Retrying Failed Requests

For any request that you want to retry automatically, we recommend that you do so using a truncated exponential backoff algorithm with jitter.

Overview

Every request you make has the possibility to fail for a variety of reasons. It is your responsibility to decide what to do when this happens. When encountering transient errors, you will typically want to retry the request. This requires some care to do correctly as incorrect retry strategies may cause many requests to be (re)sent in a short amount of time, which may eventually cause them to be throttled or otherwise permanently dropped. To mitigate this thundering herd problem, we recommend you implement retries using a truncated exponential backoff algorithm with jitter.

Truncated Exponential Backoff Algorithm With Jitter

Here is an example of algorithm that implements the recommended retry strategy:

1. Send a request.
2. Upon retryable failure, wait 1 + `jitter` seconds before retrying.
3. Upon retryable failure, wait 2 + `jitter` seconds before retrying.
4. Upon retryable failure, wait 4 + `jitter` seconds before retrying.
5. Upon retryable failure, wait 8 + `jitter` seconds before retrying.
6. And so on, with a delay of min(2^n, `max_delay`) + `jitter` seconds at each iteration.
7. After `deadline` seconds, stop retrying the request.

The `jitter` value must be picked randomly at each iteration. In this example, it could be a fractional value between 0 and 1 second.

The `max_delay` value defines the maximum time to wait between retries, excluding jitter. You should pick the highest value your use case can support. If you have no specific latency requirements, 2^6 (64) seconds should be reasonable in most situations.

The `deadline` value defines how long to keep retrying automatically until you cancel the request. You should pick a value appropriate to your use case. One way to think about it is "how long to keep trying until we need to involve a human?". For example, if you are fetching a tile to display in an interactive application, there is probably no point retrying for three minutes as the user will have given up or retried manually before that point. On the other hand, if the request is part of an automated dataset upload, it may be reasonable to retry for several hours (possibly with a larger `max_delay`) before giving up and alerting a human operator.

Permanent Failures

Some failures are permanent and there is no point in retrying the request that triggered them (e.g. if you received an HTTP 403 response because you are not allowed to perform a certain action, retrying will not help). It is your responsibility to verify the exact response code semantics based on our documentation before deciding whether to retry.

Third Party Libraries

You are welcome to use a third party library to implement your retry strategy but it is your responsibility to verify it actually implements a truncated exponential backoff algorithm with jitter with reasonable parametres under your control.

Further Information

If you have questions about this topic or need assistance in picking appropriate values in the above algorithm when implementing clients for our systems, you are welcome to contact us.

You can also find further information in the following references:

- Wikipedia: Exponential Backoff
- Google SRE Book: Addressing Cascading Failures: Retries
- AWS: Exponential Backoff And Jitter
- Google Cloud IAM: Retry failed requests
- Google: Building Secure and Reliable Systems: Mitigating Denial-of-Service Attacks: Client Retry Behavior

Previous page
Overview

Next page
Get Layer Metadata