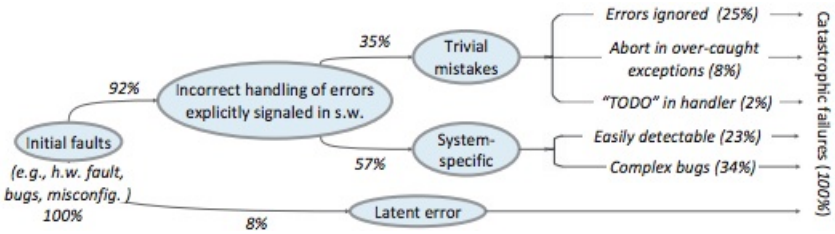
I love reading postmortems. They're educational, but unlike most educational docs, they tell an entertaining story. I've spent a decent chunk of time reading postmortems at both Google and Microsoft. I haven't done any kind of formal analysis on the most common causes of bad failures (yet), but there are a handful of postmortem patterns that I keep seeing over and over again.

Error Handling

Proper error handling code is hard. Bugs in error handling code are a major cause of *bad* problems. This means that the probability of having sequential bugs, where an error causes buggy error handling code to run, isn't just the independent probabilities of the individual errors multiplied. It's common to have cascading failures cause a serious outage. There's a sense in which this is obvious – error handling is generally regarded as being hard. If I mention this to people they'll tell me how obvious it is that a disproportionate number of serious postmortems come out of bad error handling and cascading failures where errors are repeatedly not handled correctly. But despite this being "obvious", it's not so obvious that sufficient test and static analysis effort are devoted to making sure that error handling works.

For more on this, Ding Yuan et al. have a great paper and talk: Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. The paper is basically what it says on the tin. The authors define a critical failure as something that can take down a whole cluster or cause data corruption, and then look at a couple hundred bugs in Cassandra, HBase, HDFS, MapReduce, and Redis, to find 48 critical failures. They then look at the causes of those failures and find that most bugs were due to bad error handling. 92% of those failures are actually from errors that are handled incorrectly.



Drilling down further, 25% of bugs are from simply ignoring an error, 8% are from catching the wrong exception, 2% are from incomplete TODOs, and another 23% are "easily detectable", which are defined as cases where "the error handling logic of a non-fatal error was so wrong that any statement coverage testing or more careful code reviews by the developers would have caught the bugs". By the way, this is one reason I don't mind Go style error handling, despite the common complaint that the error checking code is cluttering up the main code path. If you care about building robust systems, the error checking code is the main code!

The full paper has a lot of gems that that I mostly won't describe here. For example, they explain the unreasonable effectiveness of Jepsen (98% of critical failures can be reproduced in a 3 node cluster). They also dig into what percentage of failures are non-deterministic (26% of their sample), as well as the causes of non-determinism, and create a static analysis tool that can catch many common error-caused failures.

Configuration

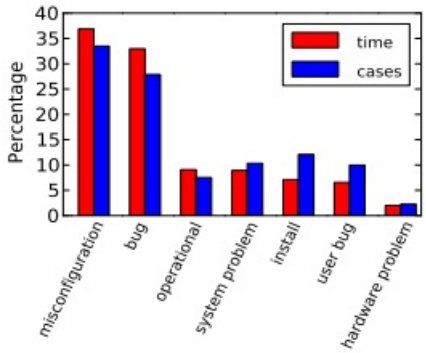
Configuration bugs, not code bugs, are the most common cause I've seen of really bad outages. When I looked at publicly available postmortems, searching for "global outage postmortem" returned about 50% outages caused by configuration changes. Publicly available postmortems aren't a representative sample of all outages, but a random sampling of postmortem databases also reveals that config changes are responsible for a disproportionate fraction of extremely bad outages. As with error handling, I'm often told that it's obvious that config changes are scary, but it's not so obvious that most companies test and stage config changes like they do code changes.

Except in extreme emergencies, risky code changes are basically never simultaneously pushed out to all machines because of the risk of taking down a service company-wide. But it seems that every company has to learn the hard way that seemingly benign config changes can also cause a company-wide service outage. For example, this was the cause of the infamous November 2014 Azure outage. I don't mean to pick on MS here; their major competitors have also had serious outages for similar reasons, and they've all put processes into place to reduce the risk of that sort of outage happening again.

