Chapter 16

Downstream resiliency

In this chapter, we will explore patterns that shield a service against failures in its downstream dependencies.

16.1 Timeout

When you make a network call, you can configure a timeout to fail the request if there is no response within a certain amount of time. If you make the call without setting a timeout, you tell your code that you are 100% confident that the call will succeed. Would you really take that bet?

Unfortunately, some network APIs don't have a way to set a timeout in the first place. When the default timeout is infinity, it's all too easy for a client to shoot itself in the foot. As mentioned earlier, network calls that don't return lead to resource leaks at best. Timeouts limit and isolate failures, stopping them from cascading to the rest of the system. And they are useful not just for network calls, but also for requesting a resource from a pool and for synchronization primitives like mutexes.

To drive the point home on the importance of setting timeouts, let's take a look at some concrete examples. JavaScript's *XMLHttpRequest* is *the* web API to retrieve data from a server asynchronously.

Its default timeout is zero[1], which means there is no timeout:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/api', true);
// No timeout by default!
xhr.timeout = 10000;
xhr.onload = function () {
 // Request finished
};
xhr.ontimeout = function (e) {
 // Request timed out
};
xhr.send(null);
```

Client-side timeouts are as crucial as server-side ones. There is a maximum number of sockets your browser[2] can open for a particular host. If you make network requests that never return, you are going to exhaust the socket pool. When the pool is exhausted, you are no longer able to connect to the host.

The *fetch* web API is a modern replacement for *XMLHttpRequest* that uses Promises. When the fetch API was initially introduced, there was no way to set a timeout at all[3]. Browsers have recently added experimental support for the Abort API[4] to support timeouts.

```
const controller = new AbortController();
const signal = controller.signal;
const fetchPromise = fetch(url, {signal});
// No timeout by default!
setTimeout(() => controller.abort(), 10000);
fetchPromise.then(response => {
 // Request finished
})
```

[1] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/timeout

[2] https://hpbn.co/primer-on-browser-networking/#connection-management-and-optimization

[3] https://github.com/whatwg/fetch/issues/951

[4] https://developer.mozilla.org/en-US/docs/Web/API/AbortController

Things aren't much rosier for Python. The popular *requests* library uses a default timeout of infinity[5]:

```
# No timeout by default!
response = requests.get('https://github.com/', timeout=10)
```

Go's *HTTP package* doesn't use timeouts[6] by default, either:

```
var client = &http.Client{
  // No timeout by default!
  Timeout: time.Second * 10,
}
response, _ := client .Get(url)
```

Modern HTTP clients for Java and .NET do a much better job and usually come with default timeouts. For example, .NET Core *HttpClient* has a default timeout of 100 seconds[7]. It's lax but better than not setting a timeout at all.

As a rule of thumb, always set timeouts when making network calls, and be wary of third-party libraries that do network calls or use internal resource pools but don't expose settings for timeouts. And if you build libraries, always set reasonable default timeouts and make them configurable for your clients.

Ideally, you should set your timeouts based on the desired false timeout rate[8]. Say you want to have about 0.1% false timeouts; to achieve that, you should set the timeout to the 99.9th percentile of the remote call's response time, which you can measure empirically.

You also want to have good monitoring in place to measure the entire lifecycle of your network calls, like the duration of the call, the status code received, and if a timeout was triggered. We will talk about monitoring later in the book, but the point I want to make here is that you have to measure what happens at the integration

[5] https://requests.readthedocs.io/en/master/user/quickstart/#timeouts

[6] https://github.com/golang/go/issues/24138

[7] https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient.timeout?view=netcore-3.1#remarks

[8] https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/

points of your systems, or you won't be able to debug production issues when they show up.

Ideally, you want to encapsulate a remote call within a library that sets timeouts and monitors it for you so that you don't have to remember to do this every time you make a network call. No matter which language you use, there is likely a library out there that implements some of the resiliency and transient fault-handling patterns introduced in this chapter, which you can use to encapsulate your system's network calls.

Using a language-specific library is not the only way to wrap your network calls; you can also leverage a reverse proxy co-located on the same machine which intercepts all the remote calls that your process makes[9]. The proxy enforces timeouts and also monitors the calls, relinquishing your process from the responsibility to do so.

16.2 Retry

You know by now that a client should configure a timeout when making a network request. But, what should it do when the request fails, or the timeout fires? The client has two options at that point: it can either fail fast or retry the request at a later time.

If the failure or timeout was caused by a short-lived connectivity issue, then retrying after some *backoff time* has a high probability of succeeding. However, if the downstream service is overwhelmed, retrying immediately will only make matters worse. This is why retrying needs to be slowed down with increasingly longer delays between the individual retries until either a maximum number of retries is reached or a certain amount of time has passed since the initial request.

[9] We talked about this in section 14.1.3 when discussing the sidecar pattern and the service mesh.

16.2.1 Exponential backoff

To set the delay between retries, you can use a *capped exponential function*, where the delay is derived by multiplying the initial backoff duration by a constant after each attempt, up to some maximum value (the cap):

$$\text{delay} = min(\text{cap}, \text{initial-backoff} \cdot 2^{\text{attempt}})$$

For example, if the cap is set to 8 seconds, and the initial backoff duration is 2 seconds, then the first retry delay is 2 seconds, the second is 4 seconds, the third is 8 seconds, and any further delay will be capped to 8 seconds.

Although exponential backoff does reduce the pressure on the downstream dependency, there is still a problem. When the downstream service is temporarily degraded, it's likely that multiple clients see their requests failing around the same time. This causes the clients to retry simultaneously, hitting the downstream service with load spikes that can further degrade it, as shown in Figure 16.1.

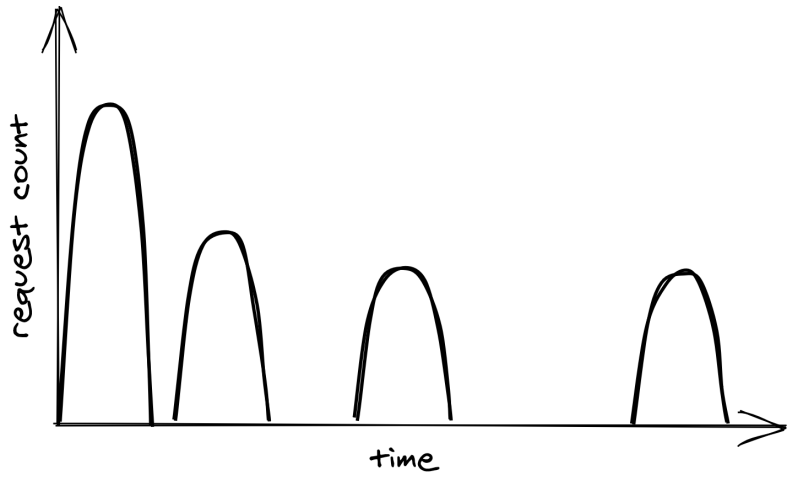

Figure 16.1: Retry storm

To avoid this herding behavior, you can introduce random jitter[10] in the delay calculation. With it, the retries spread out over time, smoothing out the load to the downstream service:

$$\text{delay} = random(0, min(\text{cap}, \text{initial-backoff} \cdot 2^{\text{attempt}}))$$

Actively waiting and retrying failed network requests isn't the only way to implement retries. In batch applications that don't have strict real-time requirements, a process can park failed requests into a *retry queue*. The same process, or possibly another, reads from the same queue later and retries the requests.

Just because a network call can be retried doesn't mean it should be. If the error is not short-lived, for example, because the process is not authorized to access the remote endpoint, then it makes no sense to retry the request since it will fail again. In this case, the process should fail fast and cancel the call right away.

You should also not retry a network call that isn't idempotent, and whose side effects can affect your application's correctness. Suppose a process is making a call to a payment provider service, and the call times out; should it retry or not? The operation might have succeeded and retrying would charge the account twice, unless the request is idempotent.

16.2.2 Retry amplification

