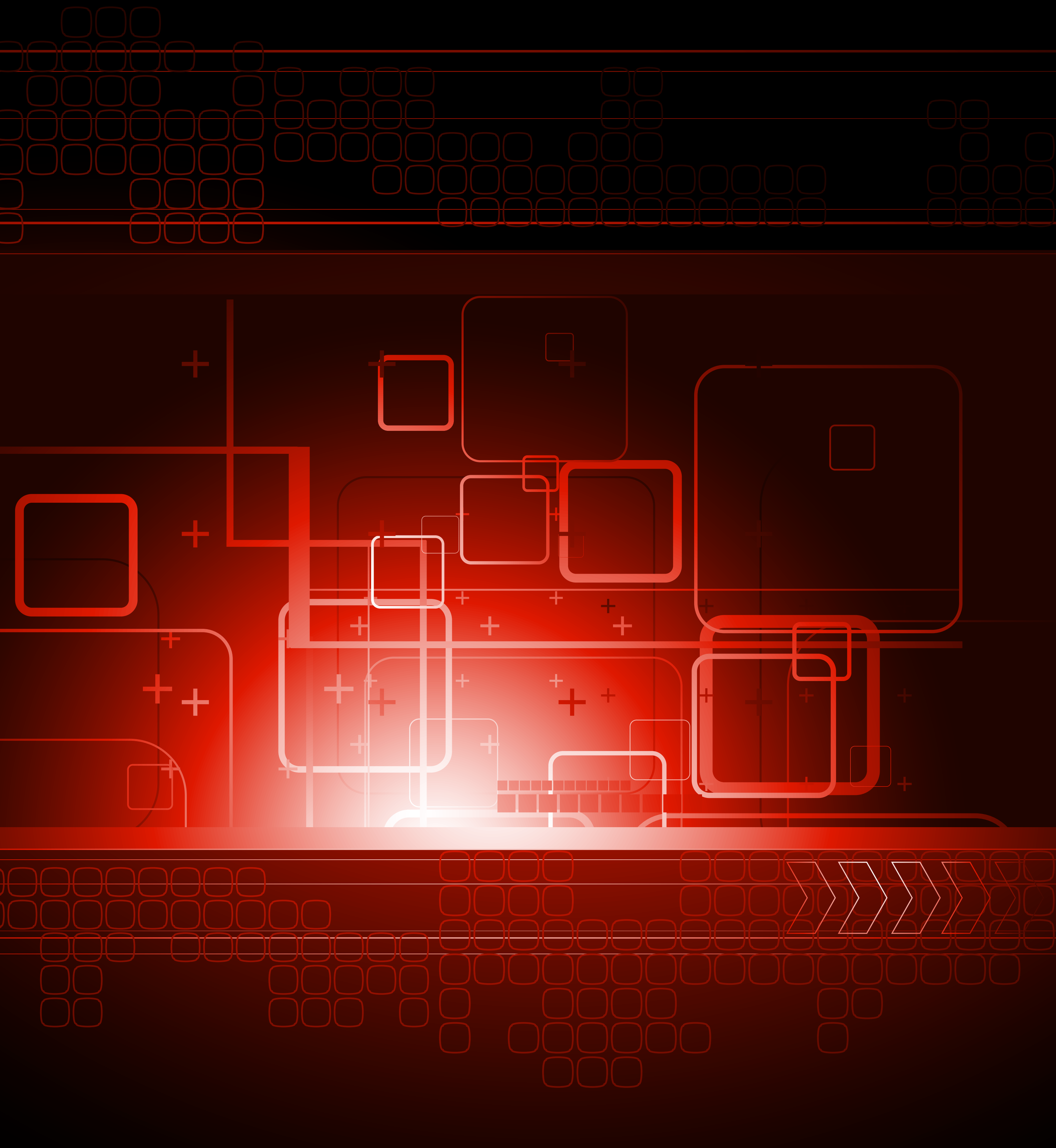
;login: logout

EXCLUSIVE ELECTRONIC EDITION NOVEMBER 2013

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION



;login: *logout*

EXCLUSIVE ELECTRONIC EDITION NOVEMBER 2013

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

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

Uncertain Infrastructures

MARK BURGESS

Mark Burgess is the CTO and Founder of CFEngine, formerly professor of Network and System Administration at Oslo University College, and the principal author of the Cfengine software. His the author of numerous books and papers on topics from physics, Network and System Administration, to fiction.
mark.burgess@cfengine.com

To our shame, one of the rarely voiced complaints one could level at the IT service industry is that we don't know how to make promises we can keep. In fact, we have not designed technology to keep anything; the main focus lies in building, tweaking, and fire-fighting, all within an increasingly fast-moving and disposable culture.

