CSE P 590 SG Assignment #1 Sample Solution

Andy Collins

DNS Spelunking

I run Linux on my desktop machine, so there will be some bias towards that. However, it seems that the Windows version of the `dig` tool is pretty much comparable.

As is always the case with DNS, we have to start with our local resolver to get anywhere. On Unix systems, you can look in `/etc/resolv.conf` to see what your configured local resolver is (along with backups and some policy related to that). Or of course you can just run `dig` without specifying a server, and you'll use the local resolver by default.

To avoid any strangeness resulting from my local resolver, however, I started off by finding a top-level domain server, using

```
dig foo
```

which got me the server

```
a.root-servers.net
```

and thereafter I always pointed `dig` at that server using `dig @a.root-servers.net ...`

1. What is the complete list of authoritative name servers for the following domains? Can you offer any guesses as to why these domains have set up their nameservers as they have?

   (a) `.com`

   (b) `.edu`

   (c) `.ca`

   (d) `.cs.washington.edu`

   (e) `.yahoo.com`

   Answer The following list of `.com` servers is the result of the command

   ```
   dig @a.root-servers.net com
   ```

   ```
   ;; AUTHORITY SECTION:
   com.                    172800  IN      NS      A.GTLD-SERVERS.NET.
   com.                    172800  IN      NS      G.GTLD-SERVERS.NET.
   com.                    172800  IN      NS      H.GTLD-SERVERS.NET.
   com.                    172800  IN      NS      C.GTLD-SERVERS.NET.
   com.                    172800  IN      NS      I.GTLD-SERVERS.NET.
   com.                    172800  IN      NS      B.GTLD-SERVERS.NET.
   com.                    172800  IN      NS      D.GTLD-SERVERS.NET.
   com.                    172800  IN      NS      L.GTLD-SERVERS.NET.
   com.                    172800  IN      NS      F.GTLD-SERVERS.NET.
   com.                    172800  IN      NS      J.GTLD-SERVERS.NET.
   com.                    172800  IN      NS      K.GTLD-SERVERS.NET.
   com.                    172800  IN      NS      E.GTLD-SERVERS.NET.
   com.                    172800  IN      NS      M.GTLD-SERVERS.NET.
   ```

Hereafter I will only list the names of the servers, and the command used to discover them, omitting all the other information returned by `dig` (which typically includes the address mappings for the nameservers, so you won't have to bother the top-level server again).

(a) `.com` (`dig @a.root-servers.net com`) — `[a-m].gtld-servers.net`

(b) `.edu` (`dig @a.root-servers.net edu`) — `[a-g,l-m]3.nstld.com`

(c) `.ca` (`dig @a.root-servers.net ca`)

- `clouso.risq.qc.ca`
- `relay.cdnnet.ca`
- `merle.cira.ca`
- `ns1cira.ca`
- `ca0[2,6].cira.ca`

(d) `.cs.washington.edu` is a little more intersting. `a.root-servers.net` was only willing to point me as far as the `edu` domain. `l3.nstld.com` pointed me to three servers for `.washington.edu`:

- `hanna.cac.washington.edu`
- `marge.cac.washington.edu`
- `ns.nts.umn.edu`

and then `hanna.cac.washington.edu` pointed me at the same three servers, plus three more:

- `trout.cs.washington.edu`
- `june.cs.washington.edu`
- `lumpy.cs.washington.edu`

and all six are authoritative for `.cs.washington.edu`. Since we know that UW is all one campus, it's not that surprising that the authority is spread out this way, but it isn't what you expect. Applying local knowledge, it's clear that the `cs.washington.edu` domain exists because we like to have `.cs` hostnames, not because we need to be autonomous within the UW. Also, it's not clear from this what `lumpy`, `trout`, and `june` actually do. No outside queries are likely to come to them, because they will stop at the washington.edu nameservers when those servers return the full mapping. It's not immediately obvious whether these machines are also the local nameservers supporting client queries from within CSE (the IP addresses don't match the servers in `/etc/resolv.conf`, but the IP addrs could easily be multiple interfaces in the same machines).

Finally note that the nameservers for the `washington.edu` domain are geographically split, with a backup at `umn.edu` (the University of Minnesota). Further investigation shows that `hanna.cac.washington.edu` (but not `marge`) is also a backup server for `umn.edu`.

