Marek's
totally not insane
idea of the day

How to resolve a million domains

28 November 2013

Few years ago Alexa started to publish a daily snapshot of top one million domains used on the internet. Using this data I wanted to analyse how the DNS assignments change over time.

The plan was simple:

1. Every day download the Alexa list of top domains.
2. Every day resolve all the domains.
3. Do something with the gathered data.
4. Profit!

The second point sounds boring, but in fact it poses an interesting technical challenge: it's not that easy to query large number of domains for the DNS `A` records. It's especially hard to do it within a constrained time, say 1 hour.

Latency vs throughput

The DNS protocol is not optimised for high throughput but for low latency. It's using an unreliable UDP datagrams. This is a good choice to reduce average latency as the recursive DNS server is usually nearby and packet loss.

Using a local DNS recursive server is a good choice when browsing the internet, but not necessarily when resolving a large number of domains.

Naive approach

The most obvious approach is to just pipe the requests to a built-in resolver in your programming language. The requests will hit a local DNS resolver eventually, but first they need to go through few intermediate layers.

Most languages use blocking and thread unsafe `gethostbyname(3)` call. The better choice is `getaddrinfo(3)`, but it's still blocking and requires multiple threads to exploit concurrency. There's also a GNU extension `getaddrinfo_a`, which should be able to run many lookups asynchronously, but I've never seen it used.

The biggest problem with all these methods is that they go through your full OS DNS resolution mechanism. That may mean parsing `/etc/nsswitch.conf`, `/etc/hosts` and `/etc/resolv.conf`.

Fortunately, in my experiments I used a `go-lang`'s built-in resolver `net.ResolveIPAddr`. It's pretty good and tries to handle all the logic internally and it's not calling glibc for help. Here's a trivial code snippet:

```
for domain := range domains {
        ip, err := net.ResolveIPAddr("ip4", domain)
        if err == nil {
                fmt.Printf("%s ok %s\n", domain, ip)
        } else {
```

```
                fmt.Printf("%s error: %s\n", domain, err)
        }
    }
```

My computer was able to resolve around 5.4 domains per second. Not very good.

Parallelise all the things!

The obvious optimisation is to run many `net.ResolveIPAddr` calls in parallel. In `go` we can achieve it easily with built-in concurrency mechanisms.

With a hundred worker goroutines the throughput goes up to 40 domains per second.

Unfortunately, at this point we hit a first road block: we're limited to a thousand workers. Every goroutine opens a separate UDP socket and the program will run out of file descriptors if we do more than 1020 requests at once. Modifying the `ulimit -n` may sound attractive, but it's not a clean solution.

Even with a thousand workers the script was using only a fraction of the available bandwidth. We clearly need better.

Use a better resolver

The next bottleneck was my ISP's recursive resolver.

I tried to use Google public DNS resolver, but were able to resolve only a small number of domains per second. Looks like it's capped at 100 requests per second from a single IP address.

Knowing that UDP requests to Google are capped I started to think about using TCP. DNS protocol allows querying over TCP, but in theory

a DNS client shouldn't do that before trying using UDP.

Obviously it appears that Google also caps queries issued over TCP.

My own resolver

Having ruled out Google DNS I decided to run my own recursive DNS server. DJB's `dnscache` is a well known local DNS recursive cache.

Unfortunately, it seems that doing recursive DNS lookups is a complex problem and `dnscache` was quickly spinning at 100% CPU.

Not my own resolver

If the really slow part is the recursive lookup itself, why should I run my own server when my ISP is already running one?

After having made a full circle, I finally settled with my final setup.

Final setup

First thing to realise is that running recursive DNS lookups can be CPU intensive. Any ISP is running a recursive DNS server, usually with a decent CPU and good pipe, so it may be a good idea to use it. It's worth mentioning that Amazon's EC2 DNS recursive servers are pretty good.

Don't use the usual OS resolving mechanisms. You want to skip all the local configuration and query the recursive server directly.

Sending a single DNS request from a unique UDP port is the secure way of doing DNS. Unfortunately it puts unneeded pressure on the kernel when running a million queries, you don't want to spend plenty of CPU time bookkeeping the open UDP sockets. Using a single UDP socket for

all the traffic is faster and allows running more than 1020 queries in parallel. It's may be necessary to write a custom DNS client for that.

The code

The result of all this experiments is a simple `go` program that is capable of flooding a DNS recursive server with queries. The script is a thin layer on top of the `go-lang` DNS packet parsing code, which I shamelessly reused.

Here's how to use it:

```
$ go get github.com/majek/goplayground/resolve
$ echo -en "google.com\nfacebook.com\nwikipedia.org\n" \
    | $GOPATH/bin/resolve -server="8.8.8.8:53"

Server: 8.8.8.8:53, sending delay: 8ms (120 pps), retry delay:
1s
google.com, 173.194.41.128 173.194.41.129 ...
facebook.com, 173.252.110.27
wikipedia.org, 208.80.154.225
Resolved 3 domains in 0.103s. Average retries 1.000. Domains per
second: 29.079
```

With this script it takes a 25 minutes to resolve all million domains on an cheap Amazon VPS.

Leave a comment.