Against the backdrop of this uncertainty, we've invested in our reliance on technology. Smart devices enable and enhance our personal freedom in ways that are just too seductive to forego, and they interface with services that lie behind the scenes. In the new age of IT-powered commerce, the continuity of that lifestyle has come centre-stage. We used to talk about business continuity and disaster recovery, now we talk about continuous delivery of products to market, as well as continuous availability of services. We are starting to realize that the modern world is always on, and we will not accept anything less.

The builders and custodians of today's infrastructure have designed technology to be managed by direct commands or remote control, replacing manual error with amplified manual error. Errors are reported by independent monitors and shot down by further manual intervention: errors slain like dragons in a gaming experience. But this will not do for mission critical infrastructure.

Systems Thinking: Clockwork Uncertainty

We can really only promise a tiny number of things about the vastly complex environments we build. This does not give cause for complete certainty, but it can help to set expectations. We can promise certain aspects of behavior, albeit with margins for error, but we should also be clear: We build systems and essentially hope for the best; nothing we do can fully determines whether a system crosses a threshold into instability.

Over the years I've had the privilege to work with many smart people at installations of varying scale and complexity, often through the lens of CFEngine, and I've seen the issues first hand and been able to learn some lessons from them. There is growing recognition that systems are composed of both humans and automated processes, and that systemic complexity amplified by scale is the main cause of uncertainty. But few voices have invested in a science to understand and describe such complexity.

There are big ideas here, far too large to fit into this brief comment; so, I decided that it was time to write them into a book. In Search of Certainty: The Science of our Information Infrastructure is my new book [1], a popular science account of what I learned over the past 20 years.

Smart but Resilient

Infrastructure is getting smarter. Why? Because we want to get at stuff faster. The less work we have to do, the more accessible marketplaces for "stuff" are, the happier we seem. That means embedded computation.

References

[1] Burgess, Mark. In Search of Certainty: The Science of our Information Infrastructure. XtAxis Press, 2013. http://markburgess.org/certainty.html

[2] Gordon, J.E. The New Science of Strong Materials: Or Why We Don't Fall Through the Floor. Princeton University Press, 2006.

Uncertain Infrastructures

We've seen a lot of progressive thinking over the past few years under the banner of software defined systems. There has been talk about "anti-fragility," with chaos monkeys prodding systems to make them fail. Of course it is not the breaking of systems that makes them stronger, but adaptive processes behind the scenes that no one is really talking about. This is where the science lies.

The thrust of my activism in system administration has been that we have to start thinking more scientifically. Computer science is weak in the broad traditions of science as a tool of explanation. Its roots lie in deductive reasoning, which is only a small part of a post hoc picture. Managing systems is not just a case of test-based development, or of analyzing big data. Test-based development is like trying to pin the tail on a mule, unless you have some guiding principles and empirical foundations on which to home in on your design.

Semantics and Dynamics = Dev and Ops

I argue that there are two aspects to systems. I shall call these semantics and dynamics. Semantics are about purpose and intent. Dynamics are about behavior and performance. In some ways, these two aspects map on the dev (development) and ops (operations) in the the current parlance. Developers think mainly about purpose and intent, whereas operations engineers have to deal with actual behavior. DevOps tries to teach the message that you need to understand both of these aspects together in a unified way if you want to understand IT services beyond a trivial scale and complexity.

In fact, there is deep science here—and not just the signal lambda calculus that has gained the unfair attention of a small crowd of developers—the study of behavior is known to us as physics and it spans a plethora of different methods and issues. I don't have time to talk about them here, but I've tried to describe the key ideas in my book.

If Only Systems Were Deterministic…

If only systems would do as they're told, developers would have their way. Many people I meet still believe that systems are deterministic. But this Newtonian dogma was shattered in the 20th century. The history of scientific thinking tells us: The world is non-deterministic, get over it.

