Retries – An interactive study of common retry methods – Encore Blog

About the author

Requests over the network can fail. This is something we cannot avoid, and to write robust software we need to handle these failures or else they may be presented to users as errors. One of the most common techniques for handling a failed request is to retry it.

In this post we're going to visually explore different methods of retrying requests, demonstrating why some common approaches are dangerous and ultimately ending up at what the best practice is. At the end of this post you will have a solid understanding of what makes safe retry behaviour, and a vivid understanding of what doesn't.

We'll be focusing on when you have control over the behaviour of the client. The advice in this post applies equally to when you're making requests to your own backend services or third-party ones. We won't be covering any server-side mitigations to the problems described in this post.

Setting the stage

Let's introduce the elements involved in our visualisation. We have:

Requests can be thought of as HTTP requests. They can succeed or fail. Failed requests have spiked edges, successful requests stay smooth.

Load balancers route requests from clients to servers.

Servers accept and serve requests.

Clients send requests to servers via a load balancer. After getting a response, they will wait an amount of time before sending another request.

Here's how all of that that looks.

We have one client sending requests periodically to one server. You could imagine this is a client periodically checking the status of some background job. The request goes through a load balancer that selects which server to send the request to. Requests either succeed or fail which you can see when they're making their return journey to the client. While the client is waiting to send its next request, it shows as a circular timer.

Adjust animation speed

If the animations are too fast or too slow for you, feel free to change them now. This will affect all animations on this page.

Basic retry handling

The simplest way to handle a failure is to do nothing. In this visualisation the server is configured to fail 100% of the time, and each client will just wait to send its next request.

Not all that exciting. Requests fail and the client just waits to send another. Let's do what people tend to do when they check Sentry and notice that they're serving 503s due to a failed request to a third-party service: retry 10 times in a tight loop.

You can see now that when a request fails, it is immediately retried. No waiting. We've configured a 100% failure rate to make the retries easier to see, but if the failure rate was 5% then the odds of 2 requests failing back to back is 1 in 400. 3 requests in a row is 1 in 8000. Retries allow you to trade latency for reliability.

However, there's a subtle side-effect of behaving this way. Every time a client retries when it would have otherwise been waiting, an extra request is generated. This increases the overall load to our service.

Now we're going to add a few more clients and introduce some buttons. The buttons control the failure rate of our servers. For now, we're just going to have 0% and 100%. When you're ready, switch from 0% to 100% and see what happens to our server.

If I'm any good at tuning my simulations, you will quickly notice the server *explode*. Even after you set the failure rate back to 0% and the server has recovered, there's a chance it will keep exploding.

What the explosion represents is a server overloading and crashing. Then it restarts a few seconds later. This can happen for all sorts of reasons in the real world, from the process running out of memory to rare segfaults that only happen under stress. Typically servers will have request queues that reject requests when the server has too much work to do, but to keep things simple we're using overload to represent any potential failure mode.

Once the server has crashed once, the extra load created by the retries can make it difficult to recover. When it comes back up, it might get quickly overwhelmed and crash again. This problem gets worse as you scale. Let's add in even more clients and a few more servers to handle the new load.

What you're likely to see here is that the moment you switch from a 0% failure rate to 100%, traffic begins to ramp up as clients begin to retry. Eventually, one of the servers will crash. As soon as one server goes, the remaining two will be unable to handle the new load.

You'll notice that setting the failure rate back to 0% here will likely have no meaningful effect. You may eventually recover, but if you're doing retries in a tight loop and you get into this overloaded state it can be very hard to get back out. In practice, the quickest way to recover is to add more servers to absorb the load. Once stabilised, you can spin the extra servers back down.

Give that a try in the next visualisation. It's tuned the same way as the previous one, but this time there's an extra toggle that lets you control the number of servers. Set the failure rate up to 100%, get into an overloaded state, then set it back down to 0% and gradually add servers until you're recovered. How many extra servers do you need in order to stabilise?

Retrying with a delay

So retrying in a tight loop is problematic and we've seen why. The next thing people do is to add a delay between each retry. 10 retries with a `sleep(1000)` between them. Let's see how that fares.

You should notice the same pattern here as with no delay between retries. When you set the failure rate to 100%, the server will crash shortly after. It may take a bit longer, but it will happen. If the rate at which your clients retry is not longer than the rate at which they normally send requests, you will see an increase in overall load.

To demonstrate, let's try a `sleep(10000)` to wait 10 seconds after a failed request. This wait is about twice as long as clients usually wait before sending their next request.

This "works" insofar as the server is unlikely to get overloaded, and if it does it is able to recover with ease. But this will lead to a bad user experience in practice. Users don't like waiting, and the longer you sleep between retries, the more likely they are to refresh manually or go and do something else. Both bad outcomes.

So what's the answer?

