Implementing health checks

David Yanacek

Services can be designed with all kinds of reliability and resiliency built in, but in order to be reliable in practice, they must also deal with predictable failures when they occur. At Amazon, we build services to be horizontally scalable and redundant, because hardware is designed to fail eventually. Any hard drive has a maximum expected lifetime, and any piece of software is susceptible to crash at some point. It may seem like a server's health is binary: it either works, or it doesn't work at all and gets out of the way. Unfortunately this isn't the case. We find that rather than just shutting down, failing servers can cause unpredictable and sometimes disproportionate harm to a system. Health checks detect and respond to these kinds of issues automatically.

This article describes how we use health checks to detect and deal with single-server failures, the things that happen when health checks are not used, and how systems that overreact to health check failures can turn small problems into complete outages. We also provide insight from our experience at Amazon about balancing the tradeoffs between various kinds of health check implementations.

Small failures with outsized impact

When I was a new software developer at Amazon, I worked on the website rendering fleet behind Amazon.com. While working on a change to add some instrumentation and get visibility into how well the software was running, I unfortunately wrote a bug. The bug triggered rarely, but when it did, it caused a given web server to render blank error pages on every request. Only restarting the web server process fixed the issue. We detected the bug and rolled back the change quickly, added plenty of tests, and improved processes to catch conditions like this in the future. But while the bug was in production, a few servers in a large fleet ended up in this broken state.

One thing that made the bug especially tricky to find was that the server didn't realize it was unhealthy. Also, the server lost the ability to report its health into monitoring systems, so it was not taken out of service automatically, and it didn't trigger its regular alarms. Making matters worse, the server became *very* fast and began producing blank error pages much faster than its peer "healthy servers" were rendering happy webpages. The load-balancing technology we used at the time favored fast servers over slow ones, so it directed a disproportionate amount of traffic to the unhealthy servers, which increased the impact even further.

Other alarms triggered, since monitoring involves measuring error rates and latency from multiple points in the system. While these kinds of monitoring systems and operational processes can serve as a backstop to contain the problem, the right health checks can significantly minimize the impact of this whole class of errors by detecting and acting on failures quickly.

Health check tradeoffs

Health checks are a way of asking a service on a particular server whether or not it is capable of performing work successfully. Load balancers ask each server this question periodically to determine which servers it is safe to direct traffic to. A service that polls messages from a queue might ask *itself* whether it is healthy before it decides to poll more work from the queue. Monitoring agents—running on each server or on an external monitoring fleet—might ask servers whether they're healthy so that they can raise an alarm or automatically deal with servers that are failing.

As we saw in my website bug example, when an unhealthy server stays in service, it can disproportionately decrease the availability of the service as a whole. With a fleet of ten servers, one bad server means that the availability of the fleet would be 90% or less. Making matters worse, some load-balancing algorithms, such as "least requests," give more work to the fastest server. When a server fails, it often begins failing requests quickly, creating a "black hole" in the service fleet by

attracting more requests than healthy servers. In some cases, we add extra protection to prevent black holes by slowing down failed requests to match the average latency of successful requests. However, there are other scenarios, such as with queue pollers, where this issue is more difficult to work around. For example, if a queue poller is polling messages as fast as it can receive them, a failed server will become a black hole as well. With such a diverse set of environments for distributing work, the way we think about protecting a partially-failed server varies from system to system.

We find that servers fail *independently* for any number of reasons, including disks that become unwritable and cause requests to fail immediately, clocks that skew abruptly and cause calls to dependencies to fail authentication, servers that fail to retrieve updated crypto material and cause decryption and encryption to fail, critical support processes that crash because of their own bugs, memory leaks, and deadlocks that freeze processing.

Servers also fail for *correlated* reasons that cause many or all servers in a fleet fail together. Correlated reasons include outage of a shared dependency and large-scale network issues.

The ideal health check will test every aspect of server and application health, perhaps even verifying that non-critical supporting processes are running. However, trouble arises when the health check fails for a *non-critical* reason and when that failure is *correlated* across servers. If automation removes servers from service when they still could have performed useful work, the automation does more harm than good.

The difficulty with health checks is this tension between, on the one hand, the benefits of thorough health checks and quickly mitigating single-server failures and, on the other hand, the harm done by a false positive failure across the entire fleet. Thus, one of the challenges of building a good health check is to guard carefully against false positives. In general, this means that the automation surrounding health checks should stop directing traffic to a single bad server but keep allowing traffic if the entire fleet *appears* to be having trouble.

Ways to measure health

There are many things that can break on a server, and there are many places in our systems where we measure server health. Some health checks can definitively report that a particular server is independently broken, while others are fuzzier and report false positives in the case of correlated failures. Some health checks are difficult to implement. Others are implemented at setup with services like Amazon Elastic Compute Cloud (Amazon EC2) and Elastic Load Balancing. Each type of health check has its strengths.

Liveness checks

Liveness checks test the basic connectivity to a service and the presence of a server process. They are often performed by a load balancer or external monitoring agent, and they are unaware of the details about how an application works. Liveness checks tend to be included with the service and do not require an application author to implement anything. Some examples of liveness checks that we use at Amazon include the following:

- Tests that confirm that a server is listening on its expected port and accepting new TCP connections.
- Tests that perform a basic HTTP requests and make sure that the server responds with a 200 status code.
- Status Checks for Amazon EC2 that test for basic things that are necessary for any system to operate, such as network reachability.

Local health checks

Local health checks go further than liveness checks to verify that the application is likely to be able to function. These health checks test resources that are not shared with the server's peers. Therefore, they are unlikely to fail on many servers in the fleet simultaneously. These health checks test for the following:

- Inability to write to or read from disk—It may be tempting to believe that a stateless service doesn't require a writable disk. However, at Amazon our services tend to use their disks for things like monitoring, logging, and publishing asynchronous metering data.
- Critical processes crashing or breaking—Some services take requests using an on-server proxy (similar to NGINX) and perform their business logic in another server process. A liveness check might only test whether the proxy process is running. A local health check process might pass through from the proxy to the application to check that both are running and answering requests correctly. Interestingly, in the website example from the beginning of the article, the existing health check was deep enough to ensure that the rendering process was running and responding but not deep enough to ensure it was responding correctly.
- Missing support processes—Hosts that are missing their monitoring daemons might leave operators "flying blind" and unaware of the health of their services. Other support processes push metering and billing usage records or receive credential updates. Servers with broken support processes put functionality at risk in subtle, difficult-to-detect ways.

Dependency health checks

Dependency health checks are a thorough inspection of the ability of an application to interact with its adjacent systems. These checks ideally catch problems local to the server, such as expired credentials, that are preventing it from interacting with a dependency. But they can also have false positives when there are problems with the dependency itself. Because of those false positives, we must be careful about how we react to dependency health check failures. Dependency health checks might test for the following:

- Bad configuration or stale metadata—If a process asynchronously looks for updates to metadata or configuration but the update mechanism is broken on a server, the server can become significantly out of sync with its peers and misbehave in an unpredictable and untested way. However, when a server doesn't see an update for a while, it doesn't know whether the update mechanism is broken or the central update system stopped publishing updates to all servers.
- Inability to communicate with peer servers or dependencies—Strange network behavior has been known to affect the ability of a subset of servers in a fleet to talk to dependencies without affecting the ability for traffic to be sent to that server. Software issues, such as deadlocks or bugs in connection pools, can also hinder network communication.
- Other unusual software bugs that require a process bounce—Deadlocks, memory leaks, or state corruption bugs can make a server spew errors.

Anomaly detection

Anomaly detection looks across all servers in a fleet to determine if any server is behaving oddly compared to its peers. By aggregating monitoring data per server, we can continuously compare error rates, latency data, or other attributes to find anomalous servers and automatically remove them from service. Anomaly detection can find divergence in the fleet that a server cannot detect about itself, such as the following:

- Clock skew—Especially when servers are under high load, their clocks have been known to skew abruptly and drastically. Security measures, such as those used to evaluate signed requests to AWS, require that the time on a client's clock is within five minutes of the actual time. If it is not, requests fail to AWS services.
- Old code—If a server is disconnected from the network or powered off for a long time and then comes back on line, it could be running dangerously outdated code that is incompatible with the rest of the fleet.
- Any unanticipated failure mode—Sometimes servers fail in such a way that they return errors that they identify error as the client's instead of theirs (HTTP 400 instead of 500). Servers may slow down instead of failing, or they may respond faster than their peers, which is a sign that they're returning false responses to their callers. Anomaly detection is an incredible catchall for unanticipated failure modes.

There are few things that must hold true for anomaly detection to work in practice:

- Servers should be doing approximately the same thing—In cases where we explicitly route different types of traffic to different types of servers, the servers might not behave similarly enough to detect outliers. However, in cases where we use load balancers to direct traffic to servers, they are likely responding in similar ways.
- Fleets should be relatively homogeneous—In fleets that include different instance types, some instances might be slower than others, which can falsely trigger passive bad server detection. To work around this scenario, we collate metrics by instance type.
- The errors or difference in behavior must be reported—Because we rely on the servers themselves to report errors, what happens when their monitoring systems are also broken? Fortunately the client of a service is a great place to add instrumentation. Load balancers like Application Load Balancer publish access logs that show which backend server was contacted on every request, the response time, and whether the request succeeded or failed.

Reacting safely to health check failures

When a server determines that it is unhealthy, there are two kinds of actions it can take. In the most extreme case, it can decide locally that it shouldn't be given any work and take itself out of service by failing a load balancer health check or by stopping polling a queue. Another way the server could react is to inform a central authority that it has a problem and let the central system decide how to handle the issue. The central system can safely address the problem without letting the automation take down the whole fleet.

There are multiple ways to implement and respond to health checks. This section describes a few patterns that we use at Amazon.

Fail open

Some load balancers can act as a smart central authority. When an individual server fails a health check, the load balancer stops sending it traffic. But when all servers fail health checks at the same time, the load balancer fails open, allowing traffic to all servers. We can use load balancers to support the safe implementation of a dependency health check, perhaps including one that queries its database and checks to ensure that its non-critical support processes are running.

For example, the AWS Network Load Balancer fails open if no servers are reporting as healthy. It also fails out of unhealthy Availability Zones if all servers in an Availability Zone reports unhealthy. (For more information about using Network Load Balancers for health checks, see the Elastic Load Balancing documentation.) Our Application Load Balancer also supports fail open, as does Amazon Route 53. (For more information about configuring health checks with Route 53, see the Route 53 documentation.)

When we rely on fail-open behavior, we make sure to test the failure modes of the dependency heath check. For example, consider a service where the servers connect to a shared data store. If that data store becomes slow or responds with a low error rate, the servers might occasionally fail their dependency health checks. This condition causes servers to flap in and out of service but does not trigger the fail-open threshold. Reasoning out and testing partial failures of dependencies with these health checks is important to avoid a situation where a failure could cause deep health checks to make matters worse.

While fail open is a helpful behavior, at Amazon we tend to be skeptical of things that we can't fully reason about or test in all situations. We haven't yet come up with general proofs that fail open will trigger as we expect for all types of overload, partial failures, or gray failures in a system or in that system's dependencies. Because of this limitation, teams at Amazon tend to restrict their fast-acting load balancer health checks to local health checks and rely on centralized systems to carefully react to deeper dependency health checks. This isn't to say we don't use fail-open behavior or prove that it works in particular cases. But when logic can act on a large number of servers quickly, we are extremely cautious about that logic.

Health checks without a circuit breaker

Allowing servers to react to their own problems may seem like the quickest and simplest path to recovery. However, it is also the riskiest path if the server is wrong about its health or doesn't see the whole picture of what's happening across the fleet. When all servers across the fleet make the same wrong decision simultaneously, it can cause cascading failures throughout adjacent services. This risk presents us with a trade-off. If there is a gap in health checking and monitoring, a server could reduce the availability of a service until the issue is detected. However, this scenario avoids a complete service outage due to unexpected health check behavior across a whole fleet.

These are the best practices we follow for implementing health checks when we don't have a circuit breaker built-in:

- Configure the work producer (load balancer, queue polling thread) to perform *liveness* and *local* health checks. Servers are taken out of service automatically by the load balancer only if they have some problem that is definitively local to that server, such as a bad disk.

- Configure other external monitoring systems to perform *dependency health checks* and *anomaly detection*. These systems could attempt to terminate instances automatically or alarm or engage an operator.

When we build systems to react automatically to dependency health check failures, we must build in the right amount of thresholding to prevent the automated system from taking drastic action unexpectedly. Teams at Amazon that operate stateful servers like Amazon DynamoDB, Amazon S3, and Amazon Relational Database Service (Amazon RDS) have important durability requirements around server replacement. They have also built cautious rate limiting and control feedback loops so that the automation stops and engages humans when thresholds are crossed. When we build such automation, we must be sure that we notice when a server fails a dependency health check. For some metrics, we rely on the servers to self-report their individual status to a central monitoring system. To compensate for cases when the server is so broken that it is unable to report its health, we also actively reach out to them to check their health.

Prioritize your health

Especially in overload conditions, it is important for servers to prioritize their health checks over their regular work. In this situation, failing or responding slowly to health checks can make a bad brownout situation even worse.

When a server fails a load balancer health check, it is asking that load balancer to take it out of service immediately and for a non-trivial amount of time. When a single server fails, that's not a problem, but in a traffic surge to the service, the last thing we want is to shrink the size of the service. Taking servers out of service during an overload can cause a downward spiral. Forcing the remaining servers take even more traffic makes them more likely to become overloaded, also fail a health check, and shrink the fleet even more.

The problem is not that overloaded servers return errors when they're overloaded. It's that servers don't respond to the load balancer ping request in time. After all, load balancer health checks are configured with timeouts, just like any other remote service call. Browned out servers are slow to respond for a number of reasons, including high CPU contention, long garbage collector cycles, or simply running out of worker threads. Services need to be configured to set resources aside to respond to health checks in a timely way instead of taking on too many additional requests.

Fortunately, there are some straightforward configuration best practices that we follow to help prevent this kind of downward spiral. Tools like iptables, and even some load balancers, support the notion of “max connections.” In this case, the OS (or load balancer) limits the number of connections to the server so that the server process is not flooded with concurrent requests that would have slowed it down.

When a service is fronted by a proxy or a load balancer that supports max connections, it seems logical to make the number of worker threads on the HTTP server match the max connections in the proxy. However, this configuration would set up the service for a downward spiral during a brownout. Proxy health checks need connections too, and so it is important to make a server's worker pool large enough to accommodate extra health check requests. Idle workers are cheap, so we tend to configure extra ones: anywhere from a handful of extra workers to double the configured proxy max connections.

Another strategy we use to prioritize health checks is for servers to implement their own maximum concurrent requests enforcement. In this case, load balancer health checks are always allowed, but

normal requests are rejected if the server is already working on some threshold. Implementations around Amazon range from a simple semaphore in Java to the more complex analysis of trends in CPU utilization.

Another way to help ensure that services respond in time to a health check ping request is to perform the dependency health check logic in a background thread and update an isHealthy flag that the ping logic checks. In this case, servers respond promptly to health checks, and the dependency health checking produces a predictable load on the external system it interacts with. When teams do this, they are extra cautious about detecting a failure of the health check thread. If that background thread exits, the server does not detect a future server failure (or recovery!).

Balancing dependency health checks with the scope of impact

Dependency health checks are appealing because they act as a thorough test of a server's health. Unfortunately they can be dangerous because a dependency can cause a cascading failure throughout a system.

We can draw some insight about handling health check dependencies by looking at our service-oriented architecture at Amazon. Each service at Amazon is designed to do a small number of things; there is no monolith that does everything. There are many reasons we like to build services this way, including faster innovation with small teams and reduced scope of impact if there is a problem with one service. This architectural design can apply to health checks too.

When one service calls another service, it's taking a dependency on that service. If a service only calls the dependency sometimes, we might consider the dependency to be a "soft dependency," since the service can still do some types of work even if it can't talk to the dependency. Without fail-open protection, implementing a health check that tests a dependency turns that dependency into a "hard dependency." If the dependency is down, the service also goes down, creating a cascading failure with increased scope of impact.

Even though we separate functionality into different services, each service likely serves multiple APIs. Sometimes, APIs on the service have their own dependencies. If one API is impacted, we prefer for the service to continue serving the other APIs. For example, a service can be both a control plane (such as occasionally-called CRUD APIs on long-living resources) and a data plane (high throughput business-super-critical APIs). We would want the data plane APIs to continue to operate even if the control plane APIs are having trouble talking to their dependencies.

Similarly, even a single API may behave differently depending on the input or state of the data. A common pattern is a Read API that queries a database but caches responses locally for some time. If the database is down, the service can still serve cached reads until the database is back online. Failing health checks if only one code path is unhealthy increases the scope of impact of a problem talking to a dependency.

This discussion of which dependency to health check raises an interesting question about the trade-offs between microservices and relatively monolithic services. There is rarely a clear-cut rule for how many deployable units or endpoints to break a service into, but the questions of "which dependencies to health check" and "does a failure then increase the scope of impact" are interesting lenses to use to determine how micro or macro to make a service.

Real things that have gone wrong with health checks

All of this may make sense in theory, but what happens to systems in practice when they don't get health checks right? We looked for patterns in stories from AWS customers and from around Amazon to help illustrate the bigger picture. We also looked into compensating factors – the sorts of "belt and suspenders" that teams implement to prevent a weakness in a health check from causing a widespread issue.

Deployments

One pattern of health check problems involves deployments. Deployment systems like AWS CodeDeploy push new code to one subset of the fleet at a time, waiting for one deployment wave to complete before moving on to the next. This process relies on servers reporting back to the deployment system once they're up and running with the new code. If they don't report back, the deployment system sees that there is something wrong with the new code and rolls back the deployment.