Computer science does itself a disservice by ignoring the main lesson of 20th century science, namely that the world is non-deterministic in fundamental ways. There is not even a well-developed theory of bugs. The push-button, imperative, API remote control approaches we use to instigate action today do not bring certainty. They offer a comfortable industrialization of process, but ultimately, by trying to remote control, we merely throw stuff over the wall and hope for the best. The only way to approach system reliability is to embrace the notion of indeterminism once and for all. It is about best effort.

Some things can help us here, such as building systems that are weakly coupled. System dependencies lead to strong coupling. If one thing fails, the system immediately transmits the failure to the next component. A weakly coupled system is fault tolerant.

Artificial Criticality

We escape from criticality by diversifying systems through redundancy. We never control systems, we merely keep their forces in balance. The knife edge of if-else programming is the radiation or asbestos of the software world. We stuff the walls full of this potentially dangerous automated reasoning, believing that it is there to protect us, when in fact it exposes us to an instability by the myriad pinpricks of a jostling environment.

Trying to conclude true or false from a highly complex environment is found to be the main cause of software unreliability. The reasoning for this is explained in the book.

Our thinking is still incredibly primitive, if we are expecting to scale reliable systems. We have given little evidence that we've understood the key issues of system automation in IT. Manufacturing and electronics have come a lot further. It's not only about how resilient the pieces are, but also about how they are put together. In several of the examples I've shown, the presence of regular maintenance could have prevented the gradual failure of the system.

The great pioneer of material science J.E. Gordon wrote [2] that: "The history of attempts to prevent cracks from spreading or evade their consequences is almost the history of engineering."

In the Comet airline disaster of 1954, microscopic cracks precipitated an avalanche failure that was so fast nothing could have prevented it from happening. In physical terms we would say that the rate of reaction dominated any process capable of preventing it. When there is a mismatch of dynamical scales like this, maintaining equilibrium is not possible. You are balancing on a knife edge. Semantics of design always give way to the dynamics of underlying reality.

There are two "answers" to this kind of failure: avoid stress concentrations, bottlenecks, and other points of failure; and use materials that catch the stress automatically by design, like the storm drain, like embedded glass and carbon fibers and alloys that spread load by deforming plastically. In IT terms, you want load balancing and failover without failure as part of the design, not as a late fire-fight.

External Intervention vs Embedded Smart Infrastructure

Why do we continue to make remote control systems that make the worst use of both humans and machines? Because we believe it's the only way to do it. But take a look at this picture of responding to a crisis.

- You wait for a crisis.
- You bring in a manual response (too late).
- You scale up the human operation by bringing power tools.

Now think about how simple drainage prevents most flooding as an entirely automated embedded system. We are obsessed by the manual intervention. It is a sign of technological immaturity. It is even more apparent in the way we attempt to orchestrate systems, using simplistic flow-chart thinking as a model of a highly parallel and distributed environment.

There are three phases to the system lifecycle that we need to rein in. We think very differently about each.

- **Planning:** Here we tend to think in terms of broad block semantics (boxes with arrows between) or workflows.
- **Operations:** A highly dynamic and parallel phase, where overt flow thinking is a hindrance / bottleneck.
- **Evaluation:** Here we look for artificial and misleading hindsight narratives about successes and failures.

Rimsky Korsakov would have rolled his eyes at contemporary descriptions of orchestration. Orchestration of total systems lies in the planning of highly parallel operations. We might only remember a specific storyline in hindsight—perhaps a good or a bad experience. This is how we usually describe the complex system, but it is not a true representation of it.

We have the opportunity to make introspective systems that merge semantics and dynamics into a unified picture. That will only happen when we remove the artificial distinction between development, configuration, operation, and monitoring.

In Search of Certainty

What does it mean to be certain about something? How do we make a reliable infrastructure for society?

Absolute certainty and determinism are myths. We can only do our best. As small forces in an environment that permits us islands of temporary calm, we must try to understand the bigger picture. There are three main issues: scale, complexity, and lack of knowledge.

Twenty years after I began CFEngine and my own research into these matters, it seemed time to tell the story of the thinking that went into it. My own interest has meandered around many topics within the scope of IT operations, and I have tried to describe how these pieces fit together in the book, but the main core of it can be understood easily as a simple-minded quest of a physicist to understand a system.

