Samantha Handal 260983914
Amélie Barsoum 260988658

October 2nd, 2023

ECSE 316 - Assignment 1 Report - Group 6

I. Introduction

In this assignment, our goal was to better understand the DNS client-server architecture and protocol, and build a process for users to request the IP address of a specified domain name, and consequently use the IP address to render the contents of the specified website. We also wanted to implement options for the user to customize their request, like choosing the type of DNS records that can be requested, such as A (Address), NS (Name Server), and MX (Mail Exchange); as well as adjusting the number of retries, the timeout time and the port. We wanted our code to correctly interpret the DNS responses, including the handling of domain name compression, and we used the timeouts and additional tries to handle the cases where there may be a slow server response, or none at all.

For us, the main challenge of this assignment was parsing the DNS response and more specifically identifying each section in the response, such as the header, question, answer, etc. Another challenge was building the DNS packet compression function, where we had to differentiate between the 3 possible cases of compressed domain names in DNS responses. We first needed to master the recognition of when domain names were referenced by a pointer, and based on this pointer, find the location where the domain name is defined. Finally, another challenge we faced was recursively finding the pointer in the sequence in the `label_to_string` function, where we had to consider the case in which a pointer points to a sequence of labels ending with a pointer. The main result of this assignment is a DNS client which can successfully query servers and parse responses, handle possible errors, use timeouts and max tries, as well as interpret compressed domain names.

II. DNS Client Program Design

a. Describe the design of your program

Rather than structuring the program into multiple classes, we structured it into a series of functions. First, we have the `main` method, which takes care of the parsing command line arguments and throws an error if the user inputs bad arguments. This information is then passed to `send_dns_query`, which returns the response packet and feeds it to `parse_dns_response`.

`Send_dns_query` takes care of the sending packet logic, which takes as arguments the server, the name, query type, timeout, max amount of tries for a response, and the port. Using the socket module, the socket is initialized, the timeout is set and the timer is initialized. The socket is initialized using SOCK_DGRAM and AF_INET. AF_INET is an address family, which uses a pair (host, port) as destination format, where host is a string representing either a hostname in internet domain notation like 'daring.cwi.nl' or an IPv4 address like '100.50.200.5', and port is an integer. SOCK_DGRAM is a datagram-based protocol (UDP). You send one datagram and get one reply and then the connection terminates. Then for the max amount of retries, `parse_packet` will build the sending packet, the socket will send the query, and, if the socket doesn't timeout, receive and return a response.

`parse_packet` takes as arguments the server name and the query type. Based on these arguments, the method creates a DNS query packet, including the Header and Question sections of the packet. Packets were built using a string representation of the hexdump, using hex digits. When sending

and receiving messages, we used library functions to convert the string representation of a hex number to a byte object, since the socket transmits messages in this format.

Once a response is successfully received, the response packet is fed to `parse_dns_response`, along with the header ID and question length. Here, the DNS response (result) parsed according to the scheme in dnsprimer.pdf. The parsing is organized by sections; header section, question section, answer section and additional section.

Every time we come across domain names in the QNAME, NAME, and RDATA fields, `packetCompression` function is called to detect whether the field is represented as a sequence of labels ending with a zero octet, a pointer, or a sequence of labels ending with a pointer, and also returns the whole QNAME, NAME, and RDATA section.

Fields ANCOUNT, ASCOUNT, and ARCOUNT dictate how many times we must iterate through the respective section to extract records. Then `label_to_string` takes care of the printing logic, converting the RDATA hexdump to legible strings, and finding and replacing the pointer in the hexdump to its corresponding label ending in '00'.

Below is a comprehensive list of all the errors handled by our code:

Input syntax, throws "Incorrect input syntax" and exits

- Validate flags: throw error if both flags given, -mx and -ns cannot be given simultaneously
- Validate IP address: from the user inputs, make sure they inputted
    - Must contain 4 parts separated by periods
    - Each part must be a number is in the range 0-255
- Validate domain name: from user inputs make sure the entered domain name has
    - Must contain only letters, numbers, hyphens, and periods.
    - Must not begin or end with a hyphen.
    - Label (sequence between periods) must not start or end with a hyphen.
    - Label must be between 1 and 63 characters long.
    - The entire domain name, including dots, should not exceed 253 characters.

