THE NETFLIX TECH BLOG

Lessons Netflix Learned from the AWS Outage

Netflix Technology Blog
Apr 29, 2011 · 8 min read

On Thursday, April 21st, Amazon experienced a large outage in AWS US-East which they describe here. This outage was highly publicized because it took down or severely hampered a number of popular websites that depend on AWS for hosting. Our post below describes our experience at Netflix with the outage, and what we've learned from it.

Some Background

Why were some websites impacted while others were not? For Netflix, the short answer is that our systems are designed explicitly for these sorts of failures. When we re-designed for the cloud this Amazon failure was exactly the sort of issue that we wanted to be resilient to. Our architecture avoids using EBS as our main data storage service, and the SimpleDB, S3 and Cassandra services that we do depend upon were not affected by the outage.

What Went Well…

The Netflix service ran without intervention but with a higher than usual error rate and higher latency than normal through the morning, which is the low traffic time of day for Netflix streaming. We didn't see a noticeable increase in customer service calls, or our customers' ability to find and start movies.

In some ways this Amazon issue was the first major test to see if our new cloud architecture put us in a better place than we were in 2008. It wasn't perfect, but we didn't suffer any big external outages and only had to do a small amount of scrambling internally. Some of the decisions we made along the way that contributed to our success include:

Stateless Services

One of the major design goals of the Netflix re-architecture was to move to stateless services. These services are designed such that any service instance can serve any request in a timely fashion and so if a server fails it's not a big deal. In the failure case requests can be routed to another service instance and we can automatically spin up a

new node to replace it.

Data Stored Across Zones

In cases where it was impractical to re-architect in a stateless fashion we ensure that there are multiple redundant hot copies of the data spread across zones. In the case of a failure we retry in another zone, or switch over to the hot standby.

Graceful Degradation

Our systems are designed for failure. With that in mind we have put a lot of thought into what we do when (not if) a component fails. The general principles are:

Fail Fast: Set aggressive timeouts such that failing components don't make the entire system crawl to a halt.

Fallbacks: Each feature is designed to degrade or fall back to a lower quality representation. For example if we cannot generate personalized rows of movies for a user we will fall back to cached (stale) or un-personalized results.

Feature Removal: If a feature is non-critical then if it's slow we may remove the feature from any given page to prevent it from impacting the member experience.

"N+1" Redundancy

Our cloud architecture is is designed with N+1 redundancy in mind. In other words we allocate more capacity than we actually need at any point in time. This capacity gives us the ability to cope with large spikes in load caused by member activity or the ripple effects of transient failures; as well as the failure of up to one complete AWS zone. All zones are active, so we don't have one hot zone and one idle zone as used by simple master/slave redundancy. The term N+1 indicates that one extra is needed, and the larger N is, the less overhead is needed for redundancy. We spread our systems and services out as evenly as we can across three of the four zones. When provisioning capacity, we use AWS reserved instances in every zone, and reserve more than we actually use, so that we will have guaranteed capacity to allocate if any one zone fails. As a result we have higher confidence that the other zones are able to grow to pick up the excess load from a zone that is not functioning properly. This does cost money (reservations are for one to three years with an advance payment) however, this is money well spent since it makes our systems more resilient to failures.

Cloud Solutions for the Cloud

We could have chosen the simplest path into the cloud, fork-lifting our existing applications from our data centers to Amazon's and simply using EC2 as if it was nothing more than another set of data centers. However, that wouldn't have given us the same level of scalability and resiliency that we needed to run our business. Instead, we fully embraced the cloud paradigm. This meant leveraging NoSQL solutions wherever possible to take advantage of the added availability and durability that they provide, even though it meant sacrificing some consistency guarantees. In addition, we also use S3 heavily as a durable storage layer and pretend that all other resources are effectively transient. This does mean that we can suffer if there are S3 related issues,

however this is a system that Amazon has architected to be resilient across individual system and zone failures and has proven to be highly reliable, degrading rather than failing.

What Didn't Go So Well...

While we did weather the storm, it wasn't totally painless. Things appeared pretty calm from a customer perspective, but we did have to do some scrambling internally. As it became clear that AWS was unlikely to resolve the issues before Netflix reached peak traffic in the early evening, we decided to manually re-assign our traffic to avoid the problematic zone. Thankfully, Netflix engineering teams were able to quickly coordinate to get this done. At our current level of scale this is a bit painful, and it is clear that we need more work on automation and tooling. As we grow from a single Amazon region and 3 availability zones servicing the USA and Canada to being a worldwide service with dozens of availability zones, even with top engineers we simply won't be able to scale our responses manually.

Manual Steps

When Amazon's Availability Zone (AZ) started failing we decided to get out of the zone all together. This meant making significant changes to our AWS configuration. While we have tools to change individual aspects of our AWS deployment and configuration they are not currently designed to enact wholesale changes, such as moving sets of services out of a zone completely. This meant that we had to engage with each of the service teams to make the manual (and potentially error prone) changes. In the future we will be working to automate this process, so it will scale for a company of our size and growth rate.

Load Balancing

Netflix uses Amazons Elastic Load Balance (ELB) service to route traffic to our front end services. We utilize ELB for almost all our web services. There is one architectural limitation with the service: losing a large number of servers in a zone can create a service interruption.

ELB's are setup such that load is balanced across zones first, then instances. This is because the ELB is a two tier load balancing scheme. The first tier consists of basic DNS based round robin load balancing. This gets a client to an ELB endpoint in the cloud that is in one of the zones that your ELB is configured to use. The second tier of the ELB service is an array of load balancer instances (provisioned directly by AWS), which does round robin load balancing over our own instances that are behind it in the same zone.

