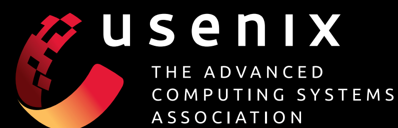
;login: logout



EXCLUSIVE ELECTRONIC EDITION

MAY 2013

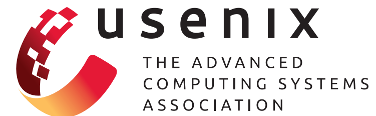
;login: *logout*

EXCLUSIVE ELECTRONIC EDITION MAY 2013

2 The Saddest Moment
James Mickens

5 The Disambiguator: Learning about Operating Systems
Selena Deckelmann

6 So many filesystems...
Rik Farrow



EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Rikki Endsley
rikki@usenix.org

PRODUCTION
Arnold Gatilao
Casey Henderson
Michele Nelson

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

©2013 USENIX Association
USENIX is a registered trademark of the USENIX Association. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. USENIX acknowledges all trademarks herein. Where those designations appear in this publication and USENIX is aware of a trademark claim, the designations have been printed in caps or initial caps.

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

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 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.

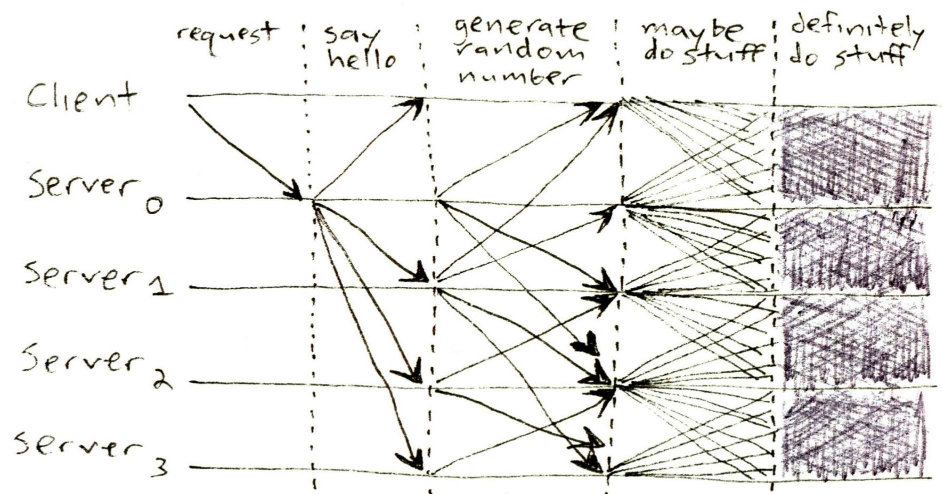


Figure 1: Typical Figure 2 from Byzantine fault paper: Our network protocol

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.

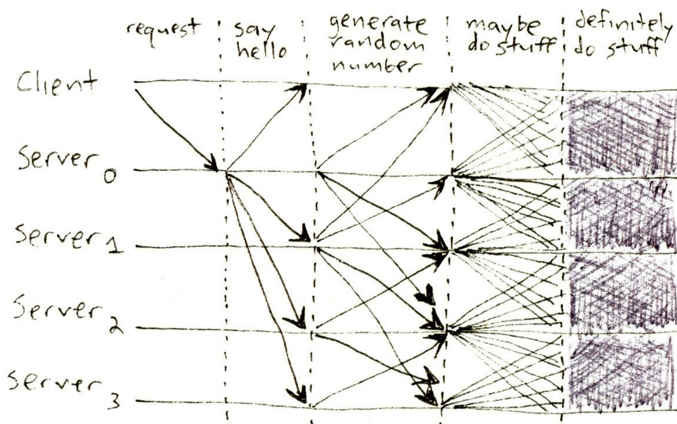
BRYAN: I verify that I heard that you want to go to lunch.

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.