(e) `.yahoo.com` (`dig @a.gtld-servers.net yahoo.com`) (same as before, have to go to the `.com` server) — `ns[1-5].yahoo.com`.

Using `traceroute`, we can discover that these five servers are diverse both geographically and ISP-wise:

name	location	ISP
`ns1.yahoo.com`	San Jose, CA	Level3
`ns2.yahoo.com`	Santa Clara, CA	Cable & Wireless
`ns3.yahoo.com`	New York, NY	Level3
`ns4.yahoo.com`	Dallas, TX	Cable & Wireless
`ns5.yahoo.com`	Washington DC	Cable & Wireless

all of which is clearly intended to protect against backhoes and other forms of network partition.

2. Which of the name servers that you discovered are willing to serve recursive queries?

Answer Serving recursive queries for unrelated hosts is not really to be expected, and of this list only `clouso.risq.qc.ca` allows it for unrelated hosts. I tested this by looking up `www.cs.berkeley.edu`. Most servers pointed me to the root servers, although `ns[1-4].yahoo.com` were able to point me to `berkeley.edu` and `ns5.yahoo.com` to `edu`, presumably because this was what they had cached at the time, which indicates that they do perform some resolutions for some people.

In addition, I am able to use all six `washington.edu` nameservers recursively, but that is only because I am inside the UW.

Curiously, in the two years since I last performed this experiment, `merle.cira.ca` has stopped accepting recursive queries, and although `ns2.uunet.ca` still accepts them, it is no longer an authoritative nameserver for the `ca` domain.

3. Which of the name servers that serve recursive queries (from 2) perform negative caching?

   Answer All of the recursive nameservers support negative caching. Here is the output from looking up `asdf.berkeley.edu` at `clouso.risq.qc.ca`, first non-recursively:

```
; <<>> DiG 9.2.2 <<>> +norecursive @clouso.risq.qc.ca asdf.berkeley.edu
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 1913
;; flags: qr ra; QUERY: 1, ANSWER: 0, AUTHORITY: 9, ADDITIONAL: 9
```

   where we get only pointed to the `edu` servers, and then recursively:

```
; <<>> DiG 9.2.2 <<>> @clouso.risq.qc.ca asdf.berkeley.edu
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 18775
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 0
```

   where we get NXDOMAIN, and then again non-recursively:

```
; <<>> DiG 9.2.2 <<>> +norecursive @clouso.risq.qc.ca asdf.berkeley.edu
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 10900
;; flags: qr ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 0
```

   where we get NXDOMAIN again, because it's in the cache

4. Try looking up non-existant `.cs.washington.edu` names using recursive lookup on the servers from 3. Is there a perceptible performance difference for names that are in the negative cache, versus those that are not? How significant do you think negative caching is for DNS performance?

   Answer (Note that this is two-year-old data, so some of these servers no longer exist or no longer support recursive queries)

   Yes, there is a clear difference in performance, but it is generally within a factor of two (not surprising, since the positive cache will make sure that the server gets to the authoritative site quickly to reject each new name). To firm up the numbers above, I looked up `aa[a-j].cs.washington.edu` from each server above, except those that are authoritative for `cs.washington.edu` and I used only `ns1.inktomi.com` because the four seem so similar). With ten runs, I computed the mean and standard deviation for the first and second lookup times:

server	first avg	first s.d.	second avg	second s.d.
clouso.risq.qc.ca	141.4 ms	0.97 ms	70.9 ms	0.32 ms
ns2.uunet.ca	907.5 ms	1538 ms	95.7 ms	2.79 ms
merle.cira.ca	203.4 ms	58.3 ms	90.6 ms	3.31 ms
ns1.inktomi.com	44.6 ms	3.72 ms	21.6 ms	1.07 ms

The initial lookups all take substantially longer (and typically about twice as long, to be expected since an ideal first query is two round-trips, to the server and back, and the second query is one), and see much greater variance (ns2.uunet.ca saw two 3 second lookups, clearly due to loss.

5. Where does Andy's e-mail get delivered? Look up all the available records for `tioga.cs.washington.edu` (note that `dig`, by default, shows only the `A` records). What happens if one of the mailservers is down?

   Answer The point here is that a proper mail agent, when presented with the address `acollins@cs.washington.edu` will look up the MX record of `cs.washington.edu`, and go to that host, rather than to `cs.washington.edu` itself, which may or may not even be a real machine (in this case it is an alias for `june`). Because I send my email from `tioga.cs.washington.edu`, it is conceivable that some email programs might take my address to be `acollins@tioga.cs.washington.edu` (although most should obey the headers). In fact the point of MX records is to make sure that it makes no difference which host within CSE people try to send my email to, because the MX records should all point to the same place.

   So if I run `dig @marge.cac.washington.edu cs.washington.edu any` and look for MX records, I get four:

```
cs.washington.edu.      86400   IN      MX      40 june.cs.washington.edu.
cs.washington.edu.      86400   IN      MX      10 mx2.cs.washington.edu.
cs.washington.edu.      86400   IN      MX      20 mx1.cs.washington.edu.
cs.washington.edu.      86400   IN      MX      30 trout.cs.washington.edu.
```

   a little bit of additional research shows that the numbers 10–40 are priorities, with lower being higher priority. Tiebreaking behavior is undefined, so it isn't surprising that all the values are distinct. So email should go to `mx2`, unless it isn't available, in which case it will fall back to the others.

   Curiously, howerver, `tioga` has a slightly different set of MX records:

```
tioga.cs.washington.edu. 86400  IN      MX      10 june.cs.washington.edu.
tioga.cs.washington.edu. 86400  IN      MX      20 trout.cs.washington.edu.
```

   so email sent there will go to `june` first. Since all of this works fine, it suggests that the four mailservers are capable of cooperating.

6. If you wanted to attack one of these name servers (meaning any that you found, not just those from parts 2 and 3), what attacks are possible and what would their effects be? To what extent do recursive query and/or negative caching support affect vulnerability to these attacks? (It should go without saying that you should **not** attempt to attack these servers in practice—you would likely get into serious trouble, and we would disavow any knowledge of your mission.)

   Answer There are many, many ways to attack nameservers. For starters, any attack that slows down or takes out the nameserver host will also stop the DNS server on the host. Vulnerability to these attacks depends on administration (likely good for the upper-level servers and not-so-good for some of the lower-level ones), and architecture. At one point the top-level servers were intentionally set up with architecture diversity, but I don't know if this is still true. Any attempt to figure it out by portscanning would likely be considered an attack in itself. In general, howerver, I will set aside full-host attacks, and focus on attacks within the DNS protocol.

DNS servers are, of course, subject to buffer overrun attacks, because they accept queries with data. This would be especially true for TCP connections, which can supply unlimited data, but even UDP allows very large datagrams, relative to the typical size of a domain name. This again is a general software security issue, rather than anything specific to the DNS protocol.

Zone transfer attacks: unless otherwise configured, anybody can connect and download a copy of any DNS server's database through a zone transfer. This can leak data that might be valuable, but it also ties up a chunk of server resources, and disproportionately on the server end.

Reverse-lookup attacks: DNS defines a parallel tree to look up names given IP numbers. Authority for the leaves of this tree belong to the owners of the IP numbers, who can pretend that their IP numbers map to any name they want, even if a forward lookup wouldn't get the same IP number. This can break any security scheme based on trusting certain hosts by name. Note that even doing a forward lookup after the reverse lookup won't help, for two reasons. For starters, there are legitimate reasons that a machine may have multiple IP numbers, and we might not expect that DNS would tell us all of them. But even worse, when responding to the inverse lookup, the attacker can supply additional records that will go into the cache and make the forward lookup succeed, without ever asking the authoritative server.

Spoofing requests: because requests are typically UDP datagrams, it is easy to spoof the source address. This can make basic DoS attacks easier to make untraceable.

For recursive servers, it also means that a server cannot tell if the domain being looked up belongs to the attacker, and therefore might try to return bogus data when the victim tries to continue the lookup. We could try to maximize the victim's use of resources by taking a long time to respond, stretching out the sequence of steps we make the victim follow, and/or returning abnormally large amounts of data to try to thrash the cache.

