Martin Baillie

Gotchas in the Go Network Packages Defaults

20210321
~27 mins
tl;dr: Things I keep forgetting about Go's network packages defaults.

Fool Me Once

I have been keeping a wee `.org` file of gotchas in the defaults of Go's various `net` packages for a while now. I pull it up each time I'm building a service with the standard library, just to make sure I don't miss something that I have already hit in the past. Let's call it learning from one's mistakes where the one in question has a shocking memory.

I've just this week added another entry after getting to the root cause of a production issue and thought to myself that there was enough in the file to merit tidying up and posting.

Well, I have nothing else to do on this infamously soggy Sunday in Sydney so what follows is the current list in all its glory, with some added explanation for good measure. Just bear in mind that the items mostly pertain to HTTP-based clients and servers from using Go at (relative) scale in service oriented architectures. Also, I think they are all still relevant as of Go 1.16 but happy to be corrected on that.

Timeouts

Set them!

The network is unreliable and the standard library default clients and servers do not set their main timeouts, and all of them interpret the zero value as infinity to boot. Timeouts are subjective to the use case and the Go core team have steered clear of making any sweeping generalisations.

> NOTE: This includes all use of the package level convenience functions too: `http.Get` and client friends, `http.ListenAndServe` and server friends.

A corollary to this is you should practically **always** have a customised `http.Client` and/or `http.Server` in a production Go service.

Client timeouts

For clients you often only need to configure the main timeout. It covers the E2E exchange and is most likely how your mental model of an RPC works:

```
c := &http.Client{
        Timeout: 5 * time.Second,
}
```

This timeout includes any HTTP `3xx` redirect durations, the reading of response body and the connection and handshake times (unless a reused connection). I find I am usually done here regarding clients.

However, for granular control over these individual properties and more, you need to drop lower to the underlying transport:

```
c := &http.Client{
        Timeout: 5 * time.Second,
        Transport: &http.Transport{
                DialContext: (&net.Dialer{
                        // This is the TCP connect timeout in this instance.
                        Timeout: 2500 * time.Millisecond,
                }).DialContext,
                TLSHandshakeTimeout: 2500 * time.Millisecond,
        },
}
```

> NOTE: Since response bodies are read after the client method has returned you need to use a `time.Timer` if you want to enforce read time limits.

There are more timeouts on the transport that I have never had a need for such as the `ResponseHeaderTimeout` (time to wait for response headers after request writing ends) and the `ExpectContinueTimeout` (time to wait for a `100-Continue` if using HTTP Expect headers).

There are also settings related to reuse, such as the transport's `IdleConnTimeout` and dialer's `KeepAlive` settings. These are deserved of their own section.

Server timeouts

In the same vein as you not wanting a server to hold your client's requests hostage because they have no timeout, when writing a Go HTTP server you have the inverse consideration: you don't want badly behaving or laggy clients holding your server's file descriptors hostage.

To avoid this, you should always have a customised `http.Server` instance:

```
s := &http.Server{
        ReadTimeout:  2500 * time.Millisecond,
        WriteTimeout: 5 * time.Second,
}
```

`ReadTimeout` here covers the time taken to read the request headers and optionally body, and `WriteTimeout` covers the duration to the end of the response write.

However, if the server is processing TLS then the `WriteTimeout` ticker actually starts as soon as that first byte of the TLS handshake is read. In practice this means you should factor in the whole `ReadTimeout` and then whatever you want to accept for writes on top of that.

Similar to the main `http.Client.Timeout` value, these are the two main server timeouts that you should think about appropriate situational values for, but there are a few others that give more granular control (such as the time to read and write headers respectively). Again, I have never had a need to use them.

These timeouts cover poorly behaving clients. But with a server, you should have a think about how long you are willing to accept as a **request handling duration** as well. I mention mental models of client timeouts above; I would argue this is the server-side version that intuitively springs to mind when you think: *"server timeout"*.

With Go's `http.Server` you could implement these timeouts with context tickers in the handler funcs themselves. You could also use the `TimeoutHandler` helper wrapper:

```
func TimeoutHandler(h Handler, dt time.Duration, msg string) Handler
```

Wrapping with this means business-as-usual until `dt` is breached at which point a 503 is written down the pipe to the client with the optional body `msg`.

HTTP Response Bodies

Close them!

As a client you might not care about the content of your response bodies or you might be anticipating empty responses. Either way, you should close them off. The standard library does not do it on your behalf and this can hold up connections in the client's pool preventing reuse (i.e. if using HTTP/1.x keep-alives) or worse, exhaust host file handles.

The standard library does however guarantee response bodies to be non-nil even in the cases of a response sans body or with a zero-length body. So, to close things out safely the following suffices:

```
res, err := client.Do(req)
if err ≠ nil {
        return err
}
defer res.Body.Close()
 ...
```

If you are not going to do anything with the body then it is still important to read it to completion. To not do so affects the propensity for reuse, particularly if the server is pushing a lot of data. Flush the body with:

```
_, err := io.Copy(ioutil.Discard, res.Body)
```

> NOTE: Depending on the scenario, it might be pertinent to make an attempt to reuse the connection, but then also more efficient to close it if the server is pushing too much data. `io.LimitedReader` can help here.

HTTP/1.x Keep-alives

Speaking of reuse, keep-alives are Go's default but sometimes you don't want them. Case in point, I had a service acting as a webhook transmitter a few years ago. It needed to make requests to many varied upstream targets (almost never the same).

The easiest way to turn the default behaviour off is to wire a custom transport into the client (which I find I'm always doing anyway for some of the other reasons in this fieldnote):

```
client := &http.Client{
    &http.Transport{
        DisableKeepAlives: true
    }
}
```

You can, however, also do this per request by telling the Go client to close it for you:

```
req.Close = true
```

Or otherwise signalling a well-behaving server to add a `Connection: close` response header with which the Go client will know what to do.

```
req.Header.Add("Connection", "close")
```

Connection Pooling

Continuing with the theme of reuse. In the micro-SOAs I find myself working in, I am actually **much less likely** to be building that webhook transmitter service above than I am a service that needs to integrate at high frequency but to only a few upstreams (e.g. a cloud datastore/queue and a dependant API or two).

I would argue in this more common scenario the Go `http.Client` defaults work against you.

By that I mean there are some properties exhibited by the client's default transport with regards to connection pooling that you should always be mindful of:

```
var DefaultTransport RoundTripper = &Transport{
        MaxIdleConns:          100,
        IdleConnTimeout:       90 * time.Second,
}
...
const DefaultMaxIdleConnsPerHost = 2
```

The relationship between these three settings can be summarised as follows: a connection pool is retained of size 100, but **only 2 per target host**, and if a connection remains unutilised for 90 seconds it will be removed and closed.

So take the scenario of 100 goroutines sharing the same or default `http.Client` to make requests to the same upstream dependency (this is not so contrived if your client is itself also a server part of a larger microservice ecosystem, forking routines per request it receives). 98 of those 100 connections get **closed immediately**.

First things first, the means your service is working harder. There are myriad connection establishment costs: kernel network stack processing and allocation; DNS lookups, of which there may be many (read about `resolv.conf(5):ndots:n` especially if you run Kubernetes clusters); as well as the TCP and TLS handshakes to get through.

This is of course not optimal, but there is another hidden cost that has bitten me in the past, rendering entire hosts useless: **closed != closed** (in Linux anyway).

The kernel actually transitions the socket to a `TIME_WAIT` state, the purpose of which being primarily to prevent delayed packets from one connection being accepted by a subsequent connection. The kernel will keep these around for ~60s (very hard to change in Linux as per RFC793 adherence).

A buildup of `TIME_WAIT` sockets can have adverse effects on the resources of a busy host.

For one, there is the additional CPU and memory to maintain the socket structure in the kernel, but most critically there is the slot in the connection table. A slot in use means another connection with the same quadruplet (source addr:port, dest addr:port) cannot exist, and this in turn can result in **ephemeral port exhaustion** — the dreaded `EADDRNOTAVAIL`.

Validating URIs

This is a small one, but as far as I'm concerned, the `url.Parse` method is essentially infallible and it trips me up all the bloody time. You almost always want `url.ParseRequestURI` and then some further checks if you are wanting to filter out relative URLs.

DNS Caching

Unlike the JVM, there is no builtin DNS cache in the Go standard runtime. This is a double edged sword. I'm personally thankful for this default after been burned countless times by that JVM cache in a past life. At the same time, it is something to always be cognisant of when trying to produce an optimised Go service.

The Go core team's stance is you should defer to the underlying host platform to support your DNS caching needs by way of something like dnsmasq. However, it is worth pointing out that you are not always in control of that situation. For example, AWS Lambda's runtime sandbox contains a single remote Route53 address in `/etc/resolv.conf` and provides no sandbox-local cache server.

Another option you have in this situation is to override the `DialContext` on `http.Transport` (as seems to be the general theme of this fieldnote) and wire in an in-memory cache. I can recommend dnscache for this purpose.

> NOTE: There is also the package singleton `net.DefaultResolver` that you may need to consider overriding if you don't have full control over your client's transport.

You might consider one of these options if you have latency-sensitive services with only a couple of upstream dependencies. Those services will otherwise need to continually dial that same unchanging couple of domains by default.

I say "by default" because you might have a well-tuned set of reused connections (perhaps even in thanks to the section on connection pooling). If that's the case then caveat emptor—I have another piece of anecdata for you.

At my previous client I had an issue where a high volume Go-based service acting (in part) as a reverse proxy kept proxying the same dead backend endpoints. This was because the backend target used DNS to do blue/green rollouts of new versions, and was occurring despite the host having a TTL-respecting DNS cache.

The problem was the service was reusing the connections so fast that the idle timeouts were never breached.

As of Go 1.16, nothing in the default runtime will force those established connections to close and thus get updated resolved IPs for hostnames, forcing you to get creative with a separate goroutine to call `transport.CloseIdleConnections()` on an interval which is less than ideal. While it is very easy to write a reverse proxy in Go, my mistake here was not deferring to something more dedicated and endpoint-aware (like the excellent Envoy proxy).

Masqueraded DualStack `net.Dial()` Errors

This one is nuanced but is a *belter* if it hits and it can surface (as it did for me) even if you're not actively using IPv6.

Take a stock Amazon EKS worker node as an example. At the time of writing this the EKS optimised AMI has the following default traits:

1. The Docker daemons do not have the experimental IPv6 flag enabled and so will not configure IPv6 addresses on the container virtual network interfaces.
2. But the kernels do have IPv6 support enabled, meaning `/proc/net` gets the IPv6 constructs (even in container namespaces) and some other things are inferred from there, most critically, `/etc/hosts` receiving two default entries for loopback: `127.0.0.1` and `::1`.

Now suppose you have a Go service that calls another over loopback, such as a process sidecar, but you naively resolve the loopback address using `localhost` hostname.

> TL;DR: This is where you went wrong. Save yourself the trouble, stop here and use `127.0.0.1` or `::1` depending on your target stack unless you have a good reason not to. Read on for why you might want to do that.

You don't notice during development, but in an integrated environment running at scale you see sporadically recurring `::1 cannot assign requested address` emitted from the dialer. However, you check the host and ephemeral ports are *absolutely fine*. There goes that theory.

What's with that `::1` IPv6 address family error though? It's concerning because from the AMI traits above, we know that a client resolving that address is going to have a bad time connecting given there's no network interface actually bound to it. But then again, if that were true why does it not fail *all* the time?

Well, it could be that the Go dialer is masquerading the real error—the **IPv4 error**!

This can happen because of a few subtle defaults:

1. Firstly, when presented with multiple addresses for the same hostname, the Go dialer sorts and selects addresses according to RFC6724 and critically, this RFC outlines a preference for the IPv6 **first**.

2. Then, because Go's default network transport also has RFC6555 support (aka. Happy Eyeballs / Dual Stack), it will try to dial both address families in parallel but give the primary IPv6 a 300 millisecond head start. If the primary fails fast (which it would always do in this case due to the non-existent IPv6 interface address), then the head start is cancelled and IPv4 is tried immediately. All good so far. However, if both addresses fail to dial, only the **primary (i.e. IPv6) error is returned**.

So if your IPv4 address is failing to dial sporadically (say, for example, a laggy upstream is causing sporadic connect timeouts) the error presented will be the irrelevant and always failing to dial IPv6 `::1 cannot assign requested address` instead of the much more helpful IPv4 `connect timeout`.

`net.IP` is Mutable

This one bit me in production, albeit when I was doing something stupid. Don't be tricked into thinking `net.IP` is an immutable data structure. It is in fact a transparent type aliased to `[]byte`. Anything you pass it to could mutate it and Sod's/Murphy's Law says it will.

Bonus: GOMAXPROCS, Containers and the CFS

A bonus entry about a default behaviour which, while not specifically related to the `net` packages, sure can affect their performance indirectly.

Go will use the `GOMAXPROCS` runtime variable value at init to decide how many real OS threads to multiplex all user-level Go routines across. By default it is set to the number of logical CPUs discovered in the host environment and I suppose this is another example of the Go core team needing to settle on a sensible default i.e. the number of logical CPUs in a generalised context is what provides the highest performance.

Unfortunately the Go runtime happens to also be unaware of CFS quotas, and multi-tenant container orchestration platforms (like Kubernetes) will default to using these to enforce their respective CPU restriction concepts on running container's cgroups.

> NOTE: CFS here being the Linux kernel's Completely Fair Scheduler—a proportional share scheduler that divides available CPU bandwidth between cgroups.

Continuing with Kubernetes as an example, a defaulted `GOMAXPROCS` value compounded by a CFS quota-unaware runtime means your Go service `Pod`, complete with carefully considered CPU resource limits and request specs, can find itself being **aggressively and unduly throttled** by the CFS.

Why is this? Well say for instance your Go service `Pod` spec has a CPU resource limit of 300m (millicpu) and taking the Linux kernel's stock CFS quota window of 100ms, this gives the Go service 30ms of CPU time per window. If it tries to use more than that it gets throttled by the CFS until the next window. This is fine and is presumably expected because you scientifically benchmarked your service process locally and landed on that 300m request in the first place (right?).

However, the key thing to remember is that 30ms is actually further subdivided between `GOMAXPROCS` OS threads, so if your `Pod` happens to land on a large commoditised host VM with, for simplicity's sake, 15 logical CPUs (as is common in binpack-styled Kubernetes cluster topologies) then your Go service will have naively set `GOMAXPROCS` to 15, giving each resulting OS thread potentially just **2ms** of CPU time per scheduling window before it gets preempted!

The result is a Go runtime scheduler wasting all its allotted CPU time on context switching and getting no useful work done because it thinks it has access to 15 logical CPUs when in fact it has access to a fractional 0.3.

> NOTE: To see if this is affecting Kubernetes services you are responsible for, the kubelet cAdvisor metric `container_cpu_cfs_throttled_periods_total` is king.

`GOMAXPROCS` is an integer so your only recourse in the example above is to hardcode it to 1. This can be done either through the environment with `export GOMAXPROCS=1`, or with the package func `runtime.GOMAXPROCS(1)`.

More generally though, I would recommend Uber's automaxprocs drop-in to make your Go runtime CFS-aware. For Kubernetes operators, you could also disable CFS quota enforcement at the kubelet level if you were so inclined.

- aws
- go
- kubernetes

C2F0 79DE D64B 7361 006A A099 2A56 EA64 591E 15E4 $id fieldnotes