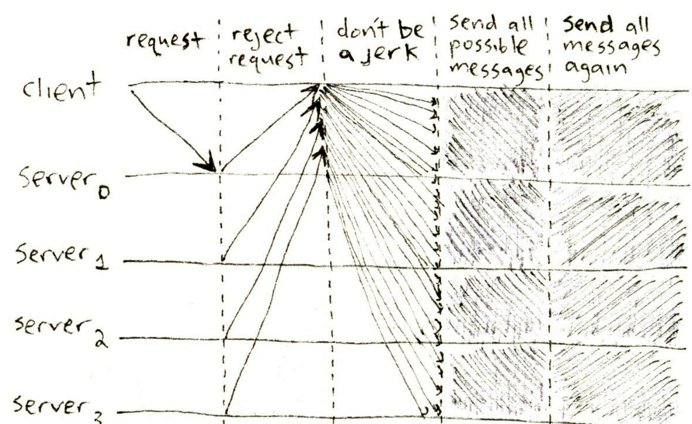


Figure 2: Our new protocol is clearly better.

The Saddest Moment

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.

The Disambiguator
Learning about Operating Systems

SELENA DECKELMANN

Selena Deckelmann is a major contributor to PostgreSQL and a data architect at Mozilla. She's been involved with free and open source software since 1995 and began running conferences for PostgreSQL in 2007. In 2012, she founded PyLadiesPDX, a portland chapter of PyLadies. Deckelmann founded Open Source Bridge and Postgres Open, and she speaks internationally about open source, databases ,and community. She is an advisor to the Ada Initiative, an organization dedicated to increasing the participation of women in open source and technology communities. You can find her on twitter (@selenamarie) and on her blog at chesnok.com. selena@chesnok.com

You want to learn about operating systems, and C appears to be the language used in examples you've seen. Now you wonder, "Do I need to learn C?"

My short answer to this is: No.

You don't need to write an operating system from scratch to learn about it. Most of the problems system administrators solve have nothing to do with C, even though many operating systems are written in C. Important concepts such as variables, flow control, loops, arrays, and input/output, are nearly the same in any language. Understand these ideas in one language, and you're on solid ground to learn them in another.

Learning about operating systems is really learning about resource management. Ask yourself questions about your own computer:

- Are the processes you think should be running actually running?
- How much RAM do your processes use over the course of a day?
- How fast are your disks?
- When will you need to upgrade your disks for speed or size?
- How do you apply what you learn to many systems, rather than just your laptop?

Answering these questions with programs you write yourself in shell scripts, Perl, Python, or Ruby will help you learn what you need to know. A book such as the UNIX and Linux System Administration Handbook [1] would also be a helpful guide.

I recently dropped into the classroom of Chris Bartlo, a high school computer science teacher. His kids learn HTML and CSS, Scratch, C++, and Java in their first three years. Now Bartlo is introducing a Python course because he saw first hand how productive a line of Python is when his kids had to solve a series of text parsing problems for a contest.

Bartlo's students can be productive in so many languages because they are in class every day, solving problems. And it's fun—they design and make games, they work in teams, and they plan out their work before they ever write a line of code.

The best way to learn a new language is to have a friend, mentor, or team learning it with you. Pick something your friends use, and you'll learn faster and have more fun.

For those of you picking up another programming language, you can find out what languages are popular on GitHub and Stack Overflow [2]; however, this is an imperfect measure. GitHub and Stack Overflow don't necessarily represent the majority of developers. These developers, though, are probably the trendsetters [3].

So, if you want to learn more about operating systems, start asking and answering questions about them. Use whatever programming language and environment works for you. For some, that could be C. But for me, Python works just fine.

References

[1] Nemeth, Evi, et al. UNIX and Linux System Administration Handbook. Prentice Hall, 2010. Print.

[2] The RedMonk Programming Language Rankings: January 2013: http://redmonk.com/sogrady/2013/02/28/language-rankings-1-13/.

[3] Developer as (Fashion) Designer: http://www.saturn-flyer.com/articles/2009/04/28/developer-as-fashion-designer/.

;login: logout

So many filesystems...

RIK FARROW

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

When I was at FAST 2009, I listened as a security researcher asked a prominent filesystem researcher why there are so many filesystems. This got me thinking about it: Why are there so many?

Early computers didn't even have filesystems: storage was tape, and some computers actually loaded card images of programs to be compiled then executed from tape. But once IBM began building dishwasher-sized disks, systems programmers needed to design data structures to organize both the data and the metadata that described the organization and attributes of that data.

