Designing and Operating Highly Available Software Systems at Scale

Ramón Medrano Llamas – @rmedranollamas
03.04.2019, EPI Gijón
What is “Site Reliability Engineering”?
Site Reliability Engineering (SRE)

Who we are: Software Engineers on a unique mission...

What we do: Make Google keep working.

How we do it: A mix of proactive and reactive engineering to make our planet-scale systems better.

Why: Users expect Google to be always available, fast, and operating correctly.
SRE: Proactive

Design
Planning
Automation
Consulting and Launches
Disaster Games
SRE: Reactive

Monitoring
Debugging & Troubleshooting
Root-cause Analysis
Performance Tuning
A Day in the Life of a Site Reliability Engineer
A “typical” day.

review code, changes, design docs.
read email. join video-conferences.
brainstorm. analyze. design. code.
meet your team in daily standup.
watch tech-talk. ask questions.
stare at system dashboards.
write road-maps and plans.
chat. eat. play. laugh.
get in the zone.
An “interesting” day.

On call responsibility for whole service.
A quiet day? Start working on project.
I get paged. Part of system is down.
Check: How many users affected?
Mitigate issue to restore service.
Deep dive. Find root cause.
Analyze last changes.
Find potential culprit.
Roll back change.
Postmortem.
Quiet.
Designing for Scale
No Query Left Behind

In the face of adversity: While we should try, we can’t literally serve every query, only 99.XXX% of them.

The cost of reliability: Shoots up exponentially with number of 9s.

What can fail: Memory, CPUs, hard drives, NICs, power supplies, cables, backhoes digging up fiber lines, trucks crashing into electrical substations ...
Suppose you just wrote a **web application** and it's running on your **server** with a **database** behind it and you can point your **web browser** to it, that means you're done, right?
Our simple service

Web server

Database

Users
Now, imagine your web application has to serve 100,000,000 users from around the world.

What needs to change?
What does 100M users mean?
What does 100M users mean?

10B requests/day
What does 100M users mean?

100k requests/second (average)
What does 100M users mean?

200k requests/second (peak)
What does 100M users mean?

200k requests/second (peak)

2M disk seeks per second (IOPS)
What do we need to serve them?

Lots of servers.

Probably lots of other stuff we aren’t going to talk much about.

Also known as “warehouse-scale computing”.

100M users need (at peak) 2M IOPS
at 100 IOPS per disk, that’s 20k disk drives
at 24 disks per server, that’s 834 servers
at 4 rack units (RU) per server, and 1¾” (4.44cm) per RU, that’s 486 ft (148m) stacked
Anatomy of a large-scale web application
Disclaimer

None of the following slides depict an actual Google service or actual Google technologies. They are indicative of the **scale** at which Google services operate, and the **technologies** needed to operate them.

All of the **problems** described on the following slides are problems that need to be solved to operate Google services. They all fall under the umbrella of the **Site Reliability Engineering** organization at Google.
Back to our simple service

Users → Web server → Database
Replication

Users

Web server

Web server

Database

Database
Meshed Topology

Not all connections shown.
One box for all replicas of each server type.
Architecture Diagram

Hierarchy of servers talking to each other.
Example Architecture

Names are entirely made up, this is not reflective of any real system, life or dead.
Cyclic?!?

Occasionally, servers talk to themselves.
Shrinking (the diagram)

We will need more space. Much more space.
How can the world reach 1000s of servers?

Can’t publish 1000s of IP addresses (one per server), add an IP load-balancer that will balance traffic over server.
Multiple Datacenters with Load Balancer Mesh

Not all connections shown.
How do we send traffic to the Front End Load Balancers?

DNS Server is IP Geo Location aware.

Users

DNS Server

DC-1

DC-2

DC-n
Monitoring

Now that we get traffic to the right place, how do we know what the servers are doing?

We need **monitoring** to figure out what the frontends are doing. How **loaded** are they? When are they getting **overloaded**? **How many requests** are they handling? Are they providing the **functionality** we expect?

No, we **don’t actually stare at screens** all day. That’s what automated alerts are for.
Service Level Indicators (SLIs)

- An **indicator** (SLI) is a *quantitative measure* of how good some *attribute* of the service is.
- An *attribute* is a dimension the service’s users care about, such as:
  - latency, how long the work takes
  - availability, how often the service can do work
  - correctness, whether the work is right
How Good Should A Service Be?

- How {fast, reliable, available, …} a service should be is fundamentally a *product* question
- “100% is the wrong reliability target for (nearly) everything”
  - cost of marginal improvements grows ~exponentially
- Can always make service better on some dimension, but involves tradeoffs with $, people, time, and other priorities
  - Product & dev management best placed for tradeoffs
Service Level Objectives

● An SLO is a mathematical relation like:
  ○ SLI ≤ target
  ○ lower bound ≤ SLI ≤ upper bound

● Error budget:
  ○ Difference of SLO to 100%
  ○ Use it for taking risk: changes, rollouts, etc.
  ○ Error budget allows velocity.
Monitoring Architecture

Servers are deeply \textit{instrumented}. Collect stats on \textbf{everything}.

Alert if human action is required. \textbf{Visualize} to help with troubleshooting.
Designing for Failure
“Failure is always an option.”

Things that fail, from low to high impact:

- machines
- switches
- power distribution units
- routers
- fiber lines
- power substations
- imperfect software
- human error
- adversarial action / malicious behavior

*We need to plan for all of these scenarios.*
Mitigate failure with Redundancy

Traditional approach: If an outage of one component causes a failure, use 2 and hope they don’t fail together.

Machine: Redundant power supplies, disk drives (raid)

Networking (redundant switches, routers)

Redundant database with two hosts.

Expensive, typically 2x plus complications.

Understand your systems – design for “defense in depth”.

Department of Redundancy Department
Failure Domain: Machine

Power supply unit (PSU) MTBF is 100,000 hours. For 10,000 machines: 1 PSU fails every 10 hours.

A myriad of other machine failure symptoms

Actions:

- Get traffic away from the failing machine and send it to repairs
- Possibly also move data elsewhere
Failure Domain: (Network) Switch

Symptom: Dozens of machines connected to one switch go offline at the same time.

Action:
- Get traffic away from affected machines
- and get someone to replace the switch.

If those machines hold any data, and the other location of that data is on the same switch: Congratulations, you’ve just lost actual data.
Failure domain: Power Distribution Unit

PDUs distribute power inside a datacenter.

Symptom: A large portion of your datacenter suddenly loses power.

Might take a large number of machines with it, as well as essential networking gear ... and sometimes produce nice sparks.

Action: umm ... tricky.

After a catastrophic power outage, machines (and drives) might not come up again.
Failure domain: Datacenter

Some failure modes can take out entire datacenters:

- hurricanes
- flooding
- earthquakes
- ...

These happen very rarely, but are the hardest to deal with.

This means that being in just one region of the world is not enough, so we need geographic diversity.
Failure Domain: Imperfect Software

This can really ruin your day

Action:
- when in doubt, roll back
- your roll-out strategy should make this easy and fast

Prevention: unit testing, integration testing, canarying, black-box testing, data integrity testing

Case study: malware blacklist outage
Back to our Datacenter Redundant Architecture

Now we can make good use of DNS IP Geo Location.

DNS Server

Users

DC-1

DC-2

DC-n
Datacenter Location Diversity

- US-East
- US-West
- EMEA
- APAC
How to deal with failure: Divert ("drain") traffic away

First we wanted to get traffic to our machines, now we want to get it away again!
because machines / datacenters / regions fail
need to figure out when to divert traffic away
see earlier slide about monitoring
can use the same mechanisms that we used for getting traffic to machines
see earlier slides about loadbalancing
Redundancy with 2 Server Instances: Steady State

Looks great, no? Each server only needs to handle 50% of the total load.
Redundancy with 2 Servers: Draining 1 Instance

Each server needs to be able to handle 100% of the load.

So we're provisioning for 200% of expected load.

100% overprovisioning is very expensive.
Redundancy with 3 Server Instances: Steady State

Now each server only needs to handle 33% of the total load.
Redundancy with 3 Servers: Draining 1 Instance

Each remaining server only needs to handle 50% of the total load.

Only needs provisioning for 150% of load expected, with only 50% overprovisioning (much better than 100% in the 2 instance case).

Conclusion:

As long as your per-server base cost is marginal, prefer large numbers of smaller instances.
Operational Considerations
Change Management

Downtime is not an option, all changes made “in flight”. Or, take a fraction of the service down.

Think about compatibility.

Deploying new code:

- Start small, get big fast if everything works
- 1 machine, 1 rack, 1 cluster, 1 region, 1 world
- Roll back on problems, never forward
- Apply this principle globally
Disaster Recovery

We talked about failures, let's talk about disasters! Examples:

- One of your major datacenters burns down
- Software bug silently corrupts data over months

Resolutions:

- ‘oops, my bad’ - probably not going to cut it

Prepare for emergency scenarios

- How to bring up the service somewhere else
- How to bring back your data and verify correctness.

Have ‘recovery plans’ not ‘backup plans’
Machines: Cattle vs. Pets

Need to manage services running on 10000+ machines.

There are not enough smurfs to name 10000+ machines, and even if there were you wouldn't be able to remember which service runs on papasmurf.example.com.

So we need a database to tell us what runs where (configuration) and a system to make services X and Y run on machine Z if the database says so (automation).
Business Continuity

Also called the "bus factor"

- Can you continue running your business if person X gets run over by a bus?

New people join the team, old-timers leave.

Everybody needs to do their share of emergency response.

Make the systems easy and safe to use, even if you don't understand 100% of their ins and outs.

Document the rest.
THANK YOU

niobium@google.com