ECSE 416 Experiment 1 Report

Lab Group 2

Jacob Shnaidman 260655643

Filmon Abraham 260637863

Introduction

The objective of this experiment was to create a Domain Name Server (DNS) client and experiment with the DNS protocol. This lab resulted in a DNS client that is capable of Address, Name Server and Mail Server queries, as well as handling Canonical Name Records in the response. The experiment showed that different domain name servers sometimes report different Internet Protocol (IP) addresses for a given domain name.

The main challenges in implementing the DNS client were to implement the handling of compressed names in the response and to implement correct error handling of all corner cases. The handling of the compressed names required a recursive algorithm which could call itself on a different location in the receive buffer if it detected an offset (pointer). The correct error handling required careful consideration of the different ways in which the DnsClient may be called or the response might be structured.

DNS Client Program Design

There is a single class for the DNS client program, and one extra class representing the DNS response header. The scope of this DNS client didn't require extensive object-oriented design. In fact, the whole client could have been written in a single class/script. The *DnsResponseHeader* class exists so that the *parseResponseHeader* method could return a single object, instead of say, an array of values representing whether the response is authoritative, how many answers there are, etc. The method *parseResponseMethod* did not necessarily need to be a method, but it was designed to be as modular as possible.

The DNS Client class is split up into various methods to maximize reusability. Although this code will probably not be reused in this class, it is good practice to modularize code as much as possible so as to separate concerns as much as possible. By keeping code in different methods, if a method is used more than once, bugs that occur in the logic contained in the method only have to be fixed once (in the method). They are also much more easily refactored and moved around. For example, there is a *parseRecord* method that is reused multiple times depending on how many records are in the response. It also makes the code much easier to read as the function itself describes what the following code does.

The Structure of the code is as follows. The code begins at the main function. It performs an interpretation of the command line arguments and sets the appropriate variables based on user input. Afterwards, it creates a UDP receive socket as well as the send socket according to IP address and port defined by the user. It creates the send and receive buffers and then calls *buildQueryHeader* and *buildQuery* methods to populate the send buffer with the correct bits. The response is sent out and a timer starts to track the response time. After a response is received, possibly after a certain amount of retries, the response header is parsed by calling *parseResponseHeader*. This function returns a *DnsResponseHeader* object which indicates how many records there are to parse. This object stores the QDCOUNT, ANCOUNT, NSCOUNT, ARCOUNT, and whether the response is authoritative. Afterwards, each record is parsed by calling *parseRecord* a number of times based on how many records are specified in the response header.

The DNS Client handles errors by throwing exceptions when errors occur. For example, if the bit in the response header indicates that recursive queries are not supported, this throws an exception. If the user is missing a required field when calling the DNS client, like the IP address of the DNS server, then an exception is thrown. If a record not found for a certain domain name, then NOTFOUND is returned as the exception message. If the RDATA type of the response header is not of one of the expected types for this assignment, then an exception is thrown like so:

```
ERROR\tUnexpected RDATA type in the response (" + String.format("%X", responseType)+
")"
```

The source code makes use of certain Java libraries to facilitate the development of the DNS client. For example, the code uses the *DatagramSocket* and *DatagramPacket* libraries. *DatagramSocket* objects allow the program to easily initialize a socket with a specific timeout in an object-oriented way. Similarly, the *DatagramPacket* library allows the program to abstract away the implementation of a UDP packet and simply specify a buffer, buffer length, server IP address and server socket to construct a *DatagramPacket* object. The source code also makes use of the *ByteBuffer* library to construct the dns query and parse the response. This library allows one to insert or read data byte-wise or 16 bits at a time. This saves time and reduces errors that can occur from doing bitwise operations.

The program handles compressed domain names by using a recursive method called *parseLabels*. This method receives a ByteBuffer as argument which essentially provides a buffer and an index at which the domain name begins and returns the domain name as a String. It works by doing the following in a loop. It checks the first byte to see if it begins with 0b11. If it does, then it is a pointer to a previous part in the DNS response. Otherwise it is a length *n* describing how many ascii characters follow. If it does not begin with 0b11, then the next *n* characters are added to the response as the domain name. The loop then continues on until a pointer is reached or a 0 is reached. If it does begin with 0b11, then it saves the current position in the buffer, changes the current position to the one indicated by the pointer, and calls itself recursively with the ByteBuffer with a modified position as argument. The return value of this recursive call is appended to the domain name that will be returned. Afterwards, the original position of the buffer that was saved is restored by setting the ByteBuffer's position to the saved position. Finally, the domain name is returned.

Testing

The DNS client features were tested using bash scripting. To ensure that different arguments were sufficiently tested, a bash script was made to create different types of requests, including -mx, -ns, -t, -r, etc. For example, -mx got the mail servers of gmail.com, -ns got the name server of mcgill.ca, etc. The bash script ensured that these tests could be replicated easily and that all test cases were covered. Retries and timeouts were tested by querying google DNS on a port that it doesn't respond to, resulting in no response.

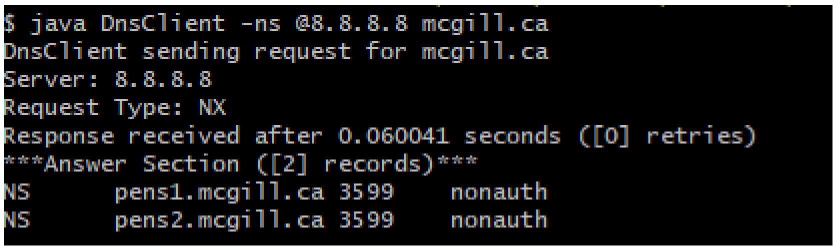
The DNS client formatting was verified by using WireShark to view the outgoing and incoming DNS requests. WireShark made it easy to see the meaning behind each bit to verify that if the request type was for an IP address, that indeed a request for an IP address was sent out. For example, it shows the QNAME in the query header as a string. It also clearly distinguishes all the particular flags and their values. It was also useful to verify that the compression was being interpreted correctly, as it allows one to see the bits in the response directly. WireShark was used to observe the individual test cases contained in the bash script mentioned above, ensuring that the formatting was proper.

The only features that were not tested completely were the -p option and the auth/nonauth parameter. The -p option was difficult to test since it was difficult to find a DNS server which responded on a port that was not 53. The -p option was validated using the debugger to observe that it actually did change the value of the port to be queried, but ultimately it proved difficult to test whether it actually could receive a response from a DNS server operating on a different port. The auth/nonauth parameter was also difficult to test in the response, since it was hard to find an authoritative DNS server. Those servers that were found to be authoritative for a certain domain name did not support recursive queries, which was the only type of query that this client could issue. It could only be verified that the authoritative bit in the header was being interpreted correctly in debug mode by changing the value in the buffer at runtime.

Experiment

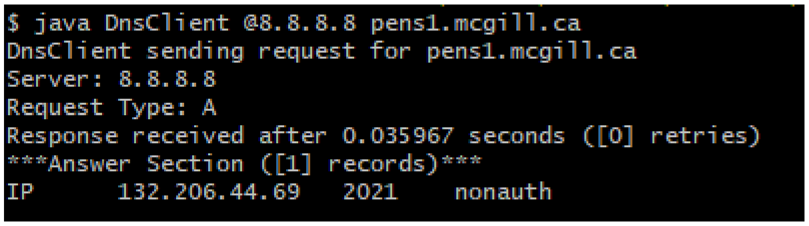
Performing a name server query on mcgill.ca returns two different domain name servers. `pens1.mcgill.ca` and `pens2.mcgill.ca`. These domain names have IP addressses 132.206.44.69 and 132.206.44.70 respectively. The response to the name server query was a string. I was expecting this to be an IP address since I assumed it would be more efficient to just store the IP rather than the domain name of the dns server. The name server lookup response is shown below in figures 1 to 3.

Figure 1: Querying Google Public DNS for mcgill.ca



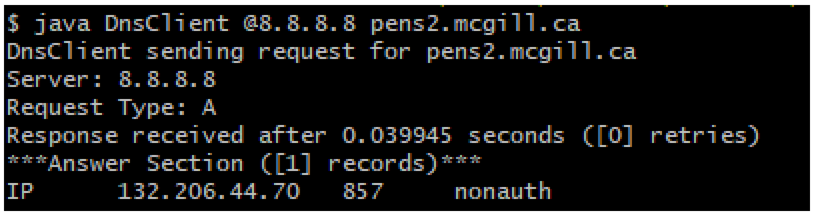
```
$ java DnsClient -ns @8.8.8.8 mcgill.ca
DnsClient sending request for mcgill.ca
Server: 8.8.8.8
Request Type: NX
Response received after 0.060041 seconds ([0] retries)
***Answer Section ([2] records)***
NS      pens1.mcgill.ca 3599    nonauth
NS      pens2.mcgill.ca 3599    nonauth
```

Figure 2: Querying Google Public DNS for the IP of the first McGill DNS server



```
$ java DnsClient @8.8.8.8 pens1.mcgill.ca
DnsClient sending request for pens1.mcgill.ca
Server: 8.8.8.8
Request Type: A
Response received after 0.035967 seconds ([0] retries)
***Answer Section ([1] records)***
IP      132.206.44.69   2021    nonauth
```

Figure 3: Querying Google Public DNS for the IP of the second McGill DNS server



```
$ java DnsClient @8.8.8.8 pens2.mcgill.ca
DnsClient sending request for pens2.mcgill.ca
Server: 8.8.8.8
Request Type: A
Response received after 0.039945 seconds ([0] retries)
***Answer Section ([1] records)***
IP      132.206.44.70   857     nonauth
```

Tables 1 and 2 below show the result of querying A records for McGill DNS and Google DNS respectively. Most websites shared the same IP addresses, however www.google.com, www.bbc.com, www.amazon.com and www.twitter.com had different IP addresses returned between using Google DNS and McGill DNS. This is because the domain name servers had different IP addresses cached for those domain names. This might be because the domain name servers are in different locations and exist on different points on the network. The IP address for each domain name might be different because the IP address that the domain name maps to is probably closer geographically.

Table 2: Querying using McGill DNS server

Website	Response
www.facebook.com	31.13.80.36
www.google.com	172.217.13.174
www.linkedin.com	108.174.10.10
www.yahoo.com	72.30.35.10
www.bbc.com	151.101.136.81
www.amazon.com	104.93.186.116
www.twitter.com	104.244.42.129

Table 2: Querying using Google DNS server

Website	Response
www.facebook.com	31.13.80.36
www.google.com	172.217.13.196
www.linkedin.com	108.174.10.10
www.yahoo.com	72.30.35.10
www.bbc.com	151.101.32.81
www.amazon.com	52.85.104.27
www.twitter.com	104.244.42.193

Discussion

It was observed that during the experiment, that Google DNS (8.8.8.8) would not respond and that all requests to that DNS would timeout while on the McGill network. Conversely, no requests to the McGill DNS (132.206.85.18) could be made from outside the network. The latter case is most likely because of security concerns. They most probably wish to limit access to their domain name server to people within the network to prevent unidentified attacks. Attacks from within the network are harder to pull off successfully. In the former case of preventing DNS requests to Google DNS servers, this might be because certain McGill Domain Names don't resolve on other domain name servers. To further investigate this issue, more DNS requests should be made using different domain name servers.

As noted in the Experiment section, the IP addresses that resolved from A records on different domain name servers were different. This was most probably because of the geographic location of the domain name servers relative to the location of the servers hosting the web sites for a domain name. If domain names were associated with IPs across the world, then it might induce a noticeable lag in resolving web pages. Since companies strive to lower the load times of their pages, if domain names could only resolve to an IP in a single location, then perhaps web pages would have different domain names depending on one's location. This is clearly not the case.

Two McGill DNS servers were found. The name server lookup responded with a two different domain names, each with their own unique IP address. These IP addresses seem to exist on the same subnet and only differ by one in the last index of their addresses. This makes sense considering that they're both associated with McGill.

There weren't many challenges implementing this DNS client, it was fairly straight forward. There were times where the response wasn't as expected because a bitwise operation was incorrect. This can be avoided by carefully thinking about the operation, and immediately checking the result in debugger. The code could be improved by having used a key value pair for the response types. Instead of having a *responseTypeString* and a *responseType* (hexadecimal code), it would have been cleaner to use a HashMap for example.