireBox design too much.

The Need for Speed

First off, you need to keep those swift cores happy, which means that data must always be ready nearby. That's a tall order, and one that hardware designers have been aware of for many years. Photonic switching at one terabit per second certainly sounds nice, and it's hard to imagine something that beats a design that already seems like science fiction based on the name. For my design, I will simply specify a message-passing network that connects all cores and their local caches to the wider world beyond. Like the FireBox design, there will be no Level 1 cache coherency or shared L1 caches. If a module running on one core wants to share data with another, it will have to be done via message passing, not by tweaking shared bits and expecting the other core to notice.

Whatever interconnect you choose, having sufficient bisection bandwidth will be key to the performance of the entire system. Can't have those cores waiting many CPU cycles for the data they need to read or write. The photonic switches have lots of bandwidth, and if they can actually switch without resorting to converting light back to electrons then to light again, they can work with a minimal of latency.

In my design, I imagine having special encryption hardware that's part of every core. That way, checking the HMAC and the decryption of messages (as well as the working in the opposite direction) can take advantage of hardware built for this very specific task. It should be possible to design very secure systems using this approach, because all communications between processes and the outside world come with verification of their source—a process that knows the correct encryption key. Like the FireBox, this system will be a service-oriented architecture, with each core providing a minimal service, again minimizing but not eliminating the probability that there will be security-affecting bugs in the code.

Some cores will be connected to the outside world, managing communications and storage. This is not that different from current approaches, where network cards for VM hosts already do a lot of coprocessing and maintain multiple queues. But something similar will need to be done for storage, as it will remain glacially slow, from the CPU's perspective, even with advances such as non-volatile memory (NVM) being available in copious amounts with speeds as fast or faster than current DRAM.

Cores will be RISC, because they are more efficient than CISC designs. (Note to self: sell Intel, buy ARM Holdings.) With Intel server CPUs like eight-core Xeon ES having 2.7 billion transistors, that's a lot of heat, much of which is used to translate CISC instructions into internal, RISC-like, microcode instructions. The AMD A1100 that was announced in January 2014 will have eight 64-bit ARM (Cortex-A57) cores and built-in SIMD, which supports AES encryption and is rated at 25 watts TDP (thermal design power), compared to 80 for the 3.2 GHz Xeon. (Hmm, buy AMD?)

Unlike the FireBox's fit-in-a-standard-rack design, my imaginary system will look like something designed by Seymour Cray, but without a couch. Cray's best-known designs were circular, because Cray was concerned about having components separated by too much distance. After all, light can only travel 29.979 cm in a nanosecond, and with CPU clock cycles measured in nanoseconds in Cray's day, distance mattered. Actually, distance matters even more today.

My design has the outward appearance of a cube. Inside, the cores will be arranged in a sphere, with I/O and support filling in the corners. Also, unlike one of Cray's designs, where you could see the refrigerant flowing around the parts, my cube will float on a fountain of water. The water will both cool and suspend the cube, while the I/O and power connections will prevent it from floating off the column of water.

At the end of Asanović's talk, I asked him why they would be using Linux. Asanović replied that Linux provides programmers with a familiar interface. That's certainly true, and I agree. But I also think that a minimal Linux shell, like that provided by a picoprocess [5], will satisfy most programs compiled to work with Linux, while being easy to support with a very simple message passing system under the hood (so to speak).

Except for some dramatic flairs, I must confess that my design is not that different from the FireBox.

Reality

One problem with my floating cube design is how to deal with broken hardware. Sometimes cores or supporting subsystems fail, and having to toss an entire cube because you can't replace failed parts isn't going to work. There's a very good reason why supercomputers today appear as long rows of rackmount servers [6]. One can hope that the reason the FireBox will fit in racks is that it contains modules that can be easily serviced and replaced.

Using water for cooling has been done before [7], but it does make maintenance more difficult than just using air. Still, even low-power RISC cores dissipate "waste" heat, and having 1000 of them translates into 3.3 kilowatts of heat—a space heater that you really don't want in your machine room. Still, that beats the 12.5 kilowatts produced by the Xeons.

Even the photonic switching network could prove problematic. In the noughts, a company named SiCortex [8] built supercomputers that used RISC cores and featured a high-speed, message-passing interconnect using a diameter-6 Kautz graph, and they failed after seven years and only selling 75 supercomputers. Perhaps the market just didn't think that having an interconnect designed to speed intercore and I/O communications was important enough.

The Lineup

We start this issue with an article from the people who built ~okeanos, a public cloud for Greek researchers. They have written for *;login:* before [9], and when I discovered that they had used Ceph as the back-end store, I asked if they would write about their experiences with Ceph. The authors describe what they needed from a back-end storage system, what they tried, and how Ceph has worked so far.

Musings

I met Tyler Harter during a poster session at FAST '14. Tyler had presented what I thought was a great paper based on traces collected from Facebook Messages. What he and his co-authors had discovered showed just how strongly layering affects write performance, resulting in a huge amount of write amplification. In this article, Harter et al. explain how they uncovered the write amplification and what can be done about it, as well as exploring whether the use of SSDs would improve the performance of this HBase over HDFS application.

I knew that Mark Lamourine had been working on Red Hat's implementation of OpenStack and thought that I might be able to convince him to explain OpenStack. OpenStack started as a NASA and Rackspace project for creating clouds and has become a relatively mature open source project. OpenStack has lots of moving parts, which makes it appear complicated, but I do know that people are using it already in production. The ~okeanos project is OpenStack compatible, and if you read both articles, you can learn more about the types of storage required for a cloud.

Tim Hockin shares an epic about debugging. What initially appeared to be a simple problem took Hockin down many false paths before he finally, after going all the way down the stack to the kernel, found the culprit—a tiny but critical change in source code.

David Lang continues to share his expertise in enterprise logging. In this issue, Lang explains how to detect and fix performance problems when using rsyslog, a system he has used and helps to maintain.

Andy Seely introduces us to some rock stars. You know, those people you worked with at the startup that didn't make it? The ones willing to work long hours for a reward that remained elusive? Seely's story is actually about how a small management change improved the lives of the people he worked with.

David Blank-Edelman delves into the world of message queues via 0MQ. I became interested in message queues when I learned, from Mark Lamourine, that they were being used in OpenStack. David shows us how simple it is to use 0MQ, as well as demonstrating just how powerful 0MQ is, using some Perl examples, in the first of a two-part series.

Dave Beazley explores a feature found in the newest version of Python, asynchronous I/O. You might think that async-io has been around for a while in Python, and you'd be right. But this is a new implementation, which considerably simplifies how event loops are used. Oh, and there's a backport of the new module to Python 2.7.

Dave Josephsen has us considering monitoring design patterns. I found his column very timely, as I know that sysadmins are questioning the design patterns they have been using for many years to collect status and information.

Dan Geer and Jay Jacobs discuss where we are today in collecting security metrics. At first, we just needed to start collecting usable data. Today, what's needed is the ability to better perform meta-analysis of publicly available data.

Robert Ferrell also explores clouds and muses about the future of advertising. Like Robert, I just can't wait until my heads-up display is showing me advertising when what I really want are the directions to where I needed to be five minutes ago.

James Mickens had written a number of columns that were only published online, and we decided to print his first one [10]. James has been experiencing deep, existential angst about issues surrounding the unreliability of untrusted computer systems. In particular, papers about Byzantine Fault Tolerance. Don't worry, as James' column will not put you to sleep.

Elizabeth Zwicky has written three book reviews. She begins with a thorough and readable tutorial on R, covers an excellent book on threat modeling, and finishes with a book on storage for photographers.

This will be Elizabeth's final column. Elizabeth has been the book reviews columnist for *;login:* since October 2005, and I, like many, have thoroughly enjoyed her erudite and droll reviews. Her work will be missed, although I hope she still finds the time to pen the occasional review.

If you want to contribute reviews to *;login:* of relevant books that you have read, please email me.

We also have summaries of FAST '14 and the Linux FAST Summit. I took notes there and converted them into a dialog that covers a lot of what happened during the summit. The summary also provides insights into how the Linux kernel changes over time.

Although the key ideas of the FireBox design appear sound to me, people have learned how to work with off-the-shelf rackmounted servers for massive data processing tasks, and the biggest change so far has been to move toward using SDN for networking. Perhaps there will be a move toward customized cores as well. When designing WSC, the ability to keep data close to where it is processed has been the key so far, whether we are discussing MapReduce or memcached.

Resources

[1] "FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers": https://www.usenix.org/conference/fast14/technical-sessions/presentation/keynote.

[2] Brendan Gregg, *Systems Performance for Enterprise and Cloud* (Prentice Hall, 2013), ISBN 978-0-13-339009-4.

[3] Rik Farrow, "Musings," *;login:*, vol. 36, no. 3, June 2011: https://www.usenix.org/login/june-2011/musings.

[4] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz, "Space-Time Partitioning in a Manycore Client OS," Hot Topics in Parallelism, 2009: https://www.usenix.org/legacy/event/hotpar09/tech/full_papers/liu/liu_html/.

[5] Jon Howell, Bryan Parno, and John R. Douceur, "How to Run POSIX Apps in a Minimal Picoprocess": http://research.microsoft.com/apps/pubs/default.aspx?id=190822.

[6] Bluefire at UCAR, in the *Proceedings of USENIX Annual Technical Conference (ATC '13)*, June 2013: http://www.ucar.edu/news/releases/2008/images/bluefire_backhalf_large.jpg.

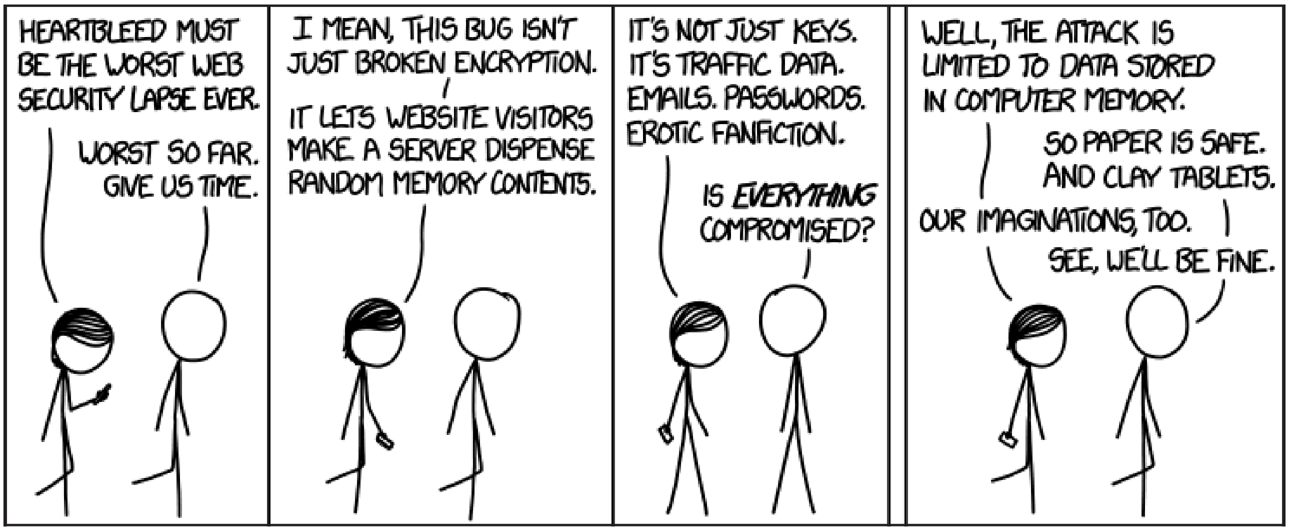
[7] Hot water-cooled supercomputer: http://www.zurich.ibm.com/news/12/superMUC.html.

[8] SiCortex supercomputer: http://en.wikipedia.org/wiki/SiCortex.

[9] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris, "Synnefo: A Complete Cloud Stack over Ganeti," *;login:,* vol. 38, no. 5, October 2013: https://www.usenix.org/login/october-2013/koukis.

[10] James Mickens, "The Saddest Moment," ;login: *logout,* May, 2013: https://www.usenix.org/publications/login-logout/may-2013/saddest-moment

xkcd


HEARTBLEED MUST BE THE WORST WEB SECURITY LAPSE EVER.
WORST SO FAR. GIVE US TIME.
I MEAN, THIS BUG ISN'T JUST BROKEN ENCRYPTION.
IT LETS WEBSITE VISITORS MAKE A SERVER DISPENSE RANDOM MEMORY CONTENTS.
IT'S NOT JUST KEYS. IT'S TRAFFIC DATA. EMAILS. PASSWORDS. EROTIC FANFICTION.
IS EVERYTHING COMPROMISED?
WELL, THE ATTACK IS LIMITED TO DATA STORED IN COMPUTER MEMORY.
SO PAPER IS SAFE. AND CLAY TABLETS.
OUR IMAGINATIONS, TOO.
SEE, WE'LL BE FINE.

xkcd.com

~okeanos
Large-Scale Cloud Service Using Ceph

FILIPPOS GIANNAKOS, VANGELIS KOUKIS, CONSTANTINOS VENETSANOPOULOS, AND NECTARIOS KOZIRIS

Filippos Giannakos is a cloud storage engineer at the Greek Research and Technology Network. His research interests include distributed and software-defined storage. Giannakos is a PhD candidate in the Computing Systems Laboratory at the School of Electrical and Computer Engineering, National Technical University of Athens. philipgian@grnet.gr

Vangelis Koukis is the technical lead of the ~okeanos project at the Greek Research and Technology Network (GRNET). His research interests include large-scale computation in the cloud, high-performance cluster interconnects, and shared block-level storage. Koukis has a PhD in electrical and computer engineering from the National Technical University of Athens. vkoukis@grnet.gr

Constantinos Venetsanopoulos is a cloud engineer at the Greek Research and Technology Network. His research interests include distributed storage in virtualized environments and large-scale virtualization management. Venetsanopoulos has a diploma in electrical and computer engineering from the National Technical University of Athens. cven@grnet.gr

Nectarios Koziris is an associate professor in the Computing Systems Laboratory at the National Technical University of Athens. His research interests include parallel architectures, interaction between compilers, OSes and architectures, OS virtualization, large-scale computer and storage systems, cloud infrastructures, distributed systems and algorithms, and distributed data management. Koziris has a PhD in electrical and computer engineering from the National Technical University of Athens. nkoziris@cslab.ece.ntua.gr

~Okeanos is a large-scale public cloud service powered by Synnefo and run by GRNET. Ceph is a distributed storage solution that targets scalability over commodity hardware. This article focuses on how we use Ceph to back the storage of ~okeanos. More specifically, we will describe what we aimed for in our storage system, the challenges we had to overcome, certain tips for setting up Ceph, and experiences from our current production cluster.

The ~okeanos Service

At GRNET, we provide ~okeanos [2, 4], a complete public cloud service, similar to AWS, for the Greek research and academic community. ~okeanos has Identity, Compute, Network, Image, Volume, and Object Storage services and is powered by Synnefo [3, 5]. ~okeanos currently holds more than 8000 VMs.

Our goals related to storage are to provide users with the following functionality:

- The ability to upload files and user-provided images by transferring only the missing data blocks (diffs).
- Persistent VMs for long-running computationally intensive tasks, or hosting services.
- Thin VM provisioning (i.e., no copy of disk data when creating a new VM) to enable fast spawning of hundreds of VMs, with zero-copy snapshot functionality for checkpointing and cheap VM backup.

We aim to run a large-scale infrastructure (i.e., serve thousands of users and tens of thousands of VMs) over commodity hardware.

A typical workflow on ~okeanos is that a user downloads an image, modifies it locally (e.g., by adding any libraries or code needed), and reuploads it by synchronizing it with the server and uploading only the modified part of the image. The user then spawns a large number of persistent VMs thinly, with zero data movement. The VM performs some computations, and the user can then take a snapshot of the disk as a checkpoint; the snapshot also appears as a regular file on the Object Storage service, which the user can sync to his/her local file system to get the output of the calculations for further offline processing or for backup.

To achieve the described workflow, however, we had to overcome several challenges regarding storage.

Storage Challenges

We stumbled upon two major facts while designing the service: (1) providing persistent VMs conflicts with the ability to scale and (2) using the same storage entities from different services requires a way to access them uniformly without copying data.

Providing persistent VMs while being able to scale is a demanding and difficult problem to solve. Persistence implies the need to live migrate VMs (change their running node while keeping the VM running) or fail them over (shut them down and restart them on another node) when a physical host experiences a problem. This implies shared storage among the nodes because the VM, and consequently the VM's host, needs to access the virtual disk's data. The most common solution to provide shared storage among the nodes is a central NAS/SAN exposed to all nodes.

However, having a central storage to rely on cannot scale and can be a single point of failure for the storage infrastructure. DRBD is another well-known enterprise-level solution to provide persistent VMs and scale at the same time. DRBD mirrors a virtual disk between two nodes and propagates VM writes that occur on one node to the replica. DRBD is essentially a RAID-1 setup over network. However, this choice imposes specific node pairs where the data is replicated and where the VM can be migrated, narrowing the administrator's options when performing maintenance. Moreover, it does not allow for either thin VM provisioning or for sharing blocks of data among VM volumes and user uploaded files, which takes us to the second problem.

The second problem we had to overcome involved presenting the different storage entities to the cloud layer uniformly. For example, a snapshot should be treated as a regular file to be synced and also as a disk image to be cloned. The actual data, though, remain the same in both cases, and only the cloud layer on top of the storage should distinguish between the two aspects. Therefore, we needed a way to access the same data from different cloud services uniformly, through a common storage layer.

To solve these problems, we needed a storage layer that would abstract the cloud aspect of a resource from its actual data and allow the ability to present this data in various ways. Additionally, we needed this mechanism to be independent from the actual data store. Because no suitable solution existed, we created Archipelago.

Archipelago

Archipelago [1] is a distributed storage layer that unifies how storage is perceived by the services, presenting the same resources in different ways depending on how the service wants to access them. It sits between the service that wants to store or retrieve data and the actual data store. Archipelago uses storage back-end drivers to interact with any shared data store. It also provides all the necessary logic to enable thin volume provisioning, snapshot functionality, and deduplication over any shared storage. Therefore, Archipelago allows us to view the cloud resources uniformly, whether these are images, volumes, snapshots, or just user files. They are just data stored in a storage back end, accessed by Archipelago.

Synnefo uses Archipelago to operate on the various representations of the same data:

- From the perspective of Synnefo's Object Storage service, images are treated as common files, with full support for remote syncing and sharing among users.
- From the perspective of the Compute service, images can be cloned and snapshotted repeatedly, with zero data movement from service to service.
- And, finally, snapshots can appear as new image files, again with zero data movement.

Backing Data Store

Archipelago acts as a middle layer that presents the storage resources and solves the resource unification problem but does not actually handle permanent storage. We needed a data store to host the data. Because we were not bound by specific constraints, we had various shared storage options. We decided to start with an existing large central NAS, exposed to all nodes as an NFS mount. This approach had several disadvantages:

- It could not scale well enough to hold the amount of users and data we aimed for.
- Having a large enterprise-level NAS imposed geographical constraints. The data reside in only one datacenter.
- It imposes a centralized architecture, which is difficult and costly to extend.

Because ~okeanos had exponential growth, we needed a different storage solution.

Ceph

Ceph is a distributed storage solution. It offers a distributed object store, called RADOS [6], block devices over RADOS called RBD, a distributed file system called CephFS, and an HTTP gateway called RadosGW. We have been following its progress and experimenting with it since early 2010.

RADOS is the core of Ceph Storage. It is a distributed object store comprising a number of OSDs, which are the software components (processes) that take care of the underlying storage of data. RADOS distributes the objects among all OSDs in the cluster. It manages object replication for redundancy, automatic data recovery, and cluster rebalancing in the presence of node failures.

RBD provides block devices from objects stored on RADOS. It splits a logical block device in a number of fixed-size objects and stores these objects on RADOS.

CephFS provides a shared file system with near-POSIX semantics, which can be mounted from several nodes. CephFS splits files into objects, which are then stored on RADOS. It also consists of one or more metadata servers to keep track of file-system metadata.

Finally, **RadosGW** is an HTTP REST gateway to the RADOS object store.

Ceph seemed a promising storage solution that provided scalable distributed storage based on commodity hardware, so we decided to evaluate it. Ceph exposes the data stored in RADOS in various forms, but it does not act on them uniformly like Archipelago does. Because we already had an HTTP gateway and VM volume representation of the data by Archipelago, we needed RADOS, which also happens to be the most stable and mature part of Ceph, to store and retrieve objects. We used Ceph's librados to implement a user-space driver for Archipelago to store our cloud data on RADOS and integrate it with Synnefo.

Things to Consider with Ceph

While evaluating RADOS, we experimented with various RADOS setup parameters and drew several conclusions regarding setup methodology and RADOS performance.

OSD/Disks Setup

Ceph's OSDs are user-space daemons that form a RADOS cluster. Each OSD needs permanent storage space where it will hold the data. This space is called "ObjectStore" in RADOS terminology. Ceph currently implements its ObjectStore using files, so we will use the term "filestore." We had several storage nodes that could host RADOS OSDs. Each node had multiple disks. So the question arose how to set up our RADOS cluster and where to place the filestores. We had numerous parameters to consider, including the number of OSDs per physical node, the number of disks per OSD, and the exact disk setup. After extensive testing, we concluded that it is beneficial to have multiple OSDs per node, and we dedicated one disk to each one. This setup ensures that one OSD does not compete with any other for the same disk and allows for multiple OSDs per node, resulting in improved aggregate performance.

Journal Mode and Filestore File System

Along with the filestore, each OSD keeps a journal to guarantee data consistency while keeping write latency low. This means that every write gets written twice in each OSD, once in the journal and once in the backing filestore. There are two modes in which the journal can operate, write-ahead and write-parallel. In write-ahead mode, each write operation is first committed to the journal and then applied to the filestore. The write operation can be acknowledged as soon as it is safely written to the journal. In write-parallel mode, each write is written to both the journal and the filestore in parallel, and the write can be acknowledged when either of the two operations is completed successfully. The write-parallel mode requires btrfs as the file system of the backing filestore, because it makes use of btrfs-specific features, such as snapshots and rollbacks, to guarantee data consistency. On the other hand, the write-ahead mode can work with all the recommended file systems, such as ext4 and XFS. Because btrfs is still not production ready and showed significant instability under heavy load, we decided to proceed with ext4 and write-ahead journal mode.

Journal Placement and Size

The RADOS journal can be placed in a file in the backing filestore, in a separate disk partition on the same disk, or in a completely separate block device. The first two options are suboptimal because they share the bandwidth and IOPS of the OSD's disk with the filestore, essentially halving the overall disk performance. Therefore, we placed the RADOS journal in a separate block device.

You might think that, because writes are confirmed when they hit the journal, an OSD can sustain improved write performance for a longer period by using a bigger journal on a fast device, falling back to filestore performance when the journal gets full. However, our experiments showed that RADOS OSDs do not work like that. If the journal media is much faster than the filestore, the OSD pauses writes when it tries to sync the journal with the filestore. This behavior can result in abnormal and erratic patterns during write bursts. Thus, a small portion of a block device with performance close to the one of the filestore should be used to hold the journal. Because we do not need large journals, multiple journals can be combined in the same block device, as long as it provides enough bandwidth for all of them.

RADOS Latency

When evaluating a storage system, especially for VM virtual disks' data, latency plays a critical role. VMs tend to perform small (4 K to 16 K) I/Os where latency becomes apparent. Our measurements showed that RADOS has a non-negligible latency of about 2 ms, so you cannot expect latency comparable with local disks. This behavior can be masked by issuing multiple requests or performing larger I/O to achieve high throughput. Also, because the requests are equally distributed among all OSDs in a cluster, the overall performance remains highly acceptable.

Data Integrity Checking

Silent data corruption caused by hardware can be a big issue on a large data store. RADOS offers a scrubbing feature, which works in two modes: regular scrubbing and deep scrubbing. Regular scrubbing is lighter and checks that the object is correctly replicated among the nodes. It also checks the object's metadata and attributes. Deep scrubbing is heavier and expands the check to the actual data. It ensures data integrity by reading the data and computing checksums.

Storage vs. Compute Nodes

Another setup choice we had was to keep our storage nodes distinct from our compute nodes, where the VMs reside. This has two main advantages. First, OSDs do not compete with the VMs for compute power. OSDs do not require a lot of CPU power normally, but it can be noticeable while scrubbing or rebalancing. Normal operations will also require more CPU power with the upcoming erasure coding feature, where CPU power is used to reconstruct objects in order to save storage space. Second, we can use the storage container RAM as cache for hot data using the Linux page-cache mechanism, which uses free RAM to cache recently accessed files on a file system. Hosting VMs on the same node would leave less memory for this purpose.

Experiences from Production

Ceph has been running in production since April 2013 acting as an Archipelago storage back end. As of this writing, Ceph's RADOS backs more than 2000 VMs and more than 16 TB of user-uploaded data on the Object Storage service.

Our current production setup comprises 20 physical nodes. Each node has

- 2 × 12-Core Xeon(R) E5645 CPU
- 96 GBs of RAM
- 12 × 2 TB SATA disks

Our storage nodes can provide more CPU power than Ceph currently needs. We are planning to use this extra power to seamlessly enable future Ceph functionality, like erasure coding, and to divert computationally intensive storage-related tasks (e.g., hashing) to the storage nodes, using the RADOS "classes" mechanism.

Each physical node hosts six OSDs. Each OSD's data resides on a RAID-1 setup between two 2 TB disks. RADOS replicates objects itself, but because it was under heavy development, we wanted to be extra sure about the safety of our data. By using this setup along with a replication level 2, we only use one fourth of our overall capacity, which covers our current storage needs. As our storage needs grow and RADOS matures, we aim to break the RAID setups and double the cluster capacity.

Using Ceph in production has several advantages:

- It allows us to use large bulk commodity hardware.
- It provides a central shared storage that can self-replicate, self-heal, and self-rebalance when a hardware node or a network link fails.
- It can scale on demand by adding more storage nodes to the cluster as demand increases.
- It enables live migration of VMs to any other node.

Using Ceph in a large-scale system also revealed some of its current weaknesses. Scrubbing and especially deep scrubbing can take a lot of time to complete. During these actions, there is significant performance drop. The cluster fills with slow requests and VMs are affected. This is a major drawback, and we had to completely disable this functionality. We plan to re-enable it as soon as it can be used without significant performance regression. Performance also drops when rebuilding the cluster after an OSD failure or when the cluster rebalances itself after the addition of a new OSD. This issue is closely related to the performance drop during deep scrubbing, and it occurs because RADOS does not throttle traffic related to recovery and deep scrubbing according to the underlying disk utilization and the rate of the incoming client I/O.

On the other hand, Ceph has survived numerous hardware failures with zero data loss and minimal service disruption. Its overall usage experience outweighs the hiccups we endured since we began using it for production purposes.

In conclusion, our experience from running a large-scale Ceph cluster shows that it has obvious potential. It can run well over commodity hardware, scale without any visible overhead, and helped us to deploy our service. However, in its current state, running it for production purposes has disadvantages because it suffers from performance problems when performing administrative actions.

Ceph is open source. Source code, more info, and extra material can be found at http://www.ceph.com.

References

[1] Archipelago: http://www.synnefo.org/docs/archipelago/latest.

[2] The ~okeanos service: http://okeanos.grnet.gr.

[3] Synnefo software: http://www.synnefo.org.

[4] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris, "~okeanos: Building a Cloud, Cluster by Cluster," *IEEE Internet Computing*, vol. 17, no. 13, May-June 2013, pp. 67–71.

[5] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris, "Synnefo: A Complete Cloud Stack over Ganeti," USENIX, *;login:*, vol. 38, no. 5 (October 2013), pp. 6–10.

[6] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn, "RADOS: A Scalable, Reliable Storage Service for Petabyte-Scale Storage Clusters," in *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07,* PDSW '07 (ACM, 2007), pp. 35–44.

Analysis of HDFS under HBase

A Facebook Messages Case Study

TYLER HARTER, DHRUBA BORTHAKUR, SIYING DONG, AMITANAND AIYER, LIYIN TANG, ANDREA C. ARPACI-DUSSEAU, AND REMZI H. ARPACI-DUSSEAU

Tyler Harter is a student at the University of Wisconsin—Madison, where he has received his bachelor's and master's degrees and is currently pursuing a computer science Ph.D. Professors Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau advise him, and he is funded by an NSF Fellowship. Tyler is interested in helping academics understand the I/O workloads seen in industry; toward this end he has interned at Facebook twice and studied the Messages application as a Facebook Fellow. Tyler has also studied desktop I/O, co-authoring an SOSP '11 paper, "A File Is Not a File...," which received a Best Paper award. harter@cs.wisc.edu

Dhruba Borthakur is an engineer in the Database Engineering Team at Facebook. He is leading the design of RocksDB, a datastore optimized for storing data in fast storage. He is one of the founding architects of the Hadoop Distributed File System and has been instrumental in scaling Facebook's Hadoop cluster to multiples of petabytes. Dhruba also is a contributor to the Apache HBase project and Andrew File System (AFS). Dhruba has an MS in computer science from the University of Wisconsin-Madison. He hosts a Hadoop blog at http://hadoopblog.blogspot.com/ and a RocksDB blog at http://rocksdb.blogspot.com. dhruba@fb.com

Large-scale distributed storage systems are exceedingly complex and time-consuming to design, implement, and operate. As a result, rather than cutting new systems from whole cloth, engineers often opt for layered architectures, building new systems upon already-existing ones to ease the burden of development and deployment. In this article, we examine how layering causes write amplication when HBase is run on top of HDFS and how tighter integration could result in improved write performance. Finally, we take a look at whether it makes sense to include an SSD to improve performance while keeping costs in check.

Layering, as is well known, has many advantages [6]. For example, construction of the Frangipani distributed file system was greatly simplified by implementing it atop Petal, a distributed and replicated block-level storage system [7]. Because Petal provides scalable, fault-tolerant virtual disks, Frangipani could focus solely on file-system-level issues (e.g., locking); the result of this two-layer structure, according to the authors, was that Frangipani was "relatively easy to build."

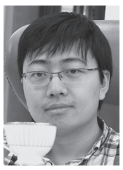
Unfortunately, layering can also lead to problems, usually in the form of decreased performance, lowered reliability, or other related issues. For example, Denehy et al. show how naïve layering of journaling file systems atop software RAIDs can lead to data loss or corruption [2]. Similarly, others have argued about the general inefficiency of the file system atop block devices [4].

In this article, we focus on one specific, and increasingly common, layered storage architecture: a distributed database (HBase, derived from Google's BigTable) atop a distributed file system (HDFS, derived from the Google File System). Our goal is to study the interaction of these important systems with a particular focus on the lower layer, which leads to our highest-level question: Is HDFS an effective storage back end for HBase?

To derive insight into this hierarchical system, and therefore answer this question, we trace and analyze it under a popular workload: Facebook Messages (FM). FM is a messaging system that enables Facebook users to send chat and email-like messages to one another; it is quite popular, handling millions of messages each day. FM stores its information within HBase (and thus, HDFS) and hence serves as an excellent case study.

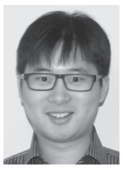
To perform our analysis, we collected detailed HDFS traces over an eight-day period on a subset of FM machines. These traces reveal a workload very unlike traditional GFS/HDFS patterns. Whereas workloads have traditionally consisted of large, sequential I/O to very large files, we find that the FM workload represents the opposite. Files are small (750 KB median), and I/O is highly random (50% of read runs are shorter than 130 KB).

We also use our traces to drive a multilayer simulator, allowing us to analyze I/O patterns across multiple layers beneath HDFS. From this analysis, we derive numerous insights. For



Siying Dong is a software engineer working in the Database Engineering team at Facebook, focusing on RocksDB. He also worked on Hive, HDFS, and some other data warehouse infrastructures. Before joining Facebook, Siying worked in the SQL Azure Team at Microsoft. He received a bachelor's degree from Tsinghua University and a master's degree from Brandeis University. siying.d@fb.com

Amitanand Aiyer is a research scientist in the Core Data team at Facebook. He focuses on building fault-tolerant distributed systems and has been working on HBase to improve its availability and fault tolerance. Amitanand is an HBase Committer and has a Ph.D. from the University of Texas at Austin. amitanand.s@fb.com



Liyin Tang is a software engineer from the Core Data team at Facebook, where he focuses on building highly available and reliable storage services, and helps the service scale in the face of exponential data growth. Liyin also works as an HBase PMC member in the Apache community. He has a master's degree in computer science from the University of Southern California and a bachelor's degree in software engineering from Shanghai Jiao Tong University. liyin.tang@fb.com

example, we find that many features at different layers amplify writes, and that these features often combine multiplicatively. For example, HBase logs introduced a 10x overhead on writes, whereas HDFS replication introduced a 3x overhead; together, these features produced a 30x write overhead. When other features such as compaction and caching are also considered, we find writes are further amplified across layers. At the highest level, writes account for a mere 1% of the baseline HDFS I/O, but by the time the I/O reaches disk, writes account for 64% of the workload.

This finding indicates that even though FM is an especially read-heavy workload within Facebook, it is important to optimize for both reads and writes. We evaluate potential optimizations by modeling various hardware and software changes with our simulator. For reads, we observe that requests are highly random; therefore, we evaluate using flash to cache popular data. We find that adding a small SSD (e.g., 60 GB) can reduce latency by 3.5x. For writes, we observe compaction and logging are the major causes (61% and 36%, respectively); therefore, we evaluate HDFS changes that give HBase special support for these operations. We find such HDFS specialization yields a 2.7x reduction in replication-related network I/O and a 6x speedup for log writes. More results and analysis are discussed in our FAST '14 paper [8].

Background and Methodology

The FM storage stack is based on three layers: distributed database over distributed file system over local storage. These three layers are illustrated in Figure 1 under "Actual Stack." As shown, FM uses HBase for the distributed database and HDFS for the distributed file system. HBase provides a simple API allowing FM to put and get key-value pairs. HBase stores these records in data files in HDFS. HDFS replicates the data across three machines and thus can handle disk and machine failures. By handling these low-level fault tolerance details, HDFS frees HBase to focus on higher-level database logic. HDFS in turn stores replicas of HDFS blocks as files in local file systems. This design enables HDFS to focus on replication while leaving details such as disk layout to local file systems. The primary advantage of this layered design is that each layer has only a few responsibilities, so each layer is simpler (and less bug prone) than a hypothetical single system that would be responsible for everything.

One important question about this layered design, however, is: What is the cost of simplicity (if any) in terms of performance? We explore this question in the context of the FM workload. To understand how FM uses the HBase/HDFS stack, we trace requests from HBase to HDFS, as shown in the Figure 1. We collect traces by deploying a new HDFS tracing framework that we built to nine FM machines for 8.3 days, recording 71 TB of HDFS I/O.

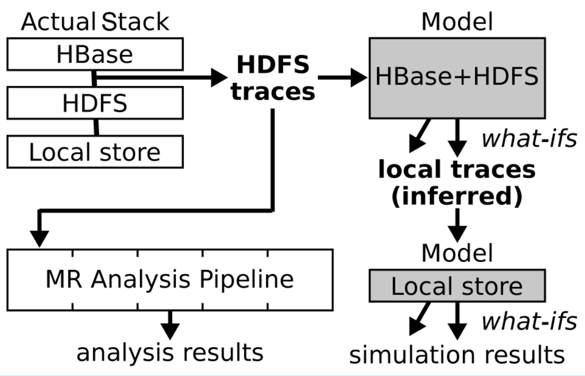


Figure 1: Tracing, analysis, and simulation

The traces record the I/O of four HBase activities that use HDFS: logging, flushing, reading, and compacting. When HBase receives a put request, it immediately logs the record to an HDFS file for persistence. The record is also added to an HBase write buffer, which, once filled, HBase flushes to a sorted data file. Data files are never modified, so when a get request arrives, HBase reads multiple data files in order to find the latest version of the data. To limit the number of files that must be read, HBase occasionally compacts old files, which involves merge sorting multiple small data files into one large file and then deleting the small files.

