Chapter 15 describes the causes of the most common failures in distributed systems: single points of failure, unreliable networks, slow processes, and unexpected load.

Chapter 16 dives into resiliency patterns that help shield a service against failures in downstream dependencies, like timeouts, retries, and circuit breakers.

Chapter 17 discusses resiliency patterns that help protect a service against upstream pressure, like load shedding, load leveling, and rate-limiting.

Chapter 15

Common failure causes

In order to protect your systems against failures, you first need to have an idea of what can go wrong. The most common failures you will encounter are caused by single points of failure, the network being unreliable, slow processes, and unexpected load. Let's take a closer look at these.

15.1 Single point of failure

A single point of failure is the most glaring cause of failure in a distributed system; if it were to fail, that one component would bring down the entire system with it. In practice, systems can have multiple single points of failure.

A service that starts up by needing to read a configuration from a non-replicated database is an example of a single point of failure; if the database isn't reachable, the service won't be able to (re)start. A more subtle example is a service that exposes an HTTP API on top of TLS using a certificate that needs to be manually renewed. If the certificate isn't renewed by the time it expires, then all clients trying to connect to it wouldn't be able to open a connection with the service.

Single points of failure should be identified when the system is

architected before they can cause any harm. The best way to detect them is to examine every component of the system and ask what would happen if that component were to fail. Some single points of failure can be architected away, e.g., by introducing redundancy, while others can't. In that case, the only option left is to minimize the blast radius.

15.2 Unreliable network

When a client makes a remote network call, it sends a request to a server and expects to receive a response from it a while later. In the best case, the client receives a response shortly after sending the request. But what if the client waits and waits and still doesn't get a response? In that case, the client doesn't know whether a response will eventually arrive or not. At that point it has only two options: it can either continue to wait, or fail the request with an exception or error.

As discussed when the concept of failure detection was introduced in chapter 7, there are several reasons why the client hasn't received a response so far:

- the server is slow;
- the client's request has been dropped by a network switch, router or proxy;
- the server has crashed while processing the request;
- or the server's response has been dropped by a network switch, router or proxy.

Slow network calls are the silent killers[1] of distributed systems. Because the client doesn't know whether the response is on its way or not, it can spend a long time waiting before giving up, if it gives up at all. The wait can in turn cause degradations that are extremely hard to debug. In chapter 16 we will explore ways to protect clients from the unreliability of the network.

[1] https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

15.3 Slow processes

From an observer's point of view, a very slow process is not very different from one that isn't running at all — neither can perform useful work. Resource leaks are one of the most common causes of slow processes. Whenever you use resources, especially when they have been leased from a pool, there is a potential for leaks.

Memory is the most well-known source of leaks. A memory leak causes a steady increase in memory consumption over time. Runtimes with garbage collection don't help much either; if a reference to an object that is no longer needed is kept somewhere, the object won't be deleted by the garbage collector.

A memory leak keeps consuming memory until there is no more of it, at which point the operating system starts swapping memory pages to disk constantly, while the garbage collector kicks in more frequently trying its best to release any shred of memory. The constant paging and the garbage collector eating up CPU cycles make the process slower. Eventually, when there is no more physical memory, and there is no more space in the swap file, the process won't be able to allocate more memory, and most operations will fail.

Memory is just one of the many resources that can leak. For example, if you are using a thread pool, you can lose a thread when it blocks on a synchronous call that never returns. If a thread makes a synchronous blocking HTTP call without setting a timeout, and the call never returns, the thread won't be returned to the pool. Since the pool has a fixed size and keeps losing threads, it will eventually run out of threads.

You might think that making *asynchronous* calls, rather than synchronous ones, would help in the previous case. However, modern HTTP clients use socket pools to avoid recreating TCP connections and pay a hefty performance fee as discussed in chapter 2. If a request is made without a timeout, the connection is never returned to the pool. As the pool has a limited size, eventually there won't be any connections left.

On top of that, the code you write isn't the only one accessing memory, threads, and sockets. The libraries your application depends on access the same resources, and they can do all kinds of shady things. Without digging into their implementation, assuming it's open in the first place, you can't be sure whether they can wreak havoc or not.

15.4 Unexpected load

Every system has a limit to how much load it can withstand without scaling. Depending on how the load increases, you are bound to hit that brick wall sooner or later. One thing is an organic increase in load that gives you the time to scale out your service accordingly, but another is a sudden and unexpected spike.

For example, consider the number of requests received by a service in a period of time. The rate and the type of incoming requests can change over time, and sometimes suddenly, for a variety of reasons:

- The requests might have a seasonality — depending on the hour of the day, the service is going to get hit by users in different countries.
- Some requests are much more expensive than others and abuse the system in ways you might have not anticipated, like scrapers slurping in data from your site at super-human speed.
- Some requests are malicious, like DDoS attacks that try to saturate your service's bandwidth, denying access to the service for legitimate users.

To withstand unexpected load, you need to prepare beforehand. The patterns in chapter 17 will teach you some techniques on how to do just that[2].

[2]These techniques might look simple but are very effective. During the COVID-19 outbreak, I have witnessed many of the systems I was responsible for at the time doubling traffic nearly overnight without causing any incidents.

15.5 Cascading failures

You would think that if your system has hundreds of processes, it shouldn't make much difference if a small percentage are slow or unreachable. The thing about failures is that they tend to spread like cancer, propagating from one process to the other until the whole system crumbles to its knees. This effect is also referred to as a *cascading failure*, which occurs when a portion of an overall system fails, increasing the probability that other portions fail.

For example, suppose there are multiple clients querying two database replicas A and B, which are behind a load balancer. Each replica is handling about 50 transactions per second (see Figure 15.1).

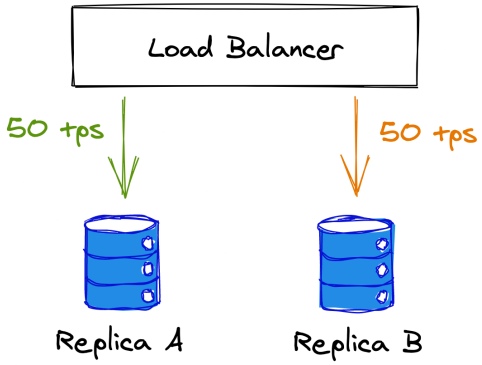

Figure 15.1: Two replicas behind an LB; each is handling half the load.

Suddenly, replica B becomes unavailable because of a network fault. The load balancer detects that B is unavailable and removes it from its pool. Because of that, replica A has to pick up the slack for replica B, doubling the load it was previously under (see Figure 15.2).

As replica A starts to struggle to keep up with the incoming requests, the clients experience more failures and timeouts. In turn, they retry the same failing requests several times, adding insult to injury.

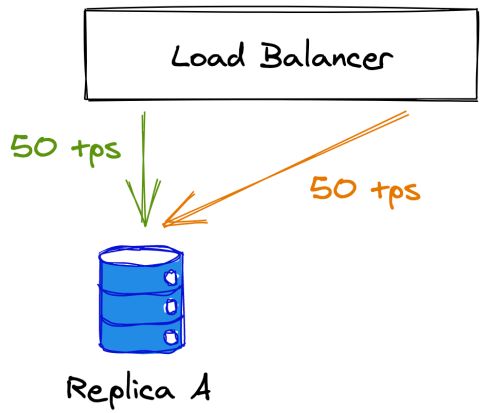

Figure 15.2: When replica B becomes unavailable, A will be hit with more load, which can strain it beyond its capacity.

Eventually, replica A is under so much load that it can no longer serve requests promptly and becomes unavailable, causing replica A to be removed from the load balancer's pool. In the meantime, replica B becomes available again and the load balancer puts it back in the pool, at which point it's flooded with requests that kill the replica instantaneously. This feedback loop of doom can repeat several times.

Cascading failures are very hard to get under control once they have started. The best way to mitigate one is to not have it in the first place. The patterns introduced in the next chapters will help you stop the cracks in the system from spreading.

15.6 Risk management

As we have just seen, a distributed system needs to embrace that failures will happen and needs to be prepared for it. Just because a failure has a chance of happening doesn't always mean you have necessarily to do something about it. The day has only so many hours, and you will need to make tough decisions about where to spend your engineering time.

Given a specific failure, you have to consider its probability of

happening and the impact it causes to your system if it does happen. By multiplying the two factors together, you get a risk score[3], which you can then use to decide which failures to prioritize and act upon (see Figure 15.3). A failure that is very likely to happen, and has an extensive impact, should be dealt with swiftly; on the other hand, a failure with a low likelihood and low impact can wait.

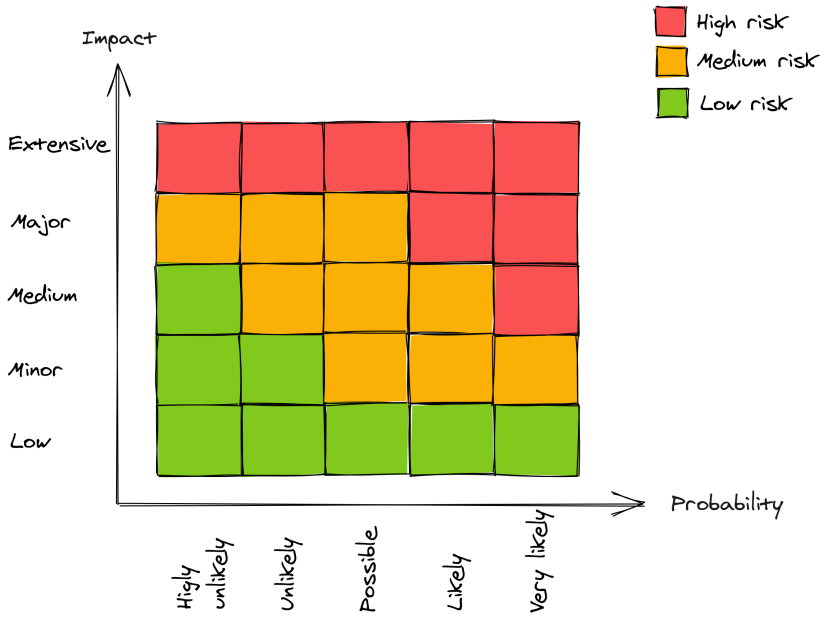

Figure 15.3: Risk matrix

To address a failure, you can either find a way to reduce the probability of it happening, or reduce its impact.

[3] https://en.wikipedia.org/wiki/Risk_matrix