Chapter 13 - Emergency Response

Emergency Response

Written by Corey Adam Baye
Edited by Diane Bates

Things break; that's life.

Regardless of the stakes involved or the size of an organization, one trait that's vital to the long-term health of an organization, and that consequently sets that organization apart from others, is how the people involved respond to an emergency. Few of us naturally respond well during an emergency. A proper response takes preparation and periodic, pertinent, hands-on training. Establishing and maintaining thorough training and testing processes requires the support of the board and management, in addition to the careful attention of staff. All of these elements are essential in fostering an environment in which teams can spend money, time, energy, and possibly even uptime to ensure that systems, processes, and people respond efficiently during an emergency.

Note that the chapter on postmortem culture discusses the specifics of how to write postmortems in order to make sure that incidents that require emergency response also become a learning opportunity (see Postmortem Culture: Learning from Failure). This chapter provides more concrete examples of such incidents.

What to Do When Systems Break

First of all, don't panic! You aren't alone, and the sky isn't falling. You're a professional and trained to handle this sort of situation. Typically, no one is in physical danger—only those poor electrons are in peril. At the very worst, half of the Internet is down. So take a deep breath…and carry on.

If you feel overwhelmed, pull in more people. Sometimes it may even be necessary to page the entire company. If your company has an incident response process (see Managing Incidents), make sure that you're familiar with it and follow that process.

Test-Induced Emergency

Google has adopted a proactive approach to disaster and emergency testing (see [Kri12]). SREs break our systems, watch how they fail, and make changes to improve reliability and prevent the failures from recurring. Most of the time, these controlled failures go as planned, and the target system and dependent systems behave in roughly the manner we expect. We identify some weaknesses or hidden dependencies and document follow-up actions to rectify the flaws we uncover. However, sometimes our assumptions and the actual results are worlds apart.

Here's one example of a test that unearthed a number of unexpected dependencies.

Details

We wanted to flush out hidden dependencies on a test database within one of our larger distributed MySQL databases. The plan was to block all access to just one database out of a hundred. No one foresaw the results that would unfold.

Response

Within minutes of commencing the test, numerous dependent services reported that both external and internal users were unable to access key systems. Some systems were intermittently or only partially accessible.

Assuming that the test was responsible, SRE immediately aborted the exercise. We attempted to roll back the permissions change, but were unsuccessful. Instead of panicking, we immediately brainstormed how to restore proper access. Using an already tested approach, we restored permissions to the replicas and failovers. In a parallel effort, we reached out to key developers to correct the flaw in the database application layer library.

Within an hour of the original decision, all access was fully restored, and all services were able to connect once again. The broad impact of this test motivated a rapid and thorough fix to the libraries and a plan for periodic retesting to prevent such a major flaw from recurring.

Findings

What went well

Dependent services that were affected by the incident immediately escalated the issues within the company. We assumed, correctly, that our controlled experiment had gotten out of hand and immediately aborted the test.

We were able to fully restore permissions within an hour of the first report, at which time systems started behaving properly. Some teams took a different approach and reconfigured their systems to avoid the test database. These parallel efforts helped to restore service as quickly as possible.

Follow-up action items were resolved quickly and thoroughly to avoid a similar outage, and we instituted periodic testing to ensure that similar flaws do not recur.

What we learned

Although this test was thoroughly reviewed and thought to be well scoped, reality revealed we had an insufficient understanding of this particular interaction among the dependent systems.

We failed to follow the incident response process, which had been put in place only a few weeks before and hadn't been thoroughly disseminated. This process would have ensured that all services and customers were aware of the outage. To avoid similar scenarios in the future, SRE continually refines and tests our incident response tools and processes, in addition to making sure that updates to our incident management procedures are clearly communicated to all relevant parties.

Because we hadn't tested our rollback procedures in a test environment, these procedures were flawed, which lengthened the outage. We now require thorough testing of rollback procedures before such large-scale tests.

Change-Induced Emergency

As you can imagine, Google has a lot of configuration—complex configuration—and we constantly make changes to that configuration. To prevent breaking our systems outright, we perform numerous tests on configuration changes to make sure they don't result in unexpected and undesired behavior. However, the scale and complexity of Google's infrastructure make it impossible to anticipate every dependency or interaction; sometimes configuration changes don't go entirely according to plan.

The following is one such example.

Details

A configuration change to the infrastructure that helps protect our services from abuse was pushed globally on a Friday. This infrastructure interacts with essentially all of our externally facing systems, and the change triggered a crash-loop bug in those systems, which caused the entire fleet to begin to crash-loop almost simultaneously. Because Google's internal infrastructure also depends upon our own services, many internal applications suddenly became unavailable as well.

Response

Within seconds, monitoring alerts started firing, indicating that certain sites were down. Some on-call engineers simultaneously experienced what they believed to be a failure of the corporate network and relocated to dedicated secure rooms (panic rooms) with backup access to the production environment. They were joined by additional engineers who were struggling with their corporate access.

Within five minutes of that first configuration push, the engineer responsible for the push, having become aware of the corporate outage but still unaware of the broader outage, pushed another configuration change to roll back the first change. At this point, services began to recover.

Within 10 minutes of the first push, on-call engineers declared an incident and proceeded to follow internal procedures for incident response. They began notifying the rest of the company about the situation. The push engineer informed the on-call engineers that the outage was likely due to the change that had been pushed and later rolled back. Nevertheless, some services experienced unrelated bugs or misconfigurations triggered by the original event and didn't fully recover for up to an hour.

Findings

What went well

There were several factors at play that prevented this incident from resulting in a longer-term outage of many of Google's internal systems.

To begin with, monitoring almost immediately detected and alerted us to the problem. However, it should be noted that in this case, our monitoring was less than ideal: alerts fired repeatedly and constantly, overwhelming the on-calls and spamming regular and emergency communication channels.

Once the problem was detected, incident management generally went well and updates were communicated often and clearly. Our out-of-band communications systems kept everyone connected even while some of the more complicated software stacks were unusable. This experience reminded us why SRE retains highly reliable, low overhead backup systems, which we use regularly.

In addition to these out-of-band communications systems, Google has command-line tools and alternative access methods that enable us to perform updates and roll back changes even when other interfaces are inaccessible. These tools and access methods worked well during the outage, with the caveat that engineers needed to be more familiar with the tools and to test them more routinely.

Google's infrastructure provided yet another layer of protection in that the affected system rate-limited how quickly it provided full updates to new clients. This behavior may have throttled the crash-loop and prevented a complete outage, allowing jobs to remain up long enough to service a few requests in between crashes.

Finally, we should not overlook the element of luck in the quick resolution of this incident: the push engineer happened to be following real-time communication channels—an additional level of diligence

that's not a normal part of the release process. The push engineer noticed a large number of complaints about corporate access directly following the push and rolled back the change almost immediately. Had this swift rollback not occurred, the outage could have lasted considerably longer, becoming immensely more difficult to troubleshoot.

What we learned

An earlier push of the new feature had involved a thorough canary but didn't trigger the same bug, as it had not exercised a very rare and specific configuration keyword in combination with the new feature. The specific change that triggered this bug wasn't considered risky, and therefore followed a less stringent canary process. When the change was pushed globally, it used the untested keyword/feature combination that triggered the failure.

Ironically, improvements to canarying and automation were slated to become higher priority in the following quarter. This incident immediately raised their priority and reinforced the need for thorough canarying, regardless of the perceived risk.

As one would expect, alerting was vocal during this incident because every location was essentially offline for a few minutes. This disrupted the real work being performed by the on-call engineers and made communication among those involved in the incident more difficult.

Google relies upon our own tools. Much of the software stack that we use for troubleshooting and communicating lies behind jobs that were crash-looping. Had this outage lasted any longer, debugging would have been severely hindered.

Process-Induced Emergency

We have poured a considerable amount of time and energy into the automation that manages our machine fleet. It's amazing how many jobs one can start, stop, or retool across the fleet with very little effort. Sometimes, the efficiency of our automation can be a bit frightening when things do not go quite according to plan.

This is one example where moving fast was not such a good thing.

Details

As part of routine automation testing, two consecutive turndown requests for the same soon-to-be-decommissioned server installation were submitted. In the case of the second turndown request, a subtle bug in the automation sent all of the machines in all of these installations globally to the Diskerase queue, where their hard drives were destined to be wiped; see Automation: Enabling Failure at Scale for more details.

Response

Soon after the second turndown request was issued, the on-call engineers received a page as the first small server installation was taken offline to be decommissioned. Their investigation determined that the machines had been transferred to the Diskerase queue, so following normal procedure, the on-call engineers drained traffic from the location. Because the machines in that location had been wiped, they were unable to respond to requests. To avoid failing those requests outright, on-call engineers drained traffic away from that location. Traffic was redirected to locations that could properly respond to the requests.

Before long, pagers everywhere were firing for all such server installations around the world. In response, the on-call engineers disabled all team automation in order to prevent further damage. They stopped or froze additional automation and production maintenance shortly thereafter.

Within an hour, all traffic had been diverted to other locations. Although users may have experienced elevated latencies, their requests were fulfilled. The outage was officially over.

Now the hard part began: recovery. Some network links were reporting heavy congestion, so network engineers implemented mitigations as choke points surfaced. A server installation in one such location was chosen to be the first of many to rise from the ashes. Within three hours of the initial outage, and thanks to the tenacity of several engineers, the installation was rebuilt and brought back online, happily accepting user requests once again.

US teams handed off to their European counterparts, and SRE hatched a plan to prioritize reinstallations using a streamlined but manual process. The team was divided into three parts, with each part responsible for one step in the manual reinstall process. Within three days, the vast majority of capacity was back online, while any stragglers would be recovered over the next month or two.

Findings

What went well

Reverse proxies in large server installations are managed very differently than reverse proxies in these small installations, so large installations were not impacted. On-call engineers were able to quickly move traffic from smaller installations to large installations. By design, these large installations can handle a full load without difficulty. However, some network links became congested, and therefore required network engineers to develop workarounds. In order to reduce the impact on end users, on-call engineers targeted congested networks as their highest priority.

The turndown process for the small installations worked efficiently and well. From start to finish, it took less than an hour to successfully turn down and securely wipe a large number of these installations.

Although turndown automation quickly tore down monitoring for the small installations, on-call engineers were able to promptly revert those monitoring changes. Doing so helped them to assess the extent of the damage.

The engineers quickly followed incident response protocols, which had matured considerably in the year since the first outage described in this chapter. Communication and collaboration throughout the company and across teams was superb—a real testament to the incident management program and training. All hands within the respective teams chipped in, bringing their vast experience to bear.

What we learned

The root cause was that the turndown automation server lacked the appropriate sanity checks on the commands it sent. When the server ran again in response to the initial failed turndown, it received an empty response for the machine rack. Instead of filtering the response, it passed the empty filter to the machine database, telling the machine database to Diskerase all machines involved. Yes, sometimes zero does mean all. The machine database complied, so the turndown workflow started churning through the machines as quickly as possible.

Reinstallations of machines were slow and unreliable. This behavior was due in large part to the use of the Trivial File Transfer Protocol (TFTP) at the lowest network Quality of Service (QoS) from the distant locations. The BIOS for each machine in the system dealt poorly with the failures.[77] Depending on the network cards involved, the BIOS either halted or went into a constant reboot cycle. They were failing to

transfer the boot files on each cycle and were further taxing the installers. On-call engineers were able to fix these reinstall problems by reclassifying installation traffic at slightly higher priority and using automation to restart any machines that were stuck.

The machine reinstallation infrastructure was unable to handle the simultaneous setup of thousands of machines. This inability was partly due to a regression that prevented the infrastructure from running more than two setup tasks per worker machine. The regression also used improper QoS settings to transfer files and had poorly tuned timeouts. It forced kernel reinstallation, even on machines that still had the proper kernel and on which Diskerase had yet to occur. To remedy this situation, on-call engineers escalated to parties responsible for this infrastructure who were able to quickly retune it to support this unusual load.

All Problems Have Solutions

Time and experience have shown that systems will not only break, but will break in ways that one could never previously imagine. One of the greatest lessons Google has learned is that a solution exists, even if it may not be obvious, especially to the person whose pager is screaming. If you can't think of a solution, cast your net farther. Involve more of your teammates, seek help, do whatever you have to do, but do it quickly. The highest priority is to resolve the issue at hand quickly. Oftentimes, the person with the most state is the one whose actions somehow triggered the event. Utilize that person.

Very importantly, once the emergency has been mitigated, do not forget to set aside time to clean up, write up the incident, and to…

Learn from the Past. Don't Repeat It.

Keep a History of Outages

There is no better way to learn than to document what has broken in the past. History is about learning from everyone's mistakes. Be thorough, be honest, but most of all, ask hard questions. Look for specific actions that might prevent such an outage from recurring, not just tactically, but also strategically. Ensure that everyone within the company can learn what you have learned by publishing and organizing postmortems.

Hold yourself and others accountable to following up on the specific actions detailed in these postmortems. Doing so will prevent a future outage that's nearly identical to, and caused by nearly the same triggers as, an outage that has already been documented. Once you have a solid track record for learning from past outages, see what you can do to prevent future ones.

Ask the Big, Even Improbable, Questions: What If…?

There is no greater test than reality. Ask yourself some big, open-ended questions. What if the building power fails…? What if the network equipment racks are standing in two feet of water…? What if the primary datacenter suddenly goes dark…? What if someone compromises your web server…? What do you do? Who do you call? Who will write the check? Do you have a plan? Do you know how to react? Do you know how your systems will react? Could you minimize the impact if it were to happen now? Could the person sitting next to you do the same?

Encourage Proactive Testing

When it comes to failures, theory and reality are two very different realms. Until your system has actually

failed, you don't truly know how that system, its dependent systems, or your users will react. Don't rely on assumptions or what you can't or haven't tested. Would you prefer that a failure happen at 2 a.m. Saturday morning when most of the company is still away on a team-building offsite in the Black Forest—or when you have your best and brightest close at hand, monitoring the test that they painstakingly reviewed in the previous weeks?

Conclusion

We've reviewed three different cases where parts of our systems broke. Although all three emergencies were triggered differently—one by a proactive test, another by a configuration change, and yet another by turndown automation—the responses shared many characteristics. The responders didn't panic. They pulled in others when they thought it necessary. The responders studied and learned from earlier outages. Subsequently, they built their systems to better respond to those types of outages. Each time new failure modes presented themselves, responders documented those failure modes. This follow-up helped other teams learn how to better troubleshoot and fortify their systems against similar outages. Responders proactively tested their systems. Such testing ensured that the changes fixed the underlying problems, and identified other weaknesses before they became outages.

And as our systems evolve the cycle continues, with each outage or test resulting in incremental improvements to both processes and systems. While the case studies in this chapter are specific to Google, this approach to emergency response can be applied over time to any organization of any size.

[77]BIOS: Basic Input/Output System. BIOS is the software built into a computer to send simple instructions to the hardware, allowing input and output before the operating system has been loaded.

previous

Chapter 12 - Effective Troubleshooting

next

Chapter 14 - Managing Incidents

Copyright © 2017 Google, Inc. Published by O'Reilly Media, Inc. Licensed under CC BY-NC-ND 4.0