We do two things with our traces of these activities. First, as Figure 1 shows, we feed them to a pipeline of MapReduce analysis jobs. These jobs compute statistics that characterize the workload. We discuss

Andrea Arpaci-Dusseau is a professor and associate chair of computer sciences at the University of Wisconsin—Madison. Andrea co-leads a research group with her husband, Remzi, and has advised 14 students through their Ph.D. dissertations; their group has received many Best Paper awards and sorting world records. She is a UW—Madison Vilas Associate and received the Carolyn Rosner "Excellent Educator" award; she has served on the NSF CISE Advisory Committee and as faculty co-director of the Women in Science and Engineering (WISE) Residential Learning Community. dusseau@cs.wisc.edu

Remzi Arpaci-Dusseau is a full professor in the Computer Sciences Department at the University of Wisconsin—Madison. Remzi co-leads a research group with his wife, Andrea; details of their research can be found at http://research.cs.wisc.edu/adsl. Remzi has won the SACM Professor-of-the-Year award four times and the Rosner "Excellent Educator" award once. Chapters from a freely available OS book he and his wife co-wrote, found at www.ostep.org, have been downloaded over 1/2 million times. Remzi has served as co-chair of USENIX ATC, FAST, OSDI, and (upcoming) SOCC conferences. Remzi has been a NetApp faculty fellow, an IBM faculty award winner, an NSF CAREER award winner, and currently serves on the Samsung DS CTO and UW Office of Industrial Partnership advisory boards. remzi@cs.wisc.edu

these characteristics in the next section and suggest ways to improve both the hardware and software layers of the stack. Second, we evaluate our suggestions via a simulation of layered storage. We feed our traces to a model of HBase and HDFS that translates the HDFS traces into inferred traces of requests to local file systems. For example, our simulator translates an HDFS write to three local file-system writes based on a model of triple replication. We then feed our inferred traces of local file-system I/O to a model of local storage. This model computes request latencies and other statistics based on submodels of RAM, SSDs, and rotational disks (each with its own block scheduler). We use these models to evaluate different ways to build the software and hardware layers of the stack.

Workload Behavior

In this section, we characterize the FM workload with four questions: What activities cause I/O at each layer of the stack? How large is the dataset? How large are HDFS files? And, is I/O sequential?

I/O Activities

We begin by considering the number of reads and writes at each layer of the stack in Figure 2. The first bar shows HDFS reads and writes, excluding logging and compaction overheads. At this level, writes represent only 1% of the 47 TB of I/O. The second bar includes these overheads. As shown, overheads are significant and write dominated, bringing the writes to 21%.

HBase tolerates failures by replicating data with HDFS. Thus, one HDFS write causes three writes to local files and two network transfers. The third bar of Figure 2 shows that this tripling increases the writes to 45%. Not all this file-system I/O will hit disk, as OS caching absorbs some of the reads. The fourth bar shows that only 35 TB of disk reads are caused by the 56 TB of file-system reads. The bar also shows a write increase, as very small file-system writes cause 4 KB-block disk writes. Because of these factors, writes represent 64% of disk I/O.

Dataset Size

Figure 3 gives a layered overview similar to that of Figure 2, but for data rather than I/O. The first bar shows 3.9 TB of HDFS data received some non-overhead I/O during tracing (data deleted during tracing is not counted). Nearly all this data was read and a small portion written. The second bar shows data touched by any I/O (including compaction and logging overheads). The third bar shows how much data is touched at the local level during tracing. This bar also shows untouched data. Most of the 120 TB of data is very cold; only a third is accessed over the eight-day period.

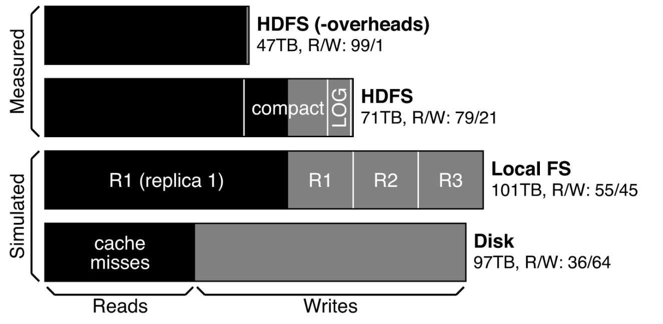


Figure 2: I/O across layers. Black sections represent reads and gray sections represent writes. The top two bars indicate HDFS I/O as measured directly in the traces. The bottom two bars indicate local I/O at the file-system and disk layers as inferred via simulation.

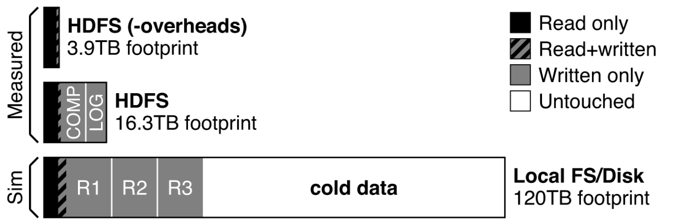


Figure 3: Data across layers. This is the same as Figure 2 but for data instead of I/O. COMP is compaction.

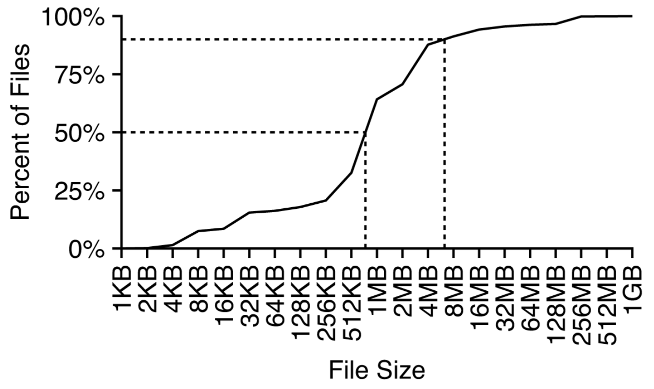


Figure 4: File-size CDF. The distribution is over the sizes of created files with 50th and 90th percentiles marked.

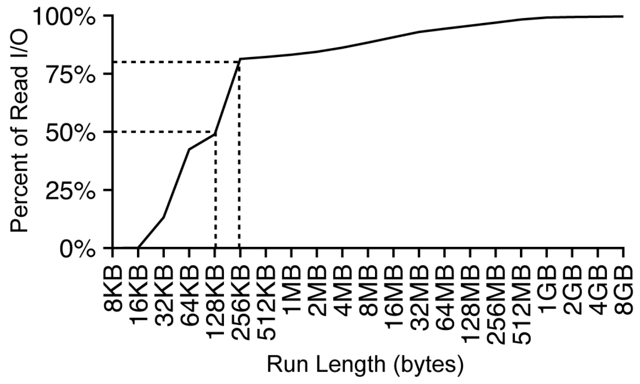


Figure 5: Run-size CDF. The distribution is over sequential read runs, with 50th and 80th percentiles marked.

File Size

GFS (the inspiration for HDFS) assumed "multi-GB files are the common case, and should be handled efficiently" [5]. Previous HDFS workload studies also show this; for example, MapReduce inputs were found to be about 23 GB at the 90th percentile (Facebook in 2010) [1].

Figure 4 reports a CDF of the sizes of HDFS files created during tracing. We observe that created files tend to be small; the median file is 750 KB, and 90% are smaller than 6.3 MB. This means that the data-to-metadata ratio will be higher for FM than for traditional workloads, suggesting that it may make sense to distribute metadata instead of handling it all with a single NameNode.

Sequentiality

GFS is primarily built for sequential I/O and, therefore, assumes "high sustained bandwidth is more important than low latency" [5]. All HDFS writes are sequential, because appends are the only type of writes supported, so we now measure read sequentiality. Data is read with sequential runs of one or more contiguous read requests. Highly sequential patterns consist of large runs, whereas random patterns consist mostly of small runs.

Figure 5 shows a distribution of read I/O, distributed by run size. We observe that most runs are fairly small. The median run size is 130 KB, and 80% of runs are smaller than 250 KB, indicating FM reads are very random. These random reads are primarily caused by get requests; the small (but significant) portion of reads that are sequential are mostly due to compaction reads.

Layering: Pitfalls and Solutions

In this section, we discuss different ways to layer storage systems and evaluate two techniques for better integrating layers.

Layering Background

Three important layers are the local layer (e.g., disks, local file systems, and a DataNode), the replication layer (e.g., HDFS), and the database layer (e.g., HBase). FM composes these in a mid-replicated pattern (Figure 6a), with the database above replication and the local stores below. The merit of this design is simplicity. The database can be built with the assumption that underlying storage will be available and never lose data. Unfortunately, this approach separates computation from data. Computation (e.g., compaction) can co-reside with, at most, one replica, so all writes involve network I/O.

Top-replication (Figure 6b) is an alternative used by Salus [9]. Salus supports the HBase API but provides additional robustness and performance advantages. Salus protects against memory corruption by replicating database computation as well as the data itself. Doing replication above the database level also reduces network I/O. If the database wants to reorganize data on disk (e.g., via compaction), each database replica can do so on its local copy. Unfortunately, top-replicated storage is complex, because the database layer must handle underlying failures as well as cooperate with other databases.

Mid-bypass (Figure 6c) is a third option proposed by Zaharia et al. [10]. This approach (like mid-replication) places the replication layer between the database and the local store; but, to improve performance, an RDD (Resilient Distributed Dataset)

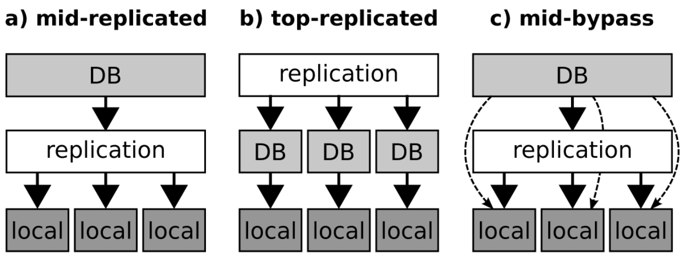


Figure 6: Layered architectures. The HBase architecture (mid-replicated) is shown, as well as two alternatives. Top-replication reduces network I/O by co-locating database computation with database data. The mid-bypass architecture is similar to mid-replication but provides a mechanism for bypassing the replication layer for efficiency.

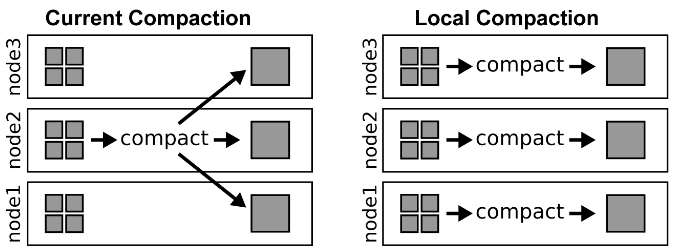


Figure 7: Local-compaction architecture. The HBase architecture (left) shows how compaction currently creates a data flow with significant network I/O, represented by the two lines crossing machine boundaries. An alternative (right) shows how local reads could replace network I/O.

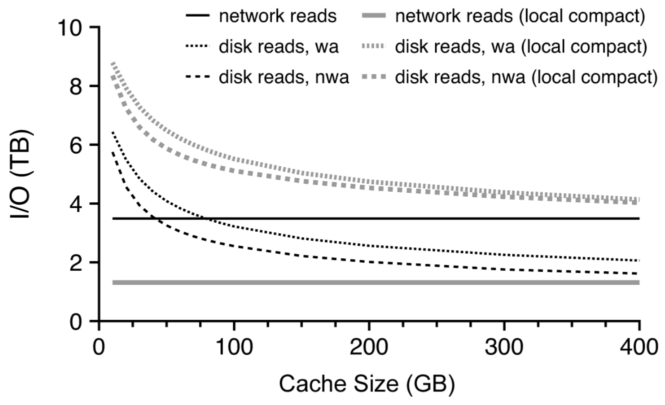


Figure 8: Local-compaction results. The thick gray lines represent HBase with local compaction, and the thin black lines represent HBase currently. The solid lines represent network reads, and the dashed lines represent disk reads; long-dash represents the no-write allocate cache policy and short-dash represents write allocate.

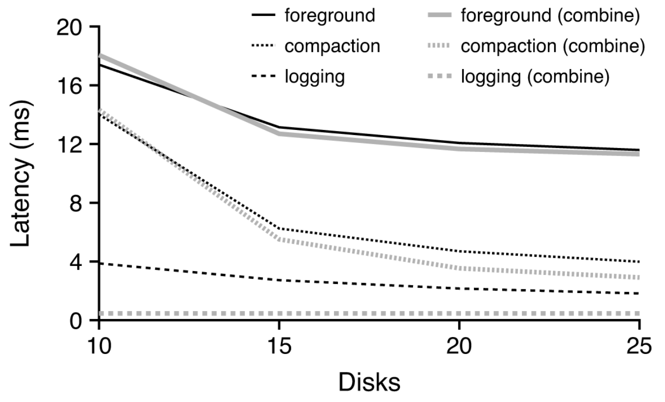


Figure 9: Combined logging results. Disk latencies for various activities are shown, with (gray) and without (black) combined logging.

API lets the database bypass the replication layer. Network I/O is avoided by shipping computation directly to the data. HBase compaction, if built upon two RDD transformations, join and sort, could avoid much network I/O.

Local Compaction

We simulate the mid-bypass approach, shipping compaction operations directly to all the replicas of compaction inputs. Figure 7 shows how local compaction differs from traditional compaction; network I/O is traded for local I/O, to be served by local caches or disks.

Figure 8 shows the result: a 62% reduction in network reads, from 3.5 TB to 1.3 TB. The figure also shows disk reads, with and without local compaction, and with either write allocate (wa) or no-write allocate (nwa) caching policies. We observe that disk I/O increases slightly more than network I/O decreases. For example, with a 100-GB cache, network I/O is decreased by 2.2 GB, but disk reads are increased by 2.6 GB for no-write allocate. This is unsurprising: HBase uses secondary replicas for fault tolerance rather than for reads, so secondary replicas are written once (by a flush or compaction) and read at most once (by compaction). Thus, local-compaction reads tend to (1) be misses and (2) pollute the cache with data that will not be read again. Even still, trading network I/O for disk I/O in this way is desirable, as network infrastructure is generally much more expensive than disks.

Combined Logging

We now consider the interaction between replication and HBase logging. Currently, a typical HDFS DataNode receives logs from three RegionServers. Because HDFS just views these logs as regular HDFS files, HDFS will generally write them to different disks. We evaluate an alternative to this design: combined logging. With this approach, HBase passes a hint to HDFS, identifying the logs as files that are unlikely to be read back. Given this hint, HDFS can choose a write-optimized layout for the logs. In particular, HDFS can interleave the multiple logs in a single write-stream on a single dedicated disk.

We simulate combined logging and measure performance for requests that go to disk; we consider latencies for logging, compaction, and foreground reads. Figure 9 reports the results for varying numbers of disks. The latency of log writes decreases dramatically with combined logging (e.g., by 6x with 15 disks). Foreground-read and compaction requests are also slightly faster in most cases due to less competition with logs for seeks. Currently, put requests do not block on log writes, so logging is a background activity. If, however, HBase were to give a stronger guarantee for puts (namely that data is durable before returning), combined logging would make that guarantee much cheaper.

Hardware: Adding a Flash Layer

Earlier, we saw FM has a very large, mostly cold dataset; keeping all this data in flash would be wasteful, costing upwards of $10k/machine (assuming flash costs $0.80/GB). Thus, because we should not just use flash, we now ask, should we use no flash or some flash? To answer this, we need to compare the performance improvements provided by flash to the corresponding monetary costs.

We first estimate the cost of various hardware combinations (assuming disks are $100 each, RAM costs $5/GB, and flash costs $0.8/GB). To compute performance, we run our simulator on our traces using each hardware combination. We try nine RAM/disk combinations, each with no flash or a 60 GB SSD; these represent three amounts of disk and three amounts of RAM. When the 60 GB SSD is used, the RAM and flash function as a tiered LRU cache.

Figure 10 shows how adding flash changes both cost and performance. For example, the leftmost two bars of the leftmost plot show that adding a 60 GB SSD to a machine with 10 disks and 10 GB of RAM decreases latency by a factor of 3.5x (from 19.8 ms to 5.7 ms latency) while only increasing costs by 5%. Across all nine groups of bars, we observe that adding the SSD always increases costs by 2–5% while decreasing latencies by 17–71%. In two thirds of the cases, flash cuts latency by more than 40%.

We have shown that adding a small flash cache greatly improves performance for a marginal initial cost. Now, we consider long-term replacement caused by flash wear, as commercial SSDs often support only 10k program/erase cycles. We consider three factors that affect flash wear. First, if there is more RAM, there will be fewer evictions to the flash level of the cache and therefore fewer writes and less wear. Second, if the flash device is large, writes will be spread over more cells, so each cell will live longer. Third, using a strict LRU policy can cause excessive writes for some workloads by frequently promoting and evicting the same items back and forth between RAM and flash.

Figure 11 show how these three factors affect flash lifetime. The black "Strict LRU" lines correspond to the same configuration used in Figure 10 for the 10 GB and 30 GB RAM results. The amount of RAM makes a significant difference. For example, for strict LRU, the SSD will live 58% longer with 30 GB of RAM instead of 10 GB of RAM. The figure also shows results for a wear-friendly policy with the gray lines. In this case, RAM and flash are each an LRU independently, and RAM evictions are inserted into flash, but (unlike strict LRU) flash hits are not repromoted to RAM. This alternative to strict LRU greatly reduces wear by reducing movement between RAM and flash. For example, with 30 GB of RAM, we observe that 240 GB SSD will last 2.7x longer if the wear-friendly policy is used. Finally, the figure shows that the amount of flash is a major factor in flash lifetime. Whereas the 20 GB SSD lasts between 0.8 and 1.6 years (depending on policy and amount of RAM), the 120 GB SSD always lasts at least five years.

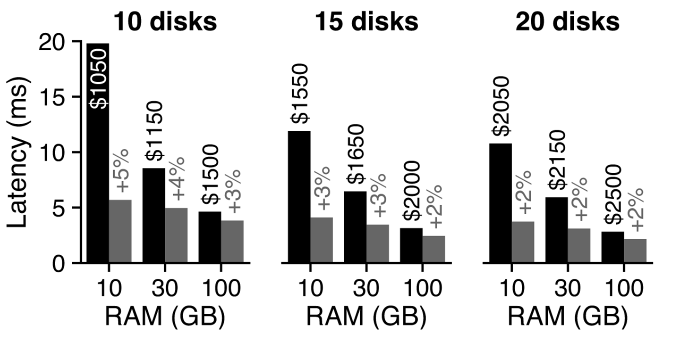


Figure 10: Flash cost and performance. Black bars indicate latency without flash, and gray bars indicate latency with a 60 GB SSD. Latencies are only counted for foreground I/O (e.g., servicing a get), not background activities (e.g., compaction). Bar labels indicate cost. For the black bars, the labels indicate absolute cost, and for the gray bars, the labels indicate the cost increase relative to the black bar.

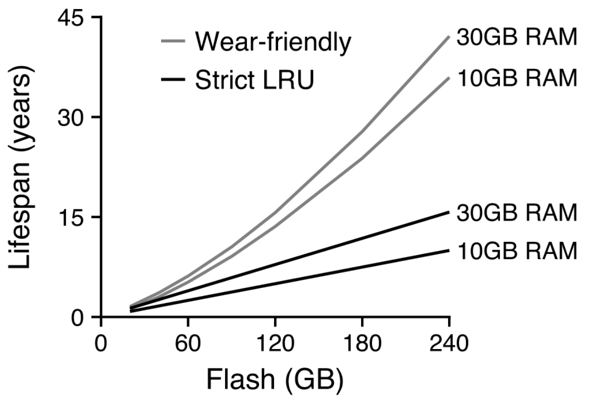


Figure 11: Flash lifetime. The relationship between flash size and flash lifetime is shown for both the keep policy (gray lines) and promote policy (black lines). There are two lines for each policy (10 or 30 GB RAM).

We conclude that adding a small SSD cache is a cost-effective way to improve performance. Adding a 60 GB SSD can often double performance for only a 5% cost increase. We find that for large SSDs, flash has a significant lifetime, and so avoiding wear is probably unnecessary (e.g., 120 GB SSDs last more than five years with a wear-heavy policy), but for smaller SSDs, it is useful to choose caching policies that avoid frequent data shuffling.

Conclusions

We have presented a detailed multilayer study of storage I/O for Facebook Messages. Our research relates to common storage ideas in several ways. First, the GFS-style architecture is based on workload assumptions, such as "high sustained bandwidth is more important than low latency" and "multi-GB files are the common case, and should be handled efficiently" [5]. We find FM represents the opposite workload, being dominated by small files and random I/O.

Second, layering storage systems is very popular; Dijkstra found layering "proved to be vital for the verification and logical soundness" of an OS [3]. We find, however, that simple layering has a cost. In particular, we show that relative to the simple layering currently used, tightly integrating layers reduces replication-related network I/O by 62% and makes log writes 6x faster. We further find that layers often amplify writes multiplicatively. For example, a 10x logging overhead (HBase level) combines with a 3x replication overhead (HDFS level), producing a 30x write overhead.

Third, flash is often extolled as a disk replacement. For example, Jim Gray has famously said that "tape is dead, disk is tape, flash is disk." For Messages, however, flash is a poor replacement for disk, as the dataset is very large and mostly cold, and storing it all in flash would cost over $10k/machine. Although we conclude that most data should continue to be stored on disk, we find small SSDs can be quite useful for storing a small, hot subset of the data. Adding a 60 GB SSD can often double performance while only increasing costs by 5%.

In this work, we take a unique view of Facebook Messages, not as a single system but as a complex composition of layered subsystems. We believe this perspective is key to deeply understanding modern storage systems. Such understanding, we hope, will help us better integrate layers, thereby maintaining simplicity while achieving new levels of performance.

Acknowledgments

We thank the anonymous reviewers and Andrew Warfield (our FAST shepherd) for their tremendous feedback, as well as members of our research group for their thoughts and comments on this work at various stages. We also thank Pritam Damania, Adela Maznikar, and Rishit Shroff for their help in collecting HDFS traces.

This material was supported by funding from NSF grants CNS-1319405 and CNS-1218405 as well as generous donations from EMC, Facebook, Fusion-io, Google, Huawei, Microsoft, NetApp, Sony, and VMware. Tyler Harter is supported by the NSF Fellowship and Facebook Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] Yanpei Chen, Sara Alspaugh, and Randy Katz, "Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads," *Proceedings of the VLDB Endowment*, vol. 5, no. 12 (August 2012), pp. 1802–1813.

[2] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, "Journal-Guided Resynchronization for Software RAID," *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)* (December 2005), pp. 87–100.

[3] E. W. Dijkstra, "The Structure of the THE Multiprogramming System," *Communications of the ACM*, vol. 11, no. 5 (May 1968), pp. 341–346.

[4] Gregory R. Ganger, "Blurring the Line Between OSes and Storage Devices," Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.

[5] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google File System," *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (October 2003), pp. 29–43.

[6] Jerome H. Saltzer, David P. Reed, and David D. Clark, "End-to-End Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, no. 4 (November 1984), pp. 277–288.

[7] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee, "Frangipani: A Scalable Distributed File System," *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)* (October 1997), pp. 224–237.

[8] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, "Analysis of HDFS under HBase: A Facebook Messages Case Study," *Proceedings of the 12th Conference on File and Storage Technologies (FAST '14)* (February 2014).

[9] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin, "Robustness in the Salus Scalable Block Store," *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)* (April 2013).

[10] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-memory Cluster Computing," *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)* (April 2012).

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Publish and Present Your Work at USENIX Conferences

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

Get more details about each of these Calls for Papers and Participation at www.usenix.org/cfp.

INFLOW '14: 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads

October 5, 2014, Broomfield, CO

Submissions due: July 1, 2014, 11:59 p.m. PDT

The goal of INFLOW '14 is to bring together researchers and practitioners working in systems, across the hardware/software stack, who are interested in the cross-cutting issues of NVM/Flash technologies, operating systems, and emerging workloads.

HotDep '14: 10th Workshop on Hot Topics in System Dependability

October 5, 2014, Broomfield, CO

Submissions due: July 10, 2014

HotDep '14 will bring forth cutting-edge research ideas spanning the domains of systems and fault tolerance/reliability. The workshop will build links between the two communities and serve as a forum for sharing ideas and challenges.

FAST '15: 13th USENIX Conference on File and Storage Technologies

February 16–19, 2015, Santa Clara, CA

Submissions due: September 23, 2014, 9:00 p.m. PDT (Hard deadline, no extensions)

The 13th USENIX Conference on File and Storage Technologies (FAST '15) brings together storage-system researchers and practitioners to explore new directions in the design, implementation, evaluation, and deployment of storage systems. The program committee will interpret "storage systems" broadly; everything from low-level storage devices to information management is of interest. The conference will consist of technical presentations, including refereed papers, Work-in-Progress (WiP) reports, poster sessions, and tutorials.

CLOUD

OpenStack

MARK LAMOURINE

Mark Lamourine is a senior software developer at Red Hat. He's worked for the last few years on the OpenShift project. He's a coder by training, a sysadmin and toolsmith by trade, and an advocate for the use of Raspberry Pi style computers to teach computing and system administration in schools. Mark has been a frequent contributor to the *;login:* Book Reviews column. markllama@gmail.com

The goal of OpenStack is nothing less than to virtualize and automate every aspect of a corporate computational infrastructure. The purpose is to provide self-service access to all of the resources that traditionally have required the intervention of a collection of teams to manage. In this article, I describe where OpenStack came from and what the various parts do.

OpenStack is an implementation of Infrastructure as a Service (annoyingly abbreviated as IaaS and amusingly pronounced "eye-ass"). Several commercial IaaS services are available, with the best known being Amazon Web Services, Google Compute Engine, and Rackspace. IaaS is also known as a "cloud" service, and there are also commercial products to create a "private cloud," most notably VMware.

OpenStack was developed as an alternative to the commercial cloud providers. With it, you can create private or public cloud services and move resources between them. OpenStack uses concepts and terminology that are compatible with Amazon Web Services and that are becoming de facto standards.

Both Red Hat, through the Red Hat Distribution of OpenStack (RDO) community project, and Canonical offer installers designed to make the installation of a demo or proof-of-concept service relatively easy for those who want to experiment with running their own cloud.

Origin, History, and Release Naming

In 2010 Rackspace and NASA created the OpenStack Foundation with the goal of producing an open source IaaS project. Since then, more than 150 other companies and organizations have joined the project. NASA dropped out in 2012 citing lack of internal progress implementing OpenStack services and redirected funding to using commercial cloud providers. In the past two years, the pace of development and improvement has increased dramatically, to the point that all three major commercial Linux distribution providers (Canonical, SuSE, and Red Hat) have customized OpenStack offerings.

OpenStack development versioning employs a code-word scheme using English alphabetic ordering. The first development cycle was code named Austin (2010.1) and was released in October 2010. New versions have been released approximately every six months since the initial release. These are the most recent development code names:

- Essex (2012.1)
- Folsom (2012.2)
- Grizzly (2013.1)
- Havana (2013.2)
- Icehouse (2014.1) (release pending as of this writing)

Once released, the stable version is numbered with the four-digit year and a single-digit serial number, (YYYY.N). No one expects more than nine releases in a calendar year. Most people continue to refer to them by their code names.

The development pace is so rapid that the OpenStack foundation only lists the current release and the previous one as supported at any given time. I see the adoption by commercial distributions as an indicator that they think the current code base is stable enough to allow economical long-term support.

Any references to capabilities here will be to the Havana (2013.2) release unless otherwise noted.

Function and Stability

OpenStack is designed as a set of agent services, each of which manages some aspect of what would otherwise be a physical infrastructure. Of course, the initial focus was on virtualized computation, but there are now subservices that manage storage (three ways), network topology, authentication, as well as a unified Web user interface. As with the development cycles, OpenStack uses code names for the active components that make up the service. The service components are detailed below.

It's been four years since the initial release, and it's really only with the Folsom (2012.2) release that enough components are available and stable to attempt to create a reliable user service. As development progresses, usage patterns are discovered that indicate the need for creating new first-class components to manage different aspects of the whole. The Folsom release was the first to include a networking component (Neutron), which had previously been part of the computation service (Nova/Quantum).

Bare metal provisioning, which has been handled by a plugin to Nova, is getting a service agent of its own (code named "Ironic," in Icehouse 2014.1), which will be more capable and flexible. Communication between the components is currently carried over an AMQP bus implemented using RabbitMQ or QPID. A new OpenStack messaging and RPC service is in the works (code named Oslo, also in Icehouse). Each of these new agent services will replace and enhance some aspect of the current systems, but one can hope that the existing components have become stable enough that most changes will be additive rather than transformative.

Components

OpenStack is enamored of code names, and for anyone interested in deploying an OpenStack service, the first task is to learn the taxonomy. There are actually some good reasons to use code names. OpenStack is meant to be a modular system; and, at least once so far, an implementation of a subsystem (Quantum) has been replaced with a completely new implementation (Neutron).

Most components are active agents. They subscribe to a messaging service (AMQP) to accept commands and return responses. Each service also has a CLI tool that can communicate directly with the agent. The active components also have a backing database for persistence across restarts.

Compute (Nova)

The core of an IaaS system is its virtualized computers. The Nova service provides the compute resources for OpenStack. It controls the placement and management of the virtual machines within the running service. Originally, Nova also contained the networking, which has since moved to Neutron.

Storage

OpenStack offers several flavors of persistent storage, each of which is designed for a specific set of tasks.

IMAGE STORAGE (GLANCE)

Glance is the image store used for creating new running instances or for storing the state of an instance that has been paused. Glance takes complete file systems and bootable disk images as input and makes them available to boot or mount on running instances.

BLOCK STORAGE (CINDER)

Cinder is the OpenStack block storage service. This is where you allocate additional disk space to your running instances. Cinder storage is persistent across reboots. Cinder can be backed by a number of traditional block storage services such as NFS or Gluster.

OBJECT STORAGE (SWIFT)

Swift offers a way to store and retrieve whole blobs of data. The data are accessed via a REST protocol, which can be coded into applications using an appropriate API library. Object storage provides a means for an application to store and share data across different instances. The object store is arranged as a hierarchical set of "container" objects, each of which can hold other containers or discrete data objects. In this way, it corresponds roughly to the structure of a file system.

Network (Neutron)

The Neutron service provides network connectivity for the Nova instances. It creates a software-defined network (SDN) that allows tight control over communication between instances, as well as access from outside of the OpenStack network.

Authentication/Authorization (Keystone)

All of the OpenStack services require user access control, and the Keystone service provides the user management and resource control policy.

User Interface (Horizon)

The Horizon user interface is one of the more recent additions. It provides a single-pane Web UI to OpenStack as a whole. It layers a task-related view of OpenStack over the functional services, which allows end users and administrators to focus on their jobs without being concerned with...well, this list of agents.

Monitoring (Ceiliometer)
Another recent addition is the monitoring function provided by the Ceiliometer service. Ceiliometer is focused on monitoring and providing metrics for the OpenStack services themselves, but the documentation claims that it is designed to be extensible. This functionality would allow implementers to add probes and metrics to reach inside instances and applications as well.

Orchestration (Heat)
The Heat service gives the OpenStack user a standardized means to define complex configurations of compute, storage, and networking resources and to apply them repeatably. Although Heat deals mainly with managing the OpenStack resources, it has interfaces with several popular OS and application-level configuration management (CM) tools, including Puppet and Chef.

Heat uses templates to define reusable configurations. The user defines a configuration including compute, storage, and networking as well as providing input to any OS configuration that will be applied by a CM system.

Installation Tools
As mentioned, OpenStack is a complex service. However, it does follow several standard patterns. Additionally, several efforts are in progress to ease the installation process and to make installs consistent.

Both RPM and Debian-based Linux systems have installer efforts.

Puppet Modules
Each of the OpenStack services has a corresponding Puppet module to aid in installation and configuration. If you're familiar with both Puppet and OpenStack, you could probably use these to create a working service, but this method isn't recommended.

RDO—Packstack (RPM)
Packstack is intended for single host or small development or demonstration setups. It's produced by the RDO foundation, which is the community version of Red Hat's implementation of OpenStack.

Packstack can run either as an interactive session, or it can accept an answer file that defines all of the responses. It can install the component services on a single host or a small set of hosts.

RDO—The Foreman
The Foreman is primarily a hardware-provisioning tool, but it also has features to make use of Puppet to define the OS and application configuration of managed systems. The RDO project has defined and packaged an installer, which makes use of the Foreman host group definitions and the OpenStack Puppet modules to create a working installation. With the Foreman, it is possible to create more complex configurations by directly using the Puppet module inputs.

Ubuntu OpenStack Installer (Debian Package)
Canonical and Ubuntu also have an OpenStack installer effort. Canonical has taken a different approach from Red Hat. They provide a bootable disk image that can be written to a USB storage device or to a DVD. On first boot, the installer walks the user through the process.

Conclusion
The idea of IaaS is too powerful to ignore in the long term. Although it is still experiencing growing pains, so many players, large and small, are now backing and contributing resources to OpenStack that it's a safe bet it will be around for a while and will improve with time.

If you're eyeing a service like Amazon Web Services and thinking, "I wish I could do that here," then you should look at OpenStack. Plan to do several iterations of installation so that you can get to know the component services and their interactions as well as your real use cases. With commercial cloud service developers and project managers starting to get used to the idea of on-demand resources, they're going to be clamoring for it soon, and OpenStack offers you the means to provide it.

References
A lot of people are working on OpenStack, so there are a lot of references. These are just a few select ones to get started.

- OpenStack: http://www.openstack.org
- OpenStack documentation repository: http://docs.openstack.org
- OpenStack Administrator's Guide: http://docs.openstack.org/admin-guide-cloud/content/
- Packstack: https://wiki.openstack.org/wiki/Packstack

SOFTWARE
- GitHub OpenStack: https://github.com/openstack
- GitHub Packstack: https://github.com/stackforge/packstack
- GitHub OpenStack Puppet modules: https://github.com/stackforge/?query=puppet-

VENDOR AND DISTRIBUTION COMMUNITIES
- Red Hat RDO: http://openstack.redhat.com/Main_Page
- Ubuntu: http://www.ubuntu.com/download/cloud/install-ubuntu-cloud

SAVE THE DATE!

LISA '14

Sponsored by USENIX in cooperation with LOPSA

NOVEMBER 9–14, 2014
SEATTLE, WA

www.usenix.org/lisa14

USENIX's LISA conference is the premier meeting place for professionals who make computing work efficiently across a variety of industries. If you're an IT operations professional, site-reliability engineer, system administrator, architect, software engineer, researcher, or otherwise involved in ensuring that IT services are effectively delivered to others—this is your conference, and we'd love to have you here.

The sessions at LISA '14 will address the topics most relevant to those working in information technology today, including:

- Systems Engineering
- Monitoring and Metrics
- DevOps
- Security
- Culture
- And much more!

