PRACTICE

The Calculus of Service Availability

By Ben Treynor, Mike Dahlin, Vivek Rau, Betsy Beyer
Communications of the ACM, September 2017, Vol. 60 No. 9, Pages 42-47
10.1145/3080202
Comments

VIEW AS: SHARE:

Credit: Optanix

As detailed in *Site Reliability Engineering: How Google Runs Production Systems*[1] (hereafter referred to as the SRE book), Google products and services seek high-velocity feature development while maintaining aggressive service-level objectives (SLOs) for availability and responsiveness. An SLO says that the service should *almost always* be up, and the service should *almost always* be fast; SLOs also provide precise numbers to define what "almost always" means for a particular service. SLOs are based on the following observation:

The vast majority of software services and systems should aim for almost-perfect reliability rather than perfect reliability—that is, 99.999% or 99.99% rather than 100%—because users cannot tell the difference between a service being 100% available and less than "perfectly" available. There are many other systems in the path between user and service (laptop, home WiFi, ISP, the power grid . . .), and those systems collectively are far less than 100% available. Thus, the marginal difference between 99.99% and 100% gets lost in the noise of other unavailability, and the user receives no benefit from the enormous effort required to add that last fractional percent of availability. Notable exceptions to this rule include antilock brake control systems and pacemakers!

For a detailed discussion of how SLOs relate to SLIs (service-level indicators) and SLAs (service-level agreements), see the "Service Level Objectives" chapter in the SRE book. That chapter also details how to choose metrics that are meaningful for a particular service or system, which in turn drives the choice of an appropriate SLO for that service.

This article expands upon the topic of SLOs to focus on service dependencies. Specifically, we look at how the availability of critical dependencies informs the availability of a service, and how to design in order to mitigate and minimize critical dependencies.

Most services offered by Google aim to offer 99.99% (sometimes referred to as the "four 9s") availability to users. Some services contractually commit to a lower figure externally but set a 99.99% target internally. This more stringent target accounts for situations in which users become unhappy with service performance well before a contract violation occurs, as the number one aim of an SRE team is to keep users happy. For many services, a 99.99% internal target represents the sweet spot that balances cost, complexity, and availability. For some services, notably global cloud services, the internal target is 99.999%.

Back to Top

99.99% Availability: Observations And Implications

Let's examine a few key observations about and implications of designing and operating a 99.99% service and then move to a practical application.

Observation 1. Sources of outages. Outages originate from two main sources: problems with the service it-self and problems with the service's critical dependencies. A critical dependency is one that, if it malfunctions, causes a corresponding malfunction in the service.

Observation 2. The mathematics of availability. Availability is a function of the frequency and the duration of outages. It is measured through:

Outage *frequency*, or the inverse: MTTF (mean time to failure).

Duration, using MTTR (mean time to repair). Duration is defined as it is experienced by users: lasting from the start of a malfunction until normal behavior resumes.

Thus, availability is mathematically defined as MTTF/(MTTF+MTTR), using appropriate units.

Implication 1. Rule of the extra 9. A service cannot be more available than the intersection of all its critical dependencies. If your service aims to offer 99.99% availability, then all of your critical dependencies must be significantly more than 99.99% available.

Internally at Google, we use the following rule of thumb: critical dependencies must offer one additional 9 relative to your service—in the example case, 99.999% availability—because any service will have several critical dependencies, as well as its own idiosyncratic problems. This is called the "rule of the extra 9."

If you have a critical dependency that does not offer enough 9s (a relatively common challenge!), you must employ mitigation to increase the effective availability of your dependency (for example, via a *capacity cache, failing open, graceful degradation* in the face of errors, and so on.)

Implication 2. The math vis-à-vis frequency, detection time, and recovery time. A service cannot be more available than its incident frequency multiplied by its detection and recovery time. For example, three complete outages per year that last 20 minutes each result in a total of 60 minutes of outages. Even if the service worked perfectly the rest of the year, 99.99% availability (no more than 53 minutes of downtime per year) would not be feasible.

This implication is just math, but it is often overlooked, and can be very inconvenient.

(Corollary to implications 1 and 2. If your service is relied upon for an availability level you cannot deliver, you should make energetic efforts to correct the situation—either by increasing the availability level of your service or by adding mitigation as described earlier. Reducing expectations (that is, the published availability) is also an option, and often it is the correct choice: make it clear to the dependent service that it should either reengineer its system to compensate for your service's availability or reduce its own target. If you do not correct or address the discrepancy, an outage will inevitably force the need to correct it.

Back to Top

Practical Application

Let's consider an example service with a target availability of 99.99% and work through the requirements for both its dependencies and its outage responses.

The numbers. Suppose your 99.99% available service has the following characteristics:

One major outage and three minor outages of its own per year. Note that these numbers sound high, but a 99.99% availability target implies a 20- to 30-minute widespread outage and several short partial outages per year. (The math makes two assumptions: that a failure of a single shard is not considered a failure of the entire system from an SLO perspective, and that the overall availability is computed with a weighted sum of regional/shard availability.)

Five critical dependencies on other, independent 99.999% services.

Five independent shards, which cannot fail over to one another.

All changes are rolled out progressively, one shard at a time.

The availability math plays out as follows.

Dependency requirements.

The total budget for outages for the year is 0.01% of 525,600 minutes/year, or 53 minutes (based on a 365-day year, which is the worst-case scenario).

The budget allocated to outages of critical dependencies is five independent critical dependencies, with a budget of 0.001% each = 0.005%; 0.005% of 525,600 minutes/year, or 26 minutes.

The remaining budget for outages caused by your service, accounting for outages of critical dependencies, is 53 – 26 = 27 minutes.

Outage response requirements.

Expected number of outages: 4 (1 full outage, 3 outages affecting a single shard only)

Aggregate impact of expected outages: (1 × 100%) + (3 × 20%) = 1.6

Time available to detect and recover from an outage: 27/1.6 = 17 minutes

Monitoring time allotted to detect and alert for an outage: 2 minutes

Time allotted for an on-call responder to start investigating an alert: five minutes. (*On-call* means that a technical person is carrying a pager that receives an alert when the service is having an outage, based on a monitoring system that tracks and reports SLO violations. Many Google services are supported by an SRE on-call rotation that fields urgent issues.)

Remaining time for an effective mitigation: 10 minutes

Implication. Levers to make a service more available. It's worth looking closely at the numbers just presented because they highlight a fundamental point: there are three main levers to make a service more reliable.

Reduce the frequency of outages—via *rollout policy*, testing, design reviews, and other tactics.

Reduce the scope of the average outage—via *sharding, geographic isolation, graceful degradation*, or *customer isolation*.

Reduce the time to recover—via monitoring, one-button safe actions (for example, *rollback* or adding emergency capacity), *operational readiness practice*, and so on.

You can trade among these three levers to make implementation easier. For example, if a 17-minute MTTR is difficult to achieve, instead focus your efforts on reducing the scope of the average outage. Strategies for minimizing and mitigating critical dependencies are discussed in more depth later in this article.

Back to Top

Clarifying the "Rule of the Extra 9" for Nested Dependencies

A casual reader might infer that each additional link in a dependency chain calls for an additional 9, such that second-order dependencies need two extra 9s, third-order dependencies need three extra 9s, and so on.

This inference is incorrect. It is based on a naive model of a dependency hierarchy as a tree with constant fan-out at each level. In such a model, as shown in Figure 1, there are 10 unique first-order dependencies, 100 unique second-order dependencies, 1,000 unique third-order dependencies, and so on, leading to a total of 1,111 unique services even if the architecture is limited to four layers. A highly available service ecosystem with that many independent critical dependencies is clearly unrealistic.

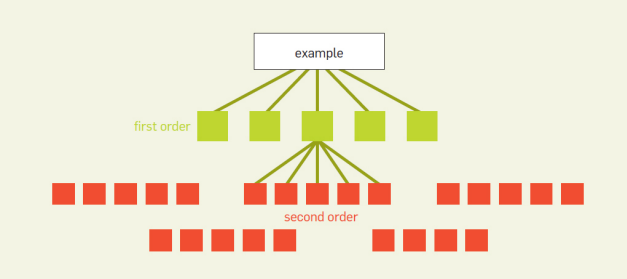

Figure 1. Dependency hierarchy: Incorrect model.

A critical dependency can by itself cause a failure of the entire service (or service shard) no matter where it appears in the dependency tree. Therefore, if a given component *X* appears as a dependency of several first-order dependencies of a service, *X* should be counted only once because its failure will ultimately cause the service to fail no matter how many intervening services are also affected.

The correct rule is as follows:

If a service has *N* unique critical dependencies, then each one contributes 1/*N* to the dependency-induced unavailability of the top-level service, regardless of its depth in the dependency hierarchy.

Each dependency should be counted only once, even if it appears multiple times in the dependency hierarchy (in other words, count only unique dependencies). For example, when counting dependencies of Service A in Figure 2, count Service B only once toward the total *N*.

For example, consider a hypothetical Service A, which has an error budget of 0.01%. The service owners are willing to spend half that budget on their own bugs and losses, and half on critical dependencies. If the service has *N* such dependencies, each dependency receives 1/Nth of the remaining error budget. Typical services often have about five to 10 critical dependencies, and therefore each one can fail only one-tenth or one-twentieth as much as Service A. Hence, as a rule of thumb, a service's critical dependencies must have one extra 9 of availability.

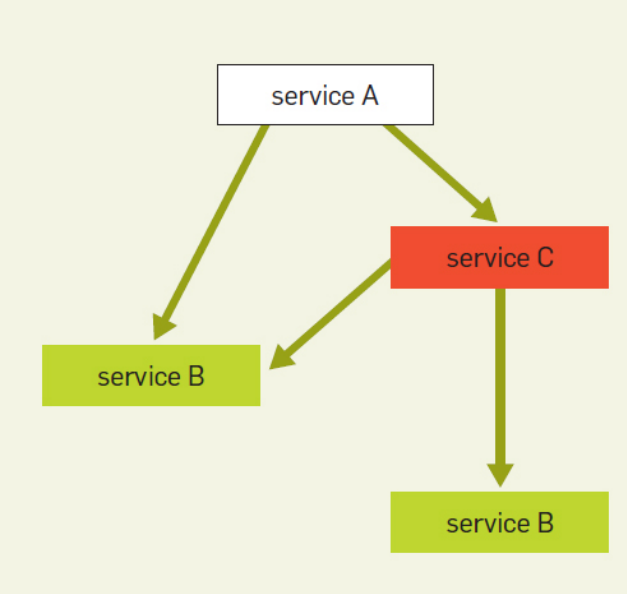

Figure 2. Multiple dependencies in the dependency hierarchy.

Back to Top

Error Budgets

The concept of error budgets is covered quite thoroughly in the SRE book,[1] but bears mentioning here. Google SRE uses error budgets to balance reliability and the pace of innovation. This budget defines the acceptable level of failure for a service over some period of time (often a month). An error budget is simply 1 minus a service's SLO, so the previously discussed 99.99% available service has a 0.01% "budget" for unavailability. As long as the service hasn't spent its error budget for the month, the development team is free (within reason) to launch new features, updates, and so on.

If the error budget is spent, the service freezes changes (except for urgent security fixes and changes addressing what caused the violation in the first place) until either the service earns back room in the budget, or the month resets. Many services at Google use sliding windows for SLOs, so the error budget grows back gradually. For mature services with an SLO greater than 99.99%, a quarterly rather than monthly budget reset is appropriate, because the amount of allowable downtime is small.

Error budgets eliminate the structural tension that might otherwise develop between SRE and product development teams by giving them a common, data-driven mechanism for assessing launch risk. They also give both SRE and product development teams a common goal of developing practices and technology that allow faster innovation and more launches without "blowing the budget."

Back to Top

Strategies for Minimizing and Mitigating Critical Dependencies

Thus far, this article has established what might be called the "Golden Rule of Component Reliability." This simply means that any critical component must be 10 times as reliable as the overall system's target, so that its contribution to system unreliability is noise. It follows that in an ideal world, the aim is to make as many components as possible noncritical. Doing so means the components can adhere to a lower reliability standard, gaining freedom to innovate and take risks.

