ECSE 316 Assignment 1 Report
Group 6
Victor Micha: 260966340
Antoine Dangeard: 260962884

1. Introduction

The objective of this assignment is to gain a better understanding of what a DNS is by developing a DNS client using sockets. DNS clients map domain names such as mcgill.ca to IP addresses. The DNS client must interact with the DNS server through socket programming, which allows processes on different machines to communicate. The DNS client issues a request for a certain domain name and waits for the DNS server to respond in a given time interval. In this assignment, the client must perform a certain amount of additional requests if the first one fails. If the DNS server returns a response within the maximum amount of request retries, then the client must parse the response and neatly display it to STDOUT. The DNS client must support queries for IP addresses, mail server, and name server records. Additionally, the DNS client must display appropriate error messages depending on the error that arises. Namely, if the response from the DNS server is not of the expected format, cannot be interpreted, or does not match the query sent by the DNS client, then the corresponding error message(s) must be printed to the screen.

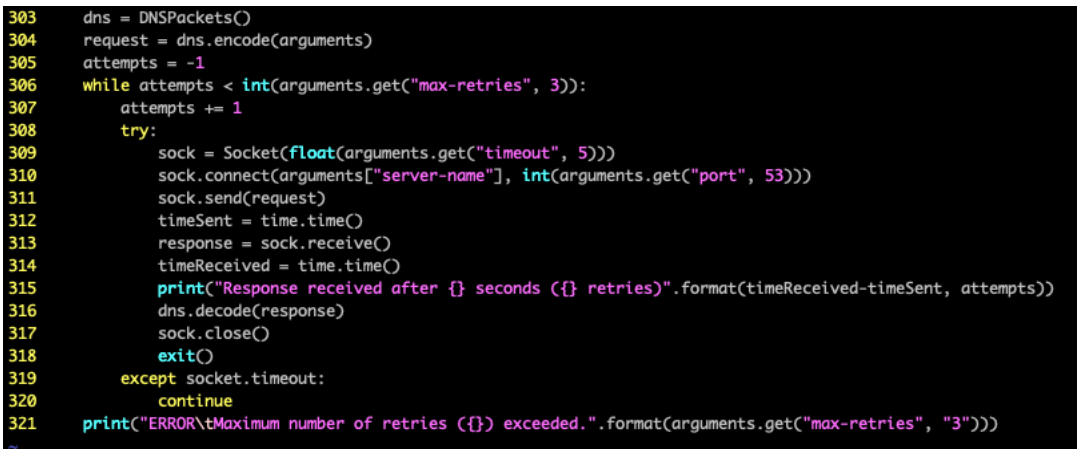
The main challenges were reading over all the documents explaining the assignment in detail, understanding what a DNS client is, as well as implementing the client efficiently with as few bugs as possible. Once the first two challenges were conquered, all that was left to do was start coding. Python makes it rather easy to ensure that the command line arguments are properly formatted. If they are, storing them in a convenient data structure was an easy next step. The website for socket programming in python was very helpful and greatly reduced the time spent on this aspect of the assignment. By consequence, we had more time to focus on issuing the request and parsing the DNS packet (if the request was successful). Parsing the DNS packet was most definitely one of the harder aspects of the assignment, as simply understanding how a DNS packet is structured is time consuming and at times confusing. Encoding the request as a DNS packet was rather straightforward, as only a header and question section was needed. Using the built in Byte objects in python was greatly helpful. The main result is that we have a much better grasp on what the DNS protocol is, and we now have our own tool that enables us to interact with the DNS infrastructure.

2. DNS Client Program Design

Our DNS client program is separated into two classes: DNSPackets and Socket. The functionality of each class will be discussed in detail below.

Firstly, the DNS client program first decodes the user CLI input into corresponding argument variables, ensuring the validity of these along the way. If the inputs are of the wrong format or some are missing, then a corresponding error message is printed for these invalid/missing arguments. If the arguments are correct however, they are formatted into a proper DNS request through the encode function of the DNSPackets class. This function also prints the first three lines of the output (indicating the destination, type, and timing of the request being sent).

The Socket class is used to connect to the right port of the specified server and to communicate the request with that server, using the class's "connect", "send", and "receive" functions, as can be seen below in Figure 1.