The 6-day program includes invited talks, workshops, panels, AMA conversations, training courses, and refereed paper and poster presentations. The on-site Hack Lab will give attendees and speakers the opportunity to demo, collaborate, and test out new ideas. Evening receptions and Birds-of-a-Feather sessions provide opportunities to meet and network with those that share your interests.

There's a better way to get your job done. Learn it at LISA.

Stay Connected...

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

When the Stars Align, and When They Don't

TIM HOCKIN

Tim Hockin attended Illinois State University to study fine art, but emerged with a BS in computer science instead. A fan of Linux since 1994, he joined Cobalt Networks after graduating and has surfed the Linux wave ever since. He joined Google's nascent Platforms group in 2004, where he created and lead the system bring-up team. Over time he has moved up the stack, most recently working as a tech lead (and sometimes manager) in Google's Cluster Management team, focusing mostly on node-side software. He is somewhat scared to say that his code runs on every single machine in Google's fleet. thockin@google.com

I got my first real bug assignment in 1999, at the beginning of my first job out of undergrad. Over a period of weeks, I went through iterations of code-reading, instrumentation, compiling, and testing, with the bug disappearing and reappearing. In the end, the problem turned out to be a subtle misalignment, something I would never have guessed when I started out.

Here's how I got started: A user reported that their database was being corrupted, but only on our platform. That's entirely possible—our platform was, for reasons that pre-dated my employment, somewhat odd. It was Linux—easy enough—but on MIPS. The MIPS port of Linux was fairly new at the time and did not have a huge number of users. To make life more fun, we used MIPS in little-endian mode, which almost nobody else did. It was supported by the toolchain and the kernel, so we all assumed it was workable. To find a bug on this rather unique platform was not surprising to me.

I took to the bug with gusto. Step one, confirm. I reproduced the user's test setup—a database and a simple HTML GUI. Sure enough, when I wrote a number, something like 3.1415, to the database through the GUI and then read it back through the GUI, the software produced a seemingly random number. Reloading the page gave me the same number. Problem reproduced.

With the arrogance of youth, I declared it a bug in the customer's GUI. I was so convinced of this that I wanted to point out exactly where the bug was and rub their noses in it. So, I spent a few hours instrumenting their HTML and Perl CGI code to print the values to a log file at each step of the process as it was committed to the database. To my great surprise, the value was correct all the way up until it was written to the database!

Clearly, then, it was a bug in the database. I downloaded source code for the database (hooray for open source!). I spent the better part of a day learning the code and finding the commit path. I rebuilt the database with my own instrumentation, and fired it up. Lo and behold, the bug was gone! The value I wrote came back perfectly. I verified that the code I was testing was the same version as in our product—it was. I verified that we did not apply any patches to it—we didn't.

Clearly, then, it was a bug in the toolchain. I spent the next week recompiling the database (to the tune of tens of minutes per compile-test cycle) with different compiler versions and flags, with no luck—everything I built worked fine. Someone suggested that maybe the Linux packaging tool (RPM) was invoking different compiler behavior than I was using manually. Great idea, I thought. I hacked up a version of the package that used my instrumented code and ran that. Bingo—the bug was back.

I began dissecting all of the differences I could figure out from the RPM build and my manual build. After a number of iterations over a number of days, I tracked it down to a single -D flag (a #define in the C code) to enable threading in the database. I was not setting this when I compiled manually, but the RPM build was.

Clearly, then, the bug was in the database's threading code. Suddenly it got a lot more daunting but also more fun. I spent even more time auditing and instrumenting the database code for threading bugs. I found no smoking gun, but iterative testing kept pushing the problem further and further down the stack. Eventually, it seemed that the database was innocent. It was writing through to the C library correctly and getting the data back corrupted.

Clearly, then, the bug was in the C library. Given the circumstances of our platform, this scenario was believable. I downloaded the source code for GNU libc and started instrumenting it. Compiling glibc was a matter of hours, not minutes, so I tried to be as surgical as I could about changes. But, every now and then, something went awry in the build and I had to "make clean" and start over—it was tedious. Several iterations later, libc was absolved of any wrongdoing—the data got all the way to the write() syscall correctly, but it was still incorrect when I read and printed it later.

Clearly, then, the bug was in the kernel. If I thought iterating the C library was painful, iterating the kernel was agony. Hack, compile, reboot, test, repeat. I instrumented all along the write() path all the way down to the disk buffers. Now several weeks into this bug, the kernel was free from blame. Maybe the data wasn't actually getting corrupted? Why this idea did not hit me before I blame on my initial overconfidence, which was much diminished by this point.

I took a new tack—could I reproduce it outside of the database test? I wrote a small program that wrote a double precision floating point value (the same as the test case) to a file, read it back, and printed it out. I linked the threading library, just like the database code, but the bug did not reproduce. That is, until I actually ran the test case in a different thread. Once I did that, the result I printed was similarly, but not identically, corrupted. I tried with non-float values, and the bug did not reproduce. I dredged my memory for details of the IEEE754 format and confirmed that the file on disk was correct—the corruption was happening in the read path, not the write path!

Given this fresh information, I instrumented the test program to print the raw bytes it had read from the file. The bytes matched the file. But when I printed the number, it was wrong. I was getting close!

Clearly, then, the bug was in printf(). I turned my attention back to the C library. I quickly found that the bytes of my float value were correct before I called printf() and were incorrect inside printf(). The printf() family of functions uses C variable-length argument lists (aka varargs)—this is implemented with compiler support to push and pop variables on the call stack. Because this process is effectively manual, printf() must trust you, the programmer, to tell it what type you pushed. It dawned on me that if I told printf() that I pushed a float when I really pushed an int, it would interpret the data that it popped incorrectly—leading to exactly this type of corruption. I verified that manually doing varargs of a double precision value corrupted the value.

To recap: the problem only occurs with (1) floating point values, (2) running in a thread, (3) with varargs.

Clearly, then, the bug was in the varargs implementation. I puzzled through the varargs code—a tangle of macros to align the memory access properly (16 bytes required for double), and I could find no flaw with the code. It followed the platform's spec, but the resulting value was still wrong. I looked at the disassembled code for pushing and popping the values, and it looked correct—the alignment was fine. In desperation, I instrumented the calling code to print the stack address as it pushed the argument and compared that with the address that varargs popped—they did not match. They were off by eight bytes!

After many hours of staring at disassembled code and cross-referencing the spec, it became clear—the logic used when pushing the value onto the stack was subtly different than the logic used to pop it off. Specifically, the push logic was using offsets relative to the frame pointer, but the pop logic used the absolute address. In the course of reading the ABI specification, I noticed that almost every value on the stack is required to be double-word (8 byte) aligned, the stack frame itself is required to be quad-word (16 byte) aligned. When I printed the frame pointer register, I found that it failed to meet this specification.

It seemed obvious: All I had to do was find the code that allocated the stack space and align it better.

I proceeded to take apart all of the LinuxThreads code related to stack setup. It allocated a block of memory for the thread stack, but the allocation was already aligned. Ah, but stacks grow down! The code placed an instance of a thread descriptor structure at the top of the stack region (base+size—sizeof(struct)) and initialized the stack to start below that. On a lark, I printed the size of that structure—it was a multiple of eight in size, but not a multiple of 16.

Gotcha!

I added a compiler directive to align the structure to 16 bytes, which has the side effect of making the size a multiple of 16, and compiled the C library one more time. The problem was gone, and the database's GUI now showed correct results. All of this work, and the fix was to add eight bytes of empty space to a structure. With a sigh, I dashed off a patch to the MIPS maintainers for GNU libc, which was met with a response to the effect of, "Oh! We've been hunting that one years." It felt like I had been, too.

Solving Rsyslog Performance Issues

DAVID LANG

David Lang is a site reliability engineer at Google. He spent more than a decade at Intuit working in the Security Department for the Banking Division. He was introduced to Linux in 1993 and has been making his living with Linux since 1996. He is an Amateur Extra Class Radio Operator and served on the communications staff of the Civil Air Patrol, California Wing, where his duties included managing the statewide digital wireless network. He was awarded the 2012 Chuck Yerkes award for his participation on various open source mailing lists. david@lang.hm

Rsyslog is a very fast modern logging daemon, but with its capabilities comes complexity: When things go wrong, it's sometimes hard to tell what the problem is and how to fix it. To continue my series of logging articles [1], this time I will be talking about how to troubleshoot performance issues with rsyslog.

Common Bottlenecks and Solutions

By far, the most common solution to poor performance is to upgrade to the current version of rsyslog. Performance is considered a key feature of rsyslog, and the improvements from one major version to another can be drastic. In many common rulesets, going from 5.x to 7.x has resulted in >10x performance improvements.

The next most common performance bottleneck is name resolution. Rsyslog will try to do a reverse lookup for the IP of any system sending it log messages. With v7, it will cache the results, but if your name lookups time out, this doesn't help much. If you cannot get a fast name server or put the names into /etc/hosts, consider disabling DNS lookups with the -x command line flag.

For people using dynamically generated file names, a very common problem is failing to increase the number of filehandles that rsyslog keeps open. Historically, rsyslog keeps only ten dynamically generated output files open per action. If you commonly have logs arriving for more than this small number, rsyslog needs to close a file (flushing pending writes), open a new file, and write to that file for each new log line that it's processing. To fix this, set the $DynaFileCacheSize parameter to some number larger than the number of files that you expect to write to (and make sure your filehandle limits allow this).

The last of the common bottlenecks is contention on the queues. If every thread accessing the queue locks it for every message, it's very possible for contention for the queue logs to become a significant bottleneck. In v5, rsyslog gained the ability to process batches of messages, and over time more modules have been getting updated to support this mode. The default batch size ($ActionQueueDequeueBatchSize) was initially 16 messages at a time; however, on dedicated, central servers, it may be appropriate to set this limit much higher. In v8, the default is being changed to 1024, and even more may be appropriate on a dedicated server. The benefit of increasing this number tapers off with size, but setting it to 1024 or so to see if there is a noticeable difference is a very reasonable early step. The discussion below will help you determine if you want to go further.

Rsyslog and Threads

Beyond these most common causes, things get more difficult. It is necessary to track down what is actually the bottleneck and address it. This task is complex because rsyslog makes heavy use of threads to decouple pieces from each other and to take advantage of modern systems, but this structure also provides some handles to use to track down the issues.

Each input to rsyslog is through one or more threads, which gather the log messages and add them to the main queue. Worker threads then pull messages off the main queue and deliver them to their destinations and/or add the message to an action queue. If there are action

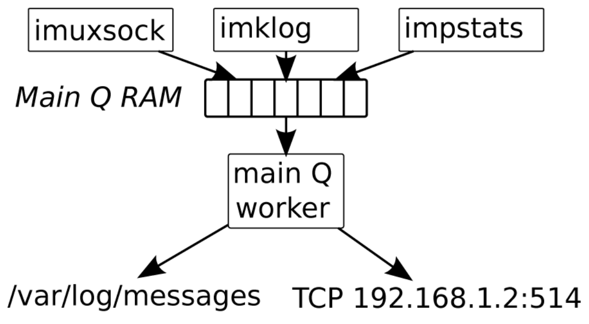


Figure 1: The flow of logs in a basic rsyslog configuration

queues, each one has its own set of worker threads pulling from the action queue and delivering to the destinations for that action.

If a worker is unable to deliver messages to a destination, all progress of that queue will block until that delivery is able to succeed (or it hits the retry limit and permanently fails). If you don't want this to block all log processing, you should make an action queue for that destination (or group of destinations).

As an example, I'll show a basic configuration that accepts logs from the kernel and from /dev/log, writes all the messages locally, and delivers them via TCP to a remote machine.

Legacy config:

```
$ModLoad imklog
$modLoad imuxsock
$SystemLogRateLimitInterval 0
$SystemLogSocketAnnotate on
*.* /var/log/messages
*.* @@192.168.1.1:514
```

Version 7 and later config:

```
module(load="imklog")
module(load="imuxsock" SysSock.RateLimit.Interval="0"
  SysSock.Annotate="on")
action(type="omfile" File="/var/log/messages")
action(type="omfwd" Target="192.168.2.11" Port="10514"
  Protocol="tcp")
```

This will create three threads in addition to the parent "housekeeping" thread. Figure 1 shows the data flow through rsyslog. The threads do not communicate directly with each other, and no one thread "owns" the queue. The housekeeping thread isn't shown here, because it doesn't have any role in the processing of log messages.

With top, you can see these threads by pressing H, and with ps, you can see these as well:

```
# ps  -eLl |grep `cat /var/run/rsyslogd.pid`
5 S 0 758  1 758  0  80   0 - 18365 poll_s ?   00:00:00 rsyslogd
1 S 0 758  1 763  0  80   0 - 18365 poll_s ?   00:00:05
  in:imuxsock
1 S 0 758  1 764  0  80   0 - 18365 syslog ?   00:00:00 in:imklog
1 S 0 758  1 765  0  80   0 - 18365 futex_ ?   00:00:02 rs:main
  Q:Reg
```

argv[0] is changed to tag each thread with what it's doing. This lets you see if any of the threads are pegging the CPU, or if the main Q worker thread is just doing nothing.

This is, of course, a "Hello World" configuration, but it shows some of the types of issues that are common for people to run into in larger setups. For example, this configuration is dependent on a remote system; if that system is down, no local logs will be processed and we will see logs queue up and eventually fill the queue, causing programs trying to write logs to block. If we want to continue logging locally, even if the remote system is down, we can create a default-sized action queue for the TCP output action. To handle longer outages, or restarts of rsyslog, the queue can be disk backed. A full discussion of queue configuration and management is a topic that will require its own article.

Legacy config:

```
$ModLoad imklog
$modLoad imuxsock
$SystemLogRateLimitInterval 0
$SystemLogSocketAnnotate on
*.* /var/log/messages
$ActionQueueType FixedArray
*.* @@192.168.1.1:514
```

Version 7 config:

```
module(load="imklog")
module(load="imuxsock" SysSock.RateLimit.Interval="0"
  SysSock.Annotate="on")
action(type="omfile" File="/var/log/messages")
action(type="omfwd" Target="192.168.2.11" Port="514"
  Protocol="tcp" queue.type="FixedArray")
```

Now, you can see an additional queue worker for the action queue:

```
# ps  -eLl |grep `cat /var/run/rsyslogd.pid`
   5 S 0 458  1 458  0  80   0 - 20414 poll_s ?  00:00:00 rsyslogd
   1 S 0 458  1 462  0  80   0 - 20414 poll_s ?  00:00:00 in:imuxsock
   1 S 0 458  1 463  0  80   0 - 20414 syslog ?  00:00:00 in:imklog
   5 S 0 458  1 464  0  80   0 - 20414 futex_ ?  00:00:00
     rs:main Q:Reg
   1 S 0 458  1 465  0  80   0 - 20414 futex_ ?  00:00:00
     rs:action  2 que
```

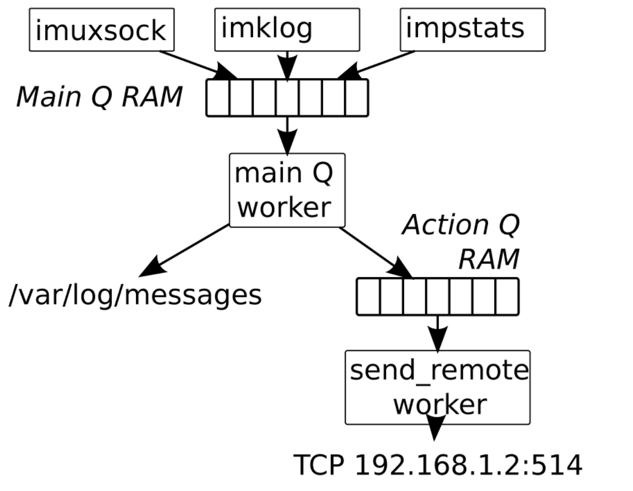


Figure 2: The flow of logs when an action queue is added to preventing blocking

With v7+, you can add a parameter to name the queue:

```
action(name="send_remote" type="omfwd" Target="192.168.2.11"
  Port="514"
  Protocol="tcp" queue.type="FixedArray" )
```

This changes the data flow to what you see in Figure 2, and the ps/top display shows the additional threads as well:

```
# ps  -eLl |grep `cat /var/run/rsyslogd.pid`
5 S 0 971  2807 971  0  80   0 - 20414 poll_s ?  00:00:00
  rsyslogd
1 S 0 971  2807 972  0  80   0 - 20414 poll_s ?  00:00:00
  in:imuxsock
1 S 0 971  2807 973  0  80   0 - 20414 syslog ?  00:00:00
  in:imklog
5 S 0 971  2807 974  0  80   0 - 20414 futex_ ?  00:00:00
  rs:main Q:Reg
1 S 0 971  2807 975  0  80   0 - 20414 futex_ ?  00:00:00
  rs:send_remote:
```

Keeping track of all the pieces can be a bit difficult when you have multiple queues in use.

Through the rest of the article, I will just give the v7 format because of the increasing complexity of specifying options with the legacy config style. Not all of the features described are going to be available on older versions.

Although ps/top lets you see how much CPU is being used by the threads, it doesn't tell you what is being done. Rsyslog includes the impstats module, which produces a lot of information about what's going on inside rsyslog.

To load impstats, add the following line to the top of your config file (it needs to be ahead of many other configuration parameters, so it's easiest to make it the very first line).

```
module(load="impstats" interval="60" resetCounters="on"
   format="legacy")
```

This statement will create a set of outputs every minute, resetting counters every time, which makes it very easy to see if a queue is backing up. A sample output looks like the following (timestamp and hostname trimmed for space):

```
rsyslogd-pstats: imuxsock: submitted=3 ratelimit.discarded=0
  ratelimit.numratelimiters=2

rsyslogd-pstats: action 1: processed=4 failed=0
  suspended=0 suspended.duration=0 resumed=0

rsyslogd-pstats: send_remote: processed=4 failed=0
  suspended=0 suspended.duration=0 resumed=0

rsyslogd-pstats: resource-usage: utime=1536 stime=60107
  maxrss=1280 minflt=386 majflt=0
  inblock=0 oublock=0 nvcsw=22 nivcsw=32

rsyslogd-pstats: send_remote: size=0 enqueued=4 full=0
  discarded.full=0 discarded.nf=0

maxqsize=1
rsyslogd-pstats: main Q: size=5 enqueued=9 full=0
  discarded.full=0
    discarded.nf=0 maxqsize=5
```

You can use an automated analyzer to find the most common types of problems. Upload your pstats logs to http://www.rsyslog.com/impstats-analyzer/, and the script will highlight several common types of problems.

If you are using JSON-formatted messages, you can change format from "legacy" to "cee" and then use the mmjsonparse module to break this down into individual variables for analysis. In this format, the logs look like:

```
rsyslogd-pstats: @cee: {"name":"imuxsock","submitted":7,
  "ratelimit.discarded":0,"ratelimit.numratelimiters":3}

rsyslogd-pstats: @cee: {"name":"action 1","processed":8,
  "failed":0,"suspended":0, "suspended.duration":0,
  "resumed":0}

rsyslogd-pstats: @cee: {"name":"action 2","processed":8,
  "failed":0,"suspended":0, "suspended.duration":0,
  "resumed":0}

rsyslogd-pstats: @cee: {"name":"send_remote","processed":8,"
  failed":0, "suspended":0, "suspended.duration":0,
  "resumed":0}
```

```
rsyslogd-pstats: @cee: {"name":"resource-usage","utime":2917,
  "stime":3237,"maxrss":1520, "minflt":406,"majflt":0,"inblock":0,
  "oublock":0,"nvcsw":30,"nivcsw":6}

rsyslogd-pstats: @cee: {"name":"send_remote","size":0,
   "enqueued":8,
    "full":0,"discarded.full":0, "discarded.nf":0,"maxqsize":1}

rsyslogd-pstats: @cee: {"name":"main Q","size":6,"enqueued":14,
  "full":0, "discarded.full":0,"discarded.nf":0,"maxqsize":6}
```

So far, you've just put the pstats logs into the main queue along with all the other logs in the system. If something causes that queue to back up, it will delay the pstats logs as well.

There are two ways to address this: First, you can configure rsyslog to write the pstat logs to a local file in addition by adding the log.file="/path/to/local/file" parameter to the module load line. The second approach is a bit more complicated, but it serves to show an example of another feature of rsyslog—rulesets and the ability to bind a ruleset to a specific input.

Take the example config and extend it to be the following:

```
module(load="impstats" interval="10" resetCounters="on"
  format="legacy" ruleset="high_p")
module(load="imklog")
module(load="imuxsock" SysSock.RateLimit.Interval="0"
  SysSock.Annotate="on")
action(type="omfile" File="/var/log/messages")
action(name="send_remote" type="omfwd" Target="192.168.2.11"
  Port="514" Protocol="tcp"queue.type="FixedArray" )

ruleset(name="high_p" queue.type="FixedArray"){
  action(type="omfile" File="/var/log/pstats")
  action(name="send_HP" type="omfwd" Target="192.168.2.11"
    Port="514"
      Protocol="tcp" queue.type="FixedArray" )
}
```

All pstat log entries will now go into a separate "main queue" named "high_p" with its own worker thread and its own separate queue to send the messages remotely. This is effectively the same as starting another stand-alone instance of rsyslog just to process these messages. There is no interaction (other than the house-keeping thread) between the threads processing the pstat messages and the threads processing other messages (see Figure 3).

```
# ps  -eLlww lgrep `cat /var/run/rsyslogd.pid`
5 S 0 827  2807 827  0  80 0 - 31181 poll_s ?  00:00:00 rsyslogd
5 S 0 827  2807 828  0  80 0 - 31181 poll_s ?  00:00:00
  in:impstats
1 S 0 827  2807 829  0  80 0 - 31181 syslog ?  00:00:00 in:imklog
1 S 0 827  2807 830  0  80 0 - 31181 poll_s ?  00:00:00
  in:imuxsock
```

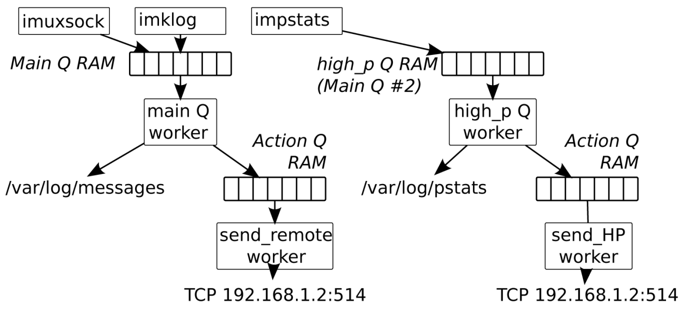


Figure 3: The flow of logs through the threads and queues with impstats bound to a ruleset

```
5 S 0 827  2807 831  0  80 0 - 31181 futex_ ?  00:00:00
  rs:main  Q:Reg
1 S 0 827  2807 832  0  80 0 - 31181 futex_ ?  00:00:00
  rs:send_remote:
5 S 0 827  2807 843  0  80 0 - 31181 futex_ ?  00:00:00
  rs:high_p:Reg
1 S 0 827  2807 844  0  80 0 - 31181 futex_ ?  00:00:00
  rs:send_  HP:Reg
```

Pstats output also shows the additional queues:

```
rsyslogd-pstats: imuxsock: submitted=0 ratelimit.discarded=0
  ratelimit.numratelimiters=0

rsyslogd-pstats: action 1: processed=0 failed=0 suspended=0
  suspended.duration=0 resumed=0

rsyslogd-pstats: send_remote: processed=0 failed=0
  suspended=0 suspended.duration=600 resumed=0

rsyslogd-pstats: action 3: processed=10 failed=0 suspended=0
  suspended.duration=0 resumed=0

rsyslogd-pstats: send_HP: processed=10 failed=0 suspended=0
  suspended.duration=600 resumed=0

rsyslogd-pstats: resource-usage: utime=26978 stime=26416
  maxrss=2056 minflt=857 majflt=0 inblock=0 oublock=400
  nvcsw=412 nivcsw=11

rsyslogd-pstats: send_remote: size=0 enqueued=0 full=0
  discarded.full=0 discarded.nf=0 maxqsize=2

rsyslogd-pstats: send_HP: size=0 enqueued=10 full=0 discarded.
  full=0 discarded.nf=0 maxqsize=10

rsyslogd-pstats: high_p: size=8 enqueued=10 full=0 discarded.
  full=0 discarded.nf=0 maxqsize=10

rsyslogd-pstats: main Q: size=0 enqueued=0 full=0 discarded.
  full=0 discarded.nf=0 maxqsize=2
```

Once you find the actual bottleneck in your configuration, what can you do about it? It boils down to a two-pronged attack.

Use More Cores to Perform the Work

This approach is tricky. In many cases enabling more worker threads will help, but you will need to check the documentation for the module that is your bottleneck to find what options you have. In many cases, the modules end up needing to serialize (e.g., you only want one thread writing to a file at a time), and if your bottleneck is in one of these areas, just adding more workers won't help. Writing to a file can be split up by moving most of the work away from the worker thread that is doing the testing of conditions, using a second thread to format the lines for the file, and, if needed, using a third thread to write the data to the file, potentially compressing it in the process.

Restructure the Configuration to Reduce the Amount of Work

This approach involves standard simplification and refactoring work for the most part. Rsyslog has lots of flexibility in terms of what modules can be written to do; so, in an extreme case, a custom module may end up being written to address a problem. In most cases, however, it's simplifying regex expressions, refactoring to reduce the number of tests needed or to make it easier for the config optimizer to detect the patterns. Like most programming, algorithmic changes usually produce gains that dwarf other optimization work, so it's worth spending time looking at ways to restructure your configuration.

Some Examples of Restructuring

If you have several actions that are related to one destination, instead of creating a separate queue for each action, you can create a ruleset containing all the actions, and then call the ruleset with a queue.

```
ruleset(name="rulesetname" queue/type="FixedArray"){
  action(type="omfwd" Target="192.168.2.11" Port="514"
    Protocol="tcp")
}
```

Then, in the main ruleset, you can replace the existing actions with:

```
call rulesetname
```

A ruleset can contain any tests and actions that you can have in a normal rsyslog ruleset, including calls to other rulesets.

With the v7 config optimizer, zero overhead is incurred in using a ruleset that doesn't have a queue, so you can also use rulesets to clarify and simplify your rulesets. If you find that you have a lot of format rules:

```
if $hostname == "host1" and $programname = "apache" then {
  /var/log/apache/host1.log
  stop
}
if $hostname == "host1" and $programname = "apache" then {
  /var/log/postfix/host2.log
  stop
}
if $hostname == "host2" and $programname = "postfix" then {
  /var/log/apache/host1.log
  stop
}
if $hostname == "host2" and $programname = "postfix" then {
  /var/log/postfix/host2.log
  stop
}
/var/log/other-logs
```

you can simplify it into:

```
$template multi_test='/var/log/%programname%/%hostname%.log'
ruleset(name="inner_test"){
  if $programname == "apache" then {
    ?multi_test
    stop
  }
  if $programname == "postfix" then {
    ?multi_test
    stop
  }
}
if $hostname = "host1" then call inner_test
if $hostname = "host2" then call inner_test
/var/log/other-logs
```

This change defines a template to be used for the file name to be written to (the Dynafile capability mentioned earlier), then defines a ruleset to use—similar to a subroutine that checks that the program name is one of the known ones. If so, it writes the log out to a file whose name is defined by the template and stops processing the log message. Then, finally, you go through a list of hosts. If the host is known, you call the ruleset subroutine to check the program name. If either the hostname or program name is not in your list of known entities, then the tests will not match and the stop action will never be reached, resulting in the log entry being put into the /var/log/other-logs file.

This specific case can be simplified further by using the rsyslog array match capability. This approach requires that the entries to be matched in the array be sorted, but it can further reduce the configuration size and speed up the processing.

```
$template multi_test='/var/log/%programname%/%hostname%.log'
ruleset(name="inner_test"){
  if $programname == ["apache", "postfix"] then {
    ?multi_test
    stop
  }
}
if $hostname = ["host1","host2"] then call inner_test
/var/log/other-logs
```

When troubleshooting performance problems with rsyslog, usually the biggest problem is finding where the bottleneck is; once the bottleneck is found, it's usually not that complicated to remove it. Something can always be done. In extreme conditions, it's even possible to use a custom module to do extensive string processing that is expensive to do in the config language. You can always ask for help on the rsyslog-users mailing list at rsyslog@lists.adiscon.com; the volunteers there are always interested in new problems to solve.

Reference
[1] Links to previous articles by David Lang about logging: https://www.usenix.org/login/david-lang-series.

/var/log/manager
Rock Stars and Shift Schedules

ANDY SEELY

Andy Seely is the manager of an IT engineering division, customer-site chief engineer, and a computer science instructor for the University of Maryland University College. His wife Heather is his init process, and his sons Marek and Ivo are always on the run queue.
andy@yankeetown.com

The stress level in a dynamic work place can be very high, even higher in a startup where every day may be the beginning or the end for the company. These types of businesses tend to hire the best and the brightest in the industry and encourage employees to give their all and more. Those same employees can be made more effective, and less susceptible to burnout, through some simple organizational tools that are obvious in hindsight but not always the first choice in the heat of the moment. A basic shift schedule can help guide effort, shepherd resources, and ultimately lead to a happier and more sustainable workforce.

I joined a company as the manager of a group of junior and mid-level system administrators. The tier-one operations team was responsible for around-the-clock monitoring and first-line response for the company's infrastructure and public-facing technology. This company was a dot-com startup that prided itself on the high quality of its workforce and the dot-com-style perks offered to them. Although we didn't actually say in our job advertisements that we only hired "rock stars," that was how we viewed ourselves. A month into the job, I started getting a creeping feeling that the surface was a rock-star party but that, underneath, there were currents preventing us from achieving our true peak of productivity. I discovered that our organization was being held captive by our own brilliance.

Brilliant People, Great Work, Bright Future

I was new to "dot-com," so lots of things were exciting and wonderful to me. We had a "nap room" where a tired sysadmin could…take a nap. We had a fully stocked kitchen, which soon started having Dr. Pepper in the rotation after I mentioned I liked Dr. Pepper. This company didn't have an actual game room like many others in the area, but we made up for the lack with plenty of desk- and cube-level games and toys.

The people around me were some of the sharpest I've ever known. I've always joked that if I find myself being the smartest person in a room, then I need a new room. This team was that new room. They were innovators, inventors, and idea people, everyone simply brimming with self-confidence and assurance. The team also took a lot of pride in how much and how hard they worked to make the company successful.

The company had solid funding and was perpetually on an upward swing in the market. The CEO had an impeccable dot-com pedigree and was always optimistic about the great places he was taking the business. Staff meetings featured talk of big names, big media, big goals, and lots of work to be done. It was intoxicating.

Everything was perfect.

Burning Bright or Just Burning Out?

My first hint of an underlying issue was the length of a "normal" workday. Most of the operations team—both tier-one and the more senior tier-two team—would be on the job at the office until seven or eight in the evening and then be online, working from home, sometimes until midnight and later. There was a pervasive sense that the work never ended, and we were being very, very productive, but… the work never ended. Then, I realized that despite putting in an average 12-hour workday, people were only really doing effective work from about noon until about seven. The rest of the time was spent in what could be called "workplace activities," which included frequent and long conversations about the long hours we were working.

The long workday issue was exacerbated by a recurring phenomenon where the response to a system problem would include several unnecessary people jumping in to help, and then those same people would be fatigued and less effective the next day. We didn't respond to system problems conservatively, but rather with enthusiasm and brilliance. After observing several such events, I learned that qualities like enthusiasm and brilliance can be consumables when overtaxed, and that junior people can feel untrusted and disenfranchised when senior people always take over.

After I'd been on the team a few months, people started warming up to me, and I started gaining more insight on the undercurrents of the workplace. The night-shift guy worked from home and got very little in-person engagement with the team, and he wondered if anyone was ever going to swap shifts with him so he could be more involved. A particular day-shift guy always jumped first to take all the hard problems, and over time started wondering why he always had to be the one to work all the hard problems. People wanted to take off for family events or vacations, but they wouldn't plan farther ahead than a couple days because long-term plans were always trumped by something operationally important. More than one team member took vacation time to visit family and ended up spending long hours online in a spare bedroom. When my own first child was being born, I was able to "take the rest of the afternoon off," which my wife and I still laugh about.

The team made mistakes, which were always opportunities to excel at fixing hard problems. We argued and fought when it got late and we got tired and it felt like no one was really in control of a big problem. We used ourselves up and compensated with willpower and pride, because we were awesome. We had to be awesome, just to survive.

Trying Some Old-Fashioned Discipline and Structure

Our "schedule" was pretty simple. Tier-one sysadmins roughly covered Monday through Friday, about 18 hours of each day. The gaps were filled in by tier-two staff being on call, with a tight rotation of a week spent on call every three or four weeks. This meant that a tier-two admin would work a normal week in the tier-two project space, plus be guaranteed to be paged by the system at least once a day after hours and sometimes dozens of times around the clock on the weekend. Serious technical or public-facing problems would result in the whole team becoming involved, which might be weekly. Every week, some event would overtax a tier-two sysadmin and leave that person less effective at anything but reactive break-fix for a day or two.

We talked and talked about our inefficiencies and how we couldn't sustain the pace. Finally, we were able to add capacity to the tier-one team. The act of growing the team gave us a moment to take a step back and rethink. I took the opportunity to do something previously unthinkable: I published an actual shift calendar.

Rock stars don't need shift schedules because they're always awesome, and awesome people don't clock in for an eight-hour shift. A few people weren't interested but got talked into playing along; others were surprisingly easy to convince, and some just had to be told to get in line. We worked together, and everyone had ownership. We worked out standard 40-hour shifts, day-swing mid-rotations, and came up with an incentive idea to have the tier-one team cover primary on call during the weekends. This didn't prevent tier two from being on call, but it did prevent tier two from being woken up constantly for tier-one-level system issues. People started talking about—and taking— vacation time when they didn't feel obligated to constantly check in. I knew we had made a cultural change when people started horse-trading shifts so they could plan their lives months in advance.

More Reliability, Less Romance

The biggest impact of the tier-one shift schedule turned out to be the increased stability of the tier-two staff. We reduced the number of after-hours and weekend calls dramatically, which resulted in stable and focused senior sysadmins who made fewer operational mistakes and were able to contribute more effectively to projects. The tier-one crew—the more junior people on the team—gained a sense of empowerment as they handled more events by themselves without the entire operations team swooping in.

Yes, it's true, those of us on "the schedule" felt less like "rock stars" and more like "employees," but we felt more confident and we were able to do better work as a result. The real impact was that everyone worked fewer clock hours and yet managed to be more effective.

It wasn't easy to change corporate culture, and it wasn't always obvious that it was going to work out. In the end, the risk paid off. Understanding how work happens and aligning people in a way that gives them the opportunity to do what they do best and create real efficiencies in the workplace can be a true challenge, especially when that realignment is counter to established norms and identities. I'm the manager, and that's my job.

COLUMNS

Practical Perl Tools

My Hero, Zero (Part 1)

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

In the last column, I spent some time exploring MongoDB, a database that challenges in some respects what it means to be a database. For this column, I'd like to take a look at a message queuing system that does the same for message queuing systems: ZeroMQ (written most often as 0MQ. Even though I'm not that hip, I'll use that representation most of the time below).

If the term "message queuing system" makes you feel all stifle-a-yawn enterprise-y, boring business service bus-ish, get-off-my-lawn-you-kids, we were doing that in the '80s-like, then I would recommend taking another look at how serious systems are getting built these days. If you are like me, you will notice again and again places in which tools are adopting message-bus architectures where you might not expect them. These architectures turn out to be an excellent way to handle the new reality of distributed systems, such as those you might find when you've launched into your favorite cloud provider. This is why message queuing systems are at the heart of packages like MCollective and Sensu. They often allow you to build loosely coupled and dynamic systems more easily than some traditional models.