Sending error

- throws "Socket timeout" until maximum number of retries is reached, throws "Maximum number of retries", then exits

Response packet, throw corresponding error then exits once whole packet is parsed

- Validate ID in header: query header id must match response header id, compare hex strings
- Validate QR bit in header: check if QR bit is 1, indicating packet is a response
- Validate RA bit in header: bit set or cleared by the server in a response message to indicate whether or not recursive queries are supported, must remain 1
- Validate RCODE in header: if RCODE is not 0, then throw the appropriate error
- Validate ANCOUNT in header: if no records in resource records in the answer section, throw not found
- Validate ARCOUNT in header: if no records in resource records in the additional section, throw not found
- Validate TYPE in answer and additional: check if record is A (IP address), NS (name server), MX (mail server), CNAME, otherwise type is unknown
- Validate CLASS in answer and additional: check if query class is Internet address

The handling of compressed domain names in DNS responses, as was briefly mentioned earlier, is handled by the function `packetCompression`. This takes the portion of the hexdump which contains domain names in the QNAME, NAME, and RDATA fields and outputs the domain name format type and

whole hex string of the domain name in its original format as well. Starting with the first 4 bytes, we convert to binary and check if this string of bits starts with "11", if it does this means the domain name is represented as a pointer, and we return. Otherwise this segment would be a part of the label, so we take the current two bytes and convert to int to iterate through the label, with the goal of finding the beginning bytes of the next. Here we check if the first two bytes are "00", if so then this means the domain name is a sequence of labels ending with a zero octet, and we return. Otherwise we check again for pointers, and if the series of labels ends in a pointer, we return.

This function is particularly useful in the `label_to_string` function so we know when to swap in labels for pointers when converting domain names to ascii characters.

III. Testing

To test our code, we used the command line format for invoking our DNS client, `python dnsClient.py [-t timeout] [-r max-retries] [-p port] [-mx|-ns] @server name`, we were able to test the different features of our DNS client, which we split into a few tests:

1) Test -mx flag : `python dnsClient.py -t 10 -r 2 -mx @8.8.8.8 mcgill.ca`

2) Test -ns flag : `python dnsClient.py -t 5 -r 3 -p 53 -ns @8.8.8.8 google.com`

3) Test the ability to parse response with multiple answers: `python dnsClient.py @8.8.8.8 www.youtube.com`

4)Test both -mx and -ns flags at once (error handling): `python dnsClient.py -t 5 -r 3 -p 5353 -mx -ns @192.168.1.10 example.com`

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ python dnsClient.py -t 5 -r 3 -p 5353 -mx -ns @192.168.1.10 example.com
ERROR    Incorrect input syntax: Both -mx and -ns cannot be given simultaneously.
```

5) Test different numbers of retries and timeout intervals by using non-existent servers: `python dnsClient.py -t 2 -r 3 -p 53 -mx @10.10.10.10 example.com`

6) Test with missing mandatory arguments: `python dnsClient.py @8.8.8.8` and `python dnsClient.py www.youtube.com`

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ python dnsClient.py @8.8.8.8
usage: dnsClient.py [-h] [-t TIMEOUT] [-r MAX_RETRIES] [-p PORT] [-mx] [-ns]
                    server name
dnsClient.py: error: the following arguments are required: name
```

Before performing these tests, we tried a simple type A request for www.mcgill.ca IP address using @132.206.85.18, which we tested on and off the McGill VPN. A response is received successfully with 0 retries on the McGill VPN, and no response is received after 3 retries off the McGill VPN. This was to ensure we ignore authoritative calls (image shows first try on the VPN, and second off the VPN)