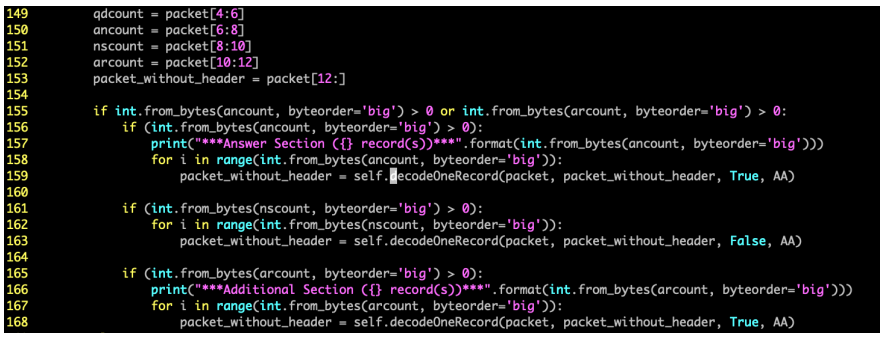


```
303     dns = DNSPackets()
304     request = dns.encode(arguments)
305     attempts = -1
306     while attempts < int(arguments.get("max-retries", 3)):
307         attempts += 1
308         try:
309             sock = Socket(float(arguments.get("timeout", 5)))
310             sock.connect(arguments["server-name"], int(arguments.get("port", 53)))
311             sock.send(request)
312             timeSent = time.time()
313             response = sock.receive()
314             timeReceived = time.time()
315             print("Response received after {} seconds ({} retries)".format(timeReceived-timeSent, attempts))
316             dns.decode(response)
317             sock.close()
318             exit()
319         except socket.timeout:
320             continue
321     print("ERROR\tMaximum number of retries ({}) exceeded.".format(arguments.get("max-retries", "3")))
```

Figure 1: Main while loop of DNS client program

The time taken for the response to be received is measured as well as the number of retries (if the initial request fails). If the time taken is longer than the socket timeout time, then the number of attempts is incremented by 1, and another iteration of the while loop is performed (as long as the number of attempts is less than the max number of retries). As can be seen on line 321 of Figure 1, if the number of attempts is greater or equal to the number of retries, then a corresponding error message is printed. If the time taken to send the request and receive the packet is less than the socket timeout, then the packet received is handled appropriately (decoded and its contents displayed). As can be seen in Figure 1, the DNSPackets' function "decode" is then called with the response as an argument. It is in this function that error messages are printed if the RA or RCODE or CLASS bits of the DNS packet have incorrect values and where the neatly formatted output is printed to STDOUT if no errors occur.

As can be seen in Figure 2, the decode function itself calls the "decodeOneRecord" function multiple times by looping over the remaining "unseen" packet data, and taking the next piece of information off the top. Thus, first, the header is removed from the packet and decoded. The "headless" packet is then passed to a for loop that "takes off" (decodes and removes) each Answer Record one by one from the beginning of the packet. Another for loop does the same for any Authority Records that are at the beginning of what remains of our packet, and finally a for loop with the same purpose decodes and Additional Records remaining in the packet. With this infrastructure, we need only to worry about decoding one record at a time, and printing any important information we decoded in the process. This program therefore can successfully decode a packet containing multiple records of different types.



```
149         qdcount = packet[4:6]
150         ancount = packet[6:8]
151         nscount = packet[8:10]
152         arcount = packet[10:12]
153         packet_without_header = packet[12:]
154
155         if int.from_bytes(ancount, byteorder='big') > 0 or int.from_bytes(arcount, byteorder='big') > 0:
156             if (int.from_bytes(ancount, byteorder='big') > 0):
157                 print("***Answer Section ({} record(s))***".format(int.from_bytes(ancount, byteorder='big')))
158                 for i in range(int.from_bytes(ancount, byteorder='big')):
159                     packet_without_header = self.decodeOneRecord(packet, packet_without_header, True, AA)
160
161             if (int.from_bytes(nscount, byteorder='big') > 0):
162                 for i in range(int.from_bytes(nscount, byteorder='big')):
163                     packet_without_header = self.decodeOneRecord(packet, packet_without_header, False, AA)
164
165             if (int.from_bytes(arcount, byteorder='big') > 0):
166                 print("***Additional Section ({} record(s))***".format(int.from_bytes(arcount, byteorder='big')))
167                 for i in range(int.from_bytes(arcount, byteorder='big')):
168                     packet_without_header = self.decodeOneRecord(packet, packet_without_header, True, AA)
```

Figure 2: Inside of DNSPackets decode function

Another feature of our program is how it deals with compression in label names and/or QNAME-formatted fields in the packet. A recursive approach was used, such that the recursive function needs only to decode the sequence of characters it was told to start at, and recursive calls

could deal with the complexity of reading labels from other places in the code and appending that to a label's final value.