Suppose that handling a request from a client requires it to go through a chain of dependencies. The client makes a call to service A, which to handle the request talks to service B, which in turn talks to service C.

If the intermediate request from service B to service C fails, should B retry the request or not? Well, if B does retry it, A will perceive a longer execution time for its request, which in turn makes it more likely to hit A's timeout. If that happens, A retries its request again, making it more likely for the client to hit its timeout and retry.

[10] https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/

Having retries at multiple levels of the dependency chain can amplify the number of retries; the deeper a service is in the chain, the higher the load it will be exposed to due to the amplification (see Figure 16.2).

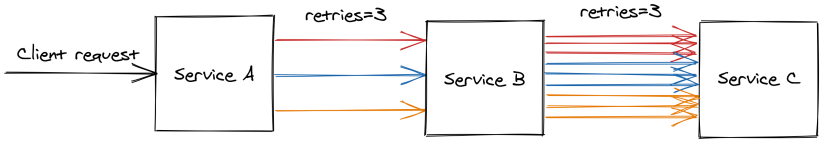

Figure 16.2: Retry amplification in action

And if the pressure gets bad enough, this behavior can easily bring down the whole system. That's why when you have long dependency chains, you should only retry at a single level of the chain, and fail fast in all the other ones.

16.3 Circuit breaker

Suppose your service uses timeouts to detect communication failures with a downstream dependency, and retries to mitigate transient failures. If the failures aren't transient and the downstream dependency keeps being unresponsive, what should it do then? If the service keeps retrying failed requests, it will necessarily become slower for its clients. In turn, this slowness can propagate to the rest of the system and cause cascading failures.

To deal with non-transient failures, we need a mechanism that detects long-term degradations of downstream dependencies and stops new requests from being sent downstream in the first place. After all, the fastest network call is the one you don't have to make. This mechanism is also called a circuit breaker, inspired by the same functionality implemented in electrical circuits.

A circuit breaker's goal is to allow a sub-system to fail without bringing down the whole system with it. To protect the system, calls to the failing sub-system are temporarily blocked. Later, when the sub-system recovers and failures stop, the circuit breaker

allows calls to go through again.

Unlike retries, circuit breakers prevent network calls entirely, which makes the pattern particularly useful for long-term degradations. In other words, retries are helpful when the expectation is that the next call will succeed, while circuit breakers are helpful when the expectation is that the next call will fail.

16.3.1 State machine

The circuit breaker is implemented as a state machine that can be in one of three states: open, closed and half-open (see Figure 16.3).

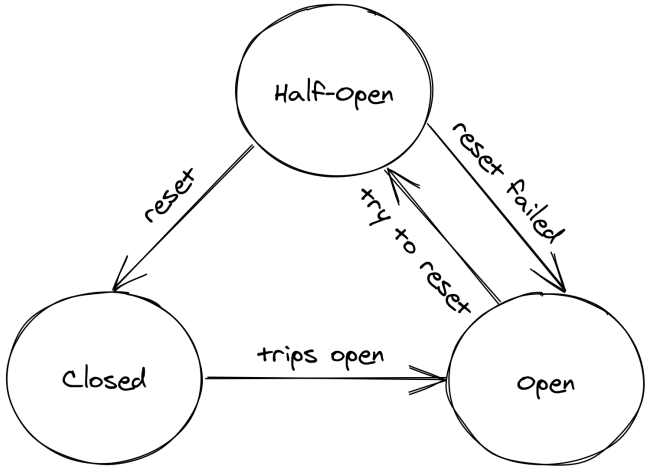

Figure 16.3: Circuit breaker state machine

In the closed state, the circuit breaker is merely acting as a pass-through for network calls. In this state, the circuit breaker tracks the number of failures, like errors and timeouts. If the number goes over a certain threshold within a predefined time-interval, the circuit breaker trips and opens the circuit.

When the circuit is open, network calls aren't attempted and fail immediately. As an open circuit breaker can have business implications, you need to think carefully what should happen when a downstream dependency is down. If the down-stream dependency is non-critical, you want your service to degrade gracefully,

rather than to stop entirely.

Think of an airplane that loses one of its non-critical sub-systems in flight; it shouldn't crash, but rather gracefully degrade to a state where the plane can still fly and land. Another example is Amazon's front page; if the recommendation service is not available, the page should render without recommendations. It's a better outcome than to fail the rendering of the whole page entirely.

After some time has passed, the circuit breaker decides to give the downstream dependency another chance, and transitions to the half-open state. In the half-open state, the next call is allowed to pass-through to the downstream service. If the call succeeds, the circuit breaker transitions to the closed state; if the call fails instead, it transitions back to the open state.

That's really all there is to understand how a circuit breaker works, but the devil is in the details. How many failures are enough to consider a downstream dependency down? How long should the circuit breaker wait to transition from the open to the half-open state? It really depends on your specific case; only by using data about past failures can you make an informed decision.

Chapter 17

Upstream resiliency

So far, we have discussed patterns that protect against downstream failures, like failures to reach an external dependency. In this chapter, we will shift gears and discuss mechanisms to protect against upstream pressure.

17.1 Load shedding

A server has very little control over how many requests it receives at any given time, which can deeply impact its performance.

The operating system has a connection queue per port with a limited capacity that, when reached, causes new connection attempts to be rejected immediately. But typically, under extreme load, the server crawls to a halt before that limit is reached as it starves out of resources like memory, threads, sockets, or files. This causes the response time to increase to the point the server becomes unavailable to the outside world.

When a server operates at capacity, there is no good reason for it to keep accepting new requests since that will only end up degrading it. In that case, the process should start rejecting excess requests[1]

[1] https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-