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ python dnsClient.py @132.206.85.18 www.mcgill.ca
DnsClient sending request for www.mcgill.ca
Server: 132.206.85.18
Request type: A
Response received after 0.076164833 seconds (0 retries)
*** Answer Section (2 records) ***
CNAME   webcache.it.mcgill.ca   2970   nonauth
IP   132.216.177.157   300   nonauth
NOTFOUND - Additional Section
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ python dnsClient.py @132.206.85.18 www.mcgill.ca
DnsClient sending request for www.mcgill.ca
Server: 132.206.85.18
Request type: A
ERROR    Socket timeout, unanswered query
ERROR    Socket timeout, unanswered query
ERROR    Socket timeout, unanswered query
ERROR    Maximum number of retries 3 exceeded
```

Figure 16: mcgill.ca @132.206.85.18 query

To validate the formatting of the DNS request packet we primarily relied on the dnsprimer.pdf for instructions. Since there is not much variability in building the request packet, translating the domain name was the biggest challenge, but comparing it to the examples in the pdf made it comprehensible. Since the packets must be sent and received in the form of a byte object, the easiest way to manage building and parsing the packets was through using a string representation of a hexadecimal number, and using library system functions to translate between string to byte, or string to byte to binary to string, and vice versa.

As for the response, we primarily used the example input given in the assignment guidelines (`python dnsClient.py -t 10 -r 2 -mx @8.8.8.8 mcgill.ca`) to parse the results and validate whether or not our logic was sound. Using print statements, we compared the header and question built in the request packet to the header and question received in the response to better understand the formatting. We also manually translated the response packet using the logic fromdnsprimer.pdf and applied this logic to our code.

Once we got the basic idea, we tested Google's DNS server (8.8.8.8) with well known name servers and mail servers like google.com, microsoft.com, or gmail.com. To validate our responses, we tested the DNS server's behavior using a well-established tool like dig (ex: `dig @8.8.8.8 NS google.com`). This allowed us to cross reference the number of records in each question, answer, authority and additional section along with the record's RDATA and TYPE.

We did not test the -p flag with non-standard ports. Otherwise, according to our tests, which are detailed in the rest of this report, the code is behaving as expected.

IV. Experiment

a. What are the IP addresses of McGill's DNS Servers? What response do you get? Does this match what you expected?

As recommended in the question, we used the Google public DNS server (8.8.8.8) to perform an NS query on mcgill.ca. This yielded the addresses `pens1.mcgill.ca` and `pens2.mcgill.ca`. The subsequent queries we had to make were type A queries on these two addresses (`python dnsClient.py @8.8.8.8 pens1.mcgill.ca` and `python dnsClient.py @8.8.8.8 pens2.mcgill.ca`), which gave us respectively the IP addresses `132.206.44.69` and `132.206.44.135`. We expected to get `132.206.85.18` as a result, since it should be authoritative for mcgill.ca, so it does not match what we expected.

b. Use your client to run DNS queries for 5 different website addresses, of your choice, in addition to www.google.com and www.amazon.com, for a total of seven addresses. Query the seven addresses using the Google public DNS server (8.8.8.8).

1. The **google.com** query gives us IP 172.217.13.206 (*Figure 1*) in the answer section, which we validated for correctness using dig (*Figure 2*).

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ python dnsClient.py @8.8.8.8 www.google.com
DnsClient sending request for www.google.com
Server: 8.8.8.8
Request type: A
Response received after 0.140947125 seconds (0 retries)
*** Answer Section (1 records) ***
IP   172.217.13.132   110   nonauth
NOTFOUND - Additional Section
```

Figure 1: Query for www.google.com

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ dig @8.8.8.8 www.google.com

; <<>> DiG 9.10.6 <<>> @8.8.8.8 www.google.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 21298
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;www.google.com.                        IN      A

;; ANSWER SECTION:
www.google.com.         297     IN      A       172.217.13.132

