How distributed systems fail

December 05, 2020

At scale, any failure that can happen will eventually happen. Hardware failures, software crashes, memory leaks - you name it. The more components you have, the more failures you will experience.

This nasty behavior is caused by *cruel math* - given an operation that has a certain probability of failing, as the total number of operations performed increases, so does the total number of failures. In other words, as you scale out your application to handle more load, the more failures it will experience.

To protect your application against failures, you first need to know what can go wrong. Assuming you are using a cloud provider and not maintaning your own datacenter, the most common failures you will encounter are caused by single points of failure, the network being unreliable, slow processes, and unexpected load.

Single Point of Failure

A single point of failure is the most glaring cause of failure in a distributed system - it's that one component that when it fails brings down the entire system with it. In practice, distributed systems can have multiple single points of failure.

A service that to start up needs to read its 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 start.

A more subtle example is a service that exposes a HTTP API on top of TLS and uses a certificate that needs to be manually renewed. If the certificate isn't renewed by the

time it expires, then most 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.

Unreliable Network

When a client make 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 an error.

Slow network calls are the silent killers 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.

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.

Memory leaks are arguably the most well-known source of leaks. A memory leak

manifests itself with a steady increase in memory consumption over time. Run-times with garbage collection don't help much either - if a reference to an object that isn't 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 the disk constantly, all the 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, and 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, the pool will eventually run out of threads.

You might think that making *asynchronous* calls, rather than a synchronous ones, would mitigate the problem in the previous case. But, modern HTTP clients use socket pools to avoid recreating TCP connections and pay a hefty performance fee. 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 to communicate with the host.

On top of all 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.

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. But one thing is an organic increase in load, which gives you the time to scale your service out accordingly, and 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:

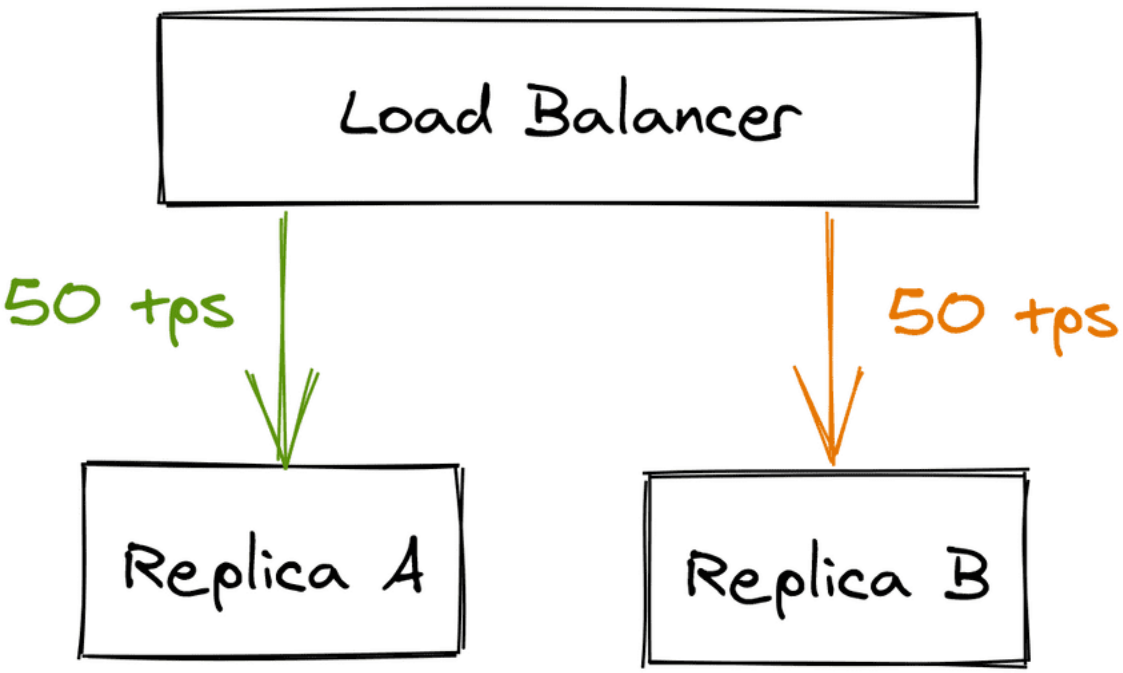
- The requests might have a seasonality - depending on the hour of the day the service is going to get hit by users in different countries.
- Some requests are much more expensive than others and abuse the system in ways you didn't really anticipate for, like scrapers slurping in data from your site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

Cascading Failures

You would think that if your system has hundreds of processes, it shouldn't make much of a difference if a small percentage are slow or unreachable. The thing about faults 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.

