CS305 · Computer Networking

Program Assignment 1

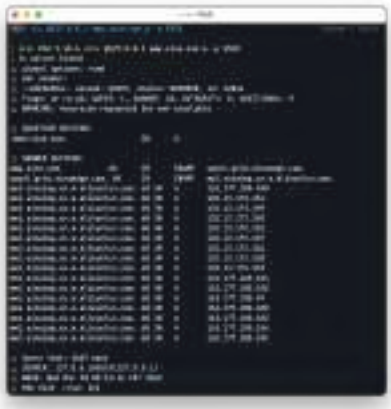
何泽安 (He Zean)
12011323
Mar. 27, 2022

Contents

1 Screenshots to Prove Workability

1.1 Correctness

The following five test cases all runs as expected, the result is similar to the reference answer and *dig @114.114.115.115*. Please also notice the warning: *recursion requested but not available*, which was what we want.



1.2 Read from Cache

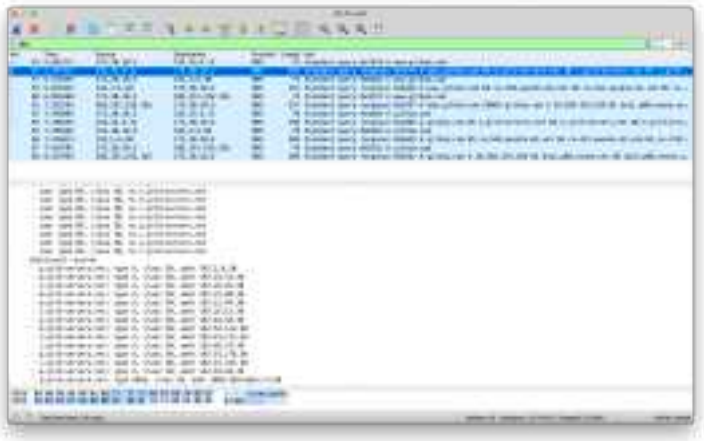
Please see section 2.1.4. But note that, if a query does need a long time to finish, dig may close the connect due to timeout. But the next same query can fortunately hit the cache and get the result.

1.3 Error Handling

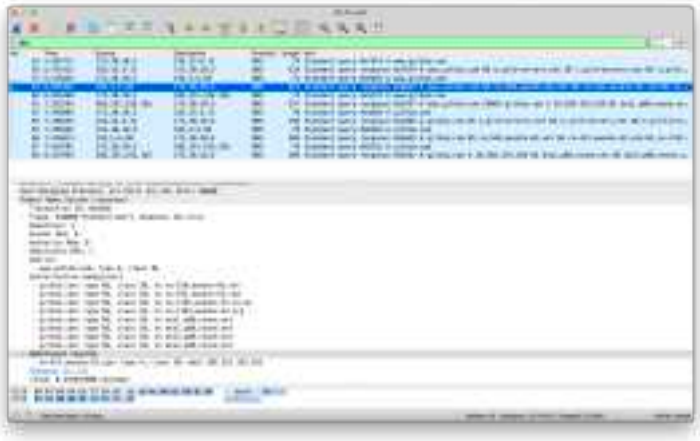
When the network condition is awful, DNS queries may often got timeout. The local DNS server will keep retrying several times (here is set as 10), if it still fails, return no answer, and set the status as 3: NXDOMAIN to indicate an error. As for wrong query names, it will not retry (unless a previous timeout before knowing the query name is not found) and return not found, with status NXDOMAIN.

1.4 Iterative Query

The following Wireshark screenshots analyzes the process of an iterative query. The DNS server (172.20.10.3) first ask the root server (192.33.4.12) about the IP address of www.github.com, but it doesn't know, and provides TLD servers' NS records in authoritative nameservers' section, indicate our DNS server to take the same query name to the TLD servers. Note that in the additional section, the root server also provides their A records, we can therefore skip a query for TLD servers and straightly go for them.



Our server happily take www.github.com to ask TLD 192.5.6.30, it also have no idea about the IP address, but it redirected our server to the authoritative server of GitHub, ns-520.awsdns-01.net etc., and provided an A record for one authoritative server, let's go for it.



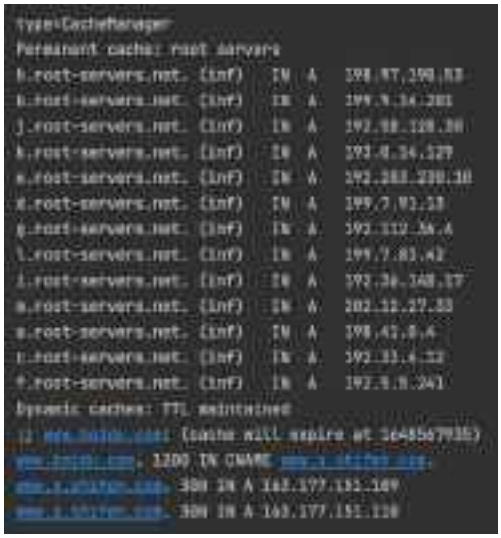
Now we come to ns-421.awsdns-52.com, this time, it gives us a CNAME record and an A record. Getting the IP address we want, we can happily ends the query and return an answer.

2 Implementation Detail

2.1 Maintain a Cache

2.1.1 Human-readable

I directly store a list of RRset related to the query name, and provide the magic function *__str__(self)* to make it more readable. In the code I submitted, I commented lines that print the cache manager, but here's a demonstration.



2.1.2 Record New Queries

```
1  if query_cache := self.cache_manager.read_cache(domain_name):
2      logger.info('read from cache')
3      return ReplyGenerator.reply(income_record, query_cache)
4  else:
5      if resp := self.query(domain_name, self.source_ip, self.source_port):
6          self.cache_manager.write_cache(domain_name, resp)
7          return ReplyGenerator.reply(income_record, resp)
8      else:
9          return ReplyGenerator.reply_not_found(income_record)
```

Please check line 6, in such case, there's no previous cache, s.t. the program runs into the *else* block and query the domain (iteratively, from the root servers). If the query fails (wrong domain, or timeout too many times), a reply indicates the error will be returned, otherwise, it save the list of RRset into cache and return a DNSRecord with RRset items inserted into the answer section.

2.1.3 Manage Expired Caches

To provide a better service, and to avoid the cache dict being too heavy, we tend to delete the expired cache items dynamically, say, since TTL is in the unit of second, we run a check once a second, which checks the timestamps indicating the min TTL in a group of RRsets and delete the expired ones. Since the cache is not that large indeed, this $O(n)$ check is fair enough.

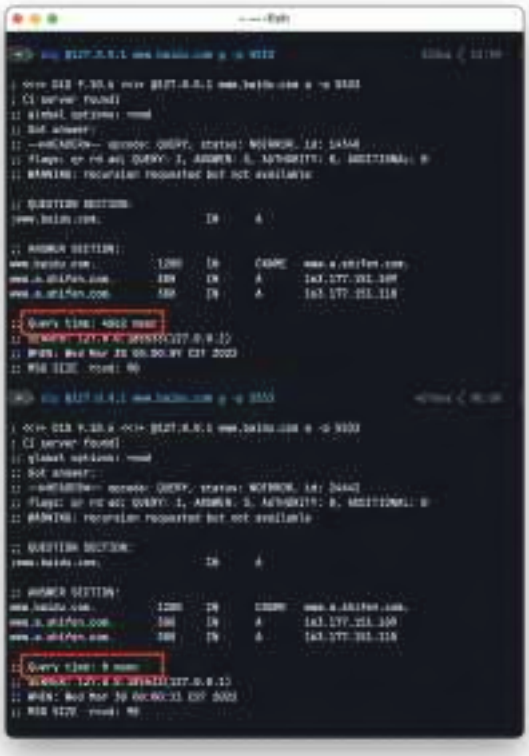
```
1  def __init__(self):
2      ...
3      # stand-alone thread for checking and deleting the cache that passes TTL
4      _thread.start_new_thread(self.daemon_ttl, ())
5
6  def daemon_ttl(self):
7      while True:
8          timestamp = round(datetime.now().timestamp())
9          for domain in list(self.cache.keys()):
10             if timestamp > self.cache[domain][1]:  # cache[domain][1] is the timestamp when the min TTL pass
11                 del self.cache[domain]
```

```
12          time.sleep(1)  # ttl manager updates pre second
```

Plus, as we all know, thanks to GIL, this build-in dict is thread-safe, which means the daemon thread shall not mess up our cache. But to be more safe, I check the TTL timestamp again in *read_cache*.

2.1.4 Reply a Cache

As the screenshot shows, the first query cost 4062ms, but the second query is so fast that dig reports it cost 0ms. Also, the console printed a log *read from cache*.



2.2 Multithreading

The socket itself, is indeed blocking, what we need to do it the let the query itself be async, and ensure that socket messages' sending will not crash if in such OS that socket is not thread safe.

The first question is easy to solve. Since the class DNSHandler is iterated from threading.Thread, we can do such modifications in DNSServer#start:

```
def start(self):
    try:
        while True:
            message, address = self.receive()
            if message == self.last_query:
                continue
            self.last_query = message
```

```
8
9               thread = threading.Thread(target=self.dns_handler.handle,
10                                        args=(message, address, self.socket, self.lock))
11              thread.daemon = True
12              thread.start()
13      except KeyboardInterrupt:
14          logger.info('bye')
15          exit(0)
```

Here's a trick that records bytes *last_query*, since the dig client will send several exactly same packages if it doesn't receive the response after some time, but there's already a thread working on the query. Thus we just skip the query. But if dig sends another query that have the same query name, it is still okay, since the id in the message is different.

As for the socket's thread safe, a simple thread lock can be applied:

```
1 @staticmethod
2 def reply(address, response: DNSRecord, sock: socket, lock: threading.Lock):
3     with lock:
4         sock.sendto(response.pack(), address)
```