A quick look at the Wikipedia page on filesystems [1] makes it clear that there are many different filesystems. In a talk by Ted Ts'o [2] in 2010, Ted points out that there is support for 66 filesystems in the Linux 2.6 kernel. I asked Ted about that, and he told me something that should have been immediately obvious to me: some filesystems have specific use cases. For example, there are network filesystems, such as nfs, nfs4, cifs, afs, and 9p. There are also filesystems for compatibility with other systems: fat/vfat/msdos, iso9660, hfs, ntfs, minixfs, hfs, qnx4, qnx6, and so on. And finally, there are cluster filesystems, such as ceph, gfs2, and ocfs2, and special purpose filesystems, for example, for working well with SSDs.

But what about the "big four" disk filesystems in Linux: ext3, ext4, XFS, and btrfs? Why is there more than a single filesystem that gets used with modern versions of Linux?

Different filesystems have different strengths, and sometimes, weaknesses. XFS was designed to work with very large files and directories, for example, and was the filesystem of choice for this reason for many years. Now ext4 and btrfs can also handle large files, although their limits are still smaller than XFS, few people will be using 16 terabyte files (the limit of testing for ext4, according to Ric Wheeler of Red Hat during his BoF at FAST '13).

Ext3 added journaling to ext2, a method that writes a journal, a list of changes, before committing those changes. The purpose of journaling was to make recovering from system crashes or power failures much quicker. Running fsck on large filesystems can take hours, but with journaling, restoring filesystem integrity takes just seconds.

The ext4 filesystem includes changes that extend the capabilities of ext3 so that it can handle larger files and directories. Ext4 also uses extents, rather than indirect blocks, to handle large files more efficiently. Before that, only XFS (of this group) used extents, essentially, ranges of virtual blocks, instead of the lists of blocks found in ext2 and ext3. Google uses ext4 for the base of its cluster filesystem, but without journaling. Google, like many cluster filesystem users, uses replication as a backup strategy; they decided not to use journaling, which includes a 10% performance hit, because they can rebuild systems instead of running fsck on large volumes.

Btrfs was designed to fulfill features created by Oracle's ZFS: snapshotting, checksums on data and metadata (for detecting silent corruption), and the ability to expand filesystems (even across device boundaries). With btrfs, you can have backups made every time a file

changes, and the checksums for these changes ripple up the directory hierarchy as well. So whereas btrfs works well for many uses, it would be a poor choice for databases and logfiles. In the 3.8 Linux kernel, support has been added for disabling checksums in files that change often.

Of these "big four", btrfs is the newest and just becoming stable enough for enterprise use. You might also have noticed that the newer filesystems, btrfs and ext4, have added features found in earlier filesystems, like those in XFS. So whereas XFS was once the only choice for very large filesystems, that has changed.

My security friend's interest in filesystems really had nothing to do with any of these issues: size, reliability, fast recovery, extensibility, or snapshotting. Simson Garfinkel was researching how much useful, and potentially dangerous, data was being left on discarded/recycled drives [4]. When you are scanning hundreds of drives looking for CAD files, personal information, and other sensitive material, having a plethora of filesystems is simply a nuisance.

References

[1] File system: http://en.wikipedia.org/wiki/File_system.

[2] Ted Ts'o, "Making Production-Ready Filesystems: A Case Study Using Ext4": ftp://ftp.kernel.org/pub/linux/kernel/people/tytso/presentations/stabilizing-ext4.pdf.

[3] Jonathan Corbet, "Why Filesystems Are Hard": http://lwn.net/Articles/370419/.

[4] Simson Garfinkel, "Read Data Corpus": http://simson.net/page/Real_Data_Corpus.

Why Join USENIX?

We support members' professional and technical development through many ongoing activities, including:

- Open access to research presented at our events
- Workshops on hot topics
- Conferences presenting the latest in research and practice
- LISA: The USENIX Special Interest Group for Sysadmins
- ;login:, the magazine of USENIX
- Student outreach

Your membership dollars go towards programs including:

- Open access policy: All conference papers and videos are immediately free to everyone upon publication
- Student program, including grants for conference attendance
- Good Works program

Helping our many communities share, develop, and adopt ground-breaking ideas in advanced technology

Join us at www.usenix.org

OPEN ACCESS

usenix

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION