NETFLIX

The Netflix Tech Blog

Friday, September 25, 2015

Chaos Engineering Upgraded

Several years ago we introduced a tool called Chaos Monkey. This service pseudo-randomly plucks a server from our production deployment on AWS and kills it. At the time we were met with incredulity and skepticism. Are we crazy? In production?!?

Our reasoning was sound, and the results bore that out. Since we knew that server failures are guaranteed to happen, we wanted those failures to happen during business hours when we were on hand to fix any fallout. We knew that we could rely on engineers to build resilient solutions if we gave them the context to *expect* servers to fail. If we could align our engineers to build services that survive a server failure as a matter of course, then when it accidentally happened it wouldn't be a big deal. In fact, our members wouldn't even notice. This proved to be the case.

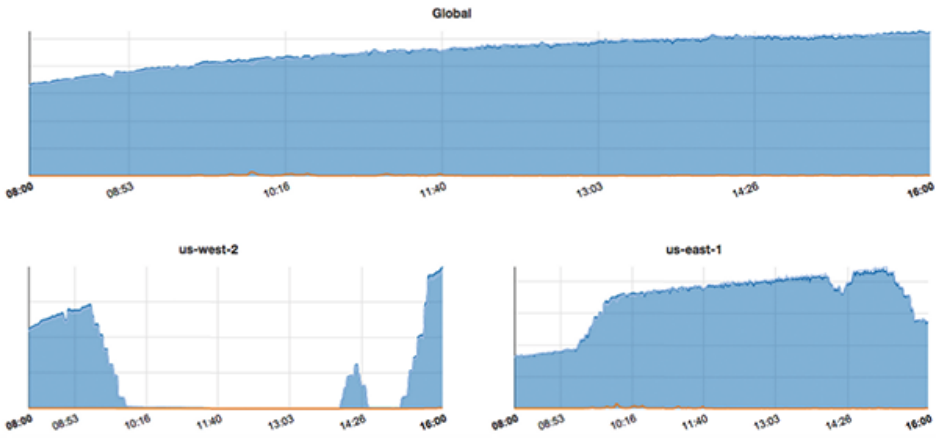
Chaos Kong

Building on the success of Chaos Monkey, we looked at an extreme case of infrastructure failure. We built Chaos Kong, which doesn't just kill a server. It kills an entire AWS Region[1].

It is very rare that an AWS Region becomes unavailable, but it does happen. This past Sunday (September 20th, 2015) Amazon's DynamoDB service experienced an availability issue in their US-EAST-1 Region. That instability caused more than 20 additional AWS services that are dependent on DynamoDB to fail. Some of the Internet's biggest sites and applications were intermittently unavailable during a six- to eight-hour window that day.

Netflix did experience a brief availability blip in the affected Region, but we sidestepped any significant impact because Chaos Kong exercises prepare us for incidents like this. By running experiments on a regular basis that simulate a Regional outage, we were able to identify any systemic weaknesses early and fix them. When US-EAST-1 actually became unavailable, our system was already strong enough to handle a traffic failover.

Below is a chart of our video play metrics during a Chaos Kong exercise. These are three views of the same eight hour window. The top view shows the aggregate metric, while the bottom two show the same metric for the west region and the east region, respectively.



Chaos Kong exercise in progress

Links

Netflix US & Canada Blog
Netflix America Latina Blog
Netflix Brasil Blog
Netflix Benelux Blog
Netflix DACH Blog
Netflix France Blog
Netflix Nordics Blog
Netflix UK & Ireland Blog
Netflix ISP Speed Index
Open positions at Netflix
Netflix Website
Facebook Netflix Page
Netflix UI Engineering
RSS Feed

About the Netflix Tech Blog

This is a Netflix blog focused on technology and technology issues. We'll share our perspectives, decisions and challenges regarding the software we build and use to create the Netflix service.

Blog Archive

► 2016 (17)
▼ 2015 (50)
► December (7)
► November (5)
► October (5)
▼ September (6)
Moving from Asgard to Spinnaker
Creating Your Own EC2 Spot Market
Chaos Engineering Upgraded
John Carmack on Developing the Netflix App for Ocu...
Announcing Electric Eye
Introducing Lemur
► August (6)
► July (3)
► June (2)

In the bottom row, you can clearly see traffic evacuate from the west region. The east region gets a corresponding bump in traffic as it steps up to play the role of savior. During the exercise, most of our attention stays focused on the top row. As long as the aggregate metric follows that relatively smooth trend, we know that our system is resilient to the failover. At the end of the exercise, you see traffic revert to the west region, and the aggregate view shows that our members did not experience an adverse effects. We run Chaos Kong exercises like this on a regular basis, and it gives us confidence that even if an entire region goes down, we can still serve our customers.

ADVANCING THE MODEL

We looked around to see what other engineering practices could benefit from these types of exercises, and we noticed that Chaos meant different things to different people. In order to carry the practice forward, we need a best-practice definition, a model that we can apply across different projects and different departments to make our services more resilient.

We want to capture the value of these exercises in a methodology that we can use to improve our systems and push the state of the art forward. At Netflix we have an extremely complex distributed system (microservice architecture) with hundreds of deploys every day. We don't want to remove the complexity of the system; we want to thrive on it. We want to continue to accelerate flexibility and rapid development. And with that complexity, flexibility, and rapidity, we still need to have confidence in the resiliency of our system.

To have our cake and eat it too, we set out to develop a new discipline around Chaos. We developed an empirical, systems-based approach which addresses the chaos inherent in distributed systems at scale. This approach specifically builds confidence in the ability of those systems to withstand realistic conditions. We learn about the behavior of a distributed system by observing it in a controlled experiment, and we use those learnings to fortify our systems before any systemic effect can disrupt the quality service that we provide our customers. We call this new discipline Chaos Engineering.

We have published the Principles of Chaos Engineering as a living document, so that other organizations can contribute to the concepts that we outline here.

CHAOS EXPERIMENT

We put these principles into practice. At Netflix we have a microservice architecture. One of our services is called Subscriber, which handles certain user management activities and authentication. It is possible that under some rare or even unknown situation Subscriber will be crippled. This might be due to network errors, under-provisioning of resources, or even by events in downstream services upon which Subscriber depends. When you have a distributed system at scale, sometimes bad things just happen that are outside any person's control. We want confidence that our service is resilient to situations like this.

We have a steady-state definition: Our metric of interest is customer engagement, which we measure as the number of video plays that start each second. In some experiments we also look at load average and error rate on an upstream service (API). The lines that those metrics draw over time are predictable, and provide a good proxy for the steady-state of the system. We have a hypothesis: We will see no significant impact on our customer engagement over short periods of time on the order of an hour, even when Subscriber is in a degraded state. We have variables: We add latency of 30ms first to 20% then to 50% of traffic from Subscriber to its primary cache. This simulates a situation in which the Subscriber cache is over-stressed and performing poorly. Cache misses increase, which in turn increases load on other parts of the Subscriber service. Then we look for a statistically significant deviation between the variable group and the control group with respect to the system's steady-state level of customer engagement.

If we find a deviation from steady-state in our variable group, then we have disproved our hypothesis. That would cause us to revisit the fallbacks and dependency configuration for Subscriber. We would undertake a concerted effort to improve the resiliency story around Subscriber and the services that it touches, so that customers can count on our service even when Subscriber is in a degraded state.

If we don't find any deviation in our variable group, then we feel more confident in our hypothesis. That translates to having more confidence in our service as a whole.

In this specific case, we did see a deviation from steady-state when 30ms latency was added to 50% of the traffic going to this service. We identified a number of steps that we could take, such as decreasing the thread pool count in an upstream service, and subsequent experiments have confirmed the bolstered resiliency of Subscriber.

CONCLUSION

We started Chaos Monkey to build confidence in our highly complex system. We don't have to simplify or even understand the system to see that over time Chaos Monkey makes the system more resilient. By purposefully introducing realistic production conditions into a controlled run, we can uncover weaknesses

► May (2)
► April (3)
► March (3)
► February (5)
► January (3)
► 2014 (37)
► 2013 (52)
► 2012 (37)
► 2011 (17)
► 2010 (8)

Labels

"cloud architecture" (3)
accelerated compositing (2)
adwords (1)
Aegisthus (1)
algorithms (3)
aminator (2)
analytics (5)
Android (2)
angular (1)
api (16)
appender (1)
Archaius (2)
architectural design (1)
architecture (1)
Asgard (1)
Astyanax (4)
authentication (1)
automation (2)
autoscaling (3)
availability (4)
AWS (30)
bake (1)
benchmark (2)
big data (11)
billing (1)
Blitz4j (1)
build (4)
Cable (1)
caching (4)
Cassandra (14)
chaos engineering (1)
chaos monkey (5)
chukwa (1)
ci (1)

before they cause bigger problems. Chaos Engineering makes our system stronger, and gives us the confidence to move quickly in a very complex system.

STAY TUNED

This blog post is part of a series. In the next post on Chaos Engineering, we will take a deeper dive into the Principles of Chaos Engineering and hypothesis building with additional examples from our production experiments. If you have thoughts on Chaos Engineering or how to advance the state of the art in this field, we'd love to hear from you. Feel free to reach out to chaos@netflix.com.

-Chaos Team at Netflix
Ali Basiri, Lorin Hochstein, Abhijit Thosar, Casey Rosenthal

1. Technically, it only simulates killing an AWS Region. For our purposes, simulating this giant infrastructure failure is sufficient, and AWS doesn't yet provide us with a way of turning off an entire region. ;-) ↩

Posted by Casey Rosenthal at 11:38 AM

Newer Post Home Older Post

classloaders (1)
Clojure (1)
cloud (25)
cloud architecture (16)
cloud prize (3)
CO2 (1)
collection (1)
complex event processing (1)
computer vision (2)
concurrency (1)
configuration (2)
configuration management (2)
conformity monkey (1)
content delivery (1)
content metadata (1)
content platform engineering (2)
content quality (1)
continuous delivery (4)
coordination (2)
cost management (1)
crypto (1)
Cryptography (2)
CSS (2)
CUDA (1)
dart (1)
data migration (1)
data pipeline (5)
data science (7)
data visualization (1)
database (5)
DataStax (2)
deadlock (1)
deep learning (1)
Denominator (2)
dependency injection (1)
device (3)
device proliferation (1)
devops (2)
distributed (10)
DNS (1)
Docker (1)
Dockerhub (1)
DSL (1)
Dyn (1)
DynECT (1)

efficiency (2)
Elastic Load Balancer (1)
elasticsearch (2)
ELB (1)
EMR (2)
encoding (4)
energy (1)
eucalyptus (1)
eureka (2)
evcache (1)
failover (2)
falcor (2)
fault-tolerance (12)
flamegraphs (2)
Flow (1)
footprint (1)
FRP (1)
functional reactive (1)
garbage (1)
garbage collection (1)
gc (1)
Genie (4)
git (1)
Governator (1)
GPU (2)
gradle (1)
green (1)
Groovy (1)
Hack Day (3)
Hadoop (12)
HBase (1)
high volume (4)
high volume distributed systems (10)
Hive (2)
HTML5 (8)
https (1)
Hystrix (5)
IBM (1)
ice (1)
IMF (3)
IMSC (2)
initialization (1)
innovation (3)
insights (1)
inter process communication (1)

Interoperable Master Format (3)
Ipv6 (2)
isolation (1)
ISP (1)
java (5)
JavaScript (19)
jclouds (1)
jenkins (2)
kafka (2)
Karyon (2)
keystone (1)
lifecycle (1)
linux (2)
lipstick (2)
load balancing (3)
localization (1)
localization platform engineering (1)
locking (1)
locks (1)
log4j (1)
logging (2)
machine learning (6)
Map-Reduce (1)
media pipeline (1)
meetup (3)
memcache (2)
memcached (1)
message security layer (1)
Mobile (2)
modules (1)
monitoring (1)
msl (1)
nebula (1)
negative keywords (1)
Netflix (17)
Netflix API (7)
netflix graph (1)
Netflix OSS (12)
NetflixOSS (12)
neural networks (1)
node.js (4)
NoSQL (5)
observability (1)
Open source (10)

operational excellence (1)
operational insight (2)
operational visibility (1)
optimization (2)
OSS (3)
outage (1)
page generation (1)
payments (1)
Paypal (1)
performance (22)
personalization (4)
phone (1)
Pig (4)
pki (1)
Playback (2)
prediction (2)
predictive modeling (3)
Presto (2)
prize (1)
pubsub (1)
pytheas (1)
python (3)
Quality (1)
quality control (1)
rca (2)
React (3)
Reactive Programming (2)
real-time insights (2)
real-time streaming (2)
Recipe (1)
recommendations (8)
Redis (3)
reinvent (2)
reliability (7)
remote procedure calls (1)
renewable (1)
research (1)
resiliency (7)
REST (2)
Ribbon (2)
Riot Games (1)
root-cause analysis (2)
Route53 (1)
rule engine (1)
Rx (2)

scalability (12)
scale (1)
scripting library (1)
search (3)
security (8)
Servo (1)
shared libraries (1)
simian army (5)
SimpleDB (3)
site reliability (1)
spark (1)
spinnaker (1)
sqoop (1)
ssd (1)
ssl (2)
STAASH (1)
Stamos (1)
stream processing (3)
streaming (2)
suro (1)
SWF (1)
synchronization (1)
tablet (2)
testability (1)
Timed Text (1)
tls (2)
traffic optimization (1)
TTML (2)
TV (5)
UI (15)
UltraDNS (1)
unit test (2)
uptime (2)
user interface (5)
Velocity (1)
video quality (2)
visualization (1)
WebKit (3)
websockets (1)
Wii U (1)
windows (1)
winner (1)
winners (1)
workflow (1)
ZeroToDocker (1)

ZooKeeper (1)

zuul (1)