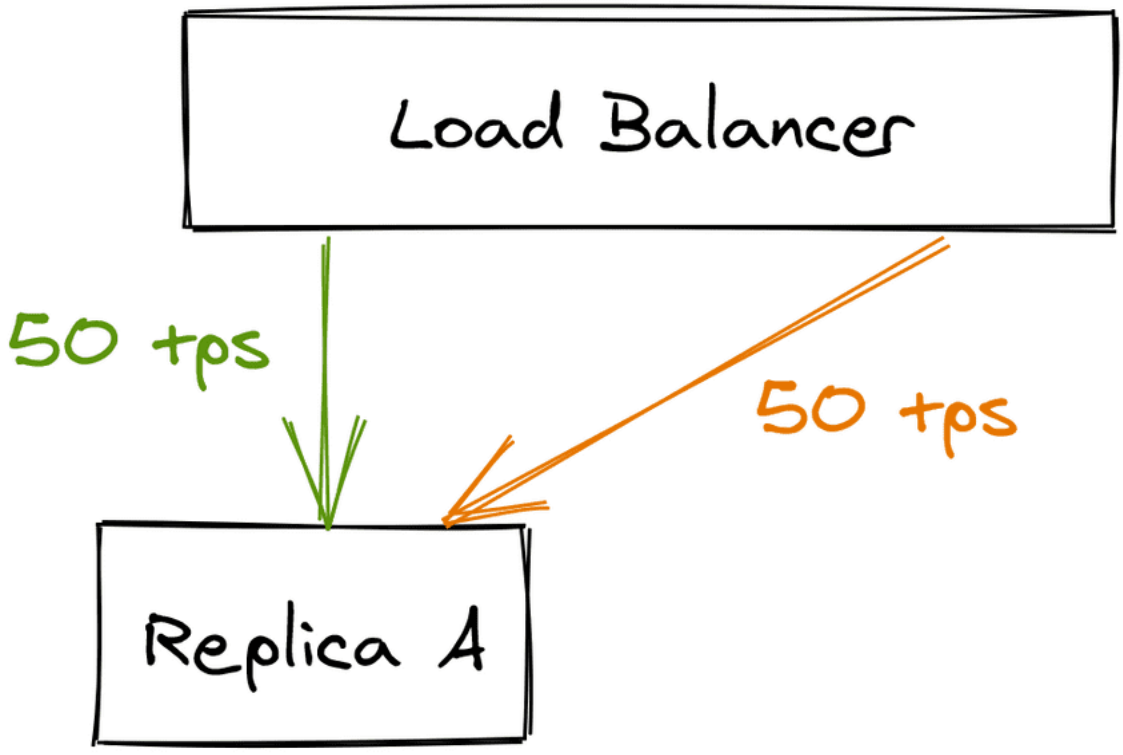
For example, suppose there are multiple clients querying two database replicas A and B, which are behind a load balancer. Each replica is handling about 50 transactions per second.

{width: 75%}



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.

{width: 75%}



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.

Eventually, replica A is under so much load that it can no longer serve requests promptly, and becomes for all intent and purposes 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 time.

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 by stopping the cracks in your services to propagate to others.

Defense Mechanisms

There is a variety of best practices you can use to mitigate failures, like circuit breakers, load shedding, rate-limiting and bulkheads. I plan to blog about those in the future, but in the meantime Google is your friend. Also, I have an entire chapter dedicated to resiliency patterns in my book about distributed systems.

Written by Roberto Vitillo