What I hope is that my book starts a discussion that shows how to apply some of the traditions of science to a subject that has ridden mainly on the coat-tails of engineering. How do we make promises we can keep? By understanding the nature of certainty itself.

If we take certainty seriously, we need to think carefully about how software is designed. We can't just throw software logic over the wall for operations to catch. We need to build for intrinsic stability from the outset through true automation. And, even then, we'll need to perform continuous maintenance, just to be sure(ish).

The Night Watch

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

As a highly trained academic researcher, I spend a lot of time trying to advance the frontiers of human knowledge. However, as someone who was born in the South, I secretly believe that true progress is a fantasy, and that I need to prepare for the end times, and for the chickens coming home to roost, and fast zombies, and slow zombies, and the polite zombies who say "sir" and "ma'am" but then try to eat your brain to acquire your skills. When the revolution comes, I need to be prepared; thus, in the quiet moments, when I'm not producing incredible scientific breakthroughs, I think about what I'll do when the weather forecast inevitably becomes RIVERS OF BLOOD ALL DAY EVERY DAY. The main thing that I ponder is who will be in my gang, because the likelihood of post-apocalyptic survival is directly related to the size and quality of your rag-tag group of associates. There are some obvious people who I'll need to recruit: a locksmith (to open doors); a demolitions expert (for when the locksmith has run out of ideas); and a person who can procure, train, and then throw snakes at my enemies (because, in a world without hope, snake throwing is a reasonable way to resolve disputes). All of these people will play a role in my ultimate success as a dystopian warlord philosopher. However, the most important person in my gang will be a systems programmer. A person who can debug a device driver or a distributed system is a person who can be trusted in a Hobbesian nightmare of breathtaking scope; a systems programmer has seen the terrors of the world and understood the intrinsic horror of existence. The systems programmer has written drivers for buggy devices whose firmware was implemented by a drunken child or a sober goldfish. The systems programmer has traced a network problem across eight machines, three time zones, and a brief diversion into Amish country, where the problem was transmitted in the front left hoof of a mule named Deliverance. The systems programmer has read the kernel source, to better understand the deep ways of the universe, and the systems programmer has seen the comment in the scheduler that says "DOES THIS WORK LOL," and the systems programmer has wept instead of LOLed, and the systems programmer has submitted a kernel patch to restore balance to The Force and fix the priority inversion that was causing MySQL to hang. A systems programmer will know what to do when society breaks down, because the systems programmer already lives in a world without law.

The Night Watch

Listen: I'm not saying that other kinds of computer people are useless. I believe (but cannot prove) that PHP developers have souls. I think it's great that database people keep trying to improve select-from-where, even though the only queries that cannot be expressed using select-from-where are inappropriate limericks from "The Canterbury Tales." In some way that I don't yet understand, I'm glad that theorists are investigating the equivalence between five-dimensional Turing machines and Edward Scissorhands. In most situations, GUI designers should not be forced to fight each other with tridents and nets as I yell "THERE ARE NO MODAL DIALOGS IN SPARTA." I am like the Statue of Liberty: I accept everyone, even the wretched and the huddled and people who enjoy Haskell. But when things get tough, I need mission-critical people; I need a person who can wear night-vision goggles and descend from a helicopter on ropes and do classified things to protect my freedom while country music plays in the background. A systems person can do that. I can realistically give a kernel hacker a nickname like "Diamondback" or "Zeus Hammer." In contrast, no one has ever said, "These semi-transparent icons are really semi-transparent! IS THIS THE WORK OF ZEUS HAMMER?"

I picked that last example at random. You must believe me when I say that I have the utmost respect for HCI people. However, when HCI people debug their code, it's like an art show or a meeting of the United Nations. There are tea breaks and witticisms exchanged in French; wearing a non-functional scarf is optional, but encouraged. When HCI code doesn't work, the problem can be resolved using grand theories that relate form and perception to your deeply personal feelings about ovals. There will be rich debates about the socioeconomic implications of Helvetica Light, and at some point, you will have to decide whether serifs are daring statements of modernity, or tools of hegemonic oppression that implicitly support feudalism and illiteracy. Is pinching-and-dragging less elegant than circling-and-lightly-caressing? These urgent mysteries will not solve themselves. And yet, after a long day of debugging HCI code, there is always hope, and there is no true anger; even if you fear that your drop-down list should be a radio button, the drop-down list will suffice until tomorrow, when the sun will rise, glorious and vibrant, and inspire you to combine scroll bars and left-clicking in poignant ways that you will commemorate in a sonnet when you return from your local farmer's market.