Our friend Wikipedia talks about message queues as "software-engineering components used for interprocess communication, or for inter-thread communication within the same process.... Message queues provide an asynchronous communications protocol, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time." Message queuing systems like ActiveMQ and RabbitMQ let you set up message broker servers so that clients can receive or exchange messages.

Now, back to the MongoDB comparison: Despite having MQ at the end of the name like ActiveMQ and RabbitMQ, ZeroMQ is a very different animal from the other MQs. I don't think I can do a better job setting up how it is different than by quoting the beginning of the official 0MQ Guide:

> ØMQ (also known as ZeroMQ, 0MQ, or zmq) looks like an embeddable networking library but acts like a concurrency framework. It gives you sockets that carry atomic messages across various transports like in-process, inter-process, TCP, and multicast. You can connect sockets N-to-N with patterns like fan-out, pub-sub, task distribution, and request-reply. It's fast enough to be the fabric for clustered products. Its asynchronous I/O model gives you scalable multicore applications, built as asynchronous message-processing tasks. It has a score of language APIs and runs on most operating systems. ØMQ is from iMatix and is LGPLv3 open source.

Let me emphasize a key part of the paragraph above. With 0MQ, you don't set up a distinct 0MQ message broker server like you might with a traditional MQ system. There's no zeromq binary, there's no /etc/init.d/zeromq, no /etc/zeromq for config files, no ZeroMQ Windows service to launch (or whatever else you think about when bringing up a server). Instead, you use the 0MQ libraries to add magic to your programs. This magic makes building your own message-passing architecture (whether it's hub and spoke, mesh, pipeline, etc.) a lot easier than you might expect. It takes a lot of the pain out of writing clients, servers, peers, and so on. I'll show exactly what this means in a moment.

I want to note one more thing before actually diving into the code. ZeroMQ looks really basic at first, probably because the network socket model *is* pretty basic. But, like anyone who has built something with a larger-than-usual construction set, such as a ton of Tinkertoys, Legos, or maybe a huge erector set (if you are as old as dirt), at a certain point you step back from a creation that has somehow grown taller than you are and say "whoa." I know I had this experience reading the ZeroMQ book (disclosure: published by O'Reilly, who also published my book). In this book, which I highly recommend if ZeroMQ interests you at all, it describes a ton of different patterns that are essentially building blocks. At some point, you'll have that "whoa" moment when you realize that these building blocks offer everything you need to construct the most elaborate or elegant architecture your heart desires. Given the length of this column, I'll only be able to look at the simplest of topologies, but I hope you'll get a hint of what's possible.

Socket to Me

Okay, that's probably the single most painful heading I've written for this column. Let's pretend it didn't happen.

Everything 0MQ-based starts with the basic network socket model, so I'll attempt a three-sentence review (at least the parts that are relevant to 0MQ). Sockets are like phone calls (and indeed the combination of an IP address and a port attempts to provide a unique phone number). To receive a connection, you create a socket and then bind to it to listen for connections (I'm leaving out accept() and a whole Stevens book, but bear with me); to make a connection, you connect() to a socket already set to receive connections. Receiving data from an incoming connection is done via a recv()-like call; sending data to the other end is performed with a send()-like call.

Those three sentences provide the key info you need to know to get started with 0MQ socket programming. Like one of those late-night commercials for medications, I feel I should tag on a whole bunch of disclaimers, modifiers, and other small print left out of the commercial, but I'm going to refrain except for these two things:

- There are lots of nitty-gritty details to (good) socket programming I'm not even going to touch. 0MQ handles a bunch of that for you, but if you aren't going to use 0MQ for some reason, be sure to check out both the non-Perl references (e.g., the Stevens books) and the Perl-related references (Lincoln Stein wrote a great book called *Network Programming with Perl* many eons ago).
- If you do want to do plain ol' socket programming in Perl, you'll want to use one of the socket-related Perl modules to make the job easier. Even though Socket.pm ships with Perl, I think you'll find IO::Socket more pleasant to use. With these modules, you can open up a socket that connects to another host easily and print() to it as if it were any other file handle. Again, this is just a "by the way" sort of thing since we're about to strap on a rocket motor and use 0MQ sockets from Perl.

To get started with 0MQ in Perl, you'll need to pick a Perl module that provides an interface to the 0MQ libraries. At the moment, there are two choices for modules worth considering, plus or minus one: ZMQ::LibZMQ3 and ZMQ::FFI. The former is the successor to what used to be called just ZeroMQ. The author of that module decided to create modules specifically targeted to specific major versions of the 0MQ libraries (v2 and v3). It offers an API that is very close to the native library (as a quick aside, the author also provides a ZMQ module which will call ZMQ::LibZMQ3 or ZMQ::LibZMQ2, but the author suggests in most cases to use the version-specific library directly, hence my statement of "plus or minus one," above). The ZMQ::FFI uses libffi to provide a slightly more abstract interface to 0MQ that isn't as 0MQ-version specific. (In case you are curious about FFI, its docs say, "FFI stands for Foreign Function Interface. A foreign function interface is the popular name for the interface that allows code written in one language to call code written in another language.")

In this column, I'll use ZMQ::LibZMQ3 because it allows me to write code that looks like the C sample code provided in the 0MQ user guide (and in the 0MQ book). For the sample code in this part of a two-part column, I'll keep things dull and create a simple echo client-server pair. The server will listen for incoming connections and echo back any messages the connected clients sent to it. First, I'll look at the server code, because it's going to give me a whole bunch of things to talk about:

```
use ZMQ::LibZMQ3;
use ZMQ::Constants qw(ZMQ_REP);

my $ctxt = zmq_init;
my $socket = zmq_socket( $ctxt, ZMQ_REP );

my $rv = zmq_bind( $socket, "tcp://127.0.0.1:8888" );

while (1) {
  my $msg = zmq_recvmsg($socket);
  print "Server received:" . zmq_msg_data($msg) . "\n";
  my $msg = zmq_msg_init_data( zmq_msg_data($msg) );
  zmq_sendmsg( $socket, $msg );
}
```

The first thing to note about this code is that loads both the 0MQ library module and a separate constants module. Depending on how you installed the library module, the constants module likely was installed at the same time for you. This module holds the definition for all of the constants and flags you'll be using with 0MQ. There are a bunch—you'll see the first one in just a couple of lines. The next line of code initializes the 0MQ environment, something you have to do before you use any other 0MQ calls. 0MQ contexts usually don't have to come into play except as a background thing unless you are doing some fairly complex programming, so I'm not going to say more about them here.

With all of that out of the way, it is time to create your first socket. Sockets get created in a context and are of a specific type (in this case ZMQ_REP, or just REP). Socket types are a very important concept in 0MQ, so I'll take a quick moment away from the code to discuss them.

I don't know the last time you played with Tinkertoys but you might recall that they consisted of a few basic connector shapes (square, circle, etc.), which accepted rods at specific angles. If you wanted to build a cube, you had to use the connectors that had holes at 90-degree angles and so on. With 0MQ, specific socket types get used for creating certain sorts of architectures. You can mix and match to a certain extent, but some combinations are more frequently used or more functional than others.

For example, in the code I am describing, I will use REQ and REP sockets (REQ for synchronously sending a REQuest, REP for synchronously providing a REPly). You will see a similar pair in an example in the next issue. You might be curious what the difference is between a REQ and REP socket because as Dr. Seuss might say, "a socket's a socket, no matter how small." The short answer is the different socket types do slightly different things around message handling (e.g., how message frames are constructed, etc.). See the 0MQ book for more details.

Now that you have a socket constructed, you can tell it to listen for connections, which is done by performing a zmq_bind(). Here you can see that I have asked to listen to unicast TCP/IP connections on port 8888 of the local host. 0MQ also knows how to handle multicast, inter-process, and inter-thread connections. At this point, you are ready to go into a loop that will listen for messages. The zmq_recvmsg() blocks until it receives an incoming message. It returns a message object, the contents of which are displayed using zmq_msg_data($msg). To reply to the message you just received, you construct a message object with the contents of the message you received and send it back over the socket via zmq_sendmsg($socket, $msg). That's all you need to construct a simple server; now, I'll look at the client code:

```
use ZMQ::LibZMQ3;
use ZMQ::Constants qw(ZMQ_REQ);

my $ctxt = zmq_init;
my $socket = zmq_socket( $ctxt, ZMQ_REQ );

zmq_connect( $socket, "tcp://127.0.0.1:8888" );

my $counter = 1;

print "I am $$\n";

while (1) {
  my $msg = zmq_msg_init_data("$counter:$$");
  zmq_sendmsg( $socket, $msg );
  print "Client sent message " . $counter++ . "\n";
  my $msg = zmq_recvmsg($socket);
  print "Client received ack:" . zmq_msg_data($msg) . "\n";
  sleep 1;
}
```

The client code starts out almost identically (note: I said almost, the socket type is different. I once spent a very frustrating hour trying to debug a 0MQ program because I had written ZMQ_REP instead of ZMQ_REQ in one place in the code). It starts to diverge from the server code because, instead of listening for a connection, it is set to initiate one by using zmq_connect(). In the loop, you construct a simple message (the message number plus the PID of the script that is running) and send it on the socket. Once you've sent the message, you wait for a response back from the server using the same exact code the server used to receive a message. REQ-REP sockets are engineered with the expectation that a request is made and a reply is returned, so you really do want to send a reply back. Finally, I should mention there is a sleep statement at the end of this loop just to keep things from scrolling by too quickly (0MQ is FAST), but you can feel free to take it out.

Let's take the code for a spin. If you start up the server, it sits and waits for connections:

```
$ ./zmqserv1.pl
```

In a separate window, start the client:

Immediately, the client prints something like:

```
I am 86989
Client sent message 1
Client received ack:1:86989
Client sent message 2
Client received ack:2:86989
Client sent message 3
Client received ack:3:86989
…
```

and the server also shows this:

```
Server received:1:86989
Server received:2:86989
Server received:3:86989
…
```

Now, you can begin to see what all of the fuss is about with 0MQ. First, if you stop the server and then start it again (while leaving the client running):

```
$ ./zmqserv1.pl
Server received:1:87110
Server received:2:87110
Server received:3:87110
^C
$ ./zmqserv1.pl
Server received:4:87110
Server received:5:87110
Server received:6:87110
Server received:7:87110
^C
$ ./zmqserv1.pl
Server received:8:87110
Server received:9:87110
Server received:10:87110
^C
```

During all of this, the client just said:

```
$ ./zmqcli1.pl
I am 87110 Client sent message 1
Client received ack:1:87110
Client sent message 2
Client received ack:2:87110
Client sent message 3
Client received ack:3:87110
Client sent message 4
Client received ack:4:87110
Client sent message 5
Client received ack:5:87110
Client sent message 6
Client received ack:6:87110
Client sent message 7
Client received ack:7:87110
Client sent message 8
Client received ack:8:87110
Client sent message 9
Client received ack:9:87110
Client sent message 10
Client received ack:10:87110
```

Here you are seeing 0MQ's sockets auto-reconnect (and do a little bit of buffering). This isn't the only magic going on behind the scenes, but it is the easiest to demonstrate.

Now, I'll make the topology a little more interesting. Suppose you want to have a single server with multiple clients connected to it at the same time. Here's the change you'd have to make to the server code:

(nada)

And the change to the client code:

(also nada)

All you have to do is start the server and spin up as many copies of the client as you desire. Let's start up five clients at once:

```
$ for ((i=0;i<5;i++)); do ./zmqcli1.pl & done
```

On the client side, you see a mishmash of the outputs from the client (that eventually return to lockstep):

```
I am 87242
I am 87241
Client sent message 1 I am 87239
Client sent message 1 I am 87243
Client sent message 1
Client received ack:1:87241
I am 87240
Client received ack:1:87243
Client sent message 1 Client sent message 1
Client received ack:1:87239
Client received ack:1:87242
Client received ack:1:87240
Client sent message 2
Client sent message 2
Client sent message 2
Client sent message 2
Client sent message 2
Client received ack:2:87239
Client received ack:2:87241
Client received ack:2:87243
Client received ack:2:87240
Client received ack:2:87242
Client sent message 3
Client sent message 3
Client sent message 3
Client sent message 3
Client sent message 3
Client received ack:3:87241
Client received ack:3:87239
Client received ack:3:87240
Client received ack:3:87243
Client received ack:3:87242
```

On the server side, the output is a little more orderly:

```
$ ./zmqserv1.pl
Server received:1:87241
Server received:1:87239
Server received:1:87243
Server received:1:87242
Server received:1:87240
Server received:2:87239
Server received:2:87241
Server received:2:87243
Server received:2:87242
Server received:2:87240
Server received:3:87241
Server received:3:87239
Server received:3:87240
Server received:3:87243
Server received:3:87242
```

It is pretty cool that the server was able to handle multiple simultaneous connections without any code changes, but one thing that may not be clear is that 0MQ is not only handling these connections, it's also automatically load-balancing between them as well. That's the sort of behind-the-scenes magic I think is really awesome.

I believe it is always good to end a show after a good magic trick, so I'll wind up part 1 of this column right here. Join me next time when I will look at other ZeroMQ socket types and build some even cooler network topologies.

Take care, and I'll see you next time.

SAVE THE DATE!

OSDI | 14
11th USENIX Symposium on Operating Systems Design and Implementation
October 6–8, 2014
Broomfield, CO

Join us in Broomfield, CO, October 6–8, 2014, for the **11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14).** The Symposium brings together professionals from academic and industrial backgrounds in what has become a premier forum for discussing the design, implementation, and implications of systems software.

Don't miss the co-located workshops on Sunday, October 5

Diversity '14: 2014 Workshop on Supporting Diversity in Systems Research

HotDep '14: 10th Workshop on Hot Topics in Dependable Systems

HotPower '14: 6th Workshop on Power-Aware Computing and Systems

INFLOW '14: 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads

TRIOS '14: 2014 Conference on Timely Results in Operating Systems

All events will take place at the Omni Interlocken Resort

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

www.usenix.org/osdi14

Python Gets an Event Loop (Again)

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009) and *Python Cookbook* (3rd Edition, O'Reilly Media, 2013). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses.
dave@dabeaz.com

March 2014 saw the release of Python 3.4. One of its most notable additions is the inclusion of the new `asyncio` module to the standard library [1]. The `asyncio` module is the result of about 18 months of effort, largely spearheaded by the creator of Python, Guido van Rossum, who introduced it during his keynote talk at the PyCon 2013 conference. However, ideas concerning asynchronous I/O have been floating around the Python world for much longer than that. In this article, I'll give a bit of historical perspective as well as some examples of using the new library. Be aware that this topic is pretty bleeding edge—you'll probably need to do a bit more reading and research to fill in some of the details.

Some Basics: Networking and Threads

If you have ever needed to write a simple network server, Python has long provided modules for socket programming, processes, and threads. For example, if you wanted to write a simple TCP/IP echo server capable of handling multiple client connections, an easy way to do it is to write some code like this:

```
# echoserver.py

from socket import socket, AF_INET, SOCK_STREAM
import threading

def echo_client(sock):
    while True:
        data = sock.recv(8192)
        if not data:
            break
        sock.sendall(data)
    print('Client closed')
    sock.close()

def echo_server(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(5)
    while True:
        client_sock, addr = sock.accept()
        print('Connection from', addr)
        t = threading.Thread(target=echo_client,
                             args=(client_sock,))
        t.start()
```

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

Although simple, this style of programming is the foundation for Python's `socketserver` module (called `SocketServer` in Python 2). `socketserver`, in turn, is the basis of Python's other built-in server libraries for HTTP, XML-RPC, and similar.

Asynchronous I/O

Even though programming with threads is a well-known and relatively simple approach, it is not always appropriate in all cases. For example, if a server needs to manage a very large number of open connections, running a program with 10,000 threads may not be practical or efficient. For such cases, an alternative solution involves creating an asynchronous or event-driven server built around low-level system calls such as `select()` or `poll()`. The underlying approach is based on an underlying event-loop that constantly polls all of the open sockets and triggers event-handlers (i.e., callbacks) on objects to respond as appropriate.

Python has long had a module, `asyncore`, for supporting asynchronous I/O. Here is an example of the same echo server implemented using it:

```
# echoasyncore.py
from socket import AF_INET, SOCK_STREAM
import asyncore

class EchoClient(asyncore.dispatcher):
    def __init__(self, sock):
        asyncore.dispatcher.__init__(self, sock)
        self._outbuffer = b''
        self._readable = True

    def readable(self):
        return self._readable

    def handle_read(self):
        data = self.recv(8192)
        self._outbuffer += data

    def handle_close(self):
        self._readable = False
        if not self._outbuffer:
            print('Client closed connection')
            self.close()

    def writable(self):
        return bool(self._outbuffer)

    def handle_write(self):
        nsent = self.send(self._outbuffer)
        self._outbuffer = self._outbuffer[nsent:]
        if not (self._outbuffer or self._readable):
            self.handle_close()

class EchoServer(asyncore.dispatcher):
    def __init__(self, address):
        asyncore.dispatcher.__init__(self)
        self.create_socket(AF_INET, SOCK_STREAM)
        self.bind(address)
        self.listen(5)

    def readable(self):
        return True

    def handle_accept(self):
        client, addr = self.accept()
        print('Connection from', addr)
        EchoClient(client)

EchoServer(('', 25000))
asyncore.loop()
```

In this code, the various objects `EchoServer` and `EchoClient` are really just wrappers around a traditional network socket. All of the important logic is found in callback methods such as `handle_accept()`, `handle_read()`, `handle_write()`, and so forth. Finally, instead of running a thread or process, the server runs a centralized event-loop initiated by the final call to `asyncore.loop()`.

Wilted Async?

Although `asyncore` has been part of the standard library since Python 1.5.2, it's always been a bit of an abandoned child. Programming with it directly is difficult—involving layers upon layers of callbacks. Moreover, the standard library doesn't provide any other support to make `asyncore` support higher-level protocols (e.g., HTTP) or to interoperate with other parts of Python (e.g., threads, queues, subprocesses, pipes, signals, etc.). Thus, if you've never actually encountered any code that uses `asyncore` in the wild, you're not alone. Almost nobody uses it—it's just too painful and low-level to be a practical solution for most programmers.

Instead, you'll more commonly find asynchronous I/O supported through third-party frameworks such as Twisted, Tornado, or Gevent. Each of these frameworks tends to be a large world unto itself. That is, they each provide their own event loop, and they provide asynchronous compatible versions of common library functions. Although it is possible to perform a certain amount of adaptation to make these different libraries work together, it's all a bit messy.

Enter asyncio

The `asyncio` library introduced in Python 3.4 represents a modern attempt to bring asynchronous I/O back into the standard library and to provide a common core upon which additional async-oriented libraries can be built. `asyncio` also aims to standardize the implementation of the event-loop so that it can be adapted to support existing frameworks such as Twisted or Tornado.

The definitive description of asyncio can be found in PEP-3156 [2]. Rather than rehash the contents of the admittedly dense PEP, I'll provide a simple example to show what it looks like to program with asyncio. Here is a new implementation of the echo server:

```
# echoasync.py

from socket import socket, AF_INET, SOCK_STREAM
import asyncio

loop = asyncio.get_event_loop()

@asyncio.coroutine
def echo_client(sock):
  while True:
     data = yield from loop.sock_recv(sock, 8192)
     if not data:
        break
     yield from loop.sock_sendall(sock, data)
  print('Client closed')
  sock.close()

@asyncio.coroutine
def echo_server(address):
  sock = socket(AF_INET, SOCK_STREAM)
  sock.bind(address)
  sock.listen(5)
  sock.setblocking(False)
  while True:
     client_sock, addr = yield from loop.sock_accept(sock)
     print('Connection from', addr)
     asyncio.async(echo_client(client_sock))

if __name__ == '__main__':
  loop.run_until_complete(echo_server(('', 25000)))
```

Carefully compare this code to the first example involving threads. You will find that the code is virtually identical except for the mysterious @asyncio.coroutine decorator and use of the yield from statements. As for those, what you're seeing is a programming style based on coroutines—in essence, a form of cooperative user-level concurrency.

A full discussion of coroutines is beyond the scope of this article; however, the general idea is that each coroutine represents a kind of user-level "task" that can be executed concurrently. The yield from statement indicates an operation that might potentially block or involve waiting. At these points, the coroutine can be suspended and then resumed at a later point by the underlying event loop. To be honest, it's all a bit magical under the covers. I previously presented a PyCon tutorial on coroutines [3]. However, the yield from statement is an even more modern development that is only available in Python 3.3 and newer, described in PEP-380 [4]. For now, just accept the fact that the yield from is required and that you've probably never seen it used in any previous Python code.

Getting Away from Low-Level Sockets

As shown, the sample echo server is directly manipulating a low-level socket. However, it's possible to write a server that abstracts the underlying protocol away. Here is a slightly modified example that uses a higher-level transport interface:

```
import asyncio
loop = asyncio.get_event_loop()

@asyncio.coroutine def
echo_client(reader, writer):
  while True:
     data = yield from reader.readline()
     if not data:
        break
     writer.write(data)
  print('Client closed')

if __name__ == '__main__':
  fut = asyncio.start_server(echo_client, '', 25000)
  loop.run_until_complete(fut)
  loop.run_forever()
```

As shown, this runs as a TCP/IP echo server. However, you can change it to a UNIX domain server if you simply change the last part as follows:

```
if __name__ == '__main__':
  fut = asyncio.start_unix_server(echo_client, '/tmp/spam')
  loop.run_until_complete(fut)
  loop.run_forever()
```

In both cases, the underlying protocol is abstracted away. The echo_client() function simply receives reader and writer objects on which to read and write data—it doesn't need to worry about the exact protocol being used to transport the bytes.

More Than Sockets

A notable feature of asyncio is that it's much more than a simple wrapper around sockets. For example, here's a modified client that feeds its data to a subprocess running the UNIX wc command and collects the output afterwards:

```
from asyncio import subprocess

@asyncio.coroutine
def echo_client(reader, writer):
  proc = yield from asyncio.create_subprocess_exec('wc',
                                  stdin=subprocess.PIPE,
                                 stdout=subprocess.PIPE)
  while True:
     data = yield from reader.readline()
```

```
        if not data:
            break
        proc.stdin.write(data)
        writer.write(data)
    proc.stdin.close()
    stats = yield from proc.stdout.read()
    yield from proc.wait()
    print("Client closed:", stats.decode('ascii'))
```

Here is an example of a task that simply sleeps and wakes up periodically on a timer:

```
@asyncio.coroutine
def counter():
    n = 0
    while True:
        print("Counting:", n)
        yield from asyncio.sleep(5)
        n += 1

if __name__ == '__main__':
    asyncio.async(counter())
    loop.run_forever()
```

There is even support for specialized tasks such as attaching a signal handler to the event loop. For example:

```
import signal

def handle_sigint():
    print('Quitting')
    loop.stop()

loop.add_signal_handler(signal.SIGINT, handle_sigint)
```

Last, but not least, you can delegate non-asynchronous work to threads or processes. For example, if you had a burning need for a task to print out Fibonacci numbers using a horribly inefficient implementation and you didn't want the computation to block the event loop, you could write code like this:

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

loop = asyncio.get_event_loop()

def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

@asyncio.coroutine
def fibonacci():
    n = 1
    while True:
        r = yield from loop.run_in_executor(pool, fib, n)
        print("Fib(%d): %d" % (n, r))
        yield from asyncio.sleep(1)
        n += 1

if __name__ == '__main__':
    pool = ThreadPoolExecutor(8)
    asyncio.async(fibonacci())
    loop.run_forever()
```

In this example, the `loop.run_in_executor()` arranges to run a user-supplied function (`fib`) in a separate thread. The first argument supplies a thread-pool or process-pool as created by the `concurrent.futures` module.

Where to Go from Here?

There is much more to `asyncio` than presented here. However, I hope the few examples here have given you a small taste of what it looks like. For more information, you might consult the official documentation [1]; if you're like me, however, you'll find the documentation a bit dense and lacking in examples. Thus, you're probably going to have to fiddle around with it as an experiment. Searching the Web for "asyncio examples" can yield some additional information and insight for the brave. In the references section, I've listed a couple of presentations and sites that have more examples [5, 6].

As for the future, it will be interesting to see whether `asyncio` is adopted as a library for writing future asynchronous libraries and applications. As with most things Python 3, only time will tell.

If you're still using Python 2.7, the Trollius project [7] is a backport of the asyncio library to earlier versions of Python. The programming interface isn't entirely the same because of the lack of support for the "yield from" statement, but the overall architecture and usage are almost identical.

References

[1] asyncio (official documentation): http://docs.python.org/dev/library/asyncio.html.

[2] PEP 3156: http://python.org/dev/peps/pep-3156/.

[3] David Beazley, "A Curious Course on Coroutines and Concurrency": http://www.dabeaz.com/coroutines.

[4] PEP 380: http://python.org/dev/peps/pep-0380/.

[5] Saul Ibarra Corretge, "A Deep Dive Into PEP-3156 and the New asyncio Module": http://www.slideshare.net/saghul/asyncio.

[6] Feihong Hsu, "Asynchronous I/O in Python 3": http://www.slideshare.net/megafeihong/tulip-24190096.

[7] Trollius project: https://pypi.python.org/pypi/trollius.

iVoyeur
Monitoring Design Patterns

DAVE JOSEPHSEN

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

Plutarch tells a great story about Hannibal, in the time of the reign of Dictator Fabius. Hannibal, a barbarian from the south, was roaming about the countryside making trouble as barbarians do, when through either some subtlety on the part of Fabius or just seriously abominable direction-giving on the part of a few sheep farmers (reports differ), Hannibal managed to get himself trapped and surrounded in a valley.

Knowing that he wasn't going to be able to escape by assault in any direction, Hannibal came up with an ingenious ploy. The story goes that he waited until nightfall, and in the darkness he sent his men up the hillsides, while keeping his livestock in the valley floor. Then, setting the horns of his cattle ablaze, he stampeded them towards Fabius' lines. The Romans, seeing what they thought was a charge of torch-wielding barbarians (but which was really a multitude of terrified flaming cattle), reformed and dug in, and in so doing, allowed Hannibal's men to escape past them on the slopes above.

Unless you're a cow, you have to agree that this was a pretty great bit of ingenuity. It's also a pretty great example of the UNIX principle of unexpected composition: that we should craft and use tools that may be combined or used in ways that we never intended. Before Hannibal, few generals probably considered the utility of flaming cattle in the context of anti-siege technology.

The monitoring systems built in the past fail pretty miserably when it comes to the principle of unexpected composition. Every one of them is born from a core set of assumptions—assumptions that ultimately impose functional limits on what you can accomplish with the tool. This is perhaps the most important thing to understand about monitoring systems before you get started designing a monitoring infrastructure of your own: Monitoring systems become more functional as they become less featureful.

Some systems make assumptions about how you want to collect data. Some of the very first monitoring systems, for example, assumed that everyone would always monitor everything using SNMP. Other systems make assumptions about what you want to do with the data once it's collected—that you would never want to share it with other systems, or that you want to store it in thousands of teensy databases on the local file system. Most monitoring systems present this dilemma: They each solve part of the monitoring problem very well but wind up doing so in a way that saddles you with unwanted assumptions, like SNMP and thousands of teensy databases.

Many administrators interact with their monitoring infrastructures like they might a bag of jellybeans—keeping the pieces they like and discarding the rest. In the past few years, many little tools have come along that make this functionally possible, enabling you to replace or augment your single, centralized monitoring system with a bunch of tiny, purpose-driven data collectors wired up to a source-agnostic storage tier (disclaimer: I work for a source-agnostic storage tier as a service company called Librato).

This strategy lets you use whatever combination of collectors makes sense for you. You can weave monitoring into your source code directly, or send forth herds of flaming cattle to collect it, and then store the monitoring data together, where it can be visualized, analyzed, correlated, alerted on, and even multiplexed to multiple destinations regardless of its source and method of collection.

Tons of open source data collectors and storage tiers are available, but instead of talking about any of them specifically, I'd like to write a little bit about the monitoring "patterns" that currently exist in the wild, because although there are now eleventybillion different implementation possibilities, they all combine the same basic five or six design patterns. In the same way I can describe Hannibal's livestock as a "diversion movement to break contact," I hope that categorizing these design patterns will make it easier to write about the specifics of data collectors and flaming cows later on.

Collection Patterns for External Metrics

I begin by dividing the target metrics themselves into two general categories: those that are derived from within the monitored process at runtime, and those that are gathered from outside the monitored process. Considering the latter type first, four patterns generally are used to collect availability and performance data from outside the monitored process.

The Centralized Polling Pattern

Anyone who has worked with monitoring systems for a while has used centralized pollers. They are the archetype design—the one that comes to mind first when someone utters the phrase "monitoring system" (although that is beginning to change). See Figure 1.

Like a grade-school teacher performing the morning roll call, the centralized poller is a monolithic program that is configured to periodically poll a number of remote systems, usually to ensure that they are available but also to collect performance metrics. The poller is usually implemented as a single process on a single server, and it usually attempts to make guarantees about the interval at which it polls each service.

Because this design predates the notion of configuration management engines, centralized pollers are designed to minimize the amount of configuration required on the monitored hosts. They may rely on external connectivity tests, or they may remotely execute agent software on the hosts they poll; in either case, however, their normal mode of operation is to periodically pull data directly from a population of monitored hosts.

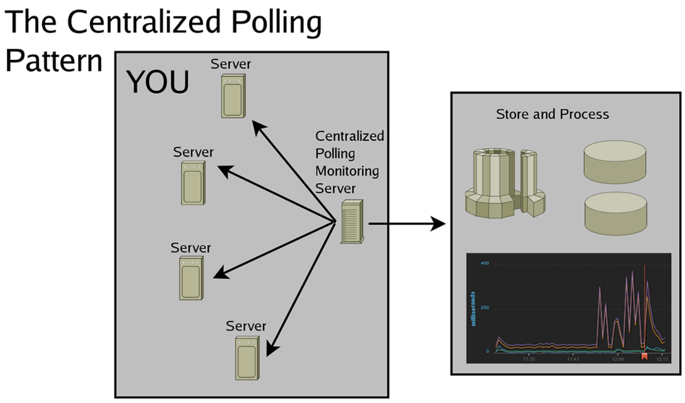

The Centralized Polling Pattern
YOU
Server
Server
Server
Server
Centralized Polling Monitoring Server
Store and Process

Figure 1: The centralized poller monitoring pattern

Centralized pollers are easy to implement but often difficult to scale. They typically operate on the order of minutes, using, for example, five-minute polling intervals, and this limits the resolution at which they can collect performance metrics. Older centralized pollers are likely to use agents with root-privileged shell access for scripting, communicate using insecure protocols, and have unwieldy (if any) failover options.

Although classic centralized pollers like Nagios, Munin, and Cacti are numerous, they generally don't do a great job of playing with others because they tend to make the core assumption that they are the ultimate solution to the monitoring problem at your organization. Most shops that use them in combination with other tools interject a metrics aggregator like statsd or other middleware between the polling system and other monitoring and storage systems.

The Stand-Alone Agent Pattern

Stand-alone agents have grown in popularity as configuration-management engines have become more commonplace. They are often coupled with centralized pollers or roll-up model systems to meet the needs of the environment. See Figure 2.

Agent software is installed and configured on every host that you want to monitor. Agents usually daemonize and run in the background, waking up at timed intervals to collect various performance and availability metrics. Because agents remain resident in memory and eschew the overhead of external connection setup and teardown for scheduling, they can collect metrics on the order of seconds or even microseconds. Some agents push status updates directly to external monitoring systems, and some maintain summary statistics that they present to pollers as needed via a network socket.

Agent configuration is difficult to manage without a CME, because every configuration change must be pushed to all applicable monitored hosts. Although they are generally designed

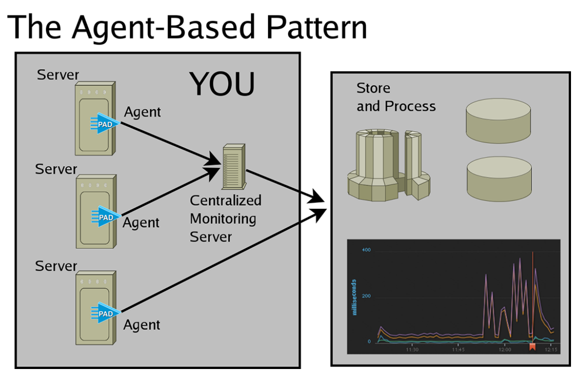


Figure 2: The stand-alone agent pattern

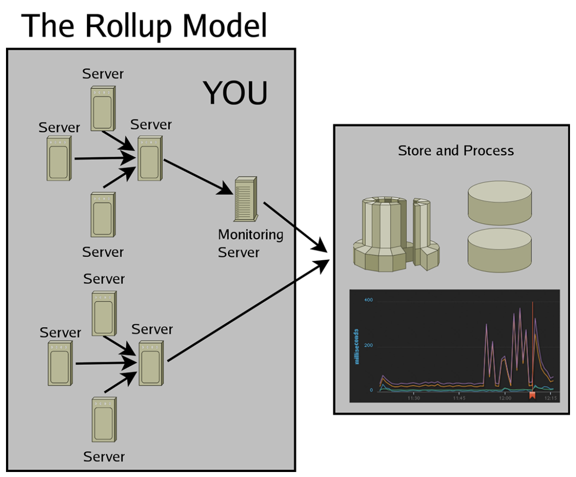


Figure 3: The roll-up pattern

to be lightweight, they can introduce a non-trivial system load if incorrectly configured. Be careful with closed-source agent software, which can introduce backdoors and stability problems. Open source agents are generally preferred because their footprint, overhead, and security can be verified and tweaked if necessary.

Collectd is probably the most popular stand-alone agent out there. Sensu uses a combination of the agent and polling pattern, interjecting a message queue between them.

The Roll-Up Pattern

The roll-up pattern is often used to achieve scale in monitoring distributed systems and large machine clusters or to aggregate common metrics across many different sources. It can be used in combination with agent software or instrumentation. See Figure 3.

The roll-up pattern is a strategy to scale the monitoring infrastructure linearly with respect to the number of monitored systems. This is usually accomplished by co-opting the monitored machines themselves to spread the monitoring workload throughout the network. Usually, small groups of machines use an election protocol to choose a proximate, regional collection host, and send all of their monitoring data to it, although sometimes the configuration is hard-coded.

The elected host summarizes and deduplicates the data, then sends it up to another host elected from a larger region of summarizers. This host in turn summarizes and deduplicates it, and so forth.

Roll-up systems scale well but can be difficult to understand and implement. Important stability and network-traffic considerations accompany the design of roll-up systems.

Ganglia is a popular monitoring project that combines stand-alone agents with the roll-up pattern to monitor massive clusters of hosts with fine-grained resolution. The statsd daemon process can be used to implement roll-up systems to hand-off in-process metrics.

Logs as Event-Streams

System and event logs provide a handy event stream from which to derive metric data. Many large shops have intricate centralized log processing infrastructure from which they feed many different types of monitoring, analytics, event correlation, and security software. If you're a Platform-as-a-Service (PaaS) customer, the log stream may be your only means to emit, collect, and inspect metric data from your application.

Applications and operating systems generate logs of important events by default. The first step in the log-stream pattern requires the installation or configuration of software on each monitored host that forwards all the logs off that host. Event-Reporter for Windows or rsyslogd on UNIX are popular log forwarders. Many programming languages also have log generation and forwarding libraries, such as the popular Log4J Java library. PaaS systems like Heroku have likely preconfigured the logging infrastructure for you.

Logs are generally forwarded to a central system for processing, indexing, and storage, but in larger environments they might be MapReduced or processed by other fan-out style parallel processing engines. System logs are easily multiplexed to different destinations, so there is a diverse collection of software available for processing logs for different purposes.

Although many modern syslog daemons support TCP, the syslog protocol was originally designed to use UDP in the transport layer, which can be unreliable at scale. Log data is usually emitted by the source in a timely fashion, but the intermediate processing systems can introduce some delivery latency. Log

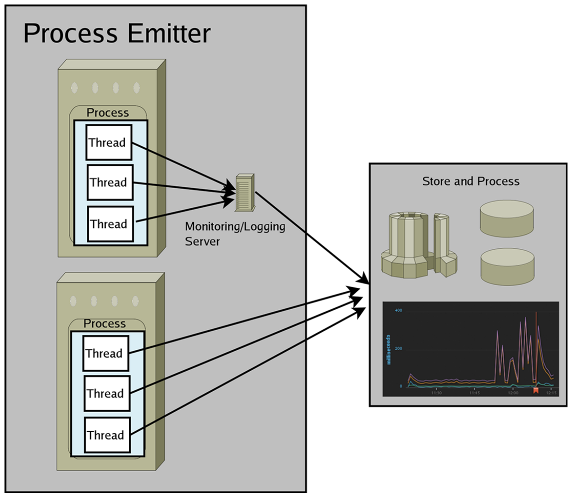


Figure 4: The process emitter pattern

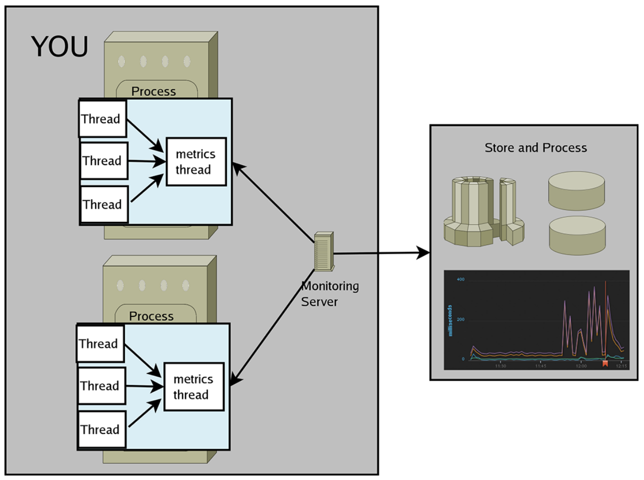


Figure 5: The process reporter pattern

data must be parsed, which can be a computationally expensive endeavor. Additional infrastructure may be required to process logging event streams as volume grows.

As I've already mentioned, with PaaS providers like Heroku and AppHarbor, logs are the only means by which to export and monitor performance data. Thus, many tools like Heroku's own log-shuttle and l2MET have grown out of that use-case. There are several popular tools for DIY enterprise log snarfing, like logstash, fluentd, and Graylog, as well as a few commercial offerings, like Splunk.

Collection Patterns for In-Process Metrics

Instrumentation libraries, which are a radical departure from the patterns discussed thus far, enable developers to embed monitoring into their applications, making them emit a constant stream of performance and availability data at runtime. This is not debugging code but a legitimate part of the program that is expected to remain resident in the application in production. Because the instrumentation resides within the process it's monitoring, it can gather statistics on things like thread count, memory buffer and cache sizes, and latency, which are difficult (in the absence of standard language support like JMX) for external processes to inspect.

Instrumentation libraries make it easy to record interesting measurements inside an application by including a wealth of instrumentation primitives like counters, gauges, and timers. Many also include complex primitives like histograms and percentiles, which facilitate a superb degree of performance visibility at runtime.

The applications in question are usually transaction-oriented; they process and queue requests from end users or external peer processes to form larger distributed systems. It is critically important for such applications to communicate their performance metrics without interrupting or otherwise introducing latency into their request cycle. Two patterns are normally employed to meet this need.

The Process Emitter Pattern

Process emitters attempt to immediately purge every metric via a non-blocking channel. See Figure 4.

The developer imports a language-specific metrics library and calls an instrumentation function like time() or increment(), as appropriate for each metric he wants to emit. The instrumentation library is effectively a process-level, stand-alone agent that takes the metric and flushes it to a non-blocking channel (usually a UDP socket or a log stream). From there, the metric is picked up by a system that employs one or more of the external-process patterns.

Statsd is a popular and widely used target for process emitters. The project maintains myriad language bindings to enable the developer to emit metrics from the application to a statsd daemon process listening on a UDP socket.

The Process Reporter Pattern

Process reporters use a non-blocking dedicated thread to store their metrics in an in-memory buffer. They either provide a concurrent interface for external processes to poll this buffer or periodically flush the buffer to upstream channels. See Figure 5.

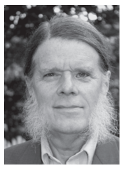
The developer imports a language-specific metrics library and calls an instrumentation function like time() or increment(), as appropriate for each metric he wants to emit. Rather than purging the metric immediately, process reporters hand the metric off to a dedicated, non-blocking thread that stores and sometimes processes summary statistics for each metric within the memory space of the monitored process. Process reporters can push their metrics on a timed interval to an external monitoring system or can export them on a known interface that can be polled on demand.

Process reporters are specific to the language in which they are implemented. Most popular languages have excellent metrics libraries that implement this pattern. Coda Hale Metrics for Java, Metriks for Ruby, and go-metrics are all excellent choices.

Thanks for bearing with me once again. I hope this article will help you identify the assumptions and patterns employed by the data collectors you choose to implement in your environment, or at least get you thinking about the sorts of things you can set aflame should you find yourself cornered by Roman soldiers. Be sure to check back with me in the next issue when I bend yet another tenuously related historical or mythical subject matter to my needs in my ongoing effort to document the monitoringosphere.

Exploring with a Purpose

DAN GEER AND JAY JACOBS



Dan Geer is the CISO for In-Q-Tel and a security researcher with a quantitative bent. He has a long history with the USENIX Association, including officer positions, program committees, etc. dan@geer.org

Jay Jacobs is the co-author of *Data-Driven Security* and a data analyst at Verizon where he contributes to their Data Breach Investigations Report. Jacobs is a cofounder of the Society of Information Risk Analysts. jay@beechplane.com

Think of [knowledge] as a house that magically expands with each door you open. You begin in a room with four doors, each leading to a new room that you haven't visited yet.... But once you open one of those doors and stroll into that room, three new doors appear, each leading to a brand-new room that you couldn't have reached from your original starting point. Keep opening doors and eventually you'll have built a palace.

Steven Johnson, *"The Genius of the Tinkerer"* [1]

Learning pays compound interest; as a person studies a subject, the more capable they become at learning even more about the subject. Just as a student cannot tackle the challenges of calculus without studying the prerequisites, we must have diligence in how we discover and build the prerequisite knowledge within cybersecurity.

Before we discuss where we are heading, let's establish where we are. Until now, we (security metricians, including the present authors) could exhort people to "Just measure *something* for heaven's sakes!" It's safe to say that such measurement has largely begun. Therefore, we have the better, if harder, problem of the meta-analysis ("research about research") of many observations, always remembering that the purpose of security metrics is decision support.

Learning from All of Us

To understand how we are at processing our observations, we turn to published industry reports. It's clear that there are a lot more of them than even two years ago. Not all reports are equal; parties have various motivations to publish, which creates divergent interpretations of what represents research worth communicating.

We suspect that most data included in industry reports are derived from convenience samples—data gathered because it is available to the researcher, not necessarily data that is representative enough to be generalizable. Not to make this a statistics tutorial, but for generalizability you need to understand (and account for) your sampling fraction, or you need to randomize your collection process. It is not that this or that industry report has a bias—all data has bias; the question is whether you can correct for that bias. A single vendor's data supply will be drawn from that vendor's customer base, and that's something to correct for. On the other hand, if you can find three or more vendors producing data of the same general sort, combining them in order to compare them can wash out the vendor-to-customer bias at least insofar as decision support is concerned.

Do not mistake our comments for a reason to dismiss convenience samples; research with a convenience sample is certainly better than "learning" from some mix of social media and headlines. This challenge in data collection is not unique to cybersecurity; performing research on automobile fatalities does not lend itself to selecting random volunteers. Studying the effects of a disease requires a convenience study of patients with the disease. It's too

Exploratory before Explanatory

Exploratory research is all about hypothesis generation, not hypothesis testing. It is all about recognizing what are the unknowns within an environment. When that environment is complex or relatively unstudied, exploratory analysis tells you where to put the real effort. Exploratory research identifies the hypotheses for explanatory research to resolve. Exploratory research does not end with "Eureka!" It ends with "If this is where I am, then which way do I go?"

early to call it, but we think it infeasible to conduct randomized clinical trials, cohort studies, and case-control research, but the time is right for such ideas to enter the cybersecurity field (and for some of you to prove us wrong).

If we are going to struggle in the design of our research and data collection, we may be doomed never to produce a single study without flaws. Although that does not preclude learning, it means that we will have to accept and even embrace the variety of conclusions each study will bring while reserving the big lessons to be drawn after the appropriate corrections for the biases in each study are made and those results aggregated. This method of learning requires the active participation of researchers who must not only understand the sampling fraction that underlies their data but also must transparently communicate it and the methodology of their research.

Learning from Each Other

Industry reports are generally data aggregated by automated means across the whole of the vendor's installed base. The variety found in these aggregation projects is intriguing, because much of the data now being harvested is in a style that we call "voluntary surveillance," such as when all the clients of Company X beacon home any potential malware that they see so that the probability of detection is heightened and the latency of countermeasures is reduced for everyone. Of course, once the client (that's you) says "Keep an eye on me," you don't have much to say about just how closely that eyeball is looking unless you actually read the whole outbound data stream yourself.

What can you learn from industry reports? Principally two things: "What is the trendline?" and "Am I different?" A measurement method can be noisy and can even contain a consistent bias without causing the trendline it traces out to yield misleading decision support. As long as the measurement error is reasonably constant, the trendline is fine. Verizon's Data Breach Investigations Report (DBIR) [2] is not based on a random sample of the world's computing plant, but that only affects the generalizability of its measured variables, not the trendline those variables trace. By contrast, public estimates of the worldwide cost of cybercrime are almost surely affected by what it takes to get newspaper headline writers to look at you. Producers of cybercrime estimates certainly claim to be based on data, but their bias and value in meta-analysis efforts must be questioned.

That trendlines are useful decision support reminds us that an ordinal scale is generally good enough for decision support. Sure, real number scales ("What do you weigh?") are good to have, but ordinal scales ("Have you lost weight?") are good enough for decision making ("Did our awareness training hold down the number of detectable cybersecurity mistakes this year?").

Whether you are different from everybody else matters only insofar as whether that difference (1) can be demonstrated with measured data and (2) has impact on the decisions that you must make. Suppose we had the full perimeter firewall logs from the ten biggest members of the Defense Industrial Base. Each one is drawn under a different sampling regime, but if they all show the same sorts of probes from the same sorts of places, then the opponent is an opportunistic opponent, which has planning implications. If, however, they all show the same trendline except for yours, then as a matter of decision support your next step is to acquire data that helps you explain what makes you special—and whether there is anything to be done about that.

Standing on the Shoulders of Giants

Medicine has a lot to teach us about combining studies done by unrelated researchers, which is a good thing, because we don't have a decade to burn reinventing those skills. The challenges facing such meta-analysis are finding multiple research efforts

1. with comparable measurements;
2. researching the same time period (environment may change rapidly);
3. publishing relevant characteristics like the sample size under observation, the data collection, and the categorization scheme.

Without the combination of all three, comparison and meta-analysis (and consequently our ability to learn) becomes significantly more difficult. To illustrate, we collected 48 industry reports; 19 of them contained a reference to "android," and five of those 19 estimated the amount of android malware to be:

- 405,140 android malware through 2012 (257,443 with a strict definition of "malware") [3]
- 276,259 total mobile malware through Q1 2013 [4]
- 50,926 total mobile malware through Q1 2013 [5]
- 350,000 total number of android malware though 2012 [6]
- over 200,000 malware for android through 2012 [7]

That's a broad range of contrast. But do not mistake the range and contrast for an indication of errors or mistakes—their studies are exploring data that they have access to and are an example of the variety of conclusions we should expect. That exploration is exactly what should be happening in this field at this point in time. What we (the security metrics people) must now do is learn how to do meta-analysis in our domain, and if we are producers, learn how to produce research consumable by other security metrics people. We have to learn how to deal with our industry's version of publication bias, learn how adroitly to discount agenda-driven "results," and learn which indicators enable us to infer study quality.

This task will not be easy, but it is timely. It is time for a cybersecurity data science. We call on those of you who can do exploratory analysis of data to do so and to publish in styles such that the tools of meta-analysis can be used to further our understanding across the entire cybersecurity field.

Thanks in advance.

Resources

[1] Steven Johnson, "The Genius of the Tinkerer," *Wall Street Journal*, September 25, 2010: tinyurl.com/ka9wotx.

[2] Verizon's Data Breach Investigations Report: http://www.verizonenterprise.com/DBIR/.

[3] F-Secure Threat Report H1 2013: retrieved from tinyurl.com/ob6t3b4.

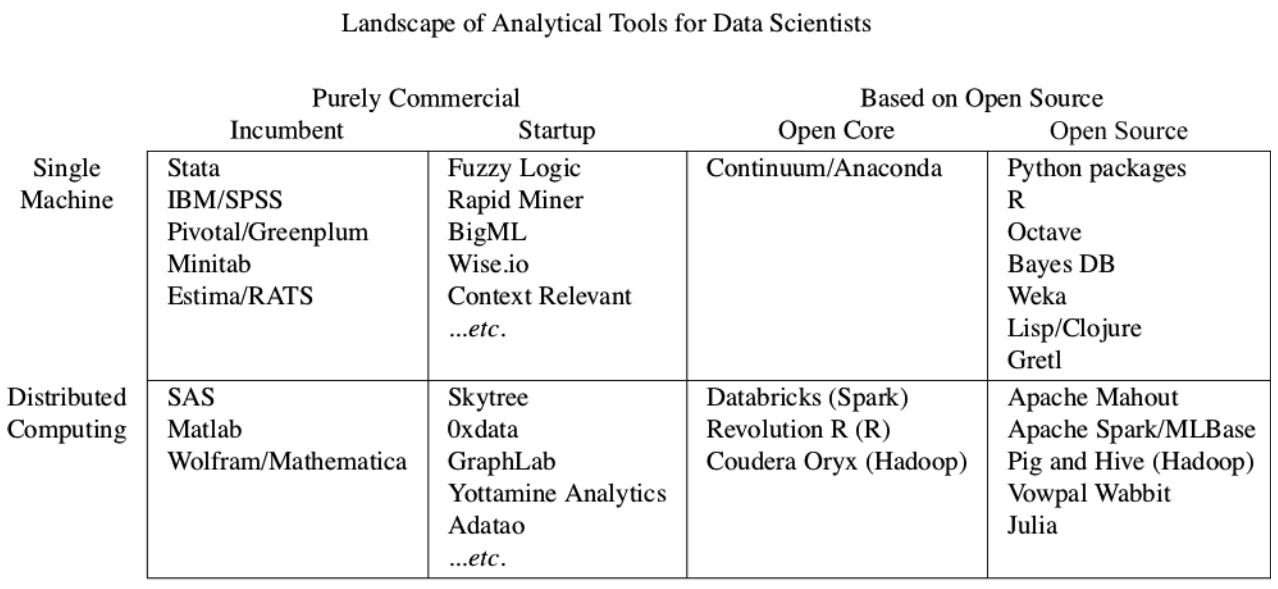
[4] Juniper Networks Third Annual Mobile Threats Report, March 2012 through March 2013: retrieved from tinyurl.com/nf654ah.

[5] McAfee Threats Report: Q1 2013: retrieved from tinyurl.com/ovtrbmg.

[6] Trend Micro: TrendLabs Q1 2013 Security roundup: retrieved from tinyurl.com/lfs2uvt.

[7] Trustwave 2013 Global Security Report: retrieved from tinyurl.com/ljt73qj.

For more resources, please visit: http://dds.ec/rsrc.



Landscape of Analytical Tools for Data Scientists

	Purely Commercial		Based on Open Source	
	Incumbent	Startup	Open Core	Open Source
Single Machine	Stata IBM/SPSS Pivotal/Greenplum Minitab Estima/RATS	Fuzzy Logic Rapid Miner BigML Wise.io Context Relevant ...*etc*.	Continuum/Anaconda	Python packages R Octave Bayes DB Weka Lisp/Clojure Gretl
Distributed Computing	SAS Matlab Wolfram/Mathematica	Skytree 0xdata GraphLab Yottamine Analytics Adatao ...*etc*.	Databricks (Spark) Revolution R (R) Coudera Oryx (Hadoop)	Apache Mahout Apache Spark/MLBase Pig and Hive (Hadoop) Vowpal Wabbit Julia

Addendum: Data science tools as of this date

/dev/random

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've looked at the cloud from both sides now:
From in and out, and still somehow,
It's the cloud's delusions I recall;
I still trust the cloud not at all.

(apologies to Joni Mitchell)

As humanity's adoption of the cloud paradigm slides ever closer to unity, detractors like yours truly grow increasingly scarce and anathematized. Rather than retreat into a bitter lepers' colony for Cloud-Luddites ("Cluddites"), I choose instead to cast my remaining stones from the unfettered, if somewhat dizzying, forefront of prophecy.

There will come a time, sayeth the prophet, when all personal computing devices will be wearable thin-clients, communicating with their data and applications via whatever is set to replace Bluetooth and/or WiFi. People will no longer buy software or storage devices but rather subscriptions to CCaaS: Cloud Computing as a (dis)Service. There will be specific application bundles available (word processing, graphics manipulation, spreadsheets, etc.), as well as omnibus cloudware plans that allow access to a broad range of data crunching techniques.

All advertising will be razor-targeted to the person, place, circumstance, and even current physiological status of its victims ^H^H^H^H^H^H^H audience. Even ad contents themselves will be written on the fly by personalized marketing microbots that have spent their entire existence studying and learning a single person's lifestyle. Because there will be no one left in the labor pool who subscribes to any consistent rules of grammar, spelling, or rhetoric, these ads will be mercifully incomprehensible to most.

Moving further afield in our technology examination, reality television will merge with products such as GoPro, Looxcie, and MeCam along with ultra-high bandwidth Internet access to create millions of around-the-clock multimedia streams starring absolutely everyone you know. For a fee, you can have writers supply you with dialogue while actors will fill in your voice to create the ultimate Shakespeare metaphor come to life: all the world really *will* be a stage.

Instead of news desks, anchors, and reporters, there will just be alerts sent out automatically by event-recognition algorithms embedded in the firmware of personal cams. They will recruit all nearby cameras into a newsworthy event cluster, or NEC. The combined streams from the cams with the best viewpoints will be labeled as a live news event and available with stream priority from the provider. Premium service subscribers will have voiceovers of commentators taking a marginally educated guess at what's going on, very similar to today's "talking head" model only not so well-dressed.

The stars of this mediaverse will be those people who manage to be near the most interesting occurrences. Naturally, a thriving "underground" business of *causing* news events will spring up, so that news personalities can maintain their fame without all that tedious roaming to and fro, looking for notoriety-inducing happenstance. The likelihood of witnessing everything from thrilling bank robberies to tragic aircraft accidents can be artificially boosted by these professional probability manipulators—for the right price.

With individual television stations obsolete and replaced by on-demand media clumps, product-hawking will be forced to evolve as well. People will be paid to carry around everything from soda cans to auto parts in the hope that someone's stream will pick them up and the omnipresent optical marketing monitor algorithms will recognize and plug same with a plethora of canned ads. With the average personal cam spitting out ten megapixels at 40 FPS, discerning one's product from the background noise of other people's inferior merchandise will not be all that difficult.

Because your wearable cam will be capable of monitoring various physiological parameters—ostensibly for your health's sake but really in order to gauge your response to various commercial ploys—any positive reaction to a product or event will cause an influx of ads for things judged by often puzzling equivalence formulae to be similar in some way. Think a dog you saw pooping on the curb was cute? Prepare to be bombarded for a full fortnight by vaguely canine and/or poop-related product and service advertisements. Carelessly employ a search engine to look up the best brand of adult diaper for an aged relative? Many hours of bodily fluid-absorbent entertainment will now be yours to enjoy in virtual 3D and ultrasurround-sound, with no means of escape sans an unthinkable disconnection from the grid.

While we're on the topic of disconnection, it won't be long before being unplugged from the grid not only is inadvisable from a mental health perspective, it will not even be possible without surgery. A routine implantation when an infant is about six months old will tie them into the worldwide grid, although a compatible interface will be necessary to do any computing. This will encourage toddlers to learn their alphabets (or at least the more popular letters and numbers) and, more importantly, emoticons as early as possible so that they can begin texting. Eventually I predict these Internet access modules will be absorbed into the body and tacked on to our DNA, creating the prospect of future pre-natal communications, where Twitter accounts are created automatically as soon as the child reaches a certain level of development.

> @TheNeatestFetus: OMG I thnk im bein born!!!
>
> @Embryoglio: hw du u no?
>
> @TheNeatestFetus: gettin squeezed out 1 end brb
>
> @Embryoglio: u still ther?
>
> @TheNeatestFetus: b**** slappd me! #BreathingRox
>
> @ Embryoglio: playa

It just keeps getting sillier from there, I'm afraid.

To bring this oblique essay back around to UNIX, let me just reassure my readers that every device I've prophesied here is running some stripped-down version of an embedded Linux kernel that I've just now decided to call Nanix (you know, because it's so small and everything).

What role will the by-now ubiquitous cloud play in all this? Every role imaginable. It won't be a cloud any longer, but rather an all-encompassing noxious ground fog that instead of creeping in on little cat feet will stomp around on enormous T-Rex talons. There will be nothing subtle or elegant about it. Data security, I must add—albeit reluctantly—will be a quaint concept of the past, like chivalry, coupon books, and pundits with something germane to say.

The Saddest Moment

JAMES MICKENS

James Mickens is a researcher in the Distributed Systems group at Microsoft's Redmond lab. His current research focuses on Web applications, with an emphasis on the design of JavaScript frameworks that allow developers to diagnose and fix bugs in widely deployed Web applications. James also works on fast, scalable storage systems for datacenters. James received his PhD in computer science from the University of Michigan, and a bachelor's degree in computer science from Georgia Tech. mickens@microsoft.com

Reprinted from ;login: *logout,* May 2013

Whenever I go to a conference and I discover that there will be a presentation about Byzantine fault tolerance, I always feel an immediate, unshakable sense of sadness, kind of like when you realize that bad things can happen to good people, or that Keanu Reeves will almost certainly make more money than you over arbitrary time scales. Watching a presentation on Byzantine fault tolerance is similar to watching a foreign film from a depressing nation that used to be controlled by the Soviets—the only difference is that computers and networks are constantly failing instead of young Kapruskin being unable to reunite with the girl he fell in love with while he was working in a coal mine beneath an orphanage that was atop a prison that was inside the abstract concept of World War II. "How can you make a reliable computer service?" the presenter will ask in an innocent voice before continuing, "It may be difficult if you can't trust anything and the entire concept of happiness is a lie designed by unseen overlords of endless deceptive power." The presenter never explicitly says that last part, but everybody understands what's happening. Making distributed systems reliable is inherently impossible; we cling to Byzantine fault tolerance like Charlton Heston clings to his guns, hoping that a series of complex

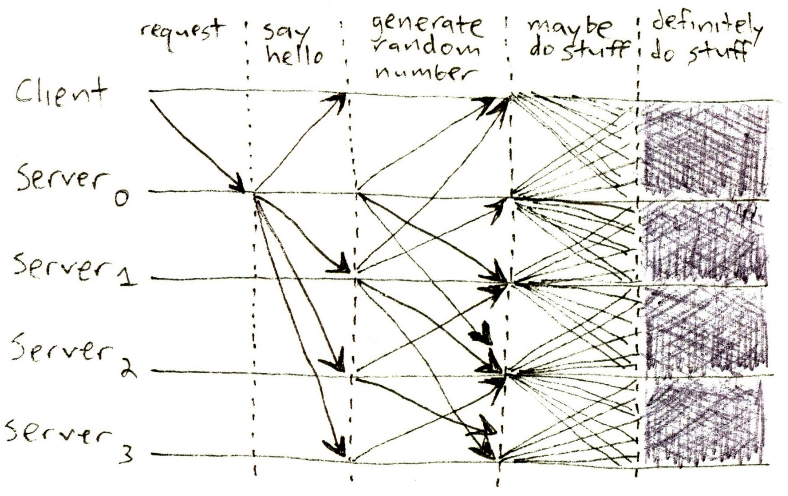


Figure 1: Typical Figure 2 from Byzantine fault paper: Our network protocol

software protocols will somehow protect us from the oncoming storm of furious apes who have somehow learned how to wear pants and maliciously tamper with our network packets.

Every paper on Byzantine fault tolerance contains a diagram that looks like Figure 1. The caption will say something like "Figure 2: Our network protocol." The caption should really say, "One day, a computer wanted to issue a command to an online service. This simple dream resulted in the generation of 16 gajillion messages. An attacker may try to interfere with the reception of 1/f of these messages. Luckily, 1/f is much less than a gajillion for any reasonable value of f. Thus, at least 15 gajillion messages will survive the attacker's interference. These messages will do things that only Cthulu understands; we are at peace with his dreadful mysteries, and we hope that you feel the same way. Note that, with careful optimization, only 14 gajillion messages are necessary. This is still too many messages; however, if the system sends fewer than 14 gajillion messages, it will be vulnerable to accusations that it only handles reasonable failure cases, and not the demented ones that previous researchers spitefully introduced in earlier papers in a desperate attempt to distinguish themselves from even more prior (yet similarly demented) work. As always, we are nailed to a cross of our own construction."

In a paper about Byzantine fault tolerance, the related work section will frequently say, "Compare the protocol diagram of our system to that of the best prior work. Our protocol is clearly better." The paper will present two graphs that look like Figure 2. Trying to determine which one of these hateful diagrams is better is like gazing at two unfathomable seaweed bundles that washed up on the beach and trying to determine which one is marginally less alienating. Listen, regardless of which Byzantine fault tolerance protocol you pick, Twitter will still have fewer than two nines of availability. As it turns out, Ted the Poorly Paid Datacenter Operator will not send 15 cryptographically signed messages before he accidentally spills coffee on the air conditioning unit and then overwrites your tape backups with bootleg recordings of Nickelback. Ted will just do these things and then go home, because that's what Ted does. His extensive home collection of "Thundercats" cartoons will not watch itself. Ted is needed, and Ted will heed the call of duty.

Every paper on Byzantine fault tolerance introduces a new kind of data consistency. This new type of consistency will have an ostensibly straightforward yet practically inscrutable name like "leap year triple-writer dirty-mirror asynchronous semi-consistency." In Section 3.2 ("An Intuitive Overview"), the authors will provide some plainspoken, spiritually appealing arguments about why their system prevents triple-conflicted write hazards in the presence of malicious servers and unexpected outbreaks of the bubonic plague. "Intuitively, a malicious server cannot lie to a client because each message is an encrypted, nested, signed, mutually-attested log entry with pointers to other encrypted and nested (but not signed) log entries."

Interestingly, these kinds of intuitive arguments are not intuitive. A successful intuitive explanation must invoke experiences that I have in real life. I have never had a real-life experience that resembled a Byzantine fault tolerant protocol. For example, suppose that I am at work, and I want to go to lunch with some of my co-workers. Here is what that experience would look like if it resembled a Byzantine fault tolerant protocol:

JAMES: I announce my desire to go to lunch.

BRYAN: I verify that I heard that you want to go to lunch.

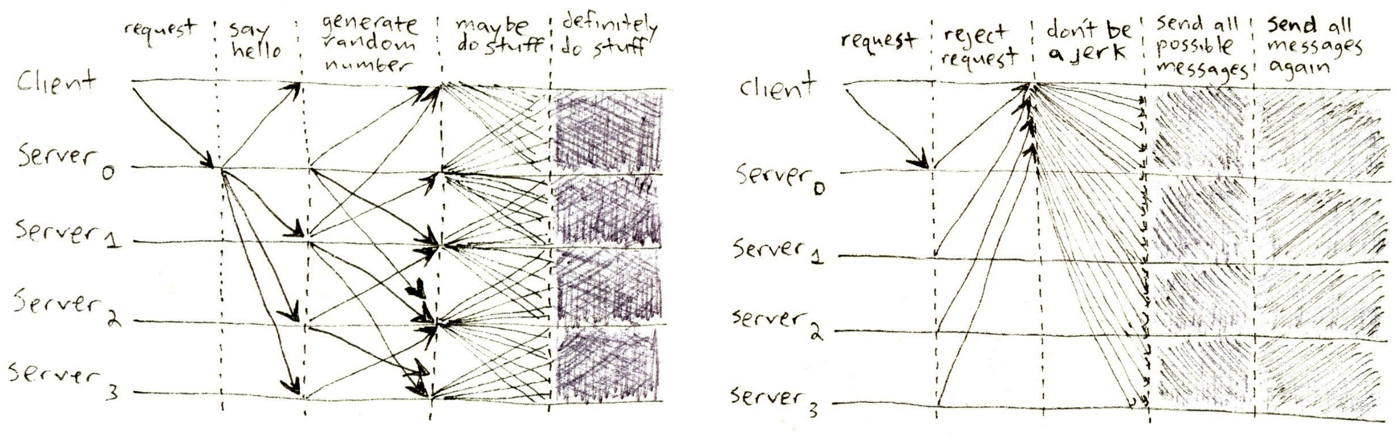


Figure 2: Our new protocol is clearly better.

RICH: I also verify that I heard that you want to go to lunch.

CHRIS: YOU DO NOT WANT TO GO TO LUNCH.

JAMES: OH NO. LET ME TELL YOU AGAIN THAT I WANT TO GO TO LUNCH.

CHRIS: YOU DO NOT WANT TO GO TO LUNCH.

BRYAN: CHRIS IS FAULTY.

CHRIS: CHRIS IS NOT FAULTY.

RICH: I VERIFY THAT BRYAN SAYS THAT CHRIS IS FAULTY.

BRYAN: I VERIFY MY VERIFICATION OF MY CLAIM THAT RICH CLAIMS THAT I KNOW CHRIS.

JAMES: I AM SO HUNGRY.

CHRIS: YOU ARE NOT HUNGRY.

RICH: I DECLARE CHRIS TO BE FAULTY.

CHRIS: I DECLARE RICH TO BE FAULTY.

JAMES: I DECLARE JAMES TO BE SLIPPING INTO A DIABETIC COMA.

RICH: I have already left for the cafeteria.

In conclusion, I think that humanity should stop publishing papers about Byzantine fault tolerance. I do not blame my fellow researchers for trying to publish in this area, in the same limited sense that I do not blame crackheads for wanting to acquire and then consume cocaine. The desire to make systems more reliable is a powerful one; unfortunately, this addiction, if left unchecked, will inescapably lead to madness and/or tech reports that contain 167 pages of diagrams and proofs. Even if we break the will of the machines with formalism and cryptography, we will never be able to put Ted inside of an encrypted, nested log, and while the datacenter burns and we frantically call Ted's pager, we will realize that Ted has already left for the cafeteria.

REGISTER TODAY

23rd USENIX Security Symposium

AUGUST 20–22, 2014 • SAN DIEGO, CA

The USENIX Security Symposium brings together researchers, practitioners, system programmers and engineers, and others interested in the latest advances in the security of computer systems and networks. The Symposium will include a 3-day technical program with refereed papers, invited talks, posters, panel discussions, and Birds-of-a-Feather sessions. Program highlights include:

Keynote Address by Phil Lapsley, author of *Exploding the Phone: The Untold Story of the Teenagers and Outlaws Who Hacked Ma Bell*

Invited Talk: "Battling Human Trafficking with Big Data" by Rolando R. Lopez, *Orphan Secure*

Panel Discussion: "The Future of Crypto: Getting from Here to Guarantees" with Daniel J. Bernstein, Matt Blaze, and Tanja Lange

Invited Talk: "Insight into the NSA's Weakening of Crypto Standards" by Joseph Menn, *Reuters*

The following co-located events will precede the Symposium on August 18–19, 2014:

EVT/WOTE '14: 2014 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections
USENIX Journal of Election Technology and Systems (JETS)
Published in conjunction with EVT/WOTE
www.usenix.org/jets

CSET '14: 7th Workshop on Cyber Security Experimentation and Test

NEW! **3GSE '14: 2014 USENIX Summit on Gaming, Games, and Gamification in Security Education**

FOCI '14: 4th USENIX Workshop on Free and Open Communications on the Internet

HotSec '14: 2014 USENIX Summit on Hot Topics in Security

HealthTech '14: 2014 USENIX Summit on Health Information Technologies
Safety, Security, Privacy, and Interoperability of Health Information Technologies

WOOT '14: 8th USENIX Workshop on Offensive Technologies

www.usenix.org/sec14

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Stay Connected...

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

Book Reviews

ELIZABETH ZWICKY

R for Everyone

Jared Lander
Pearson Education, 2014. 354 pages
ISBN 978-0-321-88803-7

R for Everyone is an introduction to using R that does not assume that you know, or want to know, a lot about programming, and concentrates on covering the range of common uses of R, from simple calculations to high-end statistics and data analysis.

In three straightforward lines of R, you can load 200,000 lines of data, calculate the minimum, maximum, mean, median, standard deviation, and quartiles for every numeric field and the most popular values with counts for the strings, and then graph them all in pairs to see how they correlate. Then you can spend 30 minutes trying to remember the arcane syntax to do the next thing you want to do, which will be easy as pie once you figure out where all the parts go. As a result, using R when manipulating numbers will not only simplify your life, it will also give you a reputation as an intellectual badass.

Of course, to get there, a guide will help. Many guides, however, are written by people who want to teach you statistics or a particular programming style, or who seem to find R intuitive, which is lucky for them but does not help the rest of us. The good and bad news is that *R for Everyone* does not want to teach you statistics. This is good news because it frees the book up to teach you R, and there are plenty of other places to learn statistics. It is bad news if you are going to feel sad when it casually mentions how to get R to produce a Poisson distribution, and you have no idea what that is. You shouldn't feel sad. Just move on, and it will be there when you need it.

Because I am more of a programmer than a statistician, I can't vouch for whether *R for Everyone* is actually sufficient for non-programmers. It is plausible that it would give a statistician enough programming background to cope, although I certainly wouldn't recommend it as a programming introduction for anyone who didn't find the prospect of easy ways to regress to the mean enticing. It is certainly gentler as a programming introduction than other R books with that aim.

It is also unusually comprehensible for an R book. I would recommend it as a first R book. You still also probably want a copy of O'Reilly's *R Cookbook*; the two books are mostly complementary, although their graphics recommendations are moderately incompatible. *R for Everyone* is more up-to-date, and the more traditional format is easier to learn from, while the *R Cookbook* is more aimed at specific problems, which makes it easier to skip through in panic looking for that missing clue.

My one complaint about *R for Everyone* is that some of the early chapters are insufficiently edited. There's some odd word usage and at least one example that is puzzlingly wrong (fortunately, it's the example of doing assignment right to left instead of left to right, which you should pretend not to know about anyway. Nobody does that; it's just confusing).

Threat Modeling: Designing for Security

Adam Shostack
John Wiley and Sons, 2014. 532 pages
ISBN 978-1-118-82269-2

This is a great book for learning to think about security in a development environment, as well as for learning to do threat modeling itself. It's a practical book, written from the point of view of an experienced practitioner, and it presents multiple approaches. (If you believe there is exactly one right way to do things, you will be annoyed. In my experience, however, people who believe in exactly one right way are people who enjoy righteous indignation, so perhaps you should read it anyway.) Also, it's written in a gently humorous style that makes it pleasant to read.

The basic problem with writing a book on threat modeling is that you have two choices. You can talk about threats, but not what you might want to do with them, which is kind of like writing an entire cookbook without ingredients lists—you get lots of techniques, sure, but you have no idea how to actually make anything. On the other hand, you can talk about threats and what to do about them, which leaves you trying to cover all of computer security in one book and still discuss threat modeling somewhere. Shostack goes for the latter approach, which is probably the better option, but it does make for a large and thinly spread book. And, because the focus is threat modeling, some great advice is buried in obscure corners.

I'd recommend this book to people new to the practice of security, or new to threat modeling, or unsatisfied with their current threat modeling practice, including people who are in other positions but are not being well served by their security people.

There are, of course, some points I disagree with or feel conflicted about. I agree that "Think like an attacker" in practice leads people down very bad roads, because they think like an attacker who also is a highly competent engineer who understands the product internals, rather than like any real attacker ever. On the other hand, there's an important kernel of truth there that needs to get through to otherwise intelligent programmers who say things like, "Oh, we don't call that interface any more, so it doesn't matter." (What matters is not what the software does when you operate it as designed, but what is possible for it to do.)

Data Protection for Photographers
Patrick H. Corrigan
Rocky Nook, 2014. 359 pages
ISBN 978-1-937538-22-4
If you are not a photographer you may be wondering why photographers would need to think specially about data protection. If you are a digital photographer, you're used to watching your disks fill and wondering how you're going to keep your pictures safe when most people's backup solutions are sized for a hundredth the amount of data you normally think about. You spend a lot of time realizing that data storage intended for homes is not going to meet the needs of your particular home. In fact, even small business systems may not suffice.

This is not a problem if your first love is system administration and photography is just your day job, but for the rest of us, it is at best annoying and at worst incomprehensible. This book will fix the incomprehensible part, explaining enough about disk and backup systems to allow photographers to make good decisions. Sadly, there's still no easy answer. You will have to choose for yourself which annoying tradeoffs to make, but at least you will make them knowingly.

I would have liked more emphasis on testing restores, and some coverage of travel options. However, this is the book you need if your data at home is outstripping the bounds of your current backup solution in either size or importance.

SAVE THE DATE!

SESA '14 **The 2014 Summit for Educators in System Administration**

NOVEMBER 11, 2014 • SEATTLE, WA

Join us at the **2014 Summit for Educators in System Administration (SESA '14),** which will be co-located with **LISA '14, the 28th Large Installation System Administration Conference.**

SESA '14 will bring together university educators to share best practices and support the creation and advancement of high-quality system administration degree programs and curricula. At SESA '14, we will discuss topics important to educators who teach system administration-related courses at universities and colleges, develop courses, and build curricula. We hope to engage all attendees to share ideas and best practices, and together create a community of sysadmin educators. System administrators who have taught courses at their university or college are also invited to join and share their experiences.

Full registration details will be available in September.

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

www.usenix.org/conference/sesa14

NOTES

USENIX Member Benefits

Members of the USENIX Association receive the following benefits:

Free subscription to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration articles, tips and techniques, practical columns on such topics as security, Perl, networks, and operating systems, book reviews, and reports of sessions at USENIXconferences.

Access to *;login:* online from December 1997 to this month: www.usenix.org/publications/login/

Access to videos from USENIX events in the first six months after the event: www.usenix.org/publications/multimedia/

Discounts on registration fees for all USENIX conferences.

Special discounts on a variety of products, books, software, and periodicals: www.usenix.org/membership/specialdisc.html.

The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.

For more information regarding membership or benefits, please see www.usenix.org/membership/ or contact office@usenix.org. Phone: 510-528-8649

USENIX Board of Directors

Communicate directly with the USENIX Board of Directors by writing to board@usenix.org.

Notice of Annual Meeting

The USENIX Association's Annual Meeting with the membership and the Board of Directors will be held on Wednesday, June 18, 2014, in Philadelphia, PA, during USENIX Federated Conferences Week, June 17–20, 2014.

Results of the Election for the USENIX Board of Directors, 2014–2016

The newly elected Board will take office at the end of the Board meeting in June 2014.

PRESIDENT
Brian Noble, *University of Michigan*

VICE PRESIDENT
John Arrasjid, *VMware*

SECRETARY
Carolyn Rowland, *National Institute of Standards and Technology (NIST)*

TREASURER
Kurt Opsahl, *Electronic Frontier Foundation*

DIRECTORS
Cat Allman, *Google*

David N. Blank-Edelman, *Northeastern University*

Daniel V. Klein, *Google*

Hakim Weatherspoon, *Cornell University*

Conference Reports

In this issue:

FAST ’14: 12th USENIX Conference on File and Storage Technologies

February 17–20, 2014 , San Jose, CA

Opening Remarks

Summarized by Rik Farrow

Bianca Schroeder (University of Toronto) opened this year’s USENIX Conference on File and Storage Technologies (FAST ’14) by telling us that we represented a record number of attendees for FAST. Additionally, 133 papers were submitted, with 24 accepted. That’s also near the record number of submissions, 137, which was set in 2012. The acceptance rate was 18%, with 12 academic, three industry, and nine collaborations in the author lists. The 28 PC members together completed 500 reviews, and most visited Toronto in December for the PC meeting.

Eno Thereska (Microsoft Research), the conference co-chair, then announced that “Log-Structured Memory for DRAM-Based Storage,” by Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout (Stanford University) had won the Best Paper award, and that Jiri Schindler (Simplivity) and Erez Zadok (Stony Brook University) would be the co-chairs of FAST ’15.

Big Memory

Summarized by Michelle Mazurek (mmazurek@cmu.edu)

Log-Structured Memory for DRAM-Based Storage

Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout, Stanford University

Awarded Best Paper!

The primary author, Stephen Rumble, was not available, so John Ousterhout presented the paper, which argued that DRAM storage systems should be log-structured, as in their previous work on RAMCloud. When building a log-structured DRAM system, an important question is how memory is allocated. Because DRAM is (relatively) expensive, high memory utilization is an important goal. Traditional operating-system allocators are non-copying; data cannot be moved after it is allocated. This results in high fragmentation and, therefore, low utilization. Instead, the authors consider a model based on garbage collection, which can consolidate memory and improve utilization. Existing garbage collectors, however, are expensive and scale poorly. They wait until a lot of free space is available (to amortize cleaning costs), which can require up to 5x over-utilization of memory. When the garbage collector does run, it can consume up to three seconds, which is slower than just resetting the system and rebuilding the RAM store from the backup log on disk.

The authors develop a new cleaning approach that avoids these problems. Because pointers in a file system are well-controlled, centrally stored, and have no circularities, it is possible to clean and copy incrementally (which would not work for a more general-purpose garbage-collection system). In the authors’ approach, the cleaner continuously finds and cleans some segments with significant free space, reducing cleaning cost and improving utilization. Further, the authors distinguish between the main log, kept in expensive DRAM with high bandwidth (targeted at 90% utilization), and the backup log, stored on disk where capacity is cheap but bandwidth is lower (targeted at 50% utilization). They use a two-level approach in which one cleaner (“compaction”) incrementally cleans one segment at a time in memory, while a second one (“combined cleaning”) less frequently cleans across segments in both memory and disk. Both cleaners run in parallel to normal operations, with limited synchronization points to avoid interference with new writes. The authors’ evaluation demonstrates that their new approach can achieve 80–90% utilization with performance degradation of only 15–20% and negligible latency overhead.

Bill Bolosky (Microsoft Research) asked how single segments cleaned via compaction can be reused before combined cleaning has occurred. Ousterhout explained that compaction creates “seglets” that can be combined into new fixed-sized chunks and allocated. A second attendee asked when cleaning occurs. Ousterhout replied that waiting as long as possible allows more space to be reclaimed at each cleaning to better amortize costs.

Strata: High-Performance Scalable Storage on Virtualized Non-Volatile Memory

Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, Geoffrey Lefebvre, Daniel Ferstay, and Andrew Warfield, Coho Data

Brendan Cully discussed the authors’ work developing an enterprise storage system that can take full advantage of fast non-volatile memory while supporting existing storage array customers who want to maintain legacy protocols (in this case, NFSv3). A key constraint is that, because flash gets consistently cheaper, enterprise customers want to wait until the last possible moment to purchase more of it, requiring the ability to add flash dynamically. The authors’ solution has three key pieces.

- Device virtualization: clients receive a virtual object name for addressing while, underneath, an rsync interface manages replication.
- Data path abstractions: for load balancing and replication, keep object descriptions in a shared database that governs paths and allows additional capacity and rebalancing, even when objects are in use.
- Protocol virtualization targeting NFSv3: the authors use a software-defined switch to control connections between clients and servers. Packets are dispatched based on their contents, allowing rebalancing without changes to the client. All nodes have the same IP and MAC address, so they appear to the client to be one node.

To evaluate the system, the authors set up a test lab that can scale from 2 to 24 nodes (the capacity of the switch) and rebalance dynamically. As nodes are added, momentary drops in IOPS are observed because nodes must both serve clients and support rebalancing operations. Overall, however, performance increases, but not linearly; the deviation from linear comes because, as more nodes are added, the probability of nodes doing remote I/O on behalf of clients (slower than local I/O) increases. This effect can be mitigated by controlling how clients and objects are assigned to promote locality; with this approach, the scaling is much closer to linear. CPU usage for this system is very high, so introduction of 10-core machines improves IOPS significantly.

An attendee asked whether requests can always be mapped to a node that holds an object, avoiding remote I/O. Cully responded that clients can't be moved on a per-request basis, and they will ask for objects that may live on different nodes. Niraj Tolia of Maginatics asked how the system deals with multiple writers. Cully responded that currently only one client can open the file for writing at a time, which prevents write conflicts but requires consecutive writers to wait; multiple readers are allowed.

Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches

Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chiu, IBM Almaden Research Center

Hyojun Kim presented a measurement study of a prototype SSD with 45 nm 1 GB phase-change memory. PCM melts and cools material to store bits: amorphous = reset, crystalline = set. Writing set bits takes longer. Write latency is typically reported at 150 ns, which is more than 1000x faster than flash and only 3x slower than DRAM, but which does not count 50 ns of additional circuit time. Write throughput is limited (4 bits per pulse) by the chip's power budget for heating the material. Kim directly compared PCM with flash, normalized for throughput, using Red Hat, a workload generator, and a statistics collector. They found that for read latency, PCM is 16x faster on average and also has much faster maximums. For write latency, however, PCM is 3.4x slower on average, with a maximum latency of 378 ms compared to 17.2 ms for flash.

The second half of the talk described how simulation was applied to assess how and whether PCM is useful for enterprise storage systems. First, the authors simulated a multi-tiered system that writes hot data to flash or PCM and cold data to a cheap hard drive, incorporating the following relative price assumptions: PCM = 24, flash = 6, disk = 1 per unit of storage. They evaluated a variety of system combinations using x% PCM, y% flash, and z% disk, based on a one-week trace from a retail store in June 2012, and measured the resulting performance in IOPS/unit cost. Using an ideal static placement based on knowing the workload traces a priori, the optimal combination was 30% PCM, 67% flash, and 3% disk. With reactive data movement based on I/O traffic (a more realistic option), the ideal combination was 22% PCM, 78% flash. A second simulation used flash or PCM as application-server-side, write-through, and LRU caching, using a 24-hour trace from customer production systems (manufacturing, media, and medical companies). The results measured average read latency. To include cost in this simulation, different combinations of flash and PCM with the same IOPS/cost were simulated. For manufacturing, the best results at three cost points were 64 GB of flash alone, 128 GB of flash alone, and 32 GB PCM + 128 GB flash. In summary, PCM has promise for storage when used correctly, but it's important to choose accurate real-world performance numbers.

One attendee asked whether the measurements had considered endurance (e.g., flash wearout) concerns as well as IOPS. Kim agreed that it's an important consideration but not one that was measured in this work. Someone from UCSD asked whether write performance for PCM is unfairly disadvantaged unless capacity-per-physical-size issues are also considered. Kim agreed that this could be an important and difficult tradeoff. A third attendee asked about including DRAM write buffers in PCM, as is done with flash. Kim agreed that the distinction is an important one and can be considered unfair to PCM, but that the current work attempted to measure what is currently available with PCM—write buffers may be available in the future. A final attendee asked for clarification about the IOPS/unit cost metric; Kim explained that it's a normalized relative metric alternative to capturing cost explicitly in dollars.

Flash and SSDs

Summarized by Jeremy C. W. Chan (cwchan@cse.cuhk.edu.hk)

Wear Unleveling: Improving NAND Flash Lifetime by Balancing Page Endurance

Xavier Jimenez, David Novo, and Paolo Ienne, Ecole Polytechnique Fédérale de Lausanne (EPFL)

Xavier Jimenez presented a technique to extend the lifetime of NAND flash. The idea is based on the observation that inside a block of a NAND flash, some pages wear out faster than others. As a result, the endurance of a block is determined by the weakest page.

To improve the lifetime, Xavier and his team introduced a fourth page state, "relieved," to indicate pages not to be programmed during a Program/Erase (P/E) cycle. By measuring the bit error rate (BER), they showed that relieved pages possess higher endurance than unrelieved ones. Xavier continued about how relief can be performed in the case of a multi-level cell (MLC). In MLC, each one of the two bits is mapped to a different page, forming an LSB and MSB page pair. A full relief means both pages are skipped in a P/E cycle, while a half relief means only the MSB is relieved. Although a half relief produces higher relative wear, it is more effective in terms of written bits per cycle.

Xavier then described two strategies on how to identify weak pages. The simple strategy is reactive, which starts relieving a page when the BER reaches a certain threshold. However, the reactive approach requires the FTL to read a whole block before erasing it, which adds an overhead to the erasing time. Therefore, the authors further proposed the proactive strategy, which predicts the number of times weak pages should be relieved to match the weakest page's extended endurance by first characterizing the endurance of the LSB/MSB pages at every position of the block. The proactive strategy is used together with adaptive planning, which predefines a number of lookup tables called plans. The plan provides the probability for each page to be relieved.

In the evaluation, Xavier and his team implemented the proactive strategy on two previously proposed FTLs, ROSE and ComboFTL, and on two kinds of 30 nm class chips, C1 and C2. C1 is an ABL chip with less interference, whereas C2 is an interleaved chip, which is faster and more flexible. The evaluation on real-world traces shows that the proactive strategy improves lifetime by 3–6% on C1 and 44–48% on C2. Only a small difference is observed in execution time because of the efficient half relief operation.

Alireza Haghdoost (University of Minnesota) asked whether Xavier's approach is applicable to a block-level FTL. Xavier explained that block-level relieving is impractical and that the evaluation is entirely based on page-level mapping. Steve Swanson (UCSD) asked if page skipping would affect the performance of reading the neighboring pages. Xavier said the endurance of the neighboring pages will be decreased by at most 2%, and because they are the stronger pages, the impact is minimal on the block's lifetime.

Lifetime Improvement of NAND Flash-Based Storage Systems Using Dynamic Program and Erase Scaling

Jaeyong Jeong and Sangwook Shane Hahn, Seoul National University; Sungjin Lee, MIT/CSAIL; Jihong Kim, Seoul National University

Jaeyong Jeong presented a system-level approach called dynamic program and erase scaling (DPES) to improve the lifetime of NAND flash-based storage systems. The approach exploits the fact that the erasure voltage and the erase time affect the endurance of NAND flash memory.

Jaeyong began the presentation with an analogy illustrated by interesting cartoons. In the analogy, NAND flash is a sheet of paper, the program action is writing on the paper with a pencil, and the erase action is using an eraser to clear out the word for the whole page. Finally, the flash translation layer (FTL) is a person called Flashman. With this analogy, Jaeyong explained why NAND endurance had decreased by 35% during the past two years despite the 100% increase in capacity. He said that advanced semiconductor technology is just like a thinner piece of paper, which wears down more easily than a thick piece of paper after a certain number of erasure cycles. However, low erase voltage and long erase time are the two main keys to improving the endurance of NAND flash.

The fundamental tradeoff between erase voltage and program time is that the lower the erase voltage, the longer the program time required. With this observation, Jaeyong and his team proposed the DPES approach, which dynamically changes the program and erase voltage/time to improve the NAND endurance while minimizing negative impact on throughput.

They implemented their idea on an FTL and called it AutoFTL. It consists of a DPES manager, which selects the program time, erase speed, and erase voltage according to the utilization of an internal circular buffer. For instance, a fast write mode is selected to free up buffer space when its utilization is high. In the evaluation, Jaeyong and his team chose six volumes with different inter-arrival times from the MSR Cambridge traces. On average, AutoFTL achieves a 69% gain on the endurance of the NAND flash with only negligible impact (2.2%) on the overall write throughput.

Geoff Kuenning (Harvey Mudd College) asked why high voltage causes electrons to get trapped in the oxide layer. Jaeyong reemphasized that the depletion of the tunnel side has an exponential relationship to the erase voltage and time. Yitzhak Birk (Technion) said that the approach of programming in small steps works but may bring adverse effects to the neighboring cells. Peter Desnoyers (Northeastern University) asked how they manage to select the appropriate reading method according to the voltage level applied. Jaeyong replied that the lookup table would be able to track the voltage level. Peter followed up that a scan for pages is not possible because you cannot read a page before knowing the voltage level.

ReconFS: A Reconstructable File System on Flash Storage

Youyou Lu, Jiwu Shu, and Wei Wang, Tsinghua University

Youyou Lu began with the novelty of ReconFS in metadata management of hierarchical file systems. This work addresses a major challenge in namespace management of file systems on solid-state drives (SSDs), which are the scattered small updates and intensive writeback required to maintain a hierarchical namespace with consistency and persistence. These writes cause write amplification that seriously hurts the lifetime of SSDs. Based on the observation that modern SSDs have high read bandwidth and IOPS and that the page out-of-band (OOB) area provides some extra space for page management, Youyou and his team built the ReconFS, which decouples maintenance of volatile and persistent directory trees to mitigate the overhead caused by scattered metadata writes.

Youyou presented the core design of ReconFS on metadata management. In ReconFS, a volatile directory tree exists in memory, which provides hierarchical namespace access, and a persistent directory tree exists on disk, which allows reconstruction after system crashes. The four triggering conditions for namespace metadata writeback are: (1) cache eviction, (2) checkpoint, (3) consistency preservation, and (4) persistence maintenance. Although a simple home-location update is used for cache eviction and checkpoint, ReconFS proposes an embedded inverted index for consistency preservation and metadata persistence logging for persistence maintenance. Inverted indexing in ReconFS is placed in the log record and the page OOB depending on the type of link. The key objective of inverted indexing is to make the data pages self-described. Meanwhile, ReconFS writes back changes to directory tree content to a log to allow recovery of a directory tree after system crashes. Together with unindexed zone tracking, ReconFS is able to reconstruct the volatile directory tree in both normal and unexpected failures.

To evaluate the ReconFS prototype, Youyou and his team implemented a prototype based on ext2 on Linux. Using filebench to simulate a metadata-intensive workload, they showed that ReconFS achieves nearly the best throughput among all evaluated file systems. In particular, ReconFS improves performance by up to 46.3% in the varmail workload. Also, the embedded inverted index and metadata persistence logging enabled ReconFS to give a write reduction of 27.1% compared to ext2.

Questions were taken offline because of session time constraints.

Conference Luncheon and Awards

Summarized by Rik Farrow

During the conference luncheon, two awards were announced. The first was the FAST Test of Time award, for work that appeared at a FAST conference and continues to have a lasting impact. Nimrod Megiddo and Dharmendra S. Modha of IBM Almaden Research Center won this year's award for "ARC: A Self-Tuning, Low Overhead Replacement Cache" (https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache).

The IEEE Reynolds and Johnson award went to John Ousterhout and Mendel Rosenblum, both of Stanford University, for their paper "The Design and Implementation of a Log-Structured File System" (http://www.stanford.edu/~ouster/cgi-bin/papers/lfs.pdf). Rosenblum commented that he was Ousterhout's student at UC Berkeley when the paper was written. Later, Rosenblum wound up working with Ousterhout at Stanford.

Personal and Mobile

Summarized by Kuei Sun (kuei.sun@utoronto.ca)

Toward Strong, Usable Access Control for Shared Distributed Data

Michelle L. Mazurek, Yuan Liang, William Melicher, Manya Sleeper, Lujo Bauer, Gregory R. Ganger, and Nitin Gupta, Carnegie Mellon University; Michael K. Reiter, University of North Carolina

Michelle Mazurek began her presentation by showing us recent events where improper access control led to mayhem and privacy invasions. The main issue is that access control is difficult, especially for non-expert users. In their previous work, the authors identified users' need for flexible policy primitives, principled security, and semantic policies (e.g., tags). To this end, they based the design of their system on two important concepts: tags, which allow users to group contents, and logical proof, which allows for fine-grained control and flexible policy. For every content access, a series of challenges and proofs needs to be made before access is granted. On each device participating in the system, a reference monitor exists to protect the content that the device owns, a device agent that performs remote proofs for enabling content transfer across the network, as well as user agents that construct proofs on behalf of the users. Michelle walked us through an example of how Bob could remotely access a photo of Alice on a remote device in this system. Michelle then described the authors' design of strong tags. Tags are first-class objects, such that access to them is independent of content access. To prevent forging, tags are cryptographically signed.

In their implementation, the authors mapped system calls to challenges. They cached recently granted permissions so that the same proof would not need to be made twice. In their evaluation, they wrote detailed policies drawn from user studies using their policy languages, all of which could be encoded in their implementation and showed that their logic had sufficient expressiveness to meet user needs. They simulated access patterns because they do not have a user study based on the perspective of the user accessing content. They ran two sets of experimental setups on their prototype system: one with a default-share user and the other with a default-protect user. The main objective of the experiments was to measure latency for system calls and see whether they were low enough for interactive users. The results showed that with the exception of readdir(), system calls fell well below the 100 ms limit that they set. The authors also showed that access control only accounted for about 5% of the total overhead. Finally, it took approximately 9 ms to show that no proof could be made (access denied!), although variance in this case can be quite high.

Tiratat Patana-anake (University of Chicago) wanted clarification on the tags. Michelle explained that you only need access to the tags required for access to the file. Someone from University of California, Santa Cruz, wanted to know which cryptographic algorithm was used, what its overhead was, and which distributed file system was used. Michelle said the proofer uses the crypto library from Java and that it doesn't add too much overhead. The distributed file system was homemade. Another person from UCSC asked whether the authors intended to replace POSIX permission standards, and the response was yes. He went on to ask whether they have reasonable defaults because most users are lazy. Michelle first explained that their user study indicates a broad disagreement among users on what they want from such a system. However, she agreed that users are generally lazy so more research into automated tagging and a better user interface would be helpful. Finally, someone asked about transitivity, where one person would take restricted content and give it to an unauthorized user. Michelle believed that there was no solution to the problem where the person to whom you grant access is not trustworthy.

On the Energy Overhead of Mobile Storage Systems

Jing Li, University of California, San Diego; Anirudh Badam and Ranveer Chandra, Microsoft Research; Steven Swanson, University of California, San Diego; Bruce Worthington and Qi Zhang, Microsoft

Jing Li began the talk by arguing that in spite of the low energy overhead of storage devices, the overhead of the full storage stack on mobile devices is actually enormous. He presented the authors' analysis of the energy consumption used by components of the storage stack on two mobile devices: an Android phone and a Windows tablet. After giving the details of the experimental setup, he showed the microbenchmark results, which revealed that the energy overhead of storage stack is 100 to 1000x higher than the energy consumed by the storage device alone. He then focused on the CPU's busy time, which showed that 42.1% of the busy time is spent in encryption APIs while another 25.8% is spent in VM-related APIs.

Li and his team first investigated the true cost of data encryption by comparing energy consumption between devices with and without encryption. Surprisingly, having encryption costs on average 2.5x more energy. Next, Li gave a short review of the benefits of isolation between applications and of using managed languages. He then showed the energy overhead for using managed languages, which is anywhere between 12.6% and 102.1% (for Dalvik on Android!). Li ended his talk with some suggestions. First, storage virtualization can be moved into the storage hardware. Second, some files, such as the OS library, do not need to be encrypted. Therefore, a partially encrypted file system would help reduce the energy overhead. Finally, hardware-based solutions (e.g., DVFS or ASIC) can be used to support encryption or hardware virtualization while keeping energy cost low.

Yonge (University of California, Santa Cruz) asked how the energy impact of DRAM was measured. Li said that they ran the benchmark to collect the I/O trace. They then replayed the I/O trace to obtain the energy overhead of the storage stack. As such, the idle power was absent from the obtained results.

ViewBox: Integrating Local File Systems with Cloud Storage Services

Yupu Zhang, University of Wisconsin–Madison; Chris Dragga, University of Wisconsin–Madison and NetApp; Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin–Madison

Yupu Zhang started by reminding us that cloud storage services are gaining popularity because of promising benefits such as reliability, automated synchronization, and ease of access. However, these systems can fail to keep data safe when local corruption or a crash arises, which causes bad data to be propagated to the cloud. Furthermore, because files are uploaded out of order, the cloud may sometimes have an inconsistent set of data. To provide a strong guarantee that the local file system's state is equal to the cloud's state, and that both states are correct, the authors developed ViewBox, which integrates the file system with the cloud storage service. ViewBox employs checksums to detect problems with local data, and utilizes cloud data to recover from local failures. ViewBox also keeps in-memory snapshots of valid file system state to ensure consistency between the local file system and the cloud.

Yupu presented the results of detailed experimentation, which revealed the shortcomings of the current setup. In their first experiment, the authors corrupted data beneath the file system to see whether it was propagated to the cloud. ZFS detected all corruptions because it performed data checksumming, but services running on top of ext4 propagated all corruptions to the cloud. In the second experiment, they emulated a crash during synchronization. Their results showed that without enabling data journaling on the local file system, the synchronization services would behave mostly erratically. All three services that the authors tested violated causal ordering that is locally enforced by fsync().

Next, Yupu gave an overview of the architecture. They modified ext4 to add checksums. Upon detecting corrupt data, ext4-cksum can communicate with a user daemon named Cloud Helper via ioctl() to fetch correct data from the cloud. After a crash, the user is given the choice of either recovering inconsistent files individually or rolling back the entire file system to the last synchronized view. The View Manager, on the other hand, creates consistent file system views and uploads them to the cloud. To provide consistency efficiently, they implemented two features: cloud journaling and incremental snapshotting. The basic concept of cloud journaling is to treat the cloud as an external journal by synchronizing local changes to the cloud at file system epochs. View Manager would continuously upload the last file system snapshot in memory to the cloud. Upon failure, it would roll the file system back to the latest synchronized view. Incremental snapshotting allows for efficient freezing of the current view by logging namespace and data changes in memory. When the file system reaches the next epoch, it will update the previous frozen view without having to interrupt the next active view.

In their evaluation, the authors showed that ViewBox could correctly handle all the types of error mentioned earlier. ViewBox has a runtime overhead of less than 5% and a memory overhead of less than 20 MB, whereas the workload contains more than 1 GB of data.

Someone asked how much they roll back after detecting corruption. Yupu responded that if you care about the whole file system, ideally, you roll back the whole file system, but generally you just roll back individual files. The same person asked what happens if an application is operating on a local copy that gets rolled back. Yupu said that the application would have to be aware of the rollback. Mark Lillibridge (HP) suggested that a correctly written application should make a copy, change that copy, and then rename the copy to avoid problems like this. Yupu agreed.

RAID and Erasure Codes

Summarized by Yue Cheng (yuec@vt.edu)

CRAID: Online RAID Upgrades Using Dynamic Hot Data Reorganization

Alberto Miranda, Barcelona Supercomputing Center (BSC-CNS); Toni Cortes, Barcelona Supercomputing Center (BSC-CNS) and Technical University of Catalonia (UPC)

Alberto Miranda presented an economical yet effective approach for performing RAID upgrading. The authors' work is motivated by the fact that storage rebalancing caused by degraded uniformity is way too expensive. Miranda proposed CRAID, a data distribution strategy that redistributes only hot data when performing RAID upgrading by using a dedicated small partition in each device as a persistent disk-based cache.

Taking advantage of the I/O access pattern monitoring that is used to keep track of data access statistics, an I/O redirector can strategically redistribute/rebalance only frequently used data to the appropriate partitions. The realtime monitoring provided by CRAID guarantees that a newly added disk will be used as long as it is added. Statistics of data access patterns are also used to effectively reduce the cost of data migration. The minor disadvantage of this mechanism is that the invalidation on the cache partition results in the loss of all the previous computation. Trace-based simulations conducted on their prototype showed that CRAID could achieve competitive performance with only 1.28% of all available storage capacity, compared to two alternative approaches.

Someone asked Alberto to comment on one of the workload characteristics that can impact the rebalancing algorithm. Miranda said that the available traces they could find were limited.

STAIR Codes: A General Family of Erasure Codes for Tolerating Device and Sector Failures in Practical Storage Systems

Mingqiang Li and Patrick P. C. Lee, The Chinese University of Hong Kong

Patrick Lee presented STAIR codes, a set of novel erasure codes that can efficiently tolerate failures in both device and sector levels. What motivates STAIR is that traditional RAID and erasure codes use multiple parity disks/parity sectors (the cost of which is prohibitive) to provide either device-level or sector-level tolerance. They proposed a general (without any restriction) and space-efficient family of erasure codes that can tolerate simultaneous device and sector failures.

The key idea of STAIR codes is to base protection against sector failure on a pattern of how sector failures occur, instead of setting a limit on tolerable sector failures. The actual code structure builds based on two encoding phases, each of which builds on any MDS code that works as long as the parameters support the code. The interesting part of their approach is the upstairs and downstairs encoding that can reuse computed parity results, thus providing space efficiency and complementary performance advantages. Evaluation results showed that STAIR codes improve encoding by up to 100% while achieving storage space efficiency. Their open-sourced coding library can be found at: http://ansrlab.cse.cuhk.edu.hk/software/stair.

Someone asked whether the authors managed to measure the overhead (reading the full stripe) of some special (extra) sectors' encoding every time they had to be recreated in a RAID device. Patrick said their solution needed to read the full stripe from the RAID so as to perform the upstairs and downstairs encoding. Umesh Maheshwari (Nimble Storage) asked whether the scenario where errors come in a burst was an assumption or a case that STAIR took care of; his concern was that in SSD the errors might end up in a random distribution. Patrick said the error burst case was an issue they were trying to address.

Parity Logging with Reserved Space: Towards Efficient Updates and Recovery in Erasure-Coded Clustered Storage

Jeremy C. W. Chan, Qian Ding, Patrick P. C. Lee, and Helen H. W. Chan, The Chinese University of Hong Kong

Patrick Lee also presented CodFS, an erasure-coded clustered storage system prototype that can achieve both high update and recovery performance. To reduce storage costs and footprint, enterprise-scale storage clusters now use erasure-coded storage rather than replication mechanisms that incur huge overheads. However, the issues faced by erasure coded storage systems are that updates are too costly and recovery is expensive as well. To deal with this, Patrick and his students propose a parity-logging scheme with reserved space that adopts a hybrid in-place and log-based update mechanism with adaptive space readjustment.

They studied two real-world storage traces and, based on the observed update characteristics, propose a novel delta-based parity logging with reserved space (PLR) mechanism that reduces disk seeks by keeping each parity chunk and its parity delta next to each other with additional space, the capacity of which can be dynamically adjusted. The challenges lie in how much reserved space is most economical and the timing of reclaiming unused reserved space.

In-place updates are basically overwriting existing data and parity chunks while log-based updates are appending changes by converting random writes to sequential writes. Their hybrid approach smartly maintains the advantages of the above two update schemes while mitigating the problems that might occur.

Konstantin Shvachko (WANdisco) asked how restrictions on I/O patterns (i.e., cluster storage systems that only support sequential reads/writes) affect the performance of workloads. Patrick said performance purely depends on workload types, and their work looked in particular at server workload that they attempted to port into cluster storage systems. Brent Welch (Google) mentioned two observations: (1) real-world workloads might consist of lots of big writes due to the fact that there are many big files distributed in storage; (2) an Object Storage Device layer is unaware of data types and is decoupled from the lower layer file system. Patrick agreed with these observations and said (1) people can choose a different coding scheme based on the segments, and (2) the decoupling problem remains an issue left to be explored in future work.

Poster Session I

Summarized by Sonam Mandal (somandal@cs.stonybrook.edu)

In-Stream Big Data Processing Platform for Enterprise Storage Management

Yang Song, Ramani R. Routray, and Sandeep Gopisetty, IBM Research

This poster presents an approach for in-stream processing of Big Data, which is becoming increasingly important because of the information explosion and subsequent escalating demand for storage capacity. The authors use Cassandra as their storage, Hadoop/HDFS for computation, RHadoop as the Machine Learning algorithm, and IBM InfoSphere Streams for their use case example. They implemented many ensemble algorithms like Weighted Linear Regression with LASSO, Weighted Generalized Linear Regression (Poisson), Support Vector Regression, Neural Networks, and Random Forest. They use their technique to identify outliers in 2.2 million backup jobs each day for jobs having 20+ metrics. They try to identify anomalies using the Contextual Local Outlier Factor algorithm before storing on HDFS/Cassandra. To do so, they leverage backup-specific domain knowledge and have shown the results of their outlier detection experiment in the poster.

Page Replacement Algorithm with Lazy Migration for Hybrid PCM and DRAM Memory Architecture

Minho Lee, Dong Hyun Kang, Junghoon Kim, and Young Ik Eom, Sungkyunkwan University

The authors came up with a page replacement algorithm to benefit hybrid memory systems using PCM by reducing the number of write operations to them. PCM is non-volatile and has in-place update and byte-addressable memory with low read latency. PCM suffers from a low endurance problem, where only a million writes are possible before it wears out, and the write latency of PCM is much slower than its read latency. PCM reduces the overhead of migration by using lazy migration. When a page fault occurs, it is always allocated in DRAM regardless of whether it was a read or write operation; and, when the DRAM fills up, pages in the DRAM are migrated to the PCM. When a write operation occurs on a page in the PCM, it attempts to migrate this page to DRAM. If there is no free space in the PCM, a page is evicted according to rules from the CLOCK algorithm.

