Timeouts, retries, and backoff with jitter

Marc Brooker

Timeouts, retries, and backoff with jitter

Failures Happen

Whenever one service or system calls another, failures can happen. These failures can come from a variety of factors. They include servers, networks, load balancers, software, operating systems, or even mistakes from system operators. We design our systems to reduce the probability of failure, but impossible to build systems that never fail. So in Amazon, we design our systems to tolerate and reduce the probability of failure, and avoid magnifying a small percentage of failures into a complete outage. To build resilient systems, we employ three essential tools: timeouts, retries, and backoff.

Many kinds of failures become apparent as requests taking longer than usual, and potentially never completing. When a client is waiting longer than usual for a request to complete, it also holds on to the resources it was using for that request for a longer time. When a number of requests hold on to resources for a long time, the server can run out of those resources. These resources can include memory, threads, connections, ephemeral ports, or anything else that is limited. To avoid this situation, clients set *timeouts*. Timeouts are the maximum amount of time that a client waits for a request to complete.

Often, trying the same request again causes the request to succeed. This happens because the types of systems that we build don't often fail as a single unit. Rather, they suffer partial or transient failures. A partial failure is when a percentage of requests succeed. A transient failure is when a request fails for a short period of time. *Retries* allow clients to survive these random partial failures and short-lived transient failures by sending the same request again.

It's not always safe to retry. A retry can increase the load on the system being called, if the system is already failing because it’s approaching an overload. To avoid this problem, we implement our clients to use *backoff*. This increases the time between subsequent retries, which keeps the load on the backend even. The other problem with retries is that some remote calls have side effects. A timeout or failure doesn't necessarily mean that side effects haven't happened. If doing the side effects multiple times is undesirable, a best practice is designing APIs to be *idempotent*, meaning they can be safely retried.

Finally, traffic doesn't arrive into Amazon services at a constant rate. Instead, the arrival rate of requests frequently has large bursts. These bursts can be caused by client behavior, failure recovery, and even by something simple as a periodic cron job. If errors are caused by load, retries can be ineffective if all clients retry at the same time. To avoid this problem, we employ *jitter*. This is a random amount of time before making or retrying a request to help prevent large bursts by spreading out the arrival rate.

Each of these solutions is discussed in the sections that follow.

Timeouts

A best practice in Amazon is to set a timeout on any remote call, and generally on any call across processes even on the same box. This includes both a connection timeout and a request timeout. Many standard clients offer robust built-in timeout capabilities.

Typically, the most difficult problem is choosing a timeout value to set. Setting a timeout too high reduces its usefulness, because resources are still consumed while the client waits for the timeout. Setting the timeout too low has two risks:

- Increased traffic on the backend and increased latency because too many requests are retried.
- Increased small backend latency leading to a complete outage, because all requests start being retried.

A good practice for choosing a timeout for calls within an AWS Region is to start with the latency metrics of the downstream service. So at Amazon, when we make one service call another service, we choose an acceptable rate of false timeouts (such as 0.1%). Then, we look at the corresponding latency percentile on the downstream service (p99.9 in this example). This approach works well in most cases, but there are a few pitfalls, described as follows:

- This approach doesn't work in cases where clients have substantial network latency, such as over the internet. In these cases, we factor in reasonable worst-case network latency, keeping in mind that clients could span the globe.

- This approach also doesn’t work with services that have tight latency bounds, where p99.9 is close to p50. In these cases, adding some padding helps us avoid small latency increases that cause high numbers of timeouts.

- We’ve encountered a common pitfall when implementing timeouts. Linux's SO_RCVTIMEO is powerful, but has some disadvantages that make it unsuitable as an end-to-end socket timeout. Some languages, such as Java, expose this control directly. Other languages, such as Go, provide more robust timeout mechanisms.

- There are also implementations where the timeout doesn't cover all remote calls, like DNS or TLS handshakes. In general, we prefer to use the timeouts built into well-tested clients. If we implement our own timeouts, we pay careful attention to the exact meaning of the timeout socket options, and what work is being done.

In one system that I worked on at Amazon, we saw a small number of timeouts talking to a dependency immediately following deployments. The timeout was set very low, to around 20 milliseconds. Outside of deployments, even with this low timeout value, we did not see timeouts happening regularly. Digging in, I found that the timer included establishing a new secure connection, which was reused on subsequent requests. Because connection establishment took longer than 20 milliseconds, we saw a small number of requests time out when a new server went into service after deployments. In some cases, the requests retried and succeeded. We initially worked around this problem by increasing the timeout value in case a connection was established. Later, we improved the system by establishing these connections when a process started up, but before receiving traffic. This got us around the timeout issue altogether.

Retries and backoff

Retries are “selfish.” In other words, when a client retries, it spends more of the server's time to get a higher chance of success. Where failures are rare or transient, that's not a problem. This is because the overall number of retried requests is small, and the tradeoff of increasing apparent availability works well. When failures are caused by overload, retries that increase load can make matters significantly worse. They can even delay recovery by keeping the load high long after the original issue is resolved. Retries are similar to a powerful medicine -- useful in the right dose, but can cause significant damage when used too much. Unfortunately, in distributed systems there's almost no way to coordinate between all of the clients to achieve the right number of retries.

The preferred solution that we use in Amazon is a backoff. Instead of retrying immediately and aggressively, the client waits some amount of time between tries. The most common pattern is an *exponential backoff*, where the wait time is increased exponentially after every attempt. Exponential backoff can lead to very long backoff times, because exponential functions grow quickly. To avoid retrying for too long, implementations typically cap their backoff to a maximum value. This is called, predictably, *capped exponential backoff*. However, this introduces another problem. Now all of the clients are retrying constantly at the capped rate. In almost all cases, our solution is to limit the number of times that the client retries, and handle the resulting failure earlier in the service-oriented architecture. In most cases, the client is going to give up on the call anyway, because it has its own timeouts.

There are other problems with retries, described as follows:

- Distributed systems often have multiple layers. Consider a system where the customer's call causes a five-deep stack of service calls. It ends with a query to a database, and three retries at each layer. What happens when the database starts failing queries under load? If each layer retries independently, the load on the database will increase 243x, making it unlikely to ever recover. This is because the retries at each layer multiply -- first three tries, then nine tries, and so on. On the contrary, retrying at the highest layer of the stack may waste work from previous calls, which reduces efficiency. In general, for low-cost control-plane and data-plane operations, our best practice is to retry at a single point in the stack.

- Load. Even with a single layer of retries, traffic still significantly increases when errors start. *Circuit breakers*, where calls to a downstream service are stopped entirely when an error threshold is exceeded, are widely promoted to solve this problem. Unfortunately, circuit breakers introduce modal behavior into systems that can be difficult to test, and can introduce significant addition time to recovery. We have found that we can mitigate this risk by limiting retries locally using a token bucket. This allows all calls to retry as long as there are tokens, and then retry at a fixed rate when the tokens are exhausted. AWS added this behavior to the AWS SDK in 2016. So customers using the SDK have this throttling behavior built in.

- Deciding when to retry. In general, our view is that APIs with side effects aren't safe to retry unless they provide idempotency. This guarantees that the side effects happen only once no matter how often you retry. Read-only APIs are typically idempotent, while resource creation APIs may not be. Some APIs, like the Amazon Elastic Compute Cloud (Amazon EC2) RunInstances API, provide explicit token-based mechanisms to provide idempotency and make them safe to retry. Good API design, and care when implementing clients, is needed to prevent duplicate side-effects.

- Knowing which failures are worth retrying. HTTP provides a clear distinction between *client* and *server* errors. It indicates that client errors should not be retried with the same request because they aren't going to succeed later, while server errors may succeed on subsequent tries. Unfortunately, eventual consistency in systems significantly blurs this line. A client error one moment may change into a success the next moment as state propagates.

Despite these risks and challenges, retries are a powerful mechanism for providing high availability in the face of transient and random errors. Judgment is required to find the right trade-off for each service. In our experience, a good place to start is to remember that retries are selfish. Retries are a way for clients to assert the importance of their request and demand that the service spend more of its resources to handle it. If a client is too selfish it can create wide-ranging problems.

Jitter

When failures are caused by overload or contention, backing off often doesn't help as much as it seems like it should. This is because of correlation. If all the failed calls back off to the same time, they cause contention or overload again when they are retried. Our solution is jitter. Jitter adds some amount of randomness to the backoff to spread the retries around in time. For more information about how much jitter to add and the best ways to add it, see Exponential Backoff and Jitter.

Jitter isn't only for retries. Operational experience has taught us that the traffic to our services, including both control-planes and data-planes, tends to spike a lot. These spikes of traffic can be very short, and are often hidden by aggregated metrics. When building systems, we consider adding some jitter to all timers, periodic jobs, and other delayed work. This helps spread out spikes of work, and makes it easier for downstream services to scale for a workload.

When adding jitter to scheduled work, we do not select the jitter on each host randomly. Instead, we use a consistent method that produces the same number every time on the same host. This way, if there is a service being overloaded, or a race condition, it happens the same way in a pattern. We humans are good at identifying patterns, and we're more likely to determine the root cause. Using a random method ensures that if a resource is being overwhelmed, it only happens - well, at random. This makes troubleshooting much more difficult.

On systems that I have worked on, like Amazon Elastic Block Store (Amazon EBS) and AWS Lambda, we found that clients frequently send requests on a regular interval, like once per minute. However, when a client has multiple servers behaving the same way, they can line up and trigger their requests at the same time. This can be the first few seconds of a minute, or the first few seconds after midnight for daily jobs. By paying attention to per-second load, and working with clients to jitter their periodic workloads, we accomplished the same amount of work with less server capacity.

We have less control over spikes in customer traffic. However, even for customer-triggered tasks, it's a good idea to add jitter where it doesn't impact the customer experience.

Conclusion

In distributed systems, transient failures or latency in remote interactions are inevitable. Timeouts keep systems from hanging unreasonably long, retries can mask those failures, and backoff and jitter can improve utilization and reduce congestion on systems.

At Amazon, we have learned that it is important to be cautious about retries. Retries can amplify the load on a dependent system. If calls to a system are timing out, and that system is overloaded, retries can make the overload worse instead of better. We avoid this amplification by retrying only when we observe that the dependency is healthy. We stop retrying when the retries are not helping to improve availability.