



acmqueue The Antifragile Organization

Embracing Failure to Improve Resilience and Maximize Availability

Ariel Tseitlin

Failure is inevitable. Disks fail. Software bugs lie dormant waiting for just the right conditions to bite. People make mistakes. Data centers are built on farms of unreliable commodity hardware. If you're running in a cloud environment, then many of these factors are outside of your control. To compound the problem, failure is not predictable and doesn't occur with uniform probability and frequency. The lack of a uniform frequency increases uncertainty and risk in the system. In the face of such inevitable and unpredictable failure, how can you build a reliable service that provides the high level of availability your users can depend on?

A naive approach could attempt to prove the correctness of a system through rigorous analysis. It could model all different types of failures and deduce the proper workings of the system through a simulation or another theoretical framework that emulates or analyzes the real operating environment. Unfortunately, the state of the art of static analysis and testing in the industry hasn't reached those capabilities.⁴

A different approach could attempt to create exhaustive test suites to simulate all failure modes in a separate test environment. The goal of each test suite would be to maintain the proper functioning of each component, as well as the entire system when individual components fail. Most software systems use this approach in one form or another, with a combination of unit and integration tests. More advanced usage includes measuring the coverage surface of tests to indicate completeness.

While this approach does improve the quality of the system and can prevent a large class of failures, it is insufficient to maintain resilience in a large-scale distributed system. A distributed system must address the challenges posed by data and information flow. The complexity of designing and executing tests that properly capture the behavior of the target system is greater than that of building the system itself. Layer on top of that the attribute of large scale, and it becomes unfeasible, with current means, to achieve this in practice while maintaining a high velocity of innovation and feature delivery.

Yet another approach, advocated in this article, is to induce failures in the system to empirically demonstrate resilience and validate intended behavior. Given that the system was designed with resilience to failures, inducing those failures—within original design parameters—validates that the system behaves as expected. Because this approach uses the actual live system, any resilience gaps that emerge are identified and caught quickly as the system evolves and changes. In the second approach just described, many complex issues aren't caught in the test environment and manifest themselves in unique and infrequent ways only in the live environment. This, in turn, increases the likelihood of latent bugs remaining undiscovered and accumulating, only to cause larger problems when the right failure mode occurs. With failure induction, the added need to model changes in the data, information flow, and deployment architecture in a test environment is minimized and presents less of an opportunity to miss problems.

Before going further, let's discuss what is meant by resilience and how to increase it.

Resilience is an attribute of a system that enables it to deal with failure in a way that doesn't cause the entire system to fail. It could involve minimizing the blast radius when a failure occurs or changing the user experience to work around a failing component. For example, if a movie recommendation service fails, the user can be presented with a nonpersonalized list of popular titles. A complex system is constantly undergoing varying degrees of failure. Resilience is how it can recover, or be insulated, from failure, both current and future.⁷

There are two ways of increasing the resilience of a system:

- **Build your application with redundancy and fault tolerance.** In a service-oriented architecture, components are encapsulated in services. Services are made up of redundant execution units (instances) that protect clients from single- or multiple-unit failure. When an entire service fails, clients of that service need fault tolerance to localize the failure and continue to function.
- **Reduce uncertainty by regularly inducing failure.** Increasing the frequency of failure reduces its uncertainty and the likelihood of an inappropriate or unexpected response. Each unique failure can be induced while observing the application. For each undesirable response to an induced failure, the first approach can be applied to prevent its recurrence. Although in practice it is not feasible to induce every possible failure, the exercise of enumerating possible failures and prioritizing them helps in understanding tolerable operating conditions and classifying failures when they fall outside those bounds.

The first item is well covered in other literature. The remainder of this article will focus on the second.

THE SIMIAN ARMY

Once you have accepted the idea of inducing failure regularly, there are a few choices on how to proceed. One option is GameDays,¹ a set of scheduled exercises where failure is manually introduced or simulated to mirror real-world failure, with the goal of both identifying the results and practicing the response—a fire drill of sorts. Used by the likes of Amazon and Google, GameDays are a great way to induce failure on a regular basis, validate assumptions about system behavior, and improve organizational response.

But what if you want a solution that is more scalable and automated—one that doesn't run once per quarter but rather once per week or even per day? You don't want failure to be a fire drill. You want it to be a nonevent—something that happens all the time in the background so that when a real failure occurs, it will simply blend in without any impact.

One way of achieving this is to engineer failure to occur in the live environment. This is how the idea for “monkeys” (autonomous agents really, but monkeys inspire the imagination) came to Netflix to wreak havoc and induce failure. Later the monkeys were grouped together and labeled the Simian Army.⁵ A description of each resilience-related monkey follows.

CHAOS MONKEY

The failure of a virtual instance is the most common type of failure encountered in a typical public cloud environment. It can be caused by a power outage in the hosting rack, a disk failure, or a

network partition that cuts off access. Regardless of the cause, the result is the same: the instance becomes unavailable. Inducing such failures helps ensure that services don't rely on any on-instance state, instance affinity, or persistent connections.

To address this need, Netflix created its first monkey: Chaos Monkey, which randomly terminates virtual instances in a production environment—instances that are serving live customer traffic.³

Chaos Monkey starts by looking into a service registry to find all the services that are running. In Netflix's case, this is done through a combination of Asgard⁶ and Edda.² Each service can override the default Chaos Monkey configuration to change termination probability or opt out entirely. Each hour, Chaos Monkey wakes up, rolls the dice, and terminates the affected instances using AWS (Amazon Web Services) APIs.

Chaos Monkey can optionally send an e-mail message to the service owner when a termination is made, but most service owners don't enable this option because instance terminations are common enough that they don't cause any service degradation.

CHAOS GORILLA

With Chaos Monkey, a system is resilient to individual instance failure, but what if an entire data center were to become unavailable? What would be the impact to users if an entire Amazon AZ (availability zone) went offline? To answer that question and to make sure such an event would have minimal customer impact, Netflix created Chaos Gorilla.

Chaos Gorilla causes an entire AZ to fail. It simulates two failure modes:

- **Network partition.** The instances in the zone are still running and can communicate with each other but are unable to communicate with any service outside the zone and are not reachable by any other service outside the zone.
- **Total zone failure.** All instances in the zone are terminated.

Chaos Gorilla causes massive damage and requires a sophisticated control system to rebalance load. For Netflix, that system is still being developed, and as a result, Chaos Gorilla is run manually, similar to the GameDay exercises mentioned previously. With each successive run, Chaos Gorilla becomes more aggressive in the way it executes the failures—the goal being to run it in an automated unattended way, like Chaos Monkey.

CHAOS KONG

A region is made up of multiple data centers (availability zones) that are meant to be isolated from one another. A robust deployment architecture has AZ redundancy by using multiple AZs. In practice, regionwide failures do occur, which makes single-region deployments insufficient to provide resilience to regionwide failures. Once a system is deployed redundantly to multiple regions, region failure must be tested analogously to instances and availability zones. Chaos Kong will serve that purpose. Netflix is working toward the goal of taking an entire region offline with Chaos Kong.

LATENCY MONKEY

Once Chaos Monkey is running and individual instance failure no longer has any impact, a new class of failures emerges. Dealing with instance failure is relatively easy: just terminate the bad instances and let new healthy instances take their places. Detecting when instances become unhealthy, but are still working, is harder, and having resilience to this failure mode is harder still.

Error rates could become elevated, but the service could occasionally return success. The service could reply with successful responses, but latency could increase, causing timeouts.

What Netflix needed was a way of inducing failure that simulated partially healthy instances. This was the genesis of Latency Monkey, which induces artificial delays in the RESTful client-server communication layer to simulate service degradation and measures if upstream services respond appropriately. In addition, very large delays can be used to simulate node downtime, or even an entire service downtime, without physically bringing instances or services down. This can be particularly useful when testing the fault tolerance of a new service by simulating the failure of its dependencies, without making these dependencies unavailable to the rest of the system.

THE REMAINING ARMY

The rest of the Simian Army, including Janitor Monkey, takes care of upkeep and other miscellaneous tasks not directly related to availability. A full reference is available at <http://techblog.netflix.com/2011/07/netflix-simian-army.html>.

MONKEY TRAINING AT NETFLIX

While the Simian Army is a novel concept and may require a shift in perspective, it isn't as hard to implement as it initially appears. Understanding what Netflix went through is illustrative for others interested in following such a path.

Netflix is known for being bold in its rapid pursuit of innovation and high availability, but not to the point of callousness. It is careful to avoid any noticeable impact to customers from these failure-induction exercises. To minimize risk, Netflix takes the following steps when introducing a monkey:

1. With the new monkey in the test environment, engineers observe the user experience. The goal is to have negligible or zero impact on the customer. If the engineers see any adverse results, then they make the necessary code changes to prevent recurrence. This step is repeated as many times as necessary until no adverse user experience is observed.
2. Once no adverse results are observed in the test environment, the new monkey is enabled in the production environment. Initially, the new monkey is run in opt-in mode. One or more services are selected to run the new monkey against, having already been run in the test environment. The new monkey runs for a few months in this mode, opting in new services over time.
3. After many services have opted in, the new monkey graduates to opt-out mode, in which all services are potential targets for the new monkey. If a service is placed on an opt-out list, the monkey avoids it.
4. The opt-out list is periodically reviewed for each monkey, and service owners are encouraged to remove their services from the list. The platform and monkey are improved to increase adoption and address reasons for opting out.

THE IMPORTANCE OF OBSERVABILITY

No discussion of resilience would be complete without highlighting the important role of monitoring. *Monitoring* here means the ability to observe and, optionally, signal an alarm on the external and internal states of the system and its components. In the context of failure induction and resilience, monitoring is important for two reasons:

- During a real, nonsimulated customer-impacting event, it's important to stabilize the system and eliminate customer impact as quickly as possible. Any automation that causes additional failure must be stopped during this time. Failing to do so can cause Chaos Monkey, Latency Monkey, and the other simians to further weaken an already unhealthy system, causing even greater adverse end-user impact. The ability to observe and detect customer-impacting service degradation is an important prerequisite to building and enabling automation that causes failure.
- Building resilient systems doesn't happen at a single point in time; it's an ongoing process that involves discovering weaknesses and dealing with them in an iterative learning cycle. Deep visibility into the system is key to understanding how the system operates and how it fails. Few root-cause investigations would succeed without metrics and insights into the operations of the system and its components. Monitoring provides a deep understanding of how the system operates, especially when it fails, and makes it possible to discover weaknesses in the system and identify anti-patterns for resilience.

One of the most important first questions to ask during a customer-impacting event is, "What changed?" Therefore, another key aspect of monitoring and observability is the ability to record changes to the state of the system. Whether a new code deployment, a change in runtime configuration, or a state change by an externally used service, the change must be recorded for easy retrieval later. Netflix built a system, internally called Chronos, for this purpose. Any event that changes the state of the system is recorded in Chronos and can be quickly queried to aid in causality attribution.

THE ANTIFRAGILE ORGANIZATION

Resilience to failure is a lofty goal. It enables a system to survive and withstand failure. There's an even higher peak to strive for, however: making the system stronger and better with each failure. In Nassim Taleb's parlance, it can become *antifragile*—growing stronger from each successive stressor, disturbance, and failure.⁸

Netflix has taken the following steps to create a more antifragile system and organization:

- 1. Every engineer is an operator of the service.** This is sometimes referred to in jest as "no ops," though it's really more "distributed ops." Separating development and operations creates a division of responsibilities that can lead to a number of challenges, including network externalities and misaligned incentives. Network externalities are caused by operators feeling the pain of problems that developers introduce. Misaligned incentives are a result of operators wanting stability while developers want velocity. The DevOps movement was started in response to this divide. Instead of separating development and operations, developers should operate their own services. They deploy their code to production and then they are the ones awakened in the middle of the night if any part of it breaks and impacts customers. By combining development and operations, each engineer can respond to failure by altering the service to be more resilient to and tolerant of future failures.
- 2. Each failure is an opportunity to learn, generating these questions:** "How could the failure have been detected more quickly?" "How can the system be more resilient to this type of failure?" "How can this failure be induced on a regular basis?" The result is that

each failure makes the system more robust and resilient, analogous to the experience a warrior gains in each battle to make him stronger and fiercer in the next. The system becomes better the more times and ways it fails.

3. **A blameless culture is fostered.** As an organization, Netflix optimizes for innovation and velocity, and it accepts that mistakes will sometimes occur, using each one as an opportunity to learn. A commonly overheard saying at Netflix is, “If we’re not making any mistakes, it means we’re not moving quickly enough.” Mistakes aren’t a bad thing, unless the same mistakes are made over and over again. The result is that people are less worried about making mistakes, and postmortems can be structured as effective opportunities to learn (see step 2).

CONCLUSION

The more frequently failure occurs, the more prepared the system and organization become to deal with it in a transparent and predictable manner. Inducing failure is the best way of ensuring both system and organizational resilience. The goal is to maximize availability, insulating users of a service from failure and delivering a consistent and available user experience. Resilience can be improved by increasing the frequency and variety of failure and evolving the system to deal better with each newfound failure, thereby increasing antifragility. Focusing on learning and fostering a blameless culture are essential organizational elements in creating proper feedback in the system.

REFERENCES

1. Robbins, J., Krishnan, K., Allspaw, J., Limoncelli, T. 2012. Resilience engineering: learning to embrace failure. *Communications of the ACM* 55(11): 40-47; <http://dx.doi.org/10.1145/2366316.2366331>.
2. Bennett, C. 2012. Edda - Learn the stories of your cloud deployments. The Netflix Tech Blog; <http://techblog.netflix.com/2012/11/edda-learn-stories-of-your-cloud.html>.
3. Bennett, C., Tseitlin, A. 2012. Chaos Monkey released into the wild. The Netflix Tech Blog; <http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html>.
4. Chandra, T. D., Griesemer, R., Redstone, J. 2007. Paxos made live: an engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*: 398-407; http://labs.google.com/papers/paxos_made_live.pdf.
5. Izrailevsky, Y., Tseitlin, A. 2011. The Netflix simian army. The Netflix Tech Blog; <http://techblog.netflix.com/2011/07/netflix-simian-army.html>.
6. Sondow, J. 2012. Asgard: Web-based cloud management and deployment. The Netflix Tech Blog; <http://techblog.netflix.com/2012/06/asgard-web-based-cloud-management-and.html>.
7. Strigini, L. 2009. Fault tolerance and resilience: meanings, measures and assessment. London, U.K.: Centre for Software Reliability, City University London; <http://www.csr.city.ac.uk/projects/amber/resilienceFTmeasurementv06.pdf>.
8. Taleb, N. 2012. *Antifragile: Things That Gain from Disorder*. Random House.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

ARIEL TSEITLIN is director of cloud solutions at Netflix, where he manages the Netflix Cloud and is responsible for cloud tooling, monitoring, performance and scalability, and cloud operations and reliability engineering. He is also interested in resilience and highly available distributed systems. Prior to joining Netflix, he was most recently VP of technology and products at Sungevity and before that was the founder and CEO of CTOWorks.

© 2013 ACM 1542-7730/13/0600 \$10.00