The authors' results show that they obtained a high hit ratio regardless of PCM size in hybrid memory architectures. They reduced the number of PCM writes by up to 75% compared to state of the art algorithms and by 40% compared to the CLOCK algorithm.

On the Fly Automated Storage Tiering

Satoshi Iwata, Kazuichi Oe, Takeo Honda, and Motoyuki Kawaba, Fujitsu Laboratories

The authors present issues with existing storage-tiering techniques and propose their own technique to overcome the shortcomings of existing approaches. Storage tiering combines fast SSDs with slow HDDs such that hot data is kept on SSDs and cold data is stored on HDDs. Less frequently accessed data in SSDs is swapped out at predefined intervals to follow changes in workloads. Existing methods have difficulty following workload changes quickly as the migration interval cannot be reduced too far. Shorter intervals lead to lower I/O performance due to more disk bandwidth consumed by migration.

The authors propose an on-the-fly agreed service time (AST) to follow any workload changes within minutes, instead of the granularity of hours or a day with previous methods. Less frequently accessed data is filtered out from migration candidates, thus decreasing bandwidth consumption, even though it results in unoccupied SSD storage space. They use a two-stage migration-filter approach. The first stage filters out hotter but not very hot segments by checking access concentrations. The second filters out segments for which the hot duration is not long enough.

When 60% of the total I/O is sent to fewer than 20 segments (approximately 10% of data size), then these segments are marked as candidates. When a segment has been marked as a candidate three times in a row, it is migrated to the SSD. The authors' evaluation results back their claims and show that workload changes can be followed in a matter of minutes using their approach.

SSD-Tailor: Customization System for Enterprise SSDs

Hyunchan Park, Youngpil Kim, Cheol-Ho Hong, and Chuck Yoo, Korea University; Hanchan Jo, Samsung Electronics

This poster presents SSD-Tailor, a customization system for SSDs. With the increasing need to satisfy customers for requirements of high performance and reliability for various workloads, it becomes difficult to design an optimal system with such a large number of potential configuration choices. SSD-Tailor determines a near-optimal design for a particular workload of an enterprise server. It requires three inputs: customer requirements, workload traces, and design options. It has three components: Design Space Explorer, Trace-driven SSD Simulator, and Fitness Analyzer, which work together iteratively in a loop to produce a near-optimal design for the given requirements and workload traces.

Design options consist of a type of flash chip, FTL policies, etc. Customer requirements may include low cost, high performance, high reliability, low energy consumption, and so on. Workload traces are full traces rather than extracted profiles.

The Design Space Explorer used genetic algorithms to find near-optimal SSD design. Genetic algorithms mimic the process of natural selection. The Trace-driven SSD Simulator (the authors used DiskSim) can help change and display the design options easily, because in spite of a reduced set of design options, too many still need to be analyzed. The Fitness Analyzer evaluates the simulation results based on scores.

The authors show that tailoring overhead is high but needs to be done only once. The benefits obtained are quite high according to their results. They compared SSD-Tailor with a brute force algorithm as their baseline.

Multi-Modal Content Defined Chunking for Data Deduplication

Jiansheng Wei, Junhua Zhu, and Yong Li, Huawei

The authors of this paper have come up with a deduplication mechanism based on file sizes and compressibility information. They identified that many file types such as mp3 and jpeg are large, hardly modified, and often replicated as is; such files should have large chunk sizes to reduce metadata volume without sacrificing too much in deduplication ratio. Other file types consist of highly compressible files, some of which are modified frequently, and these benefit from having small chunk sizes to maximize deduplication ratio.

The authors propose two methods, both of which require a preprocessing step of creating a table for size range, compressibility range, and the expected chunk size. The first method divides data objects into fixed-sized blocks and estimates their compression ratio using sampling techniques. Adjacent blocks with similar compression ratios are merged into segments. Segments are divided into chunks using content-defined chunking techniques, and these chunk boundaries may override segment boundaries. Then the chunk fingerprints are calculated.

In the second approach, many candidate chunking schemes using Content Defined Chunking (CDC) with different expected chunk sizes are generated in a single scan. One chunking scheme is used to calculate the compression ratio of its chunks, and chunks with similar compression ratios are merged together. These chunking results are directly used and their fingerprints are calculated. Their experimental results show that their Multi-Modal CDC can reduce the number of chunks by 29.1% to 92.4%.

Content-Defined Chunking for CPU-GPU Heterogeneous Environments

Ryo Matsumiya, The University of Electro-Communications; Kazushi Takahashi, Yoshihiro Oyama, and Osamu Tatebe, University of Tsukuba and JST, CREST

Chunking is an essential operation in deduplication systems, and Content-Defined Chunking (CDC) is used to divide a file into variable-sized chunks. CDC is slow as it calculates many fingerprints. The authors of this poster came up with parallelizing approaches that use both GPU and CPU to chunk a given file. Because of the difference in speed of CPU and GPU, the challenge becomes that of task scheduling. They propose two methods, Static Task Scheduling and Dynamic Task Scheduling, to efficiently use both the GPU and CPU.

Static Task Scheduling uses a user-defined parameter to determine the ratio of dividing the file such that one part is assigned to the GPU and the other part to the CPU. Each section is further divided into subsections, which are each assigned to a GPU thread or CPU thread.

Dynamic Task Scheduling consists of an initial master thread, which divides a file into distinct, fixed-sized parts called large sections. Each is assigned to a GPU thread. For detection of chunk boundaries lying across more than one segment, small subsections are created, including data parts across boundaries of large sections. Each small subsection is assigned to the CPU. Worker threads are created for GPU and CPU to handle these sections. While the task queue is not empty, each worker will perform CDC; if a queue becomes empty, then a worker will steal tasks from another worker's queue.

The authors ran experiments to find the throughput of their static and dynamic methods and compared them to CPU-only and GPU-only methods. They showed that static performs the best and dynamic follows closely behind it. The benefit of having a dynamic method is to avoid tuning parameters as is required for the static method.

Hash-Cast: A Dark Corner of Stochastic Fairness

Ming Chen, Stony Brook University; Dean Hildebrand, IBM Research; Geoff Kuenning, Harvey Mudd College; Soujanya Shankaranarayana, Stony Brook University; Vasily Tarasov, IBM Research; Arun O. Vasudevan, Stony Brook University; Erez Zadok, Stony Brook University; Ksenia Zakirova, Harvey Mudd College

This poster uncovers Hash-Cast, a networking problem that causes identical NFS clients to get unfair shares of the network bandwidth when reading data from an NFS server. Hash-Cast is a dark corner of stochastic fairness where data-intensive TCP flows are randomly hashed to a small number of physical transmit queues of NICs and hash values collide frequently. Hash-Cast influences not only NFS but also any storage servers hosting concurrent data-intensive TCP streams, such as file servers, video servers, and parallel file system servers. Hash-Cast is related to the bufferbloat problem, a phenomenon in which excessive network buffering causes unnecessary latency and poor system performance. The poster also presents a method to work around Hash-Cast by changing the default TCP congestion control algorithm from TCP CUBIC to TCP VEGAS, another algorithm that alleviates the bufferbloat problem.

MapReduce on a Virtual Cluster: From an I/O Virtualization Perspective

Sewoog Kim, Seungjae Baek, and Jongmoo Choi, Dankook University; Donghee Lee, University of Seoul; Sam H. Noh, Hongik University

The authors of this poster analyze two main questions with respect to the MapReduce framework making use of virtualized environments. They try to analyze whether Hadoop runs efficiently on virtual clusters and whether any I/O performance degradation is seen, then whether they can be mitigated by exploiting the characteristics of I/O access patterns observed in MapReduce algorithms. They ran a Terasort benchmark and found that the I/O is triggered in a bursty manner, requested intensively for a short period, and sharply increased and decreased. Some phases utilize a memory buffer. Virtual machines share I/O devices in a virtual cluster; thus, this bursty I/O may cause I/O interference among VMs. When VMs request bursty I/Os concurrently, the I/O bandwidth suffers a performance drop of about 31%, going from 1 to 4 VMs. Additionally, long seek distance and high context switch overheads exist among VMs.

To help mitigate this issue, the authors propose a new I/O scheduler for Hadoop on a virtual cluster, which minimizes the I/O interference among VMs and also exploits the I/O burstiness in MapReduce applications. Their new I/O scheduler controls bandwidth of VMs using Cgroups-blkio systems and operate at a higher layer than the existing scheduler. The Burstiness Monitor detects bursty I/O requests from each VM. The Coarse-grained Scheduler allows a bursty VM to use the I/O bandwidth exclusively for a time quantum in round-robin manner. This allows the overhead caused by context switching among multiple VMs to be reduced, along with reducing overall seek distance and execution time. Their experimental results verify the performance gains using their approach.

Keynote Presentation

FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers

Krste Asanović, University of California, Berkeley

Summarized by Tiratat Patana-anake (tiratatp@uchicago.edu)

Krste Asanović told a story about the past, present, and future of Warehouse Scale Computing (WSC), which has many applications but will gain popularity because of the migration to the extreme, with the cloud backing all devices. We moved from commercial off-the-shelf (COTS) servers to COTS parts, and we'll move to custom everything, said Asanović.

Asanović stated that the programming model for WSC needs to change. Currently, the silo model is used, but he believes that Service Oriented Architecture (SOA) is better. In SOA, each component is a service that connects via network, and each application is composed of many services. The benefits of SOA are reusability and ease of management. Moreover, by decomposing big software into small services, the services are easier to tailor to each user subset. A statistic shows that small (less than $1 million) projects have higher success rates than big projects.

Asanović then compared old wisdom to new wisdom in many related aspects. In the old wisdom, we cared only about building fault-tolerant systems. Today, we need to care about tail-latency tolerance, too, which means building predictable parts from less predictable ones. Many techniques can be used to build a tail-tolerant service. We can use software to reduce component variation by using different queues or breaking up tasks into small parts, or try to cope with variability by hedging requests when the first request result was slow, for example, or by trying requests in different queues. We can also try to improve the hardware by reducing overhead, reducing queuing, increasing network bisection bandwidth, or using partitionable resources.

The second thing that Asanović compared was the memory hierarchy. In the old days, we had DRAM, disk, and tape. Now, we have DRAM, NVRAM, and then disk. In the future, we will have a new kind of NVM that has DRAM read latency and endurance. In terms of memory hierarchy, there will be more levels in the memory hierarchy, and we might see a merging between high-capacity DRAM and flash memory into something new such as PCM.

Third, Moore's Law is dead (for logic, SRAM, DRAM, and likely 2D flash), said Asanović. The takeaway is that we have to live with this technology for a long time, and improvements in system capability will come from above the transistor level. More importantly, without Moore's Law and scaling, the cost of custom chips will come down because of the amortized cost of technology.

Fourth, Asanović discussed which ISA (Instruction Set Architecture) was better, ARM or x86. He said the real important difference was that we could build a custom chip with ARM, not Intel. He added that ISA should be an open industry standard. The goal is to have an open source chip design, such as Berkeley's RISC-V, which is an open ISA.

Moreover, Asanović said that security is very important. The key is to have all data encrypted at all times. He also said that rather than using shared memory to do inter-socket communication, message passing has won the war, which is also a better match for SOA.

Next, Asanović presented the FireBox, which is a prototype of 2020 WSC design. It is a custom "Supercomputer" for interactive and batch application that can support fault and tail tolerance, will have 1000 SoC, 1 terabit/s high radix photonic switch, and 1000 NVM modules for 100 PB total. FireBox SoC will have 100 homogeneous cores per SoC with cache coherence only on-chip and acceleration module (e.g., vector processors). NVM stack will have photonic I/O built in. Photonic switch is monolithic integrated silicon photonics with wave-division multiplexing (WDM). Photonics will have bandwidth of 1 Tb in each direction. Moreover, data will always be encrypted. Asanović said the bigger size will reduce operation expenses, can support a huge in-memory database, and has low latency network to support SOA. There are still many open questions for FireBox, such as how do we use virtualization, how do we process bulk encrypted memory, and which in-box network protocol should we use?

Finally, Asanović talked about another related project, DIABLO, which is an FPGA that simulates WSC. By using DIABLO, they found that the software stack is the overhead.

Rik Farrow wondered about the mention of Linux as the supported operating system. Asanović said that people expected a familiar API, but that FireBox would certainly include new operating system design. Kimberly Keeton (HP Labs) asked who is the "everybody" that considers using custom design chips. Asanović answered big providers are building custom chips, and if we move to an open source model, we will start seeing more. Keeton also asked a question about code portability. Asanović explained that 99.99% of code in applications doesn't need an accelerator. So, only the .01% that does need accelerators will need any changes.

Someone from VMware asked about the role of disk in FireBox. Asanović replied that disk and DC-level network are outside the scope of this project right now. Tom Spinney (Microsoft) asked about protecting keys and cryptography engines. Asanović responded that they planned on using physical mechanisms, and that this was an area of active research.

Experience from Real Systems

Summarized by Xing Lin (xinglin@cs.utah.edu)

(Big) Data in a Virtualized World: Volume, Velocity, and Variety in Cloud Datacenters

Robert Birke, Mathias Bjoerkqvist, and Lydia Y. Chen, IBM Research Zurich Lab; Evgenia Smirni, College of William and Mary; Ton Engbersen, IBM Research Zurich Lab

Mathias Bjoerkqvist started his presentation by noting that virtualization is widely used in datacenters to increase resource utilization, but the understanding of how I/O behaves in virtualized environments is limited, especially at large scales. To get a better understanding, they collected I/O traces in their private production datacenters over a three-year period. The total I/O trace was 22 PB, including 8,000 physical host machines and 90,000 virtual machines. Most of virtual machines were Windows and ran within VMware. Then they looked at how capacity and data changed and characterized read/write operations at the virtual machine layer and the host layer. They also studied the correlation between CPU, I/O, and network utilization for applications.

What's most interesting in their findings is that most contributions to the peak load came from only one third of virtual machines, which implies that we could improve the system by optimizing these few VMs. In their study, they also found a diverse set of file systems were used: For each virtual machine, as many as five file systems were used. Thus, about 20 virtual file systems were used in each host machine on average. The more CPUs and memory the host machine has, the more file systems. The data churn rate at the virtual machine layer is lower than at the host layer: 18% and 21%, respectively. They also looked at I/O amplification and deduplication rate. Comparing the number of I/Os at the virtual machine layer and the physical block layer at the host operating system, they found that amplification appears more often than deduplication. Finally, they used a k-means clustering algorithm to classify application workloads and found a strong correlation between CPU usage and I/O or network usage.

One person asked for the breakdown of true deduplication and caching effect. He suggested that the deduplication rates presented could be the combination of both. The author acknowledged that he was correct: They measured the total I/Os at the virtual machine layer and the physical host layer and were not able to distinguish between the caching effect and true deduplication. Fred Douglis (EMC) pointed out that the comparison of the data churn rate presented in this work and his own previous work was not appropriate because the workloads for this paper were primary workloads, whereas Fred's work studied backup workloads. Yaodong Yang (University of Nebraska-Lincoln) asked about the frequency of virtual machine migration in their datacenters. The authors replied that the peak load varied over time; the peak load in the middle of night could be correlated with virtual machine migration activities. Christos Karamanolis (VMware) asked a few fundamental clarification questions about their measurements, such as what the definition of a virtual I/O was, what the side-effect was from write buffering and read caching at the guest OS, and which storage infrastructure was used at the back end. The authors said they collected I/O traces at the guest OS and host OS level and suggested that people come to their poster for more details.

From Research to Practice: Experiences Engineering a Production Metadata Database for a Scale Out File System

Charles Johnson, Kimberly Keeton, and Charles B. Morrey III, HP Labs; Craig A. N. Soules, Natero; Alistair Veitch, Google; Stephen Bacon, Oskar Batuner, Marcelo Condotta, Hamilton Coutinho, Patrick J. Doyle, Rafael Eichelberger, Hugo Kiehl, Guilherme Magalhaes, James McEvoy, Padmanabhan Nagarajan, Patrick Osborne, Joaquim Souza, Andy Sparkes, Mike Spitzer, Sebastien Tandel, Lincoln Thomas, and Sebastian Zangaro, HP Storage

Kimberly Keeton characterized her team's work as special, covering their experience in transforming a research prototype (LazyBase) into a fully functional production (Express Query). Unstructured data grows quickly, at 60% every year. To make use of unstructured data, the metadata is usually used to infer the underlying structure. Standard file system search function is not feasible for providing rich metadata services, especially in scale-out file systems. The goal of their work is to design a metadata database, to allow rich metadata queries for their scale-out file systems.

LazyBase was designed to handle high update rates, at the expense of some amount of staleness. This matches well with the design of Express Query. Thus, they started with LazyBase and made three main changes to transform it into Express Query. The first change was to eliminate automatic incremental ID for long strings and ID remap because the mapping cannot be done lazily and the assignment of ID for a string cannot be done in parallel. Through experimentation, they found that ID-based lookup or joins was inefficient: minutes for ID-based versus seconds for non-ID based.

The second change was related to the transaction model in LazyBase, which allows updates to be applied asynchronously at a later time. When users delete a file, there is no way to reliably read and delete the up-to-date set of custom attributes. To deal with this problem, the authors introduced timestamps to track file operation events and attribute creations. These timestamps were then used to check for attribute validation during queries. To get the metadata about files, they put a hook in the journaling mechanism in the file system. Then the metadata was aggregated and stored into LazyBase. To support SQL-like queries, they used PostgreSQL on top of LazyBase. A REST API was designed to make Express Query easier and more flexible to use.

Scott Auchmoody (EMC) asked about using hashing to get IDs. Kimberly said that would break the locality; tables are organized according to IDs and with hashing, records for related files could be stored far away from each other. Brent Welch (Google) suggested that distributed file systems usually have an ID for each file, which could be taken and used. Kimberly acknowledged that they had taken advantage of that ID to be a unique identity for each file, and the index and sorting are based on pathname. One person noted that if metadata for files within a directory was organized as a tree structure, the tree structure could become very huge and wondered how much complexity was involved in managing large tree structures. Kimberly answered that they stored metadata as a table, and the directory structure was encoded in the path name. Shuqin Ren (Data Storage Institute, A*STAR) asked what the overhead was when adding the hook in the journaling mechanism to collect metadata. Kimberly replied that the journaling happened anyway, and they did not add any other instrumentation to the file system so the overhead was small.

Analysis of HDFS under HBase: A Facebook Messages Case Study

Tyler Harter, University of Wisconsin—Madison; Dhruba Borthakur, Siying Dong, Amitanand Aiyer, and Liyin Tang, Facebook Inc.; Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, University of Wisconsin—Madison

Tyler Harter presented his work on analyzing I/O characterizations for the layered architecture of HBase for storing and processing messages in Facebook. Reusing HBase and HDFS lowers the development cost and also reduces the complexity of each layer. However, Tyler also pointed out that layered architecture could have some performance overheads and that performance could potentially be improved if the system were layer-aware. Their work studied how each layer altered I/O requests and exploited two use cases to demonstrate that performance could be improved by making the system layer-aware.

Tyler showed that at the HDFS layer, only 1% of all read/write requests it received were writes. However, writes became 21% because of compaction and logging used in HDFS. At the local file system layer, 45% of requests were writes because of three-way replication. At the disk layer, the percentage of writes was 64%. The same trend held when they considered I/O size. Two thirds of data is cold. Then Tyler presented the distribution of file sizes. Half of files used in HBase had sizes smaller than 750 KB. For access locality, they found a high temporary locality: the hit rate was 25% if they kept accessed data for 30 minutes and 50% for two hours. Spatial locality was low, and as much as 75% reads were random. Then they looked into what hardware upgrade was most effective in terms of I/O latency and cost. They suggested that adding a few more SSDs gave the most benefit with little increase in cost; buying more disks did not help much. To demonstrate that performance is increased by making the system layer-aware, Tyler presented two optimizations: local compaction and combined logging. Instead of sending compacted data, they proposed sending the compact command to every server to initiate compaction. This change reduced the network I/O by 62%. With combined logging, a single disk in each server machine was used for logging while other disks could serve other requests. This change reduced disk head contention, and the evaluation showed a six-fold speedup for log writes and performance improvement for compaction and foreground reads.

Bill Bolosky (Microsoft Research) pointed out that file size distribution usually had a heavy tail and thus the mean file size would be three times larger than the median size. He asked whether the authors had looked into the mean size. Tyler acknowledged that in their paper they did have numbers for mean file size but he did not remember them. He also suggested that, because most files were small, what mattered for performance was the number of files, specifically metadata operations. Brad Morrey (HP Labs) noted that for their local compaction to work, related segments had to be stored in a single server. This would affect recovery performance if a server died. Tyler replied that they did not do simulation for recovery. However, the replication scheme was pluggable and they should be able to exploit different replication schemes. Margo Seltzer (Harvard) suggested that it was probably wrong to use HDFS to store small files; HDFS was not designed for that, and other systems should probably be considered. Tyler acknowledged that was a fair argument, but Facebook uses HDFS for a lot of other projects and that argues for using a uniform architecture.

Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces

Yang Liu, North Carolina State University; Raghul Gunasekaran, Oak Ridge National Laboratory; Xiaosong Ma, Qatar Computing Research Institute and North Carolina State University; Sudharshan S. Vazhkudai, Oak Ridge National Laboratory

Yang Liu started his talk by introducing the second fastest supercomputer in the world: TITAN, which has 18,000 compute nodes. More than 400 users use this supercomputer for various scientific computations, such as climate simulation. The Spider file system is used to provide file system service: In total, TITAN can store 32 PB of data and can provide 1 TB/s bandwidth. Because the supercomputer is shared by multiple users, workloads from different users could compete for the shared storage infrastructure and thus interfere with each other. Thus, it is important to understand the I/O behavior of each application. With that understanding, we can do a better job in scheduling these jobs and thus reduce I/O contention. Their work proposed an approach to extract I/O signatures for scientific workloads running from server-side tracing.

Client-side tracing is the other alternative but has a few drawbacks. Client-side tracing requires a considerable development effort, and it usually introduces some performance overhead, ranging from 2% to 8%. Different applications probably use different trace formats and may not be compatible. What's worse, client-side tracing introduces extra I/O requests. So they decided to use the coarse-grain logging at the RAID controller level at the server-side. It has no overhead and does not require any user effort. However, I/O requests are mixed at the server-side. The challenge is to extract I/O signatures for a particular application from this mixed I/O traffic. Fortunately, there are a few inherent features in scientific applications that can help to achieve that goal.

These applications usually have two distinct phases: compute phase and I/O phase. During the I/O phase, applications typically request large writes of either intermediate results or checkpointing, so there are periodic bursts for these applications. Besides, users tend to run the same application multiple times with the same configuration. Thus, I/O requests are repetitive for these applications. Given the job scheduler logs with start and end time for each job and the server-side throughput logs from Spider, they can extract the samples for a particular job. The insight from the authors is that the commonality across multiple samples tends to belong to the target application.

The challenges of extracting I/O signatures from these samples include background noise and I/O drift. To deal with these challenges, the authors proposed three stages to extract the I/O signature: (1) data preprocessing that eliminates outliers, refines the granularity of the samples, aligns durations, and reduces noise by removing light I/O traffic; (2) use of wavelet transform for each sample to make each sample smoother and to more easily distinguish bursts from background noise; and (3) use of the CLIQUE (Agrawal: SIGMOD '08) clustering algorithm to identify common bursts across samples. For evaluation, they used I/OR, a benchmark tool for parallel I/O to generate a few synthetic workloads and a real-world simulation: S3D. The signature extracted by their tool matched well with the actual I/O signature. They also compared the accuracy of their algorithm with Dynamic Time Warping (DTW) and found I/OSI outperformed DTW.

Kun Tang (Virginia Commonwealth University) asked whether it was true that users actually ran an application multiple times. Yang answered yes, based on what they observed from the job scheduler log and several previous works. One person suggested that if they already had the I/O signature, their tool would be able to reproduce that signature. Yang replied that based on their observation, each application would likely exhibit the same I/O pattern in future runs, and they could use this insight for better scheduling.

Works-in-Progress

Summarized by Dutch Meyer (dmeyer@cs.ubc.ca)

The FAST '14 Works-in-Progress session started with Taejin Kim (Seoul National University), who presented his work on "Lifetime Improvement Techniques for Multiple SSDs." He observed that write amplification in a cluster of SSDs may degrade the device's lifetime because of intermediate layers which perform replication, striping, and maintenance tasks such as scrubbing. As an example, he showed how writes could be amplified by 2.4x under Linux's RAID-5 implementation. He proposed integrating many different write prevention techniques, such as compression, deduplication, and dynamic throttling, to lower the write workload, as well as smaller stripe units, delta compression, and modifying erasure coding algorithms to use trim in place of writes.

Dong Hyun Kang (Samsung Electronics) presented "Flash-Friendly Buffer Replacement Algorithm for Improving Performance and Lifetime of NAND Flash Storages." He explained how traditional buffer replacement algorithms are based on magnetic drives and proposed an algorithm called TS-Clock to perform cache eviction. TS-Clock is designed to limit write-backs and improve cache hit rate. His preliminary results show that on DBench the TS-Clock algorithm has up to a 22.7% improvement in hit rate and extends flash lifetime by up to 40.8%.

Douglas G. Otstott (Florida International University) described his work towards developing a "holistic" approach to scheduling. In his presentation, "A Host-Side Integrate Flash Scheduler for Solid State Drives," he listed the inefficiencies of the OS to flash interface. Flash devices themselves have limited resources to consider complex scheduling. However, in the relatively more powerful OS-layer, most of the potentially useful details about individual request performance, such as read vs. write latencies, write locations, and logical to physical block mappings, are hidden. In his alternate approach, device management occurs in the

OS stack but pushes commands down to the device to balance load, ensure that writes aren't committed ahead of reads, and limit GC performance costs. His proposed interface includes isolated per-die queues for read, write, and garbage collection. He is working to validate his Linux prototype in DiskSim and is working on an FPGA implementation.

Next, Eunhyeok Park (University of Pittsburgh) described his work: "Accelerating Graph Computation with Emerging Non-Volatile Memory Technologies." Algorithms that access large graph data structures incur frequent and random storage access. After describing the CPU bottlenecks in these workloads, he described Racetrack, an approach based on an idealized memory model. Park's simulations based on Pin show that it is faster and has lower energy consumption than alternatives. His preliminary results suggest that some simple computations, for example PageRank, could be done by a CPU or FPGA on the storage device itself.

In "Automatic Generation of I/O Kernels for HPC Applications," Babak Behzad (University of Illinois at Urbana-Champaign) described his efforts to autonomously generate an I/O kernel, which is a replayable trace that includes dependency constraints. His approach is to wrap I/O activity with an application-level library to trace parallel I/O requests in high-performance computing environments. His work promises to provide better analytics for future optimization, I/O auto-tuning, and system evaluation.

Florin Isaila (Argonne National Laboratory) described "Clarisse: Cross-Layer Abstractions and Runtime for I/O Software Stack of Extreme Scale Systems." His goal is to enable global optimization in the software I/O stack to avoid the inefficiencies of I/O requests that pass through multiple layers of uncoordinated memories and nodes, each of which provides redundant function. He proposed providing cross-layer control abstractions and mechanisms for supporting data flow optimizations with a unified I/O controller. Instead of the traditional layered model, Clarisse intercepts calls at different layers. Local control modules at each stack layer talk over a backplane to a controller that coordinates buffering, aggregation, caching prefetching, optimization selection, and event processing.

Howie Huang (George Washington University) presented "HyperNVM: Hypervisor Managed Non-Volatile Memory." He observed that warehouse-scale datacenters are increasingly virtualized and that many VMs performing large amounts of I/O require high performance memory subsystems and need better systems support for emerging non-volatile memories. To solve these problems, he proposed that the hypervisor needs to be more memory aware and that the OS and applications should pass more control of memory management to the hypervisor. He proposed a system called Mortar, which is a general-purpose framework for exposing hypervisor-controlled memory to applications.

Dan Dobre (NEC Labs Europe) described his work on "Hybris: Robust and Consistent Hybrid Cloud Storage." He noted that potential users of cloud storage services are concerned about security and weak consistency guarantees. To address these two concerns, he proposed a hybrid solution that employs a private cloud backed by multiple public clouds. His approach maintains a trusted private cloud to store metadata locally, with strong metadata consistency to mask the weak data consistency offered by public clouds.

In "Problems with Clouds: Improving the Performance of Multi-Tenant Storage with I/O Sheltering," Tiratat Patana-anake (University of Chicago) described an approach to limit the performance impact of random writes in otherwise sequential workloads. In his approach, called "I/O Sheltering," random writes are initially written sequentially inline with sequential writes. Later, these sheltered writes are moved to their in-place locations. He argued that this approach would be successful because random write applications are rare, memory is abundant, and NVM provides a better location to shelter indexes. He concluded by showing significant performance improvements in multi-tenant Linux, when one random writer ran in conjunction with many sequential writers. He proposed to integrate this technique into the file system and journal in the future.

Michael Sevilla (University of California, Santa Cruz) argued that load balancing should be based on a deeper understanding of individual node resources and contention in his work on "Exploring Resource Migration Using the CephFS Metadata Cluster." Sevilla has been using Ceph as a prototyping platform to investigate migration and load balancing. He has observed that decisions about whether to migrate are often non-intuitive. Sometimes migration helps mitigate loads, but it may also hurt, because it denies access to a full metadata cache. He is attempting to develop a distributed, low-cost, multiple objective solver to make more informed migration decisions. In closing, he demonstrated the practicality of the problem by detailing an experiment where re-ordering name servers in a cluster under high load resulted in a considerable performance improvement.

With his work on "Inline Deduplication for Storage Caching," Gregory Jean-Baptiste (Florida International University) proposed enhancing existing client-side flash caches with inline deduplication to increase their effective capacity and lifespan. He has a prototype system in place that shows better performance with the IOZone benchmark over iSCSI. He noted that this approach may lead to other architectural changes, such as replacing LRU with an eviction algorithm that is aware that evicting a widely shared block could be more painful that an unshared one.

Alireza Haghdoost (University of Minnesota) presented "hfplayer: High Fidelity Block I/O Trace Replay Tool." hfplayer can reproduce previously captured SAN workload with high fidelity for the purposes of benchmarking, performance evaluation, and validation with realistic workloads. His design is based on a pool of worker threads that issue I/O requests and a harvester thread to process completion events and keep track of in-flight I/O requests. To control replay speed, a load-aware algorithm dynamically matches the number of in-flight I/O requests with the load profile in the original trace. When inter-arrival times are very short, threads bundle requests together to avoid system call overhead.

Nagapramod Mandagere (IBM Research) has been analyzing the performance of backup servers and has found some interesting opportunities for optimization. In his presentation titled "Towards a Framework for Adaptive Backup Management," he described some of his observations. He noted that backup workloads show temporal skew, where most traffic arrives at hourly boundaries but large bursts of heavy utilization are possible. Another problem is spatial skew, in which one backup server is overloaded while others are not. Both these problems stem from backups that are managed by static policies while the workload they create is very dynamic. As client backup windows get shorter and shorter (due to lack of idle times), server initiated backups are harder to schedule, but when the clients are instead allowed to push updates themselves, the backup servers can become overloaded. Furthermore, backup servers themselves need idle periods for their own maintenance. He proposes an adaptive model-based backup management framework that dynamically determines client/server pairings to minimize client backup windows.

Performance and Efficiency

Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-Aware Allocation

Hui Wang, Peter Varman, Rice University

Hui Wang started her presentation by talking about the trend of multi-tiered storage in modern datacenters with both solid state drives and traditional hard disks. This kind of combination has several advantages, including better performance in data access and lower cost. At the same time, it also brings some challenges, such as providing fair resource allocation among clients and maintaining high system efficiency. To be more specific, there is a big speed gap between SSD and HD, so scheduling proper workloads to achieve high resource utilization becomes very important. Hui Wang talked about fair resource allocation and high performance efficiency to make up for some disadvantages of heterogeneous clusters. She talked about her team's motivation using several examples: single device type, multiple devices, and dominant resource from both fairness and efficiency angles.

Based on these considerations, she presented a new allocation model, Bottleneck-Aware Allocation (BAA), based on the notion of per-device bottleneck sets. Clients bottlenecked on the same device receive throughputs in proportion to their fair shares, whereas allocation ratios between clients in different bottleneck sets are chosen to maximize system utilization. In this part, she discussed the fairness policy first, showed the bottleneck sets and fairness requirements of BAA, and then introduced their optimization algorithm of allocation.

They performed two simulations: one to evaluate the BAA's efficiency, and the other monitoring BAA's dynamic behavior under changing workloads. They also implemented a prototype, interposing BAA scheduler in the I/O path and evaluated BAA's efficiency and fairness.

Umesh Maheshwari (Nimble Storage) asked if that kind of dramatic ratio somehow changed the nature of this work. Hui Wang said, suppose it was for a single disk compared with SSD. It is most likely that HD would be the bottleneck, but if you have a large HD array, the speed gap between the two is not so dramatically large, so you would very easily have the balanced set cluster on both. Umesh asked again, assuming there was such a large difference between the two tiers, would the BBA model still hold the same performance using this approach? Hui Wang answered yes. Shuqin (Data Storage Institute Singapore) asked whether their model would work on multiple nodes. Hui Wang said she thought that their model could be easily extended to multiple nodes with coordination between schedulers. Kai Shen (University of Rochester) asked what was particularly challenging in this project. Hui Wang said the most challenging part was to accurately estimate the capacity of the system.

SpringFS: Bridging Agility and Performance in Elastic Distributed Storage

Lianghong Xu, James Cipar, Elie Krevat, Alexey Tumanov, and Nitin Gupta, Carnegie Mellon University; Michael A. Kozuch, Intel Labs; Gregory R. Ganger, Carnegie Mellon University

In this presentation, Xu introduced the concept of elasticity in distributed storage. Elastic distributed storage means storage that is able to resize dynamically as workload varies, and its advantages are the ability to reuse storage for other purposes or reduce energy usage; they can decide how many active servers to provide to work with a changing workload. By closely monitoring and reacting quickly to changes in workload, machine hours are saved. But most current storage, such as GFS and HDFS, is still not elastic. If they deactivate the nodes, it will cause some data to not be available. Before Xu talked about SpringFS, he gave two examples of prior elastic distributed storage, Rabbit and Sierra, discussing the differences between them and the disadvantages of each. Fortunately, SpringFS provides balance and fills the gap between them.

First, Xu showed a non-elastic example: in this case, almost all servers must be "active" to be certain of 100% availability, so it has no potential for elastic resizing. He then discussed the general rule of data layout in elastic storage and tradeoff space. Based on this, the authors proposed an elastic storage system, called SpringFS, which can change its number of active servers quickly, while retaining elasticity and performance goals. This model borrows the ideas of write availability and performance offloading from Rabbit and Sierra, but it expands on previous work by developing new offloading and migration schemes that effectively eliminate the painful tradeoff between agility and write performance in state-of-the-art elastic storage designs. This model, combined with the read offloading and passive migration policies used in SpringFS, minimizes the work needed before deactivation or activation of servers.

Muhammed (HGST) asked what is used to predict the performance ahead of time and whether the offload set value could be adjusted. Xu said they actually did not design this part of the workload in their paper. They just assumed they had the perfect predictor, and that it was possible to integrate some workload predictor into their work. One person asked whether this model tolerates rack fails. Xu said yes, because their model was modified from HDFS, it could tolerate rack fails as well as HDFS. Another questioner asked what offloading in SpringFS essentially means. Xu explained that offloading means redirecting requests from a heavy loaded server to a lightly loaded server.

Migratory Compression: Coarse-grained Data Reordering to Improve Compressibility

Xing Lin, Guanlin Lu, Fred Douglis, Philip Shilane and Grant Wallace, EMC Corporation-Data Protection and Availability Division, University of Utah

Lin presented Migratory Compression (MC), a coarse-grained data transformation, to improve the effectiveness of traditional compressors in modern storage systems. In MC, similar data chunks are relocated together to improve compression factors. After decompression, migrated chunks return to their previous locations. This work is motivated by two points. The first is to exploit redundancy across a large range of data (i.e., many GB) during compression by grouping similar data together before compressing. The second point is to improve the compression for long-term retention. However, the big challenge in doing MC is to identify similar chunks efficiently and scalably; common practice is to generate similarity features for each chunk because two chunks are likely to be similar if they share many features.

There are two principal use cases of MC. The first case is mzip, that is, compressing a single file by extracting resemblance information, clustering similar data, reordering data in the file, and compressing the reordered file using an off-the-shelf compressor. The second case is archival, involving data migration from backup storage systems to archive tiers or data stored directly in an archive system, such as Amazon Glacier. The evaluation result showed that adding MC to a compressor significantly improves the compression factor (23–105% for gzip, 18–84% for bzip2, 15–74% for 7z, and 11–47% for rzip), and we can also see the improvement of compression throughput with MC. In all, MC improves both compression factor and throughput by deduplication and reorganization.

Cornel Constantinescu (IBM Almaden Research Lab) asked whether this model compressed the file. Lin said that in the mzip case, they un-compacted the whole file. Constantinescu asked whether they did a comparison with other similar work, such as work used in Google's BigTable. Lin admitted he did not know that. Jacob Lorch (Microsoft Research) asked whether they were starting to look at modifying the compression algorithms. Lin said that he did not think they modified compression itself and that MC could be used as a generic preprocessing stage that could benefit all of the compression. And, if they are improving compressions, they can get the same benefits by doing this as a separate stage.

Poster Session II

Summarized by Kai Ren (kair@cs.cmu.edu)

VMOFS: Diskless and Efficient Object File System for Virtual Machines

Shesha Sreenivasamurthy and Ethan Miller, UC Santa Cruz

This project is to design an object file system for virtual machine environment by deduplicating common files shared in many VM images. In their architecture, guest machines will run a VFS layer software called VMOFS to track the mapping between inode and object, and a hypervisor manages the storage and deduplicates file objects and stores them into underlying object storage. Deduplication is achieved at a per-object level to reduce the dedup table size. This work is still under development, and no experimental results were presented.

Characterizing Large Scale Workload and Its Implications

Dongwoo Kang, Seungjae Baek, Jongmoo Choi, Donghee Lee, and Sam H. Noh, Dankook University, University of Seoul and Hongik University

This project analyzes storage workloads from a collection of traces from various cloud clusters. The observations are (1) workloads with large data are not sensitive to cache size; (2) cache allocation should be dynamically tuned due to irregular cache status changes; and (3) to achieve high cache hit ratio, one might use a policy that quickly evicts blocks with large request size and hit counts in a special period. The next part of this project will be to apply these observations to design and implement a cache service for virtual machine clusters.

Automatic Generation of I/O Kernels for HPC Applications

Babak Behzad, Farah Hariri, and Vu Dang, University of Illinois at Urbana-Champaign; Weizhe Zhang, Harbin Institute of Technology

The goal of this project is to automatically extract I/O traces from applications and regenerate these workloads to different scales of storage systems for performance measurement or testing. To fulfill such a goal, the workload generation systems have long workflows. First, it extracts traces from multiple levels of the I/O flow—application, MPI-I/O, and POSIX I/O levels—and from multiple processes. The second step is to merge these traces and understand the dependency between I/O operations presented in the traces. To construct such a dependency, it needs to understand semantics of I/O operations, which is mostly inferred from MPI-I/O library calls. For operations without clear dependency, timestamps are used to decide the order. The last step is to divide the traces, and generate binaries to replay the traces or simulate the workloads.

Poster Session II

Summarized by Qian Ding (qding8@gmail.com)

VMOFS: Breaking Monolithic VM Disks into Objects

Shesha Sreenivasamurthy and Ethan Miller, UC Santa Cruz

Shesha and Ethan propose a solution of efficient file sharing among virtual machines (VMs) through an object-based root file system. They are building a file system called VMOFS, which adopts object-level deduplication to store monolithic VM images. The hypervisor layer in the system encapsulates the OSD layer and controls the communication between file system and the OSDs via iSCSI. VMOFS is still under development so there is no evaluation at the current stage.

Characterizing Large Scale Workload and Its Implications

Dongwoo Kang, Seungjae Baek, Jongmoo Choi, Donghee Lee, and Sam H. Noh, Dankook University, University of Seoul and Hongik University

Dongwoo Kang presented work about characterizing a group of recently released storage traces and explained their observations and possible implications. The traces the team used were from MSR-Cambridge and FIU. Their first finding was that such traces were not sensitive to cache size when using a simple LRU cache but have much variation on the change of cache hit ratio and I/O size. They also found long inter-reference gaps (IRG) from the traces and provide that as a reason for low cache sensitivity. They propose two ways to improve the hit ratio: dynamic cache allocation and evicting short IRGs in the cache. They also propose ways for optimizing cache utility on a virtualized storage environment.

MicroBrick: A Flexible Storage Building Block for Cloud Storage Systems

Junghi Min, Jaehong Min, Kwanghyun La, Kangho Roh, and Jihong Kim, Samsung Electronics and Seoul National University

Jaehong Min presented the design of MicroBrick for cloud storage. The authors tried to solve the diverse resource requirement problem, such as balancing the computing and storage resource in cloud services. Each MicroBrick node adopts flexible control for both CPU-intensive and storage-intensive configuration through a PCIe switch. Thus, when the cloud system is composed of MicroBrick nodes, they can use a software management layer for autoconfiguration for different resource requirements. The preliminary evaluation of MicroBrick shows competitive results by running cloud computation (e.g., wordcount) and sort programs.

OS and Storage Interactions

Summarized by Kuei Sun (kuei.sun@utoronto.ca)

Resolving Journaling of Journal Anomaly in Android I/O: Multi-Version B-Tree with Lazy Split

Wook-Hee Kim and Beomseok Nam, Ulsan National Institute of Science and Technology; Dongil Park and Youjip Won, Hanyang University

Wook-Hee Kim began the talk by reminding us that although Android is the most popular mobile platform to date, it has severe I/O performance bottlenecks. The bottlenecks are caused by the "journaling of journal anomaly" between ext4 and SQLite, which is used by many applications. Kim gave a stunning example where one insert of 100 bytes resulted in nine random writes of 4 KB each. Currently, the database calls fsync() twice: once for journaling and once for insertion. Kim and his colleagues proposed to obviate the need for database journaling by implementing a variant of multi-version B-tree named LS-MVBT.

Kim presented several optimizations made to LS-MVBT based primarily on the characteristics of the common workloads. Lazy split seeks to reduce I/O traffic during node split by simultaneously garbage collecting dead entries so that the existing node (aka lazy node) can be reused. However, if there are concurrent transactions and the dead entries are still being accessed, then a workaround is needed. Their solution is to reserve space on each node so that new entries can still be added to the lazy node without garbage collection. To further reduce I/O traffic, instead of periodic garbage collection, LS-MVBT does not garbage collect unless space is needed. Next, Kim showed that by not updating the header pages with the most recent file change counter, only one dirty page needs to be flushed per insertion. He pointed out that this optimization would not increase overhead during crash recovery because all B-tree nodes must be scanned anyway. Lastly, they disabled sibling redistribution from the original MVBT and forced a node split whenever a node became full. Kim showed that this optimization actually reduced the number of dirty pages.

In their evaluation, Kim showed that LS-MVBT improves performance of database insertions by 70% against the original SQLite implementation (WAL mode). LS-MVBT also reduces I/O traffic by 67%, which amounts to a three-fold increase in the lifetime of NAND flash. LS-MVBT is also five to six times faster than WAL mode during recovery. Lastly, LS-MVBT outperforms WAL mode unless more than 93% of the workload are searches. At the end of the talk, Kim's colleague demonstrated a working version of their implementation, showing the performance improvement in a simulated environment.

Journaling of Journal Is (Almost) Free

Kai Shen, Stan Park, and Meng Zhu, University of Rochester

Kai Shen started by arguing that journaling of journal is a violation of the end-to-end argument and showed that adding ext4 journaling to SQLite incurs a 73% slowdown in their experiments. Existing solutions require substantial changes to either the file system or the application. The authors proposed two simple techniques.

Single-I/O data journaling attempts to minimize the number of device writes on the journal commit's critical path. In this case, data and metadata are journaled synchronously. During their experiments, they discovered a bug with ext4_sync_file(), which unnecessarily checkpoints data to be overwritten or deleted soon. The problem with the data journaling is the large volume written due to the page-sized granularity of ext4. Therefore, they proposed a second technique called file adaptive journaling, which allows each file to have a custom journaling mode. They found the solution effective for the journaling of journal problem. In particular, write-ahead logs would prefer ordered journaling due to little metadata change, and rollback logs would prefer data journaling due to heavy metadata change. In their evaluation, Shen showed that their enhanced journaling incurs either little to no cost.

Theodore Wong (Illumina, Inc.) asked what would happen if file system journaling were not used. Shen explained that protection of both file system metadata and application data still would be necessary because an untimely crash may cause the file system to become inconsistent. Because the database only protects its own data in a file, an inconsistent file system may cause the database file to become inaccessible. Kim commented on the fact that enterprise databases usually skip over the file system and operate directly on raw devices. However, lightweight databases may still prefer running on top of file systems for simplicity. Peter Desnoyer (Northeastern University) wanted to know whether these changes could affect the opportunity for inconsistency. Shen replied that as long as the failure model is fail-stop, then all of the guarantees would still apply.

Checking the Integrity of Transactional Mechanisms

Daniel Fryer, Mike Qin, Kah Wai Lee, Angela Demke Brown, Ashvin Goel, and Jack Sun, University of Toronto

Daniel Fryer began his talk by showing us that corruptions caused by file system bugs are frequently catastrophic because they are persistent, silent, and not mitigated by existing reliability techniques. The authors' previous work, Recon, ensures that file system bugs do not corrupt data on disk by checking the consistency of every transaction at runtime to prevent corrupt transactions from becoming durable. Because Recon performs its checks at commit time, it requires the underlying file system to use a transaction mechanism. Unfortunately, Recon does not detect bugs in the transaction mechanism. Their solution is to extend Recon to enforce the correctness of transaction mechanisms.

Recon already checks the consistency of transactions by verifying that metadata updates within a transaction are mutually consistent. To enforce atomicity and durability, Recon needs to also be able to catch unsafe writes and prevent them from reaching disk. Fryer defined two new sets of invariants, atomicity and durability invariants (collectively called location invariants), which govern the integrity of committed transactions. These invariants need to be checked on every write to make sure that the location of every write is correct. Fryer walked us through two types of transaction mechanisms, journaling and shadowing paging, as well as their respective invariants. Next, he presented a list of file system features that are required for efficient invariant checking at runtime.

Fryer and his team implemented location invariants for ext3 and btrfs by extending Recon. They had to retrofit ext3 with a metadata bitmap to distinguish between data and metadata, which is required to detect unsafe overwrites to metadata blocks. To evaluate the correctness of their implementation, they corrupted file system writes of various types to simulate bugs in the transaction mechanism and successfully caught most of them. The ones they missed did not affect file system consistency. Finally, they showed that adding location invariants to Recon incurs negligible overhead.

Keith Smith (NetApp) asked what could be done after a violation was detected. Fryer responded that while optimistically delaying a commit is plausible, what they've done at the moment is to return an error since their first priority is to prevent a corrupting write from reaching disk. Ted Ts'o (Google) wanted clarification on why losing some writes did not affect correctness during the corruption experiments. Fryer explained that the particular implementation of the file system was suboptimal and was checkpointing some metadata blocks unnecessarily because future committed versions of those blocks exist in the journal. Therefore, losing those writes was inconsequential. Harumi Kuno (HP Labs) was baffled by the fact that checking both consistency and location invariants resulted in better performance than just checking consistency alone. Fryer believed that it was simply due to an insufficient number of trials.

OS and Peripherals

Summarized by Matias Bjørlin (mabj@itu.dk)

DC Express: Shortest Latency Protocol for Reading Phase Change Memory over PCI Express

Dejan Vučinić, Qingbo Wang, Cyril Guyot, Robert Mateescu, Filip Blagojević, Luiz Franca-Neto, and Damien Le Moal, HGST San Jose Research Center; Trevor Bunker, Jian Xu, and Steven Swanson, University of California, San Diego; Zvonimir Bandić, HGST San Jose Research Center

Dejan Vučinić began on a side note by stating that the next big thing isn't DRAM, because of its high energy utilization, referring to the previous FireBox keynote. He then stated the motivation for his talk by showing upcoming non-volatile memories and their near DRAM access timings. He explained how they each compete with DRAM on either price or latency and showed why PCM has fast nanosecond reads but microsecond writes. He explained that to work with PCM, the team built a prototype, exposing it through a PCI-e interface.

Dejan then showed how PCI-e communicates with the host using submission and completion queues. When a new request is added to the queue, a doorbell command is issued to the PCI-e device. When received, the device sends a DMA request to which the host returns the actual data request. The request is processed, and data with a completion command at the end is sent. The handshake requires at least a microsecond before any actual data is sent. To eliminate some of the overhead, they began by removing the need for ringing the doorbell. They implemented an FPGA that continuously polls for new requests on the memory bus. Then they looked at how completion events take place, which the host could poll for when the data is finished. However, instead of the host polling, they show that completion can be inferred from the response data by inserting a predefined bit pattern in DRAM. Data is stored in DRAM from the device and should be different; thus, they could infer when all data packets have been received.

Dejan then showed that using their approach they achieve, using a single PCI-e lane, a 1.4 μs round-trip time for 512 bytes. Even with these optimizations, there continues to be large overhead involved, and thus the fundamental overhead of the communication protocol should be solved to allow PCM and other next-generation memories to be used efficiently.

Brad Morrey (HP Labs) asked why they didn't put it on the DRAM bus. Dejan replied that there is a need for queues within the DRAM. The queues are needed to prevent stalls while waiting. He wants to have it there in the future, but PCM power limitations should be taken into consideration. Peter Desnoyers (Northeastern University) asked if they had a choice on how PCI-e on PCM could evolve and what would they do? Dejan answered that it could be used with, for example, hybrid memory cubes. Finally, Ted Ts'o (Google) noted that the polling might be expensive in power. What is the power impact? Dejan said that it is very low, as you may only poll during an inflight I/O.

MultiLanes: Providing Virtualized Storage for OS-Level Virtualization on Many Cores

Junbin Kang, Benlong Zhang, Tianyu Wo, Chunming Hu, and Jinpeng Huai, Beihang University

Junbin Kang began by stating that manycore architectures exhibit powerful computing capacity and that virtualization is necessary to use this capacity. He then continued to argue paravirtualization versus operating system virtualization and ended by comparing the many layers of a traditional virtualization stack with a stack using containers. Containers are simpler in their architecture, but they expose scalability issues within the Linux kernel.

To solve the scalability issues, Junbin presented MultiLanes. The data access for containers are partitioned using a partitioned Virtual File System (pVFS) abstraction. pVFS exposes a virtualized VFS abstraction for the containers. In turn, the pVFS eliminates contention within the host VFS layer and improves locality of the VFS data structures. pVFS communicates with the host using a container-specific virtualized block driver (vDrive). The driver takes care of submitting the I/Os from the container to the host system. This allows each container data partition to be split and thereby avoid contention on host VFS locks. Junbin then explained the internal structure of the virtualized driver.

They evaluated their solution using both micro and macrobenchmarks on the file systems ext3, ext4, XFS, and btrfs, with microbenchmarks being metadata operations and sequential writes. For all of them, the operations scale significantly better with additional containers. The macrobenchmarks consist of varmail, fileserver, and MySQL, showing similar performance improvements. Finally, they discussed the overhead of their solution and show it to be negligible in most cases. However, excessive block remapping did have a cost during high throughput.

Kai Shen (University of Rochester) asked whether MultiLanes adds a large memory footprint for each of the channels they create. Junbin said that their approach is complementary to previous approaches. Yuan (UCSC) noted that for a large number of containers using XFS the performance was much better than for ext4. The answer from Junbin and Theodore Ts'o, the ext4 maintainer, was that it depends on the kernel version and other work. Further discussion was taken offline.

Linux FAST Summit '14: 2014 USENIX Research in Linux File and Storage Technologies Summit

San Jose, CA
February 20, 2014

Summarized by Rik Farrow

The Linux FAST Summit took place shortly after FAST '14 had finished. Red Hat offices in Mountain View provided a classroom that sat 30, but the room was full to the point that some of us were sitting in folding chairs at the front or back. Ric Wheeler (Red Hat) moderated the workshop.

Unlike the one Linux Kernel Summit I attended, the focus of this event was strictly on file systems and related topics. Another difference was that—instead of having mainly industry in attendance making requests of kernel developers—file system researchers, mostly students and some professors, were there to make requests for changes in how the kernel works.

The workshop began with Ric Wheeler encouraging people to submit changes to the kernel. He also explained the kernel update process, where a release candidate will come up and be followed by multiple RCs that, outside of the first two RCs, should only include bug fixes. Someone asked about rude answers from Linus Torvalds, and James Bottomley (Parallels) responded that sending in patches with new features late in the RC process is a common way to get an angry response from Linus. Also, the larger the patch set, the less likely it will be accepted. Ted Ts'o (Google) pointed out that just getting a response from Linus is a big deal and suggested seeing who is responding to such a response and who is being ignored. He also said that large intrusions to core infrastructure are less likely to be accepted. Christoph Hellwig (freelance) pointed out that changes to the core of the kernel are harder to validate and less likely to be accepted, which makes it very hard to get very high-impact changes made.

Ethan Miller (UCSC) wondered who would maintain the code that a PhD contributes after they graduate, and Christoph Hellwig responded that they want the code to be so good that the candidate gets a job supporting it later. Ric Wheeler commented on "drive-by" code drops, and said those are fine; if there is broad community support for that area, like XFS or ext4, people will review and support the change. James Bottomley said that it is important to be enthusiastic about your patch. Ted Ts'o chimed in saying that even if your patch is not perfect, it could be seen as a bug report, or some portion of it might be successful. James added that Google recruiters look for people who can take good ideas and turn them into working implementations.

Jeff Darcy (Red Hat) asked where they find tests for patches, and James said they could look at other file systems. xfstests go way beyond XFS, and the VM crew has lots of weird tests.

Ethan Miller launched a short discussion about device drivers, wondering who maintains them, and James Bottomley said that any storage device that winds up in a laptop will be accepted into the kernel. Ted Ts'o said that there were lots of device drivers with no maintainer, and James Bottomley suggested that Fengguang Wu (Intel) runs all code in VMs for regression testing. Andreas Dilger (Google) pointed out that his is just basic testing. James said that SCSI devices might stay there forever, or until someone wants to change the interface.

Ric Wheeler suggested covering how the staging tree for the kernel works. James explained that there are about 300 core maintainer trees, and kernel releases are on a two- to three-month cycle. Ted Ts'o said that RC1 and RC2 are where patches go in, with patches accepted to RC3 being rare, and that by RC7 they better be really critical patches, as that is just before a version release. Ted also suggested that if you are a commercial device builder, you want to test your device with RC2 to see if changes have broken the driver for your device.

Erez Zadok (Stony Brook) asked for a brief history of the memory management (MM) tree. Andrew Morton (Linux Foundation) explained that MM had become a catchall tree where stuff also falls through. Andrew collects these patches and pushes them up to Linus.

After a short break, Ric Wheeler opened the discussion about shingled drives. Ted Ts'o had suggested the *;login:* article by Tim Feldman (Seagate) and Garth Gibson (CMU) [1] as good preparation for this part of the summit. Briefly, Shingled Magnetic Recording (SMR) means that written tracks overlap, like shingles on a roof. These tracks can still be read, but writing must occur at the end of a band of these shingled tracks, or a band can be completely rewritten, starting at the beginning. Random writes are not allowed. Vendors can make SMR drives that appear like normal drives (managing the changes), partially expose information about the drives (restricted), or allow the operating system to control writing the drives (host-aware).

Ted Ts'o explained that he had proposed a draft interface for SMR on the FSDEVEL list. The goal is to make the file system friendly to having large erase blocks like flash, which matches host-aware SMR behavior. If the device is restricted, you still need the operating system to be aware that it is an SMR drive. Ted suggested that this could be part of devmapper (if part of the OS) and could begin as a shim layer.

Erez Zadok said that a group at Stony Brook had been working with Western Digital and had received some SMR drives with a firmware that does more and more, along with a vanilla drive. One of his graduate students reports to Jim Molina, CTO of Western Digital. Erez wanted to share what they've learned so far with the Linux file system community.

First, the vendors will not let us put active code into drives. They will provide a way of knowing when garbage collection (GC) is about to start, and some standards are evolving. Vendors are conscious of the desire for more visibility into the 500k lines of code already in drives.

Because the drives come to Erez Zadok preformatted, Ric Wheeler wondered what percentage remains random, as opposed to shingled, bands. Erez said that the vendors wanted less than 1% of SMR drives as random (traditional tracks) because of the economics involved. They have already done some work using NULLFS with the drives, and Jeff Darcy said he had experimented with using a shim in the device mapper. Ted Ts'o pointed out there was a real research opportunity here in building a basic redirection layer that makes a restricted drive appear like a plain device.

James Bottomley thought that anything they wrote wouldn't last long, as the vendors would move the code into the drive. Eric Reidel (Seagate) objected, saying that they need to determine the boundary between the drive and some amount of software. That boundary needs to take full advantage of the technology, covering the limitations and exposing the benefits. Eric encouraged people to think about this envelope of some hardware and some exposed interfaces.

Someone suggested using XFS's block allocation mechanism. Ted Ts'o said that if we could solve multiple problems at once with Dave Chinner's block allocation scheme, it would give us a lot of power. Both ext4 and XFS have block allocation maps that keep track of both logical and physical block addresses, but keeping track of physical block addresses relies on getting information back from drives.

Sage Weil (Inktank) wondered how many hints the drivers would need to send to drives; For example, block A should be close to B, or A will be short-lived. James Bottomley mentioned that most people assume that the allocation table is at the beginning of the disk. Erez Zadok replied that if we can produce a generic enough abstraction layer, lots of people will use it. It could be used with SMR, but also with raw flash.

Error messages are another issue. Erez said that some drives produce an error if the block has not been written before being read, and someone else said that the driver could return all zeroes for initialized blocks. Erez said that currently, writing to the first block in a zone automatically resets all blocks in that zone, but is that the correct behavior? Bernard Metzler (IBM Zurich) wondered whether it's best to expose the full functionality of this storage class, as was done with flash by Fusion-io. In 10 years, non-volatile memory (NVM) will replace DRAM, so should we treat NVM like block devices or memory?

The mention of NVM quickly sidetracked the discussion. Christoph Hellwig suggested that addressing NVM should not be an either-or decision, while Ric Wheeler suggested that the user base could decide. James Bottomley wondered about error handling of NVM. Someone said they had tried using NVM as main memory, but it was still too slow, so they tried it via PCI. And they did try mmap, and the performance was not very good. Christoph responded that what they have called mmap needs to be fixed. The idea is to allow writes directly to a page in memory and use DMA for backing store.

James Bottomley pointed out that putting NVM on the PCI bus causes problems, because it makes it cross-domain. Ric Wheeler said this was a good example of the types of problems we need to know more about in that vendors can't talk about what they are doing, but then kernel developers and researchers don't know how to prepare for the new technology. Ted Ts'o said that he knows that Intel is paying several developers to work on using NVM as memory, and not on the PCI bus.

Erez Zadok said that this would be the first time we had a byte-addressable persistent memory, with a different point in the device space. He hoped it would not wind up like flash, where there are very limiting APIs controlling access.

After another short break, two students from the University of Wisconsin, Vijay Chidambaram and Thanumalayan Sankaranarayana Pillai, along with Jeff Darcy (Red Hat), addressed the group. They have a shared interest in new ways to flush data from in-memory cache (pages) to disk, although for different reasons. Vijay started the discussion by outlining the problem they had faced when needing to have ordered file system writes. The normal way of forcing a sync is via an fsync call on a particular file handle, but this has side effects. Sage Weil asked if they were trying to achieve ordering in a single file, and Vijay said yes. They had added a system call called osync to make ordered writes durable and had modified ext4 and were able to do this in a fairly performant way. Christoph Hellwig said that there really was no easy way of doing this without a sync, and that no one actually had written a functioning fbarrier (write data before metadata) call. Vijay said that they had created a new mode for a file, and Ted Ts'o asked if, when they osync, it involves a journal command. Vijay responded that the osync wraps the data in a journal command but does not sync it.

Jeff Darcy, a lead programmer for Gluster, said that he just wants to know what ext4 has done, because they need the correlation between user requests and journal commits. Christoph said they could easily do that in the kernel because each time they commit, there is a log sequence number. They could wait for the return from disk and use the log sequence number to implement osync. Ted said that it's the name that's the problem, like a cookie to identify the osync write. Jeff said that getting a cookie back that can be used to check if a write has been committed would be enough for their purposes.

Vijay stated that their main concern was knowing when data has become durable (not that different from Darcy's concern). Kai Shen (Rochester) said he liked the idea, but that it's not easy to use in comparison with atomicity. Kai had worked on a paper about msync, for doing an atomic write to one file. Ric Wheeler pointed out that Chris Mason (btrfs lead) has been pushing for an atomic write, which has not reached upstream (submitted for a kernel update) yet, but is based on work done by Fusion-io. Vijay complained that with ext4, you have metadata going to the journal and data going to the disk, but they need a way of doing this atomically. Sage Weil pointed out that there is no such thing as a rollback in file systems. Jeff Darcy said that all distributed databases have the same problem with durability.

Thanumalayan finally spoke, saying that they had discovered people doing fsync after every write and had been searching for patterns of fsync behavior themselves. When they shared the patterns they had uncovered with application developers, the developers' convictions about how things work were so strong they almost convinced him. The room erupted in laughter. On a more serious note, Thanumalayan asked, if he does a number of operations, such as renames, do they occur in order? Andreas Dilger said they don't have to, but get bunched up. Ted Ts'o mentioned that an fsync on one file implies that all metadata gets sync'd in ext4 and XFS, but not zfs or btrfs. However, that could change, and we might add a flag to control this behavior later.

Ric Wheeler liked this idea and suggested having a flag for async vs. fsync behaviors and being able to poll a selector for completion. Christoph Hellwig thought this wouldn't buy you much with existing file systems, but Ric thought it was worth something. Ted Ts'o said they could add a new asynchronous I/O type, and Christoph said that AIO sync has new opcode, IOCB_CMD_FSYNC, exposed to user space, that is not "wired up." Ethan Miller said it would be nice if you could request ordered writes. Greg Ganger (CMU) explained that there's a good reason why databases use transactions, and that they had tried doing this, and it's really difficult.

Erez Zadok commented that it's interesting to listen to the comments, as he has done 10 years of work on transactional storage, worked with umpteen PhDs on theses, and so on. The simplest interface to expose to users is start transaction/end transaction. To do this, you must go through the entire storage stack, from drivers to the page cache, which must be flushed in a particular order. Once he managed to do all that, he found that people didn't need fsync. Vijay piped up with "that's what we've been trying to do," and Sage Weil said they had tried doing this with btrfs and wondered if they ever got transactions to work. Erez replied that they had gotten transactional throughput faster than some, but Thanumalayan jumped in to say they had run into limits on how large a transaction could be. Vijay said that if you start writing, then use osync—it works like a transaction.

Ric Wheeler then commented that this is where kernel people throw up their hands and say show us how easy this is to do. Ted Ts'o wondered why it couldn't be done in user space (FUSE), and Ric replied that they need help from the kernel. Sage Weil had tried making every transaction checkpoint a btrfs snapshot, so they could roll back if they needed to. Jeff Darcy said that if you need multiple layers working together, you will have conflicting events. Thanumalayan just wants a file system with as much ordering as possible. He was experimenting with what happens to a file when a crash occurs, and said that POSIX allows a totally unrelated file to disappear. Christoph Hellwig shouted out that POSIX says nothing about crashes and power outages, and Jeff agreed that POSIX leaves this behavior undefined.

Jeff claimed he would be happy if he could get a cookie or transaction number, flush it with waiting, and select on that cookie for completion. Greg Ganger suggested he actually wanted more, to commit what he wants first. Vijay explained that the NoSQL developers they've talked to don't need durability for all things, but for some things, and that they have code, used for an SOSP 2013 paper, as an example. Jeff thought that NoSQL developers make assumptions about durability all the time, and about how fsync works. Ted Ts'o thought that all they needed to do was add new flags, or perhaps a new system call. Christoph Hellwig pointed out that there is sync_file_range, for just synchronizing file data within the given range, but not metadata, so it isn't very useful.

Kai Shen suggested that perhaps fsync should work more like msync, which includes an invalidate flag that makes msync appear atomic. Christoph said that older systems didn't have a unified buffer, so you had to write from the virtual memory system to the file system buffers. Ric Wheeler mentioned user space file systems, and Sage Weil suggested that it would be nice if you could limit the amount stored in the caches based on cgroup membership. James Bottomley stated that the infrastructure is almost all there for doing this with cgroups already.

Sage brought up another problem: that when a sync occurs, the disk gets really busy. Because of this, they (Inktank) actually try to keep track of how many blocks might be dirty so they can guess how long a sync might take. Jeff Darcy concurred, saying that they do something similar with Gluster. They buffer up as much as they can in the FUSE layer before pushing it into the kernel via a system call. James said that the Postgres people want the ability to manage write-ahead as well. Christoph clarified this, saying that they want to use mmap for reads, but control when to commit data (write) when it changes. They don't want the kernel storing the data on its own, but rather they want a backing store in place. This would be like mmap private.

James Bottomley said that the problem is related to journal lock scaling, when you want a lock for each subtree. You want to make the journal lock scale per subtree. Ted Ts'o pointed out that the last paper at FAST [2] covered this very topic. James pointed out that this would be a problem for containers, which are like hypervisors with only one kernel (like Solaris has in some form).

Kirill Korotaev (Parallels) then went to the front of the room to introduce another topic: FUSE performance. Most people think FUSE is slow, said Kirill, but they have seen up to 1 GB/s in real life. After that, bottlenecks slow more performance gains. To get past that, they need some interface to do kernel splicing and believe that would be quite a useful interface. Copying using pipes is very slow because pipes use mutex locks, and pipes don't work with UNIX sockets. What they basically need is the ability to do random reads of data with these buffers and to send them to a socket. Kirill also questioned why there's a requirement that data must be aligned in memory. Andrew Morton said that was a very old requirement for some devices, and James Bottomley said that's why there is a bounce buffer for doing alignment under the hood.

Kirill wanted to revive IOBuff, where they could attach to pages in user space, then send them to sockets. James pointed out that except for DirectIO, everything goes through the page cache, and moving data from one file to another is hard. Ric Wheeler asked if the interface was doable, and Kirill said that they think it is. Andrew Morton wondered if this was different from sendfile, and Sage Weil thought perhaps splice would work as well, but Kirill responded that neither work as well. Sage thought the problems could be fixed, but Kirill ended on the note that if it were easier, it would have been done before.

George Amvrosiadis (University of Toronto) introduced the topic of maintenance and traces. For durability, storage systems perform scrubbing (background reads) and fsck for integrity, but today those things need to be done online. The issue becomes how to do this without disturbing the actual workload. For scrubbing in btrfs, you get an upper bound on the number of requests that can be processed during scrubbing. Ric Wheeler, who worked at EMC before moving to Red Hat, asked how often do you want to scrub, and how much performance do you want to give up, and for how long.

George Amvrosiadis wanted to monitor traces and then use the amount of activity to decide when to begin scrubbing during what appeared to be idle time. Ric Wheeler said that while he was at EMC, he saw disks that were busy for years, and the only "idle time" was when the disks performed self-checks. Andreas Dilger asked if there was an idle priority, and George said they wanted to do this for btrfs. The problem is that all requests look the same, as maintenance requests look like other requests. Ric responded that they need a maintenance hint. Ted Ts'o said you'd need to tag the request all the way down to the block device layer, orthogonal to priority. George then stated that all they want to do is optimize when to schedule scrubbing. James Bottomley said they already have a mechanism for increasing the priority of requests, and perhaps they could also support decreasing priority. Ric replied that this sounded like the "hints" stuff from last year's Linux Filesystem Summit. If the file system supports it, although we can't guarantee it, we can at least allow it, so this sounds easy.

Ethan Miller wondered if you have devices with built-in intelligence, do you really want both the device and the kernel doing scrubbing? Ric answered that you want to scrub from the application down to the disk, checking the entire path. George Amvrosiadis then asked if scrubbing could cause more errors or increase the probability of errors. Ethan said that scrubbing has no effects; it's just reading. But Eric Reidel pointed out that all disk activity has some small probability of causing a problem, like smearing a particle on a disk platter. These probabilities are small, compared to the MTBF for a drive.

George Amvrosiadis still wanted some hint from the file system that work was about to begin. This brought up a tangent, where Ric Wheeler pointed out that batching up work for a device has caused problems in the past. Andreas Dilger explained that if just one thread were writing, you could get 1000 transactions per second, but if the application were threaded, and two threads were writing, the rate would go down to 250 per second. The problem occurred because the file system would wait a jiffy (4 ms at the time) trying to batch writes from threaded applications.

Greg Ganger said that there is a temptation to do all types of things at the SCSI driver level so it can do scheduling, and Ric Wheeler responded that the storage industry has been looking at hints from the file system for a long time. James Bottomley replied that the storage industry wanted hints about everything, a hundred hints, and Ted Ts'o added, not just hints, but 8–16 levels for each hint.

George Amvrosiadis then asked about getting traces of read/write activity, and Ethan Miller agreed, saying that they would like to have interesting traces shared. Ric Wheeler said that people have worked around this using filebench, and Greg added that it would be nice to have both the file and block level trace data.

Erez Zadok, the co-chair of FAST '15, said that USENIX is interested in promoting openness, so for the next FAST, when you submit a paper you can include whether you plan on sharing traces and/or code. Christoph Hellwig said just include a link to the code, and Ted Ts'o added that could be done once a paper has been accepted. Ethan Miller said that anonymization of traces is really hard, and they had experience working with NetApp on wireshark traces for a project in 2013. Erez pointed out that HP developed a standard, the Data Series format, and also developed public tools for converting to this format. He continued saying that EMC plans on releasing some traces they have been collecting for several years, and that a past student, Vasily Tarasov, had spent a week at their datacenters collecting statistics.

Several other topics were covered during the final hour: RDMA, dedup, and scalability, but your correspondent missed this in order to catch a flight to SCaLE 12x in Los Angeles.

I did appreciate getting to watch another summit in progress, and was reminded of evolution. The Linux kernel evolves, based on both what people want, but even more on what contributors actually do. And, as with natural evolution, most steps are small ones, because changing a lot at once is a risky maneuver.

References

[1] Tim Feldman and Garth Gibson, "Shingled Magnetic Recording: Areal Density Increase Requires New Data Management," *;login:* vol. 38, no. 3 (June 2013): https://www.usenix.org/publications/login/june-2013-volume-38-number-3/shingled-magnetic-recording-areal-density-increase.

[2] Junbin Kang, Benlong Zhang, Tianyu Wo, Chunming Hu, and Jinpeng Huai, "MultiLanes: Providing Virtualized Storage for OS-level Virtualization on Many Cores": https://www.usenix.org/conference/fast14/technical-sessions/presentation/kang.

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

REGISTER TODAY!

23rd USENIX Security Symposium

AUGUST 20–22, 2014 • SAN DIEGO, CA

The USENIX Security Symposium brings together researchers, practitioners, system programmers and engineers, and others interested in the latest advances in the security of computer systems and networks. The Symposium will include a 3-day technical program with refereed papers, invited talks, posters, panel discussions, and Birds-of-a-Feather sessions. Program highlights include:

Keynote Address by Phil Lapsley, author of *Exploding the Phone: The Untold Story of the Teenagers and Outlaws Who Hacked Ma Bell*

Invited Talk: "Battling Human Trafficking with Big Data" by Rolando R. Lopez, *Orphan Secure*

Panel Discussion: "The Future of Crypto: Getting from Here to Guarantees" with Daniel J. Bernstein, Matt Blaze, and Tanja Lange

Invited Talk: "Insight into the NSA's Weakening of Crypto Standards" by Joseph Menn, *Reuters*

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

www.usenix.org/sec14

Stay Connected...

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