The most basic and obvious strategy to reduce critical dependencies is to eliminate single points of failure (SPOFs) whenever possible. The larger system should be able to operate acceptably without any given component that's not a critical dependency or SPOF.

In reality, you likely cannot get rid of all critical dependencies, but you can follow some best practices around system design to optimize reliability. While doing so isn't always possible, it is easier and more effective to achieve system reliability if you plan for reliability during the design and planning phases, rather than after the system is live and impacting actual users.

Conduct architecture/design reviews. When you are contemplating a new system or service, or refactoring or improving an existing system or service, an architecture or design review can identify shared infrastructure and internal vs. external dependencies.

Shared infrastructure. If your service is using shared infrastructure—for example, an underlying database service used by multiple user-visible products—think about whether or not that infrastructure is being used correctly. Be explicit in identifying the owners of shared infrastructure as additional stakeholders. Also, beware of overloading your dependencies—coordinate launches carefully with the owners of these dependencies.

Internal vs. external dependencies. Sometimes a product or service depends on factors beyond company control—for example, code libraries, or services or data provided by third parties. Identifying these factors allows you to mitigate the unpredictability they entail.

Engage in thoughtful system planning and design. Design your system with the following principles in mind.

Redundancy and isolation. You can seek to mitigate your reliance upon a critical dependency by designing that dependency to have multiple independent instances. For example, if storing data in one instance provides 99.9% availability for that data, then storing three copies in three widely distributed instances provides a theoretical availability level of 1 – 0.013, or nine 9s, if instance failures are independent with zero correlation.

In the real world, the correlation is never zero (consider network backbone failures that affect many cells concurrently), so the actual availability will be nowhere close to nine 9s but is much higher than three 9s. Also note that if a system or service is "widely distributed," geographic separation is not always a good proxy for uncorrelated failures. You may be better off using more than one system in nearby locations than the same system in distant locations.

Similarly, sending an RPC (remote procedure call) to one pool of servers in one cluster may provide 99.9% availability for results, but sending three concurrent RPCs to three different server pools and accepting the first response that arrives helps increase availability to well over three 9s (noted earlier). This strategy can also reduce *tail latency* if the server pools are approximately equidistant from the RPC sender. (Since there is a high cost to sending three RPCs concurrently, Google often stages the timing of these calls strategically: most of our systems wait a fraction of the allotted time before sending the second RPC, and a bit more time before sending the third RPC.)

Failover and fallback. Pursue software rollouts and migrations that *fail safe* and are automatically isolated should a problem arise. The basic principle at work here is that by the time you bring a human online to trigger a *failover*, you have likely already exceeded your error budget.

Where concurrency/voting is not possible, automate failover and *fallback*. Again, if the issue needs a human to check what the problem is, the chances of meeting your SLO are slim.

Asynchronicity. Design dependencies to be asynchronous rather than synchronous where possible so that they don't accidentally become critical. If a service waits for an RPC response from one of its noncritical dependencies and this dependency has a spike in latency, the spike will unnecessarily hurt the latency of the parent service. By making the RPC call to a noncritical dependency asynchronous, you can decouple the latency of the parent service from the latency of the dependency. While asynchronicity may complicate code and infrastructure, this trade-off will be worthwhile.

Capacity planning. Make sure that every dependency is correctly provisioned. When in doubt, overprovision if the cost is acceptable.

Configuration. When possible, standardize configuration of your dependencies to limit inconsistencies among subsystems and avoid one-off failure/error modes.

Detection and troubleshooting. Make detecting, troubleshooting, and diagnosing issues as simple as possible. Effective monitoring is a crucial component of being able to detect issues in a timely fashion. Diagnosing a system with deeply nested dependencies is difficult. Always have an answer for mitigating failures that doesn't require an operator to investigate deeply.

Fast and reliable rollback. Introducing humans into a mitigation plan substantially increases the risk of missing a tight SLO. Build systems that are easy, fast, and reliable to roll back. As your system matures and you gain confidence in your monitoring to detect problems, you can lower MTTR by engineering the system to automatically trigger safe rollbacks.

Systematically examine all possible failure modes. Examine each component and dependency and identify the impact of its failure. Ask yourself the following questions:

> Can the service continue serving in degraded mode if one of its dependencies fails? In other words, design for graceful degradation.