We need a way of retrying that retries quickly in case the error is low probability, thus protecting the user experience, but recognises when things are really wrong and waits longer to prevent unrecoverable overload.

We need "**exponential backoff.**" There are lots of things you can configure when calculating exponential backoff, but if you imagine we started off waiting for 1 second and waited twice as long each retry, 10 retries would look like this:

1 second

2 seconds

4 seconds

8 seconds

16 seconds

32 seconds

1 minute and 4 seconds

2 minutes and 8 seconds

4 minutes and 16 seconds

8 minutes and 32 seconds

This would be an enormous amount of time to wait, so in practice exponential backoff is tuned to start lower than 1 second, and often has a lower multiplier. Google's Java HTTP Client Library, for example, starts at 0.5 seconds and has a multiplier of 1.5. This yields the following retry intervals:

0.5 seconds

0.75 seconds

1.125 seconds

1.687 seconds

2.53 seconds

3.795 seconds

5.692 seconds

8.538 seconds

12.807 seconds

19.210 seconds

Enough mathematics, how does this look in practice? All of the following examples use the Google HTTP library backoff defaults (0.5 second initial delay, 1.5 multiplier).

As soon as you flip over to 100% failure rate you'll notice the usual ramp up in requests, but as those requests are retried you will then notice that the backoff kicks in and things calm down. The server may crash but the clients give it space to recover. When you flip back to 0% failure rate, the server is able to return to normal service quickly.

For fun, let's also see it in action at scale. I'm going to give you some more failure rates to play with, too. Go wild.

You may have struggled to get any of the servers to crash in this example, even at a 100% failure rate. This is exponential backoff at work, helping your clients recognise trouble and getting them to give your servers space to recover.

Jitter

We've seen the power of exponential backoff at work, but there's one last thing we can do with our retries to make them truly best practice.

"Jitter" is the process of randomising how long we wait between retries to within a specific range. To follow the Google HTTP client library example, they add 50% jitter. So a retry interval can be between 50% lower and 50% higher than the calculated figure. Here's how that affects our numbers from before:

0.5 seconds, ± 0.25 seconds

0.75 seconds, ± 0.375 seconds

1.125 seconds, ± 0.5625 seconds

1.687 seconds, ± 0.8435 seconds

2.53 seconds, ± 1.265 seconds

3.795 seconds, ± 1.8975 seconds

5.692 seconds, ± 2.846 seconds

8.538 seconds, ± 4.269 seconds

12.807 seconds, ± 6.4035 seconds

19.210 seconds, ± 9.605 seconds

This jitter helps prevent clients from synchronising with each other and sending surges of requests.

Putting this in to code

So you've read this post and realised you're either not making use of retries, or you're doing them dangerously. Here's some example Go code that implements the retry strategy we've built up to, exponential backoff with jitter, that you can use in your own projects.

```
package main

import (
        "encoding/json"
        "fmt"
        "net/http"
        "time"

        "github.com/cenkalti/backoff/v4"
)
```

```
func main() {
        bo := backoff.NewExponentialBackOff()
        bo.InitialInterval = 500 * time.Millisecond
        bo.Multiplier = 1.5
        bo.RandomizationFactor = 0.5

        err := backoff.Retry(func() error {
                resp, err :=
http.Get("https://jsonplaceholder.typicode.com/todos/1")
                if err != nil {
                        return err
                }
                defer resp.Body.Close()

                var result map[string]interface{}
                if err := json.NewDecoder(resp.Body).Decode(&result); err
!= nil {
                        return err
                }

                fmt.Printf("%+v\n", result)
                return nil
        }, bo)

        if err != nil {
                fmt.Println("Request failed:", err)
        }
}
```

Wrapping Up

I hope that this post has helped visually cement how different retry behaviours work in practice, and given you a good, intuitive understanding of the failure modes. We can't always prevent failure, but we can set ourselves up to have the best chance of recovering when it does happen.

To recap what we've learned:

Retrying in a tight loop is dangerous. You risk getting into overload situations that are difficult to recover

from.

Retrying with a delay helps a little bit but is still **dangerous.**

Exponential backoff is a much safer way of retrying, balancing user experience with safety.

Jitter adds an extra layer of protection, preventing clients from sending synchronised surges of requests.

If you have questions or feedback, please reach out on Slack, via email at hello@encore.dev, or @encoredotdev on Twitter.

Sam Rose has been programming professionally for over 10 years, with a focus on the backend and SRE domains. He has worked at a wide range of companies, from large ones like Google to smaller ones like Nebula.

If you enjoyed this post, Sam has a collection of similarly visual and interactive posts on his personal site. He has written about hashing, memory allocation, and load balancing so far, with more planned.

To keep up to date with his work you can follow him on Twitter, and if you want to support what he does he also has Patreon.

Playground

As a final treat, here's the visualisation with the debug UI exposed so that you can tweak all of the parameters in whatever way you like. Enjoy 😄