This is not the world of the systems hacker. When you debug a distributed system or an OS kernel, you do it Texas-style. You gather some mean, stoic people, people who have seen things die, and you get some primitive tools, like a compass and a rucksack and a stick that's pointed on one end, and you walk into the wilderness and you look for trouble, possibly while using chewing tobacco. As a systems hacker, you must be prepared to do savage things, unspeakable things, to kill runaway threads with your bare hands, to write directly to network ports using telnet and an old copy of an RFC that you found in the Vatican. When you debug systems code, there are no high-level debates about font choices and the best kind of turquoise, because this is the Old Testament, an angry and monochromatic world, and it doesn't matter whether your Arial is Bold or Condensed when people are covered in boils and pestilence and Egyptian pharaoh oppression. HCI people discover bugs by receiving a concerned email from their therapist. Systems people discover bugs by waking up and discovering that their first-born children are missing and "ETIMEDOUT " has been written in blood on the wall.

What is despair? I have known it—hear my song. Despair is when you're debugging a kernel driver and you look at a memory dump and you see that a pointer has a value of 7. THERE IS NO HARDWARE ARCHITECTURE THAT IS ALIGNED ON 7. Furthermore, 7 IS TOO SMALL AND ONLY EVIL CODE WOULD TRY TO ACCESS SMALL NUMBER MEMORY. Misaligned, small-number memory accesses have stolen decades from my life. The only things worse than misaligned, small-number memory accesses are accesses with aligned buffer pointers, but impossibly large buffer lengths. Nothing ruins a Friday at 5 P.M. faster than taking one last pass through the log file and discovering a word-aligned buffer address, but a buffer length of NUMBER OF ELECTRONS IN THE UNIVERSE. This is a sorrow that lingers, because a 2^{893} byte read is the only thing that both Republicans and Democrats agree is wrong. It's like, maybe Medicare is a good idea, maybe not, but there's no way to justify reading everything that ever existed a jillion times into a mega-jillion sized array. This constant war on happiness is what non-systems people do not understand about the systems world. I mean, when a machine learning algorithm mistakenly identifies a cat as an elephant, *this is actually hilarious*. You can print a picture of a cat wearing an elephant costume and add an ironic caption that will entertain people who have middling intellects, and you can hand out copies of the photo at work and rejoice in the fact that everything is still fundamentally okay. There is nothing funny to print when you have a misaligned memory access, because your machine is dead and there are no printers in the spirit world. An impossibly large buffer error is even worse, because these errors often linger in the background, quietly overwriting your state with evil; if a misaligned memory access is like a criminal burning down your house in a fail-stop manner, an impossibly large buffer error is like a criminal who breaks into your house, sprinkles sand atop random bedsheets and toothbrushes, and then waits for you to slowly discover that your world has been tainted by madness. Indeed, the common discovery mode for an impossibly large buffer error is that your program seems to be working fine, and then it tries to display a string that should say “Hello world,” but instead it prints “#a[5]:3!” or another syntactically correct Perl script, and you’re like WHAT THE HOW THE, and then you realize that your prodigal memory accesses have been stomping around the heap like the Incredible Hulk when asked to write an essay entitled “Smashing Considered Harmful.”

You might ask, “Why would someone write code in a grotesque language that exposes raw memory addresses? Why not use a modern language with garbage collection and functional programming and free massages after lunch?” Here’s the answer: Pointers are real. They’re what the hardware understands. Somebody has to deal with them. You can’t just place a LISP book on top of an x86 chip and hope that the hardware learns about lambda calculus by osmosis. Denying the existence of pointers is like living in ancient Greece and denying the existence of Krackens and then being confused about why none of your ships ever make it to Morocco, or Ur-Morocco, or whatever Morocco was called back then. Pointers are like Krackens—real, living things that must be dealt with so that polite society can exist. Make no mistake, I don’t *want* to write systems software in a language like C++. Similar to the Necronomicon, a C++ source code file is a wicked, obscure document that’s filled with cryptic incantations and forbidden knowledge. When it’s 3 A.M., and you’ve been debugging for 12 hours, and you encounter a virtual static friend protected volatile templated function pointer, you want to go into hibernation and awake as a werewolf and then find the people who wrote the C++ standard and bring ruin to the things that they love. The C++ STL, with its dyslexia-inducing syntax blizzard of colons and angle brackets, guarantees that if you try to declare any reasonable data structure, your first seven attempts will result in compiler errors of Wagnerian fierceness:

```
Syntax error: unmatched thing in thing from std::nonstd::__
map<_Cyrillic, _$$$dollars>const basic_string< epic_
mystery,mongoose_traits &lt; char>, __default_alloc_<casual_
Fridays = maybe>>
```