How do you deal with unavailability of a dependency in different scenarios? Upon startup of the service? During runtime?

Conduct thorough testing. Design and implement a robust testing environment that ensures each dependency has its own test coverage, with tests that specifically address use cases that other parts of the environment expect. Here are a few recommended strategies for such testing:

Use *integration testing* to perform fault injection—verify that your system can survive failure of any of its dependencies.

Conduct disaster testing to identify weaknesses or hidden/unexpected dependencies. Document follow-up actions to rectify the flaws you uncover.

Don't just load test. Deliberately overload your system to see how it degrades. One way or another, your system's response to overload *will* be tested; better to perform these tests yourself than to leave load testing to your users.

Plan for the future. Expect changes that come with scale: a service that begins as a relatively simple binary on a single machine may grow to have many obvious and nonobvious dependencies when deployed at a larger scale. Every order of magnitude in scale will reveal new bottlenecks—not just for your service, but for your dependencies as well. Consider what happens if your dependencies cannot scale as fast as you need them to.

Also be aware that system dependencies evolve over time and that your list of dependencies may very well grow over time. When it comes to infrastructure, Google's typical design guideline is to build a system that will scale to 10 times the initial target load without significant design changes.

Back to Top

Conclusion

While readers are likely familiar with some or many of the concepts this article has covered, assembling this information and putting it into concrete terms may make the concepts easier to understand and teach. Its recommendations are uncomfortable but not unattainable. A number of Google services have consistently delivered better than four 9s of availability, not by superhuman effort or intelligence, but by thorough application of principles and best practices collected and refined over the years (see SRE's Appendix B: A Collection of Best Practices for Production Services).

Back to Top

Acknowledgments

Thank you to Ben Lutch, Dave Rensin, Miki Habryn, Randall Bosetti, and Patrick Bernier for their input.

Related articles
on queue.acm.org

There's Just No Getting Around It: You're Building a Distributed System
Mark Cavage
http://queue.acm.org/detail.cfm?id=2482856

Eventual Consistency Today: Limitations, Extensions, and Beyond
Peter Bailis and Ali Ghodsi
http://queue.acm.org/detail.cfm?id=2462076

A Conversation with Wayne Rosing
David J. Brown
http://queue.acm.org/detail.cfm?id=945162

Back to Top

References

1. Beyer, B., Jones, C., Petoff, J., Murphy, N.R. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016; https://landing.google.com/sre/book.html.

Back to Top

Authors

Ben Treynor started programming at age six and joined Oracle as a software engineer at age 17. He has also worked in engineering management at E.piphany, SEVEN, and Google (2003-present). His current team of

approximately 4,200 at Google is responsible for Site Reliability Engineering, networking, and datacenters worldwide.

Mike Dahlin is a distinguished engineer at Google, where he has worked on Google's Cloud Platform since 2013. Prior to joining Google, he was a professor of computer science at the University of Texas at Austin.

Vivek Rau is an SRE manager at Google and a founding member of the Launch Coordination Engineering sub-team of SRE. Prior to joining Google, he worked at Citicorp Software, Versant, and E.piphany. He currently manages various SRE teams tasked with tracking and improving the reliability of Google's Cloud Platform.

Betsy Beyer is a technical writer for Google, specializing in Site Reliability Engineering. She has previously written documentation for Google's Data Center and Hardware Operations Teams. She was formerly a lecturer on technical writing at Stanford University.

Back to Top

Sidebar: Key Definitions

Some of the terms and concepts used throughout this article may not be familiar to readers who don't specialize in operations.

Capacity cache: A cache that serves precomputed results for API calls or queries to a service, generating cost savings in terms of compute/IO resource needs by reducing the volume of client traffic hitting the underlying service.

Unlike the more typical performance/latency cache, a capacity cache is considered critical to service operation. A drop in the cache hit rate or cache ratio below the SLO is considered a capacity loss. Some capacity caches may even sacrifice performance (for example, redirecting to remote sites) or freshness (for example, CDNs) in order to meet hit rate SLOs.

