Diving into Go's HTTP server timeouts

Recently, I was adding timeouts to a Go HTTP server and ended up exploring how the different settings and approaches act and interact. I'm going to publish my notes here, along with the code I used for testing. Hopefully this will help someone else (or myself) in the future.

The timeout testing client can be found here: github.com/adam-p/httptimeout. There is a server in the examples directory that you can make requests to.

I link to it below, but I'm going to recommend here that you read Filippo Valsorda's post "So you want to expose Go on the Internet". It's essential, but I didn't find it had enough quite enough detail about timeouts, hence the below examination.

There are two different, overlapping levels of timeout in our HTTP server:

Read, write, and idle timeouts on the http.Server

The ServeHTTP timeout (this middleware)

The http.Server timeouts are overlapping and somewhat confusing (to me and others) so I'll test and detail how they work (or seem to). (Another important but insufficiently thorough reference is the Cloudflare post "So you want to expose Go on the Internet".)

IdleTimeout: "IdleTimeout is the maximum amount of time to wait for the next request when keepalives are enabled. If IdleTimeout is zero, the value of ReadTimeout is used." Not relevant to request timeouts.

ReadTimeout: "The maximum duration for reading the entire request, including the body." It's implemented in net/http by calling SetReadDeadline immediately after Accept.

ReadHeaderTimeout: "ReadHeaderTimeout is the amount of time allowed to read request headers." Implemented as above.

WriteTimeout: "WriteTimeout is the maximum duration before timing out writes of the response. It is reset whenever a new request's header is read." This effectively covers the lifetime of the ServeHTTP handler stack.

Observations:

The documentation makes a big deal out of ReadHeaderTimeout allowing for per-request timeouts based on the headers. "The connection's read deadline is reset after reading the headers and the Handler can decide what is considered too slow for the body." "Because ReadTimeout does not let Handlers make per-request decisions on each request body's acceptable deadline or upload rate, most users will prefer to use ReadHeaderTimeout." But since http.Request doesn't provide access to the underlying net.Conn, I don't see a way to set a connection deadline from the handler level. (Perhaps it intends the per-request timeout to be done via mw/context rather than via conn deadlines.)

Our TLS terminates at the load balancer, so mention of different TLS behaviour you might see doesn't apply.

The zero values mean no timeout. These shouldn't be used for anything but toy servers.

A timeout during header or body read means that there's no response to the client. This is unfortunate but expected.

A timeout during header read means that there's no server log written for the request. This is even more unfortunate but also not unexpected. The handler stack (including logging middleware) is not set up until the headers are read.

http.Server timeouts do not themselves cancel the request context. However, if a body read follows the timeout, the resulting error will trigger a context cancellation.

A ReadTimeout during body read results in a log with status of 503. This is initially somewhat surprising. The timeout interrupts the read connection, then the failed read attempt cancels the request context, then the http.TimeoutHandler (discussed below) receives the signal of that cancellation and sends the 503 response.

This is okay, but I'd prefer more control over it. (This might be a bigger problem later, when we try to handle "context canceled" with more nuance.)

The previous point illustrates (I think) that the read and write channels of the connection are severed by the timeouts separately (the response can be written even though the read is interrupted).

ReadHeaderTimeout by itself works as expected. The header read is deadlined, but nothing else is.

ReadTimeout by itself works as expected. The timeout is shared between the header read and body read.

ReadHeaderTimeout and ReadTimeout together:

If set to the same value, behaviour is indistinguishable from just ReadTimeout being set.

If ReadHeaderTimeout is a different value from ReadTimeout:
If the header read is too slow, then ReadHeaderTimeout is used.

If the body read is too slow, then ReadTimeout is used. The time allowed for the body read is the total ReadTimeout minus the time spent reading headers. (As in the ReadTimeout-by-itself case.)

I haven't figured out (in a reasonable amount of time) a way to emulate/implement a slow read. I don't know how to stream the response and read of it.

But if the WriteTimeout is set to 1ns the client gets EOF immediately.

A sleep longer than WriteTimeout before writing the response results in the client getting no data, but the client still takes the sleep-time to disconnect rather than the timeout-time, which seems very strange to me.

One of the sources led me to believe that ReadHeaderTimeout+WriteTimeout would cover the whole request ("ReadHeaderTimeout … covers up to the request headers … WriteTimeout normally covers the time from the end of the request header read to the end of the response write (a.k.a. the lifetime of the ServeHTTP)"). What actually happens is that the header read timeout is correct, the write timeout is correct, but there's no body read timeout. So the request can spend forever reading the body but when it goes to write the response the write connection has deadlined.

I believe that what's happening is that the WriteTimeout is reset every time a read happens, so it's not actually starting as long as there's a body read. (The documentation says WriteTimeout "is reset whenever a new request's header is read." But that doesn't seem to be exactly accurate.)

Otherwise rough testing with combinations of the read timeouts with WriteTimeout suggests they behave as expected (no interaction).

In addition to the http.Server timeouts we use a timeout middleware, which is basically a wrapper around http.TimeoutHandler. Here are some observations when the timeout middleware is in play and has a timeout shorter than the connection timeouts:

Unsurprisingly, the timeout mw's timeout doesn't start ticking until the handler stack is set up, so not until after the headers are read.

http.TimeoutHandler uses "503 Service Unavailable" as its timeout response. It seems like "408 Request Timeout" would be a more semantically appropriate response. We could intercept the response write to change that code, but it would get hack-y to distinguish between http.TimeoutHandler returning 503 and, say, our Ping endpoint returning it intentionally. Additionally, returning a 5xx error means that our clients will automatically retry the request, which is a good thing (probably).
We could also use a copy of http.TimeoutHandler (~220 lines) to return whatever value we want.

It may seem silly to worry about sending a response to the client when its connection is so degraded that it probably can't read it. But: a) the timeout response might be a lot smaller than whatever the client is trying to send, b) the client's down pipe might be faster than its up pipe, and c) the timeout might actually be due to our server taking too long to the process, rather than a problem with the client.

Whether the client receives the timeout mw 503 response depends on what it's doing. (My test client that gets interrupted writing slowly can't read the response, but if it's trying to read when the timeout happens the response is received okay.)

A slow body read is interrupted by the mw timeout with an "i/o timeout" error. I believe this is due to the request context being canceled by the timeout.

A long time.Sleep isn't magically interrupted, unsurprisingly. But selecting on ctx.Done and time.After ends early due to the context cancellation.

There are two cases when TimeoutHandler returns 503. The first is, of course, when the deadline it set on the context fires (it could have been set somewhere else, in theory). The other is if the context was canceled for some other reason (such as the client leaving). They are distinguishable from the client side because there's no response body in the latter case.

Note that it is important that the timeout mw have a shorter timeout than the http.Server timeouts. We want the client to receive a response, if possible, rather than just having its connection severed.

This is not as simple as it might seem. The handler mw timeout must be shorter than either the WriteTimeout or the time remaining to the ReadTimeout after header reading. But at the handler level we don't know how long the header read took, except that it took less than ReadHeaderTimeout. So our mw timeout should be `min(WriteTimeout, ReadTimeout-ReadHeaderTimeout)`.

…Except that calculation ends up feeling very unnatural in practice. Instead, it makes more sense to first choose the desired handler timeout, then set the http.Server connection timeouts based on that. I think that it's reasonable to use 0.5x the timeout for ReadHeaderTimeout and 1x the timeout for ReadTimeout and WriteTimeout.

We certainly can't rely on the timeout mw while reading headers (because there is no middleware at that point), but it's possible that body read and response write timeouts are redundant. Severing the connection seems safer than cancelling the context and hoping something checks it, so we'll set the other timeouts anyway.

## Addendum

### Let's work through the timeout math

Let's say we want, generally, a 10-second request timeout. So we set TimeoutHandler's timeout to 10 seconds.

We need to pick a ReadHeaderTimeout that is basically independent from that (because the handler timeout doesn't start until *after* the header read is complete). It seems reasonable to pick 5 seconds.

As discussed above, we prefer the ReadTimeout to be longer than the handler timeout, so the client has a chance of getting the response. Because ReadTimeout ticks away during the header read, the calculation for this is something like:

```
ReadTimeout := handler_timeout + ReadHeaderTimeout + wiggle_room


e.g.,
= 10s + 5s + 200ms
```

So even if the header read takes 4.9s, we are still left with 10.3s for the body read – slightly longer than the handler timeout.

WriteTimeout covers from the end of the reads until the end of writing. If there's no body to read, this is the whole post-header request time. So, we want it to be `hander_timeout + wiggle_room`, so something like 10.2s.

IdleTimeout… is independent of any of this stuff. It seems common to set it to a couple of minutes.

### AWS observations

Using an AWS load balancer in front of your Go server muddies the behaviour of some of these timeouts, but doesn’t completely obviate them.

ALB seems to buffer all the incoming headers, so ReadHeaderTimeout does nothing. ALB’s timeout for reading headers appears to be 60 seconds.

ALB doesn’t seem to have a body-read timeout (or at least not one shorter than a couple of minutes). It does seem to be buffering some of the incoming body, since the client can still send some data after the backend server has given up the connection. About 30 seconds after the server drops the connection, the load balance responds with 502 Bad Gateway.

I didn’t test the write timeout, but I bet there isn’t one.

The ALB idle timeout seems to be 60 seconds.