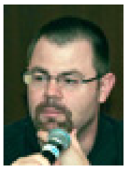
One time I tried to create a list<map<int>>, and my syntax errors caused the dead to walk among the living. Such things are clearly unfortunate. Thus, I fully support high-level languages in which pointers are hidden and types are strong and the declaration of data structures does not require you to solve a syntactical puzzle generated by a malevolent extraterrestrial species. That being said, if you find yourself drinking a martini and writing programs in garbage-collected, object-oriented Esperanto, be aware that the only reason that the Esperanto runtime works is because there are systems people who have exchanged any hope of losing their virginity for the exciting opportunity to think about hex numbers and their relationships with the operating system, the hardware, and ancient blood rituals that Bjarne Stroustrup performed at Stonehenge.

Perhaps the worst thing about being a systems person is that other, non-systems people think that they understand the daily tragedies that compose your life. For example, a few weeks ago, I was debugging a new network file system that my research group created. The bug was inside a kernel-mode component, so my machines were crashing in spectacular and vindictive ways. After a few days of manually rebooting servers, I had transformed into a shambling, broken man, kind of like a computer scientist version of Saddam Hussein when he was pulled from his bunker, all scraggly beard and dead eyes and florid, nonsensical ramblings about semi-imagined enemies. As I paced the hallways, muttering Nixonian rants about my code, one of my colleagues from the HCI group asked me what my problem was. I described the bug, which involved concurrent threads and corrupted state and asynchronous message delivery across multiple machines, and my coworker said, “Yeah, that sounds bad. Have you checked the log files for errors?” I said, “Indeed, I would do that if I hadn’t broken *every component that a logging system needs to log data*. I have a network file system, and I have broken the network, and I have broken the file system, and my machines crash when I make eye contact with them. I HAVE NO TOOLS BECAUSE I’VE DESTROYED MY TOOLS WITH MY TOOLS. My only logging option is to hire monks to transcribe the subjective experience of watching my machines die as I weep tears of blood.” My coworker, in an earnest attempt to sympathize, recounted one of his personal debugging stories, a story that essentially involved an addition operation that had been mistakenly replaced with a multiplication operation. I listened to this story, and I said, “Look, I get it. Multiplication is not addition. This has been known for years. However, multiplication and addition are at least related. Multiplication is like addition, but with more addition. Multiplication is a grown-up pterodactyl, and addition is a baby pterodactyl. Thus, in your debugging story, your code is wayward, but it basically has the right idea. In contrast, there is no family-friendly GRE analogy that relates what my code should do, and what it is actually doing. I had the modest goal of translating a file read into a network operation, and now my machines have tuberculosis and orifice containment issues. Do you see the difference between our lives? When you asked a girl to the prom, you discovered that her father was a cop. When I asked a girl to the prom, I DISCOVERED THAT HER FATHER WAS STALIN.”

In conclusion, I’m not saying that everyone should be a systems hacker. GUIs are useful. Spell-checkers are useful. I’m glad that people are working on new kinds of bouncing icons because they believe that humanity has solved cancer and homelessness and now lives in a consequence-free world of immersive sprites. That's exciting, and I wish that I could join those people in the 27th century. But I live here, and I live now, and in my neighborhood, *people are dying in the streets*. It's like, French is a great idea, but nobody is going to invent French if they're constantly being attacked by bears. Do you see? SYSTEMS HACKERS SOLVE THE BEAR MENACE. Only through the constant vigilance of my people do you get the freedom to think about croissants and subtle puns involving the true father of Louis XIV. So, if you see me wandering the halls, trying to explain synchronization bugs to confused monks, rest assured that every day, in every way, it gets a little better. For you, not me. I'll always be furious at the number 7, but such is the hero's journey.