Customer isolation: Isolating customers from each other may be advantageous so that the behavior of one customer doesn't impact other customers. For example, you might isolate customers from one another based on their global traffic. When a given customer sends a surge of traffic beyond what they're provisioned for, you can start throttling or rejecting this excess traffic without impacting traffic from other customers.

Failing safe/failing open/failing closed: Strategies for gracefully tolerating the failure of a dependency. The "safe" strategy depends on context: failing open may be the safe strategy in some scenarios, while failing closed may be the safe strategy in others.

Failing open: When the trigger normally required to authorize an action fails, failing open means to let some action happen, rather than making a decision. For example, a building exit door that normally requires badge verification "fails open" to let you exit without verification during a power failure.

Failing closed is the opposite of falling open. For example, a bank vault door denies all attempts to unlock it if its badge reader cannot contact the access-control database.

Failing safe means whatever behavior is required to prevent the system from falling into an unsafe mode when expected functionality suddenly doesn't work. For example, a given system might be able to *fail open* for a while by serving cached data, but then *fail closed* when that data becomes stale (perhaps because past a certain point, the data is no longer useful).

Failover: A strategy that handles failure of a system component or service instance by automatically routing incoming requests to a different instance. For example, you might route database queries to a replica database, or route service requests to a replicated server pool in another datacenter.

Fallback: A mechanism that allows a tool or system to use an alternative source for serving results when a given component is unavailable. For example, a system might fall back to using an in-memory cache of previous results. While the results may be slightly stale, this behavior is better than outright failure. This type of fallback is an example of graceful degradation.

Geographic isolation: You can build additional reliability into your service by isolating particular geographic zones to have no dependencies on each other. For example, if you separate North America and Australia into separate serving zones, an outage that occurs in Australia because of a traffic overload won't also take out your service in North America. Note that geographic isolation does come at increased cost: isolating these geographic zones also means that Australia cannot borrow spare capacity in North America.

Graceful degradation: A service should be "elastic" and not fail catastrophically under overload conditions and spikes—that is, you should make your applications do something reasonable even if not all is right. It is better to give users limited functionality than an error page.

Integration testing: The phase in software testing in which individual software modules are combined and tested as a group to verify that they function correctly together. These "parts" may be code modules,

individual applications, client and server applications on a network, among others. Integration testing is usually performed after unit testing and before final validation testing.

Operational readiness practice: Exercises designed to ensure the team supporting a service knows how to respond effectively when an issue arises, and that the service is resilient to disruption. For example, Google performs disaster-recovery test drills continuously to make sure that its services deliver continuous uptime even if a large-scale disaster occurs.

Rollout policy: A set of principles applied during a service rollout (a deployment of any sort of software component or configuration) to reduce the scope of an outage in the early stages of the rollout. For example, a rollout policy might specify that rollouts occur progressively, on a 5%/20%/100% timeline, so that a rollout proceeds to a larger portion of customers only when it passes the first milestone without problems. Most problems will manifest when the service is exposed to a small number of customers, allowing you to minimize the scope of the damage. Note that for a rollout policy to be effective in minimizing damage, you must have a mechanism in place for rapid rollback.

Rollback: This is the ability to revert a set of changes that have been previously rolled out (fully or not) to a given service or system. For example, you can revert configuration changes or run a previous version of a binary that's known to be good.

Sharding: Splitting a data structure or service into shards is a management strategy based on the principle that systems built for a single machine's worth of resources don't scale. Therefore, you can distribute resources such as CPU, memory, disk, file handles, and so on across multiple machines to create smaller, faster, more easily managed parts of a larger whole.

Tail latency: When setting a target for the latency (response time) of a service, it is tempting to measure the average latency. The problem with this approach is that an average that looks acceptable can hide a "long tail" of very large outliers, where some users may experience terrible response times. Therefore, the SRE best practice is to measure and set targets for 95^{th}- and/ or 99^{th}-percentile latency, with the goal of reducing this tail latency, not just average latency.

No entries found

Comment on this article

Signed comments submitted to this site are moderated and will appear if they are relevant to the topic and not abusive. Your comment will appear with your username if published. View our policy on comments

(Please sign in or create an ACM Web Account to access this feature.)

Create an Account

SUBMIT FOR REVIEW