3. Testing

To test each feature of our program, as we built the different elements making up the code, we ensured that what was written previously was fully functional and robust before adding on another layer. For example, when reading in the command line arguments, we made sure to test the code and error-detection in arguments with as many edge cases as we could think of before deciding to move on to packet decoding, encoding, socket communication, etc. For packet encoding and decoding, it was more difficult to generate edge cases, since the variety of the input was so large, so we settled on printing the values of variables and packets that were being encoded or decoded, and ensuring that every time what we saw print was a value that made sense for the case at hand. It became more difficult to test this out for the edge cases, but through finding different addresses to test on and printing to STDOUT, we were able to convince ourselves of the robustness of the code before moving on to error-detection in packets, decoding compressed labels, etc. Once again, for these challenges, we decided unit testing on various addresses for various different record types would give a broad enough set of cases to eliminate the possibility of our code having an edge-case error somewhere. This was something we can not demonstrate, but have made ourselves confident about through our unit testing.

4. Experiment

EXPERIMENT 1:

```
C:\Users\antoine\Documents\SCUOLA\WINTER 2023\ECSE 316\A1\ECSE316_DNS>python3 DnsClient.py -ns @8.8.8.8 mcgill.ca
DnsClient sending request for mcgill.ca
Server: 8.8.8.8
Request type: NS
Response received after 0.023802280426025390 seconds (0 retries)
***Answer Section (1 record(s))***
MX      mcgill-ca.mail.protection.outlook.com   10      1927    nonauth
```

EXPERIMENT 2: (follow up to mcgill.ca name server request)

```
C:\Users\antoine\Documents\SCUOLA\WINTER 2023\ECSE 316\A1\ECSE316_DNS>python3 DnsClient.py @8.8.8.8 mcgill-ca.mail.protection.outlook.com
DnsClient sending request for mcgill-ca.mail.protection.outlook.com
Server: 8.8.8.8
Request type: A
Response received after 0.058955907821655273 seconds (0 retries)
***Answer Section (2 record(s))***
IP      104.47.75.228   10      nonauth
IP      104.47.75.164   10      nonauth
```

Expectation: Since McGill's email system is hosted with Microsoft Outlook, it would make sense that their email server would also be an outlook server
The results match with our expectations.

EXPERIMENT 3: google.com

```
C:\Users\antoine\Documents\SCUOLA\WINTER 2023\ECSE 316\A1\ECSE316_DNS>python3 DnsClient.py @8.8.8.8 google.com
DnsClient sending request for google.com
Server: 8.8.8.8
Request type: A
Response received after 0.015426635742187500 seconds (0 retries)
***Answer Section (1 record(s))***
IP      172.217.13.206  37      nonauth
```

EXPERIMENT 4: amazon.com

```
C:\Users\antoine\Documents\SCUOLA\WINTER 2023\ECSE 316\A1\ECSE316_DNS>python3 DnsClient.py @8.8.8.8 amazon.com
DnsClient sending request for amazon.com
Server: 8.8.8.8
Request type: A
Response received after 0.014767885208129883 seconds (0 retries)
***Answer Section (3 record(s))***
IP      205.251.242.103 11      nonauth
IP      54.239.28.85    11      nonauth
IP      52.94.236.248   11      nonauth
```

EXPERIMENT 5: ebay.com

```
C:\Users\antoine\Documents\SCUOLA\WINTER 2023\ECSE 316\A1\ECSE316_DNS>python3 DnsClient.py @8.8.8.8 ebay.com
DnsClient sending request for ebay.com
Server: 8.8.8.8
Request type: A
Response received after 0.016542673110961914 seconds (0 retries)
***Answer Section (3 record(s))***
IP      209.140.136.254 39      nonauth
IP      209.140.139.232 39      nonauth
IP      209.140.136.23  39      nonauth
```