What Every Admin Should Know About Email

JOE BROCKMEIER



Joe Brockmeier works on the Open Source and Standards team for Red Hat, and is a PMC member for the Apache CloudStack project. Joe has a long history of involvement with Linux and open source, and has also worked for Novell as the openSUSE community manager. Brockmeier is a recovering technology journalist, and has written for ReadWriteWeb, Linux.com, LWN, Linux Magazine, NetworkWorld, ZDNet, and many others.

I'm regularly taken aback by how far computers and computing have come since I started futzing with computers in 1995. The tools available today are *astounding* compared to what I was using in 1995.

One of the minor exceptions, of course, is email. Yes, email clients have improved in the past 18 years, but not by a lot. The basics are pretty much the same.

Sadly, not only has the software failed to evolve significantly, people's use of email has largely not improved since 1995, either. Actually, its use has degraded significantly in the interim. By that, I mean that what was widely regarded as "good netiquette" in 1995 is largely disregarded by folks sending email in a corporate setting. That's a pity, because what was good practice in 1995 is still best practice today, though perhaps for different reasons.

For instance, sending large attachments via email used to be considered a no-no to because many folks would be connecting via dial-up. Really, really slow dial-up. Today? We may all have super-speedy Internet at home, but when we're on the road? Not necessarily. Spotty mobile coverage, lousy hotel Wi-Fi, and ridiculous data roaming charges for international travelers are all good reasons for users to consider the size of their messages before sending.

Because we all spend, literally, hours every day corresponding with people via email, how others send email is not just a matter of preference; it's actually a difference of "you're making my life easier" or "you're making my life harder."

Work vs. Personal Mail

Note that this list is related to email exchanged in a work setting (including open source developer mailing lists where work is being done) and not personal email. What's appropriate for casual, one-on-one conversations is different from what's appropriate for productive conversations.

For instance, if someone top-posts a response to "Can we meet for the movie at 7 P.M.?," it's really no biggie. Top-posting that requires a recipient to scroll backwards through a six-message conversation trying to figure out what the hell the conversation was about is just rude.

We all know top-posting is evil, but there's more to good email etiquette than not top-posting:

- **Don't shotgun emails.** Just because a person has more than one email account, it doesn't mean you should send a piece of mail to all of them. Pick one. Otherwise you're just creating a mess the other person has to clean up twice.
- **Avoid CC'ing people in emails to a list.** Some lists and/or mail clients are configured so that hitting Reply will send a note to the original sender rather than the list. Others are configured to send mail just to the list. In as much as possible, if you're

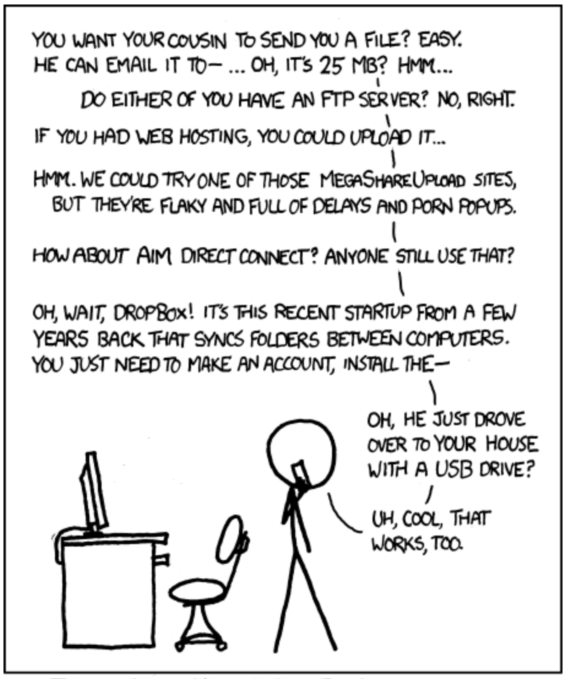
having a conversation on a list, don't also CC individual users to whom you may be replying directly. They don't need two copies of the message.