For servers that support recursive queries, there are even more potential attacks. For starters, the DoS potential is greater, because you can make the server consume more computational and network resources. In addition, you can attack the negative cache, by sending many invalid requests to thrash it. On it's own, I don't think this would be terribly effective, because the cached lookups are not that much slower than the uncached lookups. There are, however, some secondary effects that such an attack might have:

- If the negative cache shares memory with the positive cache, then we could thrash the regular caching behavior, which would be far more serious.
- If the server is heavily loaded, then it is possible that doubling the amount of time spent per connection could overwhelm connection state.
- It is possible that the additional compute work (that might have been saved were the negative cache working) could overwhelm the server, but recall that most of the time is spent in message delays, not computation.

As is always the case, other sorts of attacks may still be possible, and if I see any good ones I'll post them too.

7. Research (using your favorite method—mine is Google) how many `.com` registrations happen per day.

   Recall that we saw that a huge source of traffic to TLD servers is answering queries for bogus domain names. Let us consider a design where all DNS resolvers store the *complete* list of NS records for all second-level domain names (e.g. `yahoo.com`), and can therefore resolve all valid names *and* recognize all invalid names without contacting a TLD server (they would, of course, still need to talk to the second-level nameservers). To simplify the problem, we will consider only the `.com` domain (We can either imagine that we do the same thing for other top-level domains, or that we fall back on "normal" DNS for non-`.com` names; in either case we have much less data about the other top-level domains, so we won't worry about them.)

   Assuming that all second-level `.com` NS records have a two-day TTL (meaning that our new DNS must similarly ensure changes in second-level names are reflected within a two-day window), characterize the circumstances in which it would be cheaper (in terms of total network packets) to maintain and use this complete list of second-level names rather than interrogating the `.com` TLD server(s) directly.

Answer The basic architecture we are proposing is one where the top-level domain servers maintain an authoritative list of all second-level domains (as they do now), and serves out the entire list, on demand, to any resolver who wants it. We still (I think) want a client-driven system like this, rather than one where the servers know the resolvers and push out the data eagerly.

To avoid sending the entire database, most of which does not change very often, we would use diff-coding. The servers would maintain both the current database, and at least two days worth of change logs, detailing individual updates or groups of updates. Resolvers would only request the complete (timestamped) database once, and thereafter would request, at no more than a two-day interval, the list of updates. To make a request, a resolver would supply the timestamp of its most recent update, and would get back all updates after that time. If the server's update log doesn't go back that far, then the client would have to re-request the entire database, but we will assume that doesn't happen. Note that, if resolvers are to be able to change which server they talk to, then we will need some of the synchronization algorithms we have seen, to make sure that nothing slips through the cracks.

An update consists of the domain name and the new server for the domain (and the IP address of that server, otherwise we won't have saved the top-level server from getting hit). According to Mockapetris and Dunlap, labels in the DNS hierarchy are limited to 63 characters, so we can easily fit what we need from an NS record in 75 bytes (domain name, omitting the .com, plus IP address for nameserver plus a few bytes to spare), so we could fit 20 mappings in a 1500-byte packet.

If we assume about 45,000 updates per day (near the top end of the range of values people found), then, every two days, each resolver must download 4,500 packets of updates and apply them to the local database. We assume that the local storage and computation are negligible. So basically, an individual resolver would be better off using this scheme if it gets more than 2,250 requests per day for second-level domains which wouldn't be cached in the current DNS.

Whether or not *that* is reasonable depends on three factors:

- The popularity of domain names. This will almost certainly be a heavy-tailed distribution, so it's likely that 10–50 percent of total requests will be for unpopular domains, which wouldn't be in the cache.
- The number of clients a resolver serves. Typically there is about a resolver per subnet, so this could be anywhere from 10–1000 clients.
- The rate at which clients request domain lookups. For web-browser clients, this is the number of distinct sites they visit. We will approximate this as the rate at which humans jump to new places.

If we take 10–50 percent as the range of unpopularity, and 10–1000 as the range of population size, then we get a range of 4.5–2,250 super-requests/day/client, or 0–1.6 requests/min/client, so this is not out-of-whack, although a resolver with a small number of clients would probably prefer the current system.

Also note that this is calcuated in *packets* rather than *bytes*. Our packets are all mostly full, while typical DNS packets are mostly empty. Therefore, were we to repeat this analysis using bytes transferred as the metric, we would likely come up with a required request rate 10–20 times greater than we did, which would mean that most smaller resolvers probably would not benefit.