Engineering Reliability into Sites

Dr Alexander Perry
Staff Software Engineer in Site Reliability Engineering
Google
Los Angeles, CA

RAM IX, Huntsville, AL

Abstract

This talk introduces Site Reliability Engineering (SRE) at Google,
explaining its purpose and describing the challenges it addresses.

SRE teams manage Google's many services and web sites from our offices in Pittsburgh, New York, London, Sydney, Zurich, Los Angeles, Dublin, Mountain View, ...

They draw upon the Linux based computing resources
that are distributed in data centers around the world.

Outline

1. What do we mean by **site** and its Reliability, Availability, Maintainability (RAM)

2. Hardware Reliability

3. The Challenge of Redundancy

4. Planning for Degradation of a Component

5. Systems consisting of non-identical Components

6. Hope is Not a Strategy

Site - An Integrated Deployment

- SRE Teams ensure user-visible reliability and availability
  - Need authority over relevant software and systems
- SRE Teams develop automation to deliver maintainability
  - In depth knowledge of the details is necessary

- Steep learning curve for engineers, mostly due to complexity
  - Continuous retraining, sites always being improved
- Specializations among teams for shared cloud infrastructure
  - Ensure those components are delivering on their service level objectives

Availability, Maintainability … at Google

Availability:

- Google Apps offers a 99.9% Service Level Agreement (SLA)
  - for covered services
  - in recent years we’ve exceeded this promise
  - In 2013, Gmail achieved 99.978% availability

Maintainability:

- Google Apps has no scheduled downtime or maintenance windows.
  - we do not plan for our applications to be unavailable
  - even when we're upgrading our services or maintaining our systems.

Reliability … at Google

How can Google be so reliable?

- Google's application and network architecture is designed
  - for maximum reliability and uptime
- Google's computing platform assumes ongoing hardware failure
  - And it uses robust software failover to withstand disruption
- All Google systems are inherently redundant by design
  - each subsystem is not dependent on any particular physical or logical server
- Data is replicated multiple times
  - across Google's clustered active servers

Assume Ongoing Hardware Failure

- Reliability Engineering at component level
  - Bathtub curve, mortality statistics, failure domain correlation
  - Service life, aging metrics, mission driven scheduling
- Best practices for defining and validating the models
  - Check for fit and residue for confidence of applicability
  - New sub-component technologies can suddenly break assumptions
- Concrete measurable aging metrics for reporting service life
  - Estimators inherently assume test stand use cases
  - Having more than one automatically logged parameter is fine

Assume Ongoing Changes in Customer Behavior

- Customer adapts requirements
  - For example when the available alternatives change
- Operational envelope and maintenance cadence gets restated
  - With successive block releases, and as historical data accumulates
- Customer may adapt chosen mission profile within envelope
  - But only if cost savings or other benefits justify the effort involved
  - Low duty cycle users likely stick with older mission profiles
- Each aging metric grows differently according to mission profile
  - Estimators prorated on test stand lifecycle data need big error bars
  - Models that incorporate a known mission as a parameter cannot adapt

Customer Behavior example - Light Aircraft

- Rental by “Hobbs” which is clock time with fuel included
  - Cost efficient at high speed, fly fast and high, high speed descents
  - Engine runs very hot for climb, hot for cruise, shock cool for descents
- Rental by “Tach” which is engine crankshaft rotations
  - Cost efficient at 50% power, fly slower, more at lower altitudes
  - Engine generally cool, gets carbon deposits, more airframe wear
- Ownership which is total cost including maintenance
  - Dominated by engine overhaul, which is tach time plus thermal transients
- Changing the **asset accounting** modifies the **ratio between aging metrics**
  - Hobbs, Tach, Thermal cycles, Thermal transients, Fuel burned, etc

Redundant Subsystems have more ways to fail

Duplicating for redundancy, such as multiple engines on aircraft means:

1. All duplicates fail together for common risks
   - Such as fuel exhaustion, volcanic ash, contaminated fuel
2. Mean time between failure of any duplicate is proportionally shorter
   - More maintenance cycles and corresponding risk of unrelated subsystem damage
3. Additional critical subsystems for input distribution, and for output selection
   - Such as fuel valves, auto-feather, propeller synchronization, failure identification
4. When degraded by a failure, some situations are catastrophic
   - Such as VMC for the minimum controllable airspeed, or the single engine service ceiling
5. Even safe operations must be subtly different when degraded by failure
   - Such as VYSE for the single engine climb speed, more care with bank angles

Redundant by Design

All of those factors need to be addressed somehow:

1. Assume a risk is common unless you can prove it independent
2. Multiply the known risk of maintenance faults by number of replicas
3. Additional critical subsystems must be simple, enabling reliability analysis
4. Catastrophic failure situations should be avoided in normal planning
5. Forward planning needs to comply with the degraded performance data

Humans are fallible, so use continuous testing and monitoring for compliance

Simplifying Redundancy for Reliability

- By default we added two critical subsystems for every redundant subsystem
  - Complexity often impairs reliability due to accidental engineering inconsistencies
- We can combine an output selector with the next input distributor
  - Obviously this only works if two adjacent components are both separately redundant
- We can make the output-input subsystem itself redundant too
  - The input subsystem also tells the next component which output subsystem replica to use
- With both of those, there might be nothing which isn’t redundant

- Care is needed: Even Master Election is hard to get right

Master Election, aka distributed consensus

- How hard can it be to reliably choose exactly one master?
    - Paxos protocol is the industry standard provably correct solution
    - But surprisingly difficult to cover all the corner cases in implementation
- Replace the correct but unintuitive protocol with an intuitive one
    - Delegate the conversion between distributed protocols to a separate service
    - It takes less engineering effort to validate one service than all other clients
- It is more effort than for a single protocol instance
    - Have to prove **all use cases** for the service, not just one
    - Have to prove the state machine’s safety for the intuitive protocol

Redundant by Design … for Computer Systems

Four major forms of redundancy are available:

1. Hardware and architectural redundancy, just like non computer systems
2. Information redundancy, such as error detection and correction methods
3. Time redundancy, sequentially performing the same operation multiple times
4. Software redundancy, multiple functionally equivalent implementations

Each only covers a subset of the faults against which redundancy is effective

- Expect to need more than one of them for each subsystem!

Planning for degradation is hard

The component having internal redundancy offers **at least two** specifications

- We can hope to get the nicer one, and usually we will … *two engines*
- Randomly, and with basically no warning, we will get the other one … *one engine*

Systems level planning needs to design for viability with the degraded one

- Do we always use the degraded one? This is safe, easy, but wasteful
- If not, any planning for other components must cover both cases
- An exponential number of combinations of maybe-degraded components
    - One engine is more critical than the other, propeller governor failure, high or low altitude, etc.

Manage Component Quality

- Operational monitoring should include replication behavior
  - Each Input distributor, how many replicas are being sent data
  - Each Output selector, how many replica are providing results, and how many are valid
- Three numbers for each instance of each replicating component
  - Probably thousands of numbers describing a realistic system’s live configuration
- Each combination of values is basically a fresh concrete component design
  - Statistical analysis over time to determine whether reliability goals are being met
  - Sensitivity analysis to determine whether that statistical analysis is flawed
  - Experiment analysis to form a probabilistic model for available performance

Component requirements need to be stated in terms of Service Levels

Identical Components will be Different

- Software is developed as human readable source files
  - Equivalent to mechanical drawings for brackets, bolts, etc
- Build tools compile human readable files to efficient binaries
  - CAM software converts a 3D model to toolpaths in gcode, etc
- Actual behavior of one given binary varies
  - Differences between parts machined from the same toolpath
- Binary assumes an ideal virtual machine
  - Virtual Machine is guaranteed to be perfect, but subject to unbounded latency
  - It is hard for a binary to find out which guarantee is driving its own latency up
  - Any persistent imperfection is hidden from the binary, by crashing it!
- Worst case execution time analysis - it’s not just scheduling

Manage Integrated System Quality

- A system is a special case of a high level Component
  - The probabilistic performance and service level becomes product reliability
  - There is nowhere to failover to, no replicas
  - Is there no way to avoid crashing if the load exceeds available performance?
- Individual requirements should be specified with distinct Service Levels
  - If so, the system may choose which Service Level Objective to abandon
- Execution Time Analysis informs the marginal cost of the Objectives
  - The marginal cost varies according to why the system is degraded
- Distinguish between offering an output to actually delivering the output
  - Delivering the output immediately is cheaper overall, as well as lower latency
  - Offering the output is cheaper on average, if the client sometimes skips making the request

Testing Validates Models

- Product Reliability is driven by Probabilistic Performance
  - Which is computed from resource interactions and component reliability
  - If that raw data is not trusted, the result should not be relied on

- Load testing a binary does not provide relevant data
  - Sensitivity analysis for redundancy configuration will usually fail

Realistic Testing

- Configurable virtual machines can run hermetic tests
  - Each test provides another data point verifying the resource model

- Continuous testing observes more binary versions
  - Changes in one component might interact with another’s performance

- After release and deployment, continue observing those versions
  - Accumulates an even larger diversity of verifying data points

Releases and Monitoring

- Each binary release has test-based measurement of reliability parameters
  - These are larger error bars than the parameters from ongoing operations of the last release

- The release is made available gradually to successive customer groups
  - Those with the largest remaining budget of Service Level Objective available
  - Proceed to the next group as the error bars show improvement in confidence

- A key part of Monitoring is accumulating all that Service Level data
  - Monitoring qualifying Releases quickly … drives better Maintainability

In Summary

- Design software architecture for availability
- Model components for reliability
- Unit Test replicated subsystems for sensitivity
- Regression Test binaries for downgrade probability
- Serialise releases for maintainability
- Monitor operations for validity

Conclusion

- Each SRE Team ensures user-visible reliability and availability
  - As a matter of routine engineering, not heroic operational efforts

- Each SRE Team develops automation to deliver maintainability
  - Arrange for predictable risks to apply when customers care less

- Horizontal collaboration across SRE Teams for shared infrastructure
  - Individual teams are customers of that systems product (which also has an SLA)

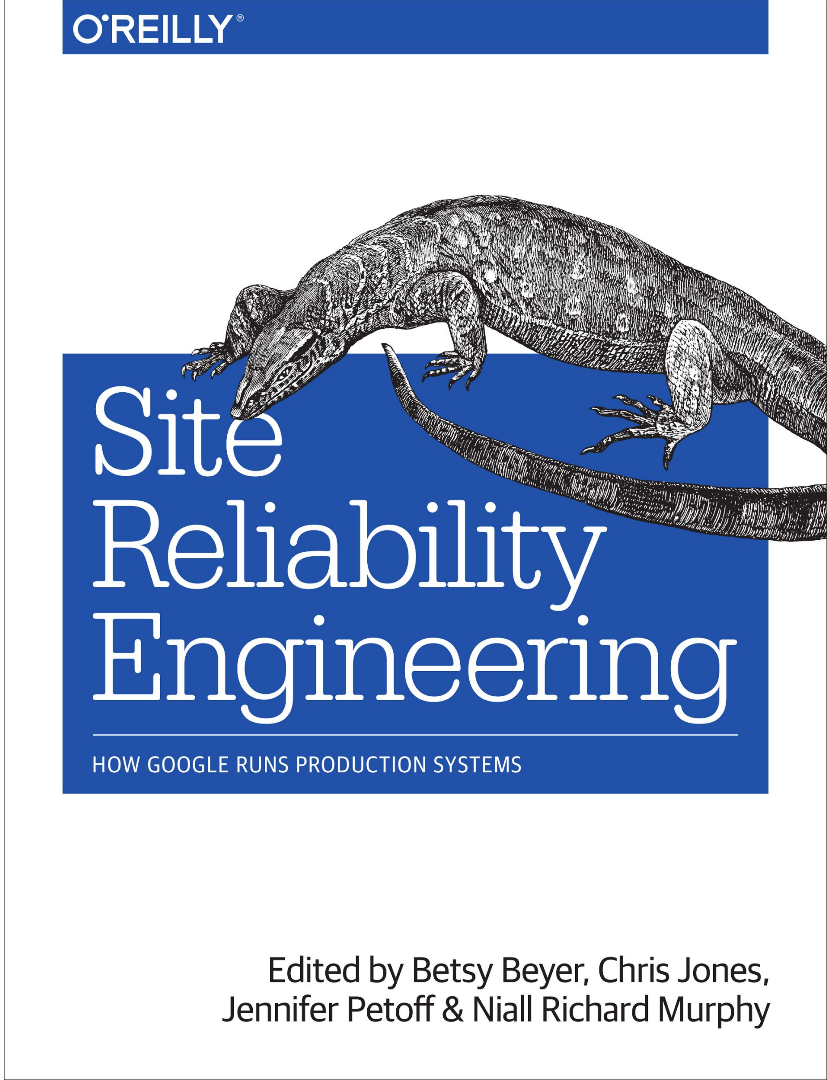
Google

Thank you

Questions?

landing.google.com/sre/book.html

Chapter 17 on Testing for Reliability jointly written by Alex.Perry@Google.com



O'REILLY®

Site Reliability Engineering

HOW GOOGLE RUNS PRODUCTION SYSTEMS

Edited by Betsy Beyer, Chris Jones, Jennifer Petoff & Niall Richard Murphy