- **Use meaningful subject lines.** When you compose an email, try to make sure the subject is useful to the recipients and descriptive of the message you're sending. If it's a short message, you can even put the entire thing in the subject and put "[EOM]" afterwards.
- **If the topic of a thread changes, change the subject.** This goes back to long email threads on the corporate side, or long discussions on the -dev lists for projects. You start on Topic A, but mid-way through the discussion, someone decides to bring up Topic B, which is totally unrelated or only quasi-related to the topic at hand. This means that people skimming email have *no clue* that the thread with the subject about the first topic has changed to something relevant to them. Or, equally annoying, a topic they *were* interested in following has now devolved into something else entirely. (This can also be known as attempting to hijack a discussion.)
- **Don't just reply to an unrelated message to send an email.** This one drives me *bonkers* because I'll try to arrange my inbox by conversation, and an email about one thing will be buried in a long-dead conversation about something totally unrelated.
- **Trim your mails.** If you're replying to one sentence in a 3,000-word email, cut out everything but the sentence you're replying to and reply to that.
- **Don't use HTMLized email.** Yeah, I'm a crusty old Linux guy and still use the Mutt client to read a lot of my email. For far too many reasons to go into in this article, I despise HTMLized email. (Again, work. Personal use? Whatever makes you happy. But it doesn't belong in a professional setting.)
- **Have a signature.** Have an email signature, preferably one that gives a clue who you are, and perhaps other methods of reaching you. Keep it short, though. Under no circumstances should you include a bunch of logos or images in your signature. (See above about "don't use HTMLized email.")
- **Drop the legal boilerplate.** A footer on your email telling someone how to handle your message when they haven't agreed to your terms is not likely to be enforceable. It's doubly annoying when the footer is longer than the message itself.
- **Avoid surprise CCs.** Generally, adding someone to a discussion without announcing it is rude; however, there are exceptions, for example, when the original sender specifically requests that other relevant parties be added if necessary.
- **Avoid improper use of CC.** If you need to send a blanket announcement or forward to a bunch of people, use BCC instead of CC. I don't want 20 follow-up replies that are totally irrelevant to me just because people blindly click "Reply All."
- **Use Reply All sparingly.** The corollary to the above rule is to think before hitting Reply All. Do *all* the people in the discussion need to see your reply? Maybe, but think twice.
- **Do not reply to digests.** Frankly, I am against allowing digests for mailing lists, but they're probably here to stay. If you want to lurk, fine, have fun. If you wish to reply? Do *not* reply to a digest, especially without changing the subject to be appropriate or trimming the message so that everyone else has to slog through a day or week's worth of email to read your reply to *one* message in the bunch.
- **Follow instructions for using mailing lists.** People who reply to a mailing list with "unsubscribe" instead of following the instructions clearly printed in the footer of about 98% of mailing list messages should be deprived of computer access for at least a week.

I could go on. And on. Using business email boils down to being considerate of others in your communications. I understand, for instance, that top-posting is perfectly reasonable for a two-word reply sent from a phone. It is not, however, a reasonable approach when replying to a lengthy email with a likewise lengthy reply addressing multiple parts of the email.

Now if you'll excuse me, I have a bunch of email to process.

xkcd


YOU WANT YOUR COUSIN TO SEND YOU A FILE? EASY. HE CAN EMAIL IT TO— ... OH, IT'S 25 MB? HMM...
DO EITHER OF YOU HAVE AN FTP SERVER? NO, RIGHT.
IF YOU HAD WEB HOSTING, YOU COULD UPLOAD IT...
HMM. WE COULD TRY ONE OF THOSE MEGASHAREUPLOAD SITES, BUT THEY'RE FLAKY AND FULL OF DELAYS AND PORN POPUPS.
HOW ABOUT AIM DIRECT CONNECT? ANYONE STILL USE THAT?
OH, WAIT, DROPBOX! IT'S THIS RECENT STARTUP FROM A FEW YEARS BACK THAT SYNCS FOLDERS BETWEEN COMPUTERS. YOU JUST NEED TO MAKE AN ACCOUNT, INSTALL THE—
OH, HE JUST DROVE OVER TO YOUR HOUSE WITH A USB DRIVE?
UH, COOL, THAT WORKS, TOO.

I LIKE HOW WE'VE HAD THE INTERNET FOR DECADES, YET "SENDING FILES" IS SOMETHING EARLY ADOPTERS ARE STILL FIGURING OUT HOW TO DO.

xkcd.com

Why Join USENIX?

OPEN ACCESS

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

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION