Golang custom transports and timeouts

PUBLISHED ON APRIL 9TH, 2015 – FILED UNDER GOLANG, TIMEHOP, BARE METAL

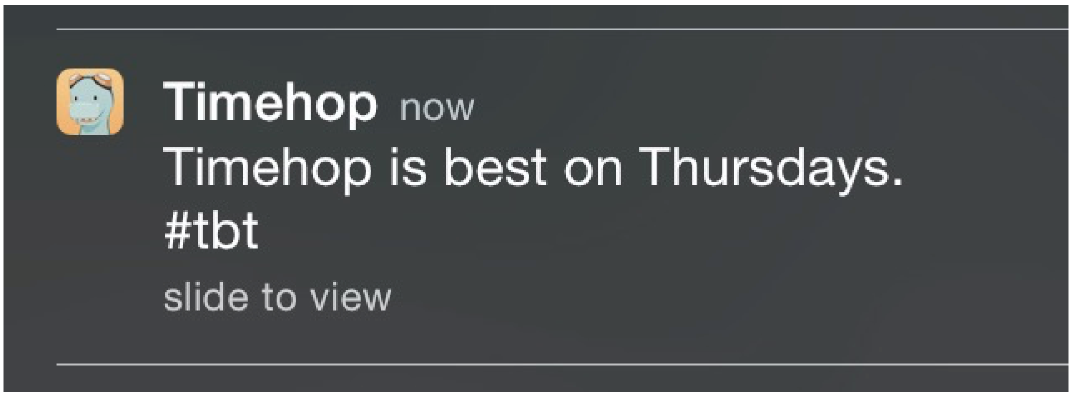
This post is a *post-mortem* of a Go production system crash. We'll use the logs to drill down into the guts of Go's network source code to find a leak that affected a core part of our stack at Timehop. The root cause was the use of custom transports without explicit timeouts for requests and DNS lookups, as well as using an older version of glibc under Linux.

TL;DR

- Always use explicit timeouts – especially when using custom transports
- If you're running under Linux, upgrade glibc to version 2.20

"Your Timehop day is ready"

One of the core elements to the Timehop experience is the morning push notification you get on your phone:


Timehop now
Timehop is best on Thursdays. #tbt
slide to view

The morning push

Behind that notification is a complex set of server-side tasks we call *prepares*. They ensure that everything is ready to go when you check the app in the morning.

For the sake of not venturing too far outside the scope of this post, I'll skip a few details on what actually happens on these *prepares*; here's what matters:

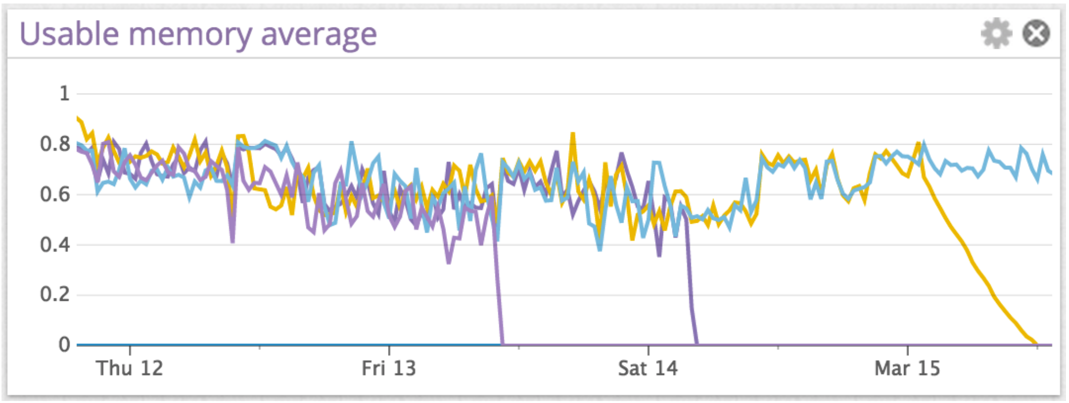
1. At the time of writing, we run over 11 million prepares a day
2. Each prepare takes, on average, around 300ms
3. Each prepare is highly concurrent and network I/O intensive (what @kevrone wrote here should give you a hint of the scale of how much data we deal with)
4. Each machine runs 30 prepares concurrently
5. We have 4 c3.large machines
6. We use DataDog as our primary monitoring tool for our AWS CloudFormation stacks

Everything was fine (for a while)

As we went over 10 million prepares a day, we started observing DataDog metrics dying off, while the prepare jobs were still being executed – we keep an eye out on the queue sizes with a not-so-redundant, alternate method.

At first this was so infrequent that we disregarded its importance; maybe it was just something wrong with the chef recipe that restarted the DD agent or DD agent itself. Either way, we'd just kill that machine (which was effectively working, just not reporting DD stats) and CloudFormation would bring another one up. *Voilá*, "fixed".

Then it started happening more frequently:



Daily acute drops in available memory, evidenced by the lines sharply pointing down. These are times when DataDog reporting died completely.

A sudden and rather severe memory leak – all the while, the *prepares* were still being performed.

I jumped on the DD logs:

```
2015-03-14 08:23:33 UTC | ERROR | dd.collector | collector(agent.py:354)
  | Uncaught error running the Agent
error: [Errno 12] Cannot allocate memory
```

Then onto our own Go logs:

```
fatal error: out of memory (stackcacherefill)
runtime stack:
runtime.throw(0xbb7bc0)
  /usr/local/go/src/pkg/runtime/panic.c:520 +0x69
stackcacherefill()
  /usr/local/go/src/pkg/runtime/stack.c:52 +0x94
```

Oops.

Tracing the leak

As the Go program crashed, it dumped its runtime stack onto the logs – all 4.5 million lines of it.

After spending a couple hours tracing and accounting for every goroutine and function call we had written, the only thing that really stood out was this:

```
goroutine 281550311 [semacquire, 27 minutes]:
sync.runtime_Semacquire(0xc2267cadc0)
  /usr/local/go/src/pkg/runtime/sema.goc:199 +0x30
sync.(*WaitGroup).Wait(0xc2337c8f80)
  /usr/local/go/src/pkg/sync/waitgroup.go:129 +0x14b
net.(*singleflight).Do(0xbbdc50, 0xc22f602e40, 0x20, ...)
  /usr/local/go/src/pkg/net/singleflight.go:37 +0x127
net.lookupIPMerge(0xc22f602e40, 0x20, 0x0, 0x0, 0x0, 0x0, 0x0)
  /usr/local/go/src/pkg/net/lookup.go:42 +0xae
net.func025()
  /usr/local/go/src/pkg/net/lookup.go:80 +0x3e
```

```
created by net.lookupIPDeadline
  /usr/local/go/src/pkg/net/lookup.go:82 +0x2d8
```

The number of occurrences of this pattern, as well as the length for which the go routines had been stuck (~30 minutes) struck me as odd, so I counted the occurrences...

```
$ grep 'net.lookupIPDeadline' crash.log | wc -l
420563
```

Woah. That's **a lot** of stuck lookups.

But they have a deadline and clean up after themselves, right?

Nope. Here's `lookupIPDeadline`:

```
// lookupIPDeadline looks up a hostname with a deadline.
func lookupIPDeadline(host string, deadline time.Time)
                     (addrs []IPAddr, err error) {
  if deadline.IsZero() {
    return lookupIPMerge(host)
  }

  // We could push the deadline down into the name resolution
  // functions. However, the most commonly used implementation
  // calls getaddrinfo, which has no timeout.

  timeout := deadline.Sub(time.Now())
  if timeout <= 0 {
    return nil, errTimeout
  }
  t := time.NewTimer(timeout)
  defer t.Stop()

  ch := lookupGroup.DoChan(host, func() (interface{}, error) {
    return lookupIP(host)
  })

  select {
  case <-t.C:
    // The DNS lookup timed out for some reason. Force
    // future requests to start the DNS lookup again
    // rather than waiting for the current lookup to
    // complete. See issue 8602.
    lookupGroup.Forget(host)

    return nil, errTimeout

  case r := <-ch:
    return lookupIPReturn(r.v, r.err, r.shared)
  }
}
```

A few very interesting things:

- If no deadline is specified, there will be no guarantee of this function ever returning
- When a deadline is specified, the timeout is a Go timeout, not a lower (OS) level timeout
- That Issue #8602 mention, which we'll get back to later on

I was kind of baffled there wasn't any sort of lower level timeout while performing a DNS lookup... So, following through to `lookupIPMerge`:

```
func lookupIPMerge(host string) (addrs []IP, err error) {
  addrsi, err, shared :=
    lookupGroup.Do(host, func() (interface{}, error) {
      return lookupIP(host)
    })
  // ...
```

Now the `lookupIP` function is platform-dependent. In this case it's defined in `lookup_unix.go` (the binary was running on a linux machine.)

```
func lookupIP(host string) (addrs []IP, err error) {
  addrs, err, ok := cgoLookupIP(host)
  if !ok {
    addrs, err = goLookupIP(host)
  }
  return
}
```

`cgoLookupIP` (which defers to `cgoLookupIPCNAME`) is defined in `cgo_unix.go`

Nowhere along this path will you see a possible timeout being raised from below. Looking up the documentation of `getaddrinfo`, there isn't a single mention of "timeout".

Basically, if the `lookupIP(host)` call running inside that `lookupGroup`'s `DoChan()` hangs — which apparently it can since this is called —, it'll take a Go routine with it. Forever. Bye.

The only way to truly fix this is to push the timeouts down the stack. I went back to Issue #8602 and on the fix commit, noticed this *TODO*:

```
// TODO(bradfitz): consider pushing the deadline down into the
// name resolution functions. But that involves fixing it for
// the native Go resolver, cgo, Windows, etc.
```

And how it got replaced by this:

```
// We could push the deadline down into the name resolution
// functions. However, the most commonly used implementation
// calls getaddrinfo, which has no timeout.
```

You read that right:

> `getaddrinfo`, **which has no timeout**.

At the end of the day, the only thing preventing a real leak is the implementation of `getaddrinfo` having some sort of hard limit on DNS lookup timeouts and eventually returning.

While this struck me as a severe oversight at first, understanding that there is no portable cross-platform solution to pushing the timeouts down the stack made me realize that I'd probably also end up erring on the side of trusting the underlying implementation to return in a timely fashion. It's either that or re-invent the wheel and end up writing a resolver.

In our case, this was a bug with glibc's `getaddrinfo`, fixed in v2.20, so the only real solution was to update glibc.

To check which version of glibc you're running on your (Linux) system:

```
$ ldd --version
ldd (Ubuntu EGLIBC 2.19-0ubuntu6) 2.19
```

Now that the real probleam was solved, it was time to go back and make things a little better...

Use explicit timeouts across the board

When you do something like this...

```
foo := &http.Transport{}
```

... wherever this transport ends up being used, it may hang for an indeterminate period of time performing lookups and/or TLS handshake.

For unencrypted connections, the `Transport` instance will use whichever function is assigned to the `Dial` field to establish the connection. If no function is assigned, it'll default to `net.Dial`, which creates a temporary `Dialer` that may or may not have a timeout – keyword here is *may*.

And since you'll highly likely be using this transport with an `http.Client`, I'd also recommend setting a cap for the `Timeout` field. Just keep in mind this is a hard global timeout on the entire request cycle – dial, send the request and read the response.

I traced the DNS lookups back up to our code and found this:

```
client := &http.Client{
  Transport: &http.Transport{},
}
```

There it was, that second line. The quick & dirty fix would be to simply replace `&http.Transport{}` with `&http.DefaultTransport` but you may want to consider something a bit more explicit and/or aggressive for production systems:

```
secs := 3 // rather aggressive
client := &http.Client{
  Transport: &http.Transport{
    Proxy: http.ProxyFromEnvironment,
    Dial: (&net.Dialer{
      Timeout:   secs * time.Second,
      KeepAlive: 30 * time.Second,
    }).Dial,
    TLSHandshakeTimeout: secs * time.Second,
  },
}
```

Conclusion

- Use explicit timeouts everywhere — take the driver's seat
- `http.DefaultClient` is relatively safe since it uses `http.DefaultTransport`, which has dial timeout set (it won't have request timeouts though, so tread carefully)
- Make sure your `http.Client`'s `Dial` function has timeouts and set `TLSHandshakeTimeout`
- Update to glibc 2.20 or higher

« previous | index | next »