The most basic service startup deployment script would simply fork the server process and immediately respond "deployment done" to the deployment system. However this is dangerous because so many things can go wrong with the new code: the new code could crash right after launching, get hung up and fail to start listening on a server socket, fail to load configuration needed to process requests successfully, or encounter a bug. When a deployment system isn't configured to test against a dependency health check, it doesn't realize that it is pushing a bad deployment. It marches along breaking one server after another.

Fortunately, in practice Amazon teams implement multiple mitigating systems to prevent this scenario from taking out their whole fleet. One such mitigation is to configure alarms that trigger whenever the overall fleet size is too small or running at high load, or when there is high latency or error rate. If any of these alarms trigger, the deployment system halts the deployment and rolls back.

Another type of mitigation is to use phased deployments. Instead of deploying the whole fleet in a single deployment, the service can be configured to deploy a subset, perhaps an Availability Zone, before pausing and running a full suite of integration tests against that zone. This deployment-per-Availability Zone alignment is convenient because services are already designed to be able to keep operating if there are problems with a single Availability Zone.

And of course before deploying to production, Amazon teams push those changes through test environments and run automated integration tests that would catch this type of failure. However, subtle and unavoidable differences between production and test environments may exist, so it is important to combine many layers of deployment safety to catch all kinds of problems before causing impact in production. While health checks are important to protect services against bad deployments, we make sure to not stop there. We think about the "belt and suspenders" approaches that serve as backstops to protect fleets from these and other mistakes.

Asynchronous processors

Another pattern of failure is around asynchronous message processing, such as a service that gets its work by polling an SQS Queue or Amazon Kinesis Stream. Unlike in systems that take requests from load balancers, there isn't anything automatically performing health checks to remove servers from service.

When services don't have deep enough health checks, individual queue worker servers can have failures like disks filling up or running out of file descriptors. This issue won't stop the server from

pulling work off the queue, but it will stop the server from being able to successfully process messages. This issue has resulted in delayed message processing, where the bad server pulls off work from the queue quickly and fails to deal with it.

In these kinds of situations, there are often several compensating factors to help contain the impact. For example, if a server fails to process the message that it pulls off SQS, then SQS redelivers that message to another server after a configured message visibility timeout. End-to-end latency increases, but messages are not dropped. Another compensating factor is an alarm that goes off when there are too many errors processing messages, alerting an operator to investigate.

Disks filling up

Another class of failures we see is when disks on servers fill up, causing both processing and logging to fail. This failure leads to a gap in monitoring visibility, since the server might not be able to report its failures to the monitoring system.

Again, several mitigating controls keep services from “flying blind” and mitigate impact quickly. Systems fronted by a proxy such as an Application Load Balancer or API Gateway will have error rate and latency metrics produced by that proxy. In this case, alarms fire even if the server isn’t reporting them. For queue-based systems, services like Amazon Simple Queue Service (Amazon SQS) report metrics that indicate that processing is delayed for some messages.

The thing that these solutions have in common is that there are multiple layers of monitoring. The server itself reports errors, but so does an external system. The same principle is important with health checks. An external system can test the health of a given system more accurately than it can test itself. This is why with AWS Auto Scaling, teams configure a load balancer to do external ping health checks.

Teams also write their own custom health check system to periodically ask each server if it is healthy and report to AWS Auto Scaling when a server is unhealthy. One common implementation of this system involves a Lambda function that runs every minute, testing the health of every server. These health checks can even save their state between each run in something like DynamoDB so that they don’t inadvertently mark too many servers as unhealthy at once.

Zombies

Another pattern of problems includes zombie servers. Servers can become disconnected from the network for periods of time but remain running, or they can power off for extended periods and later be rebooted.

When zombie servers come back to life they can be significantly out of sync with the rest of the fleet, which can cause serious problems. For example, if a zombie server is running a much older, incompatible software version, it can cause failures when it tries to interact with a database with different schema or it can use the wrong configuration.

To deal with zombies, systems often reply to health checks with their currently running software version. Then a central monitoring agent then compares the responses across the fleet to look for anything running an unexpectedly out of date version and prevents these servers from moving back into service.

In conclusion

Servers, and the software that runs on them, fail for all kinds of weird reasons. Hardware eventually physically breaks. As software developers, we eventually write some bug like the one I describe above that puts the software into a broken state. Multiple layers of checks, from lightweight liveness checks to passive monitoring of per-server metrics, are needed to catch all types of unexpected failure modes.

When these failures happen, it is important to detect them and take the affected servers out of service quickly. However, as with any fleet automation, we add rate-limiting, thresholding, and circuit breakers that turn off automation and involve humans in situations of uncertainty or in extreme situations. Failing open and building centralized actors are strategies for reaping the benefits of deep health checking with the safety of rate-limited automation.