I don't mean to pick on large cloud companies, either. If anything, the situation there is better than at most startups, even very well funded ones. Most of the "unicorn" startups that I know of don't have a proper testing/staging environment that lets them test risky config changes. I can understand why – it's often hard to set up a good QA environment that mirrors prod well enough that config changes can get tested, and like driving without a seatbelt, nothing bad happens the vast majority of the

time. If I had to make my own seatbelt before driving my car, I might not drive with a seatbelt either. Then again, if driving without a seatbelt were as scary as making config change, I might consider it.

Back in 1985, Jim Gray observed that "operator actions, system configuration, and system maintence was the main source of failures – 42%". Since then, there have been a variety of studies that have found similar results. For example, Rabkin and Katz found the following causes for failures:



Hardware

Basically every part of a machine can fail. Many components can also cause data corruption, often at rates that are much higher than advertised. For example, Schroeder, Pinherio, and Weber found DRAM error rates were more than an order of magnitude worse than advertised. The number of silent errors is staggering, and this actually caused problems for Google back before they switched to ECC RAM. Even with error detecting hardware, things can go wrong; relying on ethernet checksums to protect against errors is unsafe and I've personally seen malformed packets get passed through as valid packets. At scale, you can run into more undetected errors than you expect, if you expect hardware checks to catch hardware data corruption.

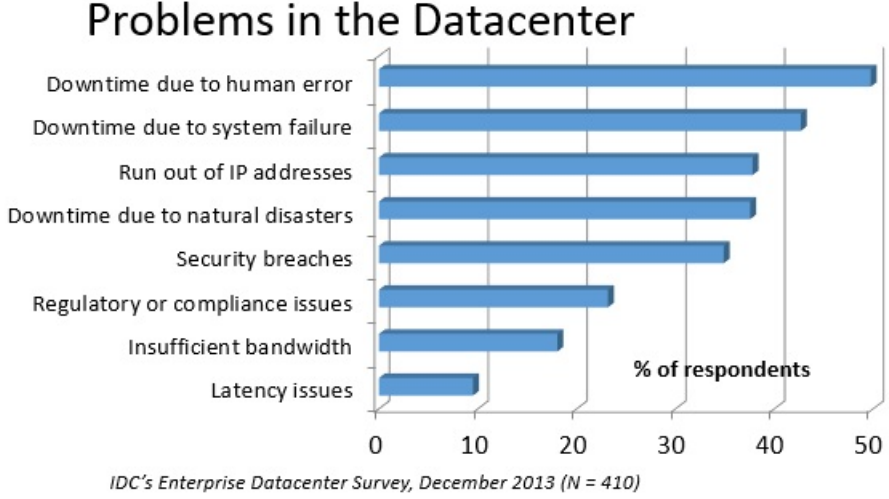
Failover from bad components can also fail. This AWS failure tells a typical story. Despite taking reasonable sounding measures to regularly test the generator power failover process, a substantial fraction of AWS East went down when a storm took out power and a set of backup generators failed to correctly provide power when loaded.

Humans

This section should probably be called process error and not human error since I consider having humans in a position where they can accidentally cause a catastrophic failure to be a process bug. It's generally accepted that, if you're running large scale systems, you have to have systems that are robust to hardware failures. If you do the math on how often machines die, it's obvious that systems that aren't robust to hardware failure cannot be reliable. But humans are even more error prone than machines. Don't get me wrong, I like humans. Some of my best friends are human. But if you repeatedly put a human in a position where they can cause a catastrophic failure, you'll eventually get a catastrophe. And yet, the following pattern is still quite common:

> Oh, we're about to do a risky thing! Ok, let's have humans be VERY CAREFUL about executing the risky operation. Oops! We now have a global outage.

Postmortems that start with "Because this was a high risk operation, foobar high risk protocol was used" are ubiquitous enough that I now think of extra human-operated steps that are done to mitigate human risk as an ops smell. Some common protocols are having multiple people watch or confirm the operation, or having ops people standing by in case of disaster. Those are reasonable things to do, and they mitigate risk to some extent, but in many postmortems I've read, automation could have reduced the risk a lot more or removed it entirely. There are a lot of cases where the outage happened because a human was expected to flawlessly execute a series of instructions and failed to do so. That's exactly the kind of thing that programs are good at! In other cases, a human is expected to perform manual error checking. That's sometimes harder to automate, and a less obvious win (since a human might catch an error case that the program misses), but in most cases I've seen it's still a net win to automate that sort of thing.



In an IDC survey, respondents voted human error as the most troublesome cause of problems in the datacenter.