EXPERIMENT 6: yahoo.com

```
C:\Users\antoine\Documents\SCUOLA\WINTER 2023\ECSE 316\A1\ECSE316_DNS>python3 DnsClient.py @8.8.8.8 yahoo.com
DnsClient sending request for yahoo.com
Server: 8.8.8.8
Request type: A
Response received after 0.014252185821533203 seconds (0 retries)
***Answer Section (6 record(s))***
IP      74.6.231.20     1532    nonauth
IP      98.137.11.163   1532    nonauth
IP      74.6.143.25     1532    nonauth
IP      98.137.11.164   1532    nonauth
IP      74.6.143.26     1532    nonauth
IP      74.6.231.21     1532    nonauth
```

EXPERIMENT 7: tiktok.com

```
C:\Users\antoine\Documents\SCUOLA\WINTER 2023\ECSE 316\A1\ECSE316_DNS>python3 DnsClient.py @8.8.8.8 tiktok.com
DnsClient sending request for tiktok.com
Server: 8.8.8.8
Request type: A
Response received after 0.015581130981445312 seconds (0 retries)
***Answer Section (4 record(s))***
IP      13.225.195.66   60      nonauth
IP      13.225.195.67   60      nonauth
IP      13.225.195.117  60      nonauth
IP      13.225.195.48   60      nonauth
```

EXPERIMENT 8: facebook.com

```
C:\Users\antoine\Documents\SCUOLA\WINTER 2023\ECSE 316\A1\ECSE316_DNS>python3 DnsClient.py @8.8.8.8 facebook.com
DnsClient sending request for facebook.com
Server: 8.8.8.8
Request type: A
Response received after 0.014907121658325195 seconds (0 retries)
***Answer Section (1 record(s))***
IP      31.13.80.36     121     nonauth
```

EXPERIMENT 9: open-ai.com

```
C:\Users\antoine\Documents\SCUOLA\WINTER 2023\ECSE 316\A1\ECSE316_DNS>python3 DnsClient.py @8.8.8.8 open-ai.com
DnsClient sending request for open-ai.com
Server: 8.8.8.8
Request type: A
Response received after 0.11884617805480957 seconds (0 retries)
***Answer Section (1 record(s))***
IP      64.190.63.111   300     nonauth
```

A DNS server helps to translate domain names into IP addresses so that we don't have to remember these complicated numbers when trying to access webpages on the internet. The DNS client will issue a query to the server containing a domain name. The server will then interact with other servers to find the domain name's associated IP address. The server hence acts as a client when querying other servers for the IP address. If the IP address of the domain name in the query is cached in the first server contacted by the client, then this server can simply respond directly to the client with the value of the IP address held in its cache. Therefore the server will not have to query other servers to find the IP address of the domain name if it is already in its cache, greatly speeding up the process of retrieving a domain name's IP address.

5. Discussion

The main results or learning outcomes of this assignment are that we learned a great deal about the DNS client server architecture as well as the low level design of DNS packets. Without knowing which bits in a packet mean what, it can be hard to truly understand what is being transmitted in each packet. We also learned the importance of DNS servers and their role in the internet, specifically how necessary they are in order for IP addresses to be abstracted from humans. Lastly, we also learned the difference between recursive and iterative DNS queries. For the former, the DNS client issues a request to the DNS server, who in turns issues requests to another server, which will request another server, etc. until the corresponding IP address is found. In this case notice that the client only interacts with one server. However, for iterative DNS queries, the DNS client will issue requests to servers one by one, each time receiving a pointer to the next server who might have the DNS pair, until the correct IP address translation is found.

One main challenge we faced was the ambiguity in the format of data received from/expected by the DNS servers. We often found ourselves confused as to whether our value had actually been converted into bytes or not, and if we were even on the right track at some points. It was difficult to validate our efforts until a packet could successfully be sent, received, and decoded. This left room for ambiguity in what we were doing incorrectly or not, and made it difficult to move forward at the beginning of the assignment until a significant amount of code had been written. This problem could be addressed in the future by perhaps providing clearer guidelines on what a correctly formatted packet looks like in code, and having more concrete examples and step-by-step “value checks” (print statements to ensure that each step has successfully executed) so that code can be written and confirmed as correct without the need for many other features to be implemented. Another way this problem could be addressed (specifically for Python, this is a problem we encountered) is by ensuring that when a byte-value is printed in hexadecimal format, individual bytes are not converted into their corresponding ASCII characters unless explicitly requested to do so. This would make it much easier to read/understand byte arrays which were printed to STDOUT, and make it much clearer if labels were being correctly encoded into bit strings.