;; Query time: 78 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Mon Oct 02 13:51:04 EDT 2023
;; MSG SIZE  rcvd: 59
```

Figure 2: Output validation using Dig

2. The **www.amazon.com** query gave us CNAMEs tp.47cf2c8c9-frontier.amazon.com, d3ag4hukkh62yn.cloudfront.net and IP 3.160.23.134 (*Figure 3*), which we also validated using dig (*Figure 4*).

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ python dnsClient.py @8.8.8.8 www.amazon.com
DnsClient sending request for www.amazon.com
Server: 8.8.8.8
Request type: A
Response received after 0.13517812499999998 seconds (0 retries)
*** Answer Section (3 records) ***
CNAME   tp.47cf2c8c9-frontier.amazon.com   1494   nonauth
CNAME   d3ag4hukkh62yn.cloudfront.net   60   nonauth
IP   3.160.6.130   60   nonauth
NOTFOUND - Additional Section
```

Figure 3: Query for www.amazon.com

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ dig @8.8.8.8 www.amazon.com

; <<>> DiG 9.10.6 <<>> @8.8.8.8 www.amazon.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 58272
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;www.amazon.com.                        IN      A

;; ANSWER SECTION:
www.amazon.com.         1738    IN      CNAME   tp.47cf2c8c9-frontier.amazon.com.
tp.47cf2c8c9-frontier.amazon.com. 60 IN CNAME   d3ag4hukkh62yn.cloudfront.net.
d3ag4hukkh62yn.cloudfront.net. 60 IN    A       3.160.23.134

;; Query time: 91 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Mon Oct 02 13:46:14 EDT 2023
;; MSG SIZE  rcvd: 138
```

Figure 4: Output validation using Dig

Now for the queries of our choice:

3. We queried **mcgill.com**, with a NS type request to test our additional section. This yielded name servers pens1.mcgill.ca and pens2.mcgill.ca in the answer section, and IPs 132.206.44.69 and 132.206.44.135 in the additional section (*Figure 5*). We tested this using Dig (*Figure 6*).

```
DnsClient sending request for mcgill.com
Server: 8.8.8.8
Request type: NS
Response received after 0.04396845799055882 seconds (0 retries)
*** Answer Section (2 records) ***
NS   pens2.mcgill.ca   2255   nonauth
NS   pens1.mcgill.ca   2255   nonauth
*** Additional Section (2 records) ***
IP   132.206.44.69   2748   nonauth
IP   132.206.44.135   1579   nonauth
```

Figure 5: Query for amazon.com

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ dig @8.8.8.8 NS mcgill.com

; <<>> DiG 9.10.6 <<>> @8.8.8.8 NS mcgill.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 37711
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;mcgill.com.                    IN      NS

;; ANSWER SECTION:
mcgill.com.             3600    IN      NS      pens1.mcgill.ca.
mcgill.com.             3600    IN      NS      pens2.mcgill.ca.

;; Query time: 86 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Mon Oct 02 14:11:47 EDT 2023
;; MSG SIZE  rcvd: 88
```

Figure 6: Output validation using Dig

4. We queried **un.org**, with custom values for the timeout (5), retry (3) and port (53) options. This yielded name servers IPs 157.150.185.49 and 157.150.185.92 in the additional section (Figure 1). We tested this using a DNS Lookup website which is referenced in our resources at the end of this report.

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ python dnsClient.py -t 5 -r 3 -p 53 @8.8.8.8 un.org
DnsClient sending request for un.org
Server: 8.8.8.8
Request type: A
Response received after 0.23672258300000001 seconds (0 retries)
*** Answer Section (2 records) ***
IP   157.150.185.49   160   nonauth
IP   157.150.185.92   160   nonauth
NOTFOUND - Additional Section
```

Figure 7: Query for un.org

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ dig @8.8.8.8 +time=5 +retry=3 -p 53 un.org

; <<>> DiG 9.10.6 <<>> @8.8.8.8 +time=5 +retry=3 -p 53 un.org
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 59803
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;un.org.                                IN      A

;; ANSWER SECTION:
un.org.                 300     IN      A       157.150.185.92
un.org.                 300     IN      A       157.150.185.49

;; Query time: 391 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Mon Oct 02 14:18:38 EDT 2023
;; MSG SIZE  rcvd: 67
```

Figure 8: Output validation using Dig

5. We queried **www.netflix.ca**, with a MX type request. This yielded CNAMEs detour.netflix.net and detour.prod.netflix.net in the answer section (*Figure 9*). We tested this using Dig (*Figure 10*).