One thing I find interesting is how underrepresented human error seems to be in public postmortems. As far as I can tell, Google and MS both have substantially more automation than most companies, so I'd expect their postmortem databases to contain proportionally fewer human error caused outages than I see in public postmortems, but in fact it's the opposite. My guess is that's because companies are less likely to write up public postmortems when the root cause was human error enabled by risky manual procedures. A prima facie plausible alternate reason is that improved technology actually increases the fraction of problems caused by humans, which is true in some industries, like flying. I suspect that's not the case here due to the sheer number of manual operations done at a lot of companies, but there's no way to tell for sure without getting access to the postmortem databases at multiple companies. If any company wants to enable this analysis (and others) to be done (possibly anonymized), please get in touch.

Monitoring / Alerting

The lack of proper monitor is never the sole cause of a problem, but it's often a serious contributing factor. As is the case for human errors, these seem underrepresented in public postmortems. When I talk to folks at other companies about their worst near disasters, a large fraction of them come from not having the right sort of alerting set up. They're often saved having a disaster bad enough to require a public postmortem by some sort of ops heroism, but heroism isn't a scalable solution.

Sometimes, those near disasters are caused by subtle coding bugs, which is understandable. But more often, it's due to blatant process bugs, like not having a clear escalation path for an entire class of failures, causing the wrong team to debug an issue for half a day, or not having a backup oncall, causing a system to lose or corrupt data for hours before anyone notices when (inevitably) the oncall person doesn't notice that something's going wrong.

The Northeast blackout of 2003 is a great example of this. It could have been a minor outage, or even just a minor service degredation, but (among other things) a series of missed alerts caused it to become one of the worst power outages ever.

Not a Conclusion

This is where the conclusion's supposed to be, but I'd really like to do some serious data analysis before writing some kind of conclusion or call to action. What should I look for? What other major classes of common errors should I consider? These aren't rhetorical questions and I'm genuinely interested in hearing about other categories I should think about. Feel free to ping me here. I'm also trying to collect public postmortems here.

One day, I'll get around to the serious analysis, but even without going through and classifying thousands of postmortems, I'll probably do a few things differently as a result of having read a bunch of these. I'll spend relatively more time during my code reviews on errors and error handling code, and relatively less time on the happy path. I'll also spend more time checking for and trying to convince people to fix "obvious" process bugs.

One of the things I find to be curious about these failure modes is that when I talked about what I found with other folks, at least one person told me that each process issue I found was obvious. But these "obvious" things still cause a lot of failures. In one case, someone told me that what I was telling them was obvious at pretty much the same time their company was having a global outage of a multi-billion dollar service, caused by the exact thing we were talking about. Just because something is obvious doesn't mean it's being done.

Elsewhere

Richard Cook's How Complex Systems Fail takes a more general approach; his work inspired The Checklist Manifesto,

which has saved lives.

Allspaw and Robbin's Web Operations: Keeping the Data on Time talks about this sort of thing in the context of web apps. Allspaw also has a nice post about some related literature from other fields.

In areas that are a bit closer to what I'm used to, there's a long history of studying the causes of failures. Some highlights inlcude Jim Gray's Why Do Computers Stop and What Can Be Done About It? (1985), Oppenheimer et. al's Why Do Internet Services Fail, and What Can Be Done About It? (2003), Nagaraja et. al's Understanding and Dealing with Operator Mistakes in Internet Services (2004), part of Barroso et. al's The Datacenter as a Computer (2009), and Rabkin and Katz's How Hadoop Clusters Break (2013), and Xu et. al's Do Not Blame Users for Misconfigurations.

There's also a long history of trying to understand aircraft reliability, and the story of how processes have changed over the decades is fascinating, although I'm not sure how to generalize those lessons.

Just as an aside, I find it interesting how hard it's been to eke out extra uptime and reliability. In 1974, Ritchie and Thompson wrote about a system "costing as little as $40,000" with 98% uptime. A decade later, Jim Gray uses 99.6% uptime as a reasonably good benchmark. We can do much better than that now, but the level of complexity required to do it is staggering.

Acknowledgements

Thanks to Leah Hanson, Anonymous, Marek Majkowski, Nat Welch, and Julia Hansbrough for providing comments on a draft of this. Anonymous, if you prefer to not be anonymous, send me a message on zulip. For anyone keeping score, that's three folks from Google, one person from Cloudflare, and one anyonymous commenter. I'm always open to comments/criticism, but I'd be especially interested in comments from folks who work at companies with less scale. Do my impressions generalize?

Thanks to gwern and Dan Reif for taking me up on this and finding some bugs in this post.

← Reviewing Steve Yegge's prediction record

Slashdot and Sourceforge →

Archive *Popular* *About (hire me!)*

Twitter *RSS*