In the case where you are in 3 zones and and you have many service instances go down then the rest of the nodes in that AZ have to pick up the slack and handle the extra load. Eventually, if you couldn't launch more nodes to bring capacity up to previous levels you would likely suffer a cascading failure where all the nodes in the zone go down. If this happened then a third of your traffic would essentially go into a "black hole" and fail.

The net effect of this is that we have to be careful to make sure that our zones stay evenly balanced with frontend servers so that we don't serve degraded traffic from any

one zone. This meant that when the outage happened last week we had to manually update all of our ELB endpoints to completely avoid the failed zone, and change the autoscaling groups behind them to do the same so that the servers would all be in the zones that were getting traffic.

It also appears that the instances used by AWS to provide ELB services were dependent on EBS backed boot disks. Several people have commented that ELB itself had a high failure rate in the affected zone.

For middle tier load balancing Netflix uses its own software load balancing service that does balance across instances evenly, independent of which zone they are in. Services using middle tier load balancing are able to handle uneven zone capacity with no intervention.

Lessons Learned

This outage gave us some valuable experience and helped us to identify and prioritize several ways that we should improve our systems to make it more resilient to failures.

Create More Failures

Currently, Netflix uses a service called "Chaos Monkey" to simulate service failure. Basically, Chaos Monkey is a service that kills other services. We run this service because we want engineering teams to be used to a constant level of failure in the cloud. Services should automatically recover without any manual intervention. We don't however, simulate what happens when an entire AZ goes down and therefore we haven't engineered our systems to automatically deal with those sorts of failures. Internally we are having discussions about doing that and people are already starting to call this service "Chaos Gorilla".

Automate Zone Fail Over and Recovery

Relying on multiple teams using manual intervention to fail over from a failing AZ simply doesn't scale. We need to automate this process, making it a "one click" operation, that can be invoked if required.

Multiple Region Support

We are currently re-engineering our systems to work across multiple AWS regions as part of the Netflix drive to support streaming in global markets. As AWS launches more regions around the world, we want to be able to migrate the support for a country to a newly launched AWS region that is closer to our customers. This is essentially the same operation as a disaster recovery migration from one region to another, so we would have the ability to completely vacate a region if a large scale outage occurred.

Avoid EBS Dependencies

We had already decided that EBS performance was an issue, so (with one exception) we have avoided it as a primary data store. With two outages so far this year we are taking steps to further reduce dependencies by switching our stateful instances to S3 backed AMIs. These take a lot longer to create in our build process, but will make our Cassandra storage services more resilient, by depending purely on internal disks. We

already have plans to migrate our one large MySQL EBS based service to Cassandra.

Conclusion

We set out to build a highly available Netflix service on AWS and this outage was a major test of that decision. While we do have some lessons learned and some improvements to make, we continue to be confident that this is the right strategy for us.

Authors:

- Adrian Cockroft — Director: Architecture, ECommerce and System Engineering
- Cory Hicks — Sr. Manager: Applied Personalization Algorithms
- Greg Orzell — Sr. Manager: Streaming Service and Insight Engineering

See Also:

5 Lessons We've Learned Using AWS

Dorothy, you're not in Kansas anymore.

medium.com

Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region

For customers with an attached EBS volume or a running RDS database instance in the affected Availability Zone in the...

aws.amazon.com

. . .

Originally published at techblog.netflix.com on April 19, 2011.

Cloud Computing | AWS | Netflix

86 claps

WRITTEN BY
Netflix Technology Blog
Follow
Learn more about how Netflix designs, builds, and operates our systems and engineering organizations

Netflix TechBlog
Follow
Learn about Netflix's world class engineering efforts, company culture, product developments and more.

See responses (2)

More From Medium

Byte Down: Making Netflix's Data Infrastructure Cost-Effective

Netflix Technology Blog in Netflix TechBlog

Netflix Studio Engineering Overview

Netflix Technology Blog in Netflix TechBlog

Artwork Personalization at Netflix

Netflix Technology Blog in Netflix TechBlog

Beyond Interactive: Notebook Innovation at Netflix

Netflix Technology Blog in Netflix TechBlog

Full Cycle Developers at Netflix

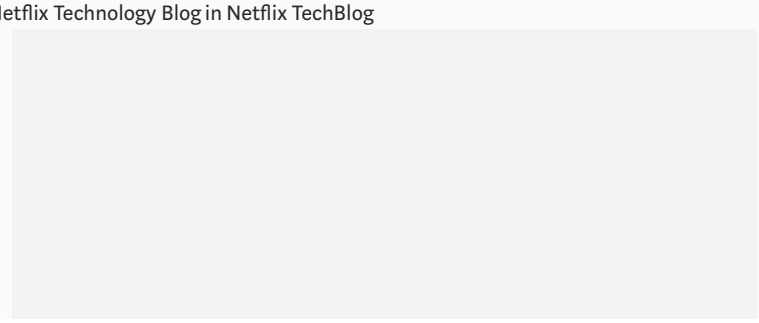
Netflix Technology Blog in Netflix TechBlog

Our learnings from adopting GraphQL

Netflix Technology Blog in Netflix TechBlog

Growth Engineering at Netflix — Accelerating Innovation

Netflix Technology Blog in Netflix TechBlog



Netflix FlameScope

Netflix Technology Blog in Netflix TechBlog

Medium

About

Help

Legal