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ python dnsClient.py -mx @8.8.8.8 www.netflix.ca
DnsClient sending request for www.netflix.ca
Server: 8.8.8.8
Request type: MX
Response received after 0.28399937500000005 seconds (0 retries)
*** Answer Section (2 records) ***
CNAME   detour.netflix.net   300   nonauth
CNAME   detour.prod.netflix.net   300   nonauth
NOTFOUND - Additional Section
```

Figure 9: Query for www.netflix.ca

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ dig @8.8.8.8 MX www.netflix.ca

; <<>> DiG 9.10.6 <<>> @8.8.8.8 MX www.netflix.ca
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 37441
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 1, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;www.netflix.ca.                        IN      MX

;; ANSWER SECTION:
www.netflix.ca.         300     IN      CNAME   detour.netflix.net.
detour.netflix.net.     300     IN      CNAME   detour.prod.netflix.net.

;; AUTHORITY SECTION:
prod.netflix.net.       60      IN      SOA     ns-803.awsdns-36.net. awsdns-hostmaster.amazon.com. 1 7200 900 1209600 86400

;; Query time: 214 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Mon Oct 02 14:21:26 EDT 2023
;; MSG SIZE  rcvd: 182
```

Figure 10: Output validation using Dig

6. We queried **www.netflix.com**, with a MX type request. This yielded CNAMEs www.dradis.netflix.com, www.us-east-2.internal.dradis.netflix.com,

apiproxy-website-nlb-prod-1-f5850ec0efac7a14.elb.us-east-2.amazonaws.comin the answer section (*Figure 11*). We tested this using Dig (*Figure 12*).

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ python dnsClient.py -mx @8.8.8.8 www.netflix.com
DnsClient sending request for www.netflix.com
Server: 8.8.8.8
Request type: MX
Response received after 0.07135674999999998 seconds (0 retries)
*** Answer Section (3 records) ***
CNAME   www.dradis.netflix.com   164   nonauth
CNAME   www.us-east-2.internal.dradis.netflix.com   44   nonauth
CNAME   apiproxy-website-nlb-prod-1-f5850ec0efac7a14.elb.us-east-2.amazonaws.com   44   nonauth
NOTFOUND - Additional Section
```

Figure 11: Query for www.netflix.com

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ dig @8.8.8.8 MX www.netflix.com

; <<>> DiG 9.10.6 <<>> @8.8.8.8 MX www.netflix.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 35257
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 1, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;www.netflix.com.               IN      MX

;; ANSWER SECTION:
www.netflix.com.        166     IN      CNAME   www.dradis.netflix.com.
www.dradis.netflix.com. 46      IN      CNAME   www.us-east-2.internal.dradis.netflix.com.
www.us-east-2.internal.dradis.netflix.com. 46 IN CNAME apiproxy-website-nlb-prod-2-98ca77053f29aaca.elb.us-east-2.amazonaws.com.

;; AUTHORITY SECTION:
elb.us-east-2.amazonaws.com. 60 IN      SOA     ns-1513.awsdns-61.org. awsdns-hostmaster.amazon.com. 1 7200 900 1209600 60

;; Query time: 100 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Mon Oct 02 14:21:20 EDT 2023
;; MSG SIZE  rcvd: 271
```

Figure 12: Output validation using Dig

7. We queried **www.jetbrains.com**, with an A type request. This yielded IPs 3.162.3.74, 3.162.3.22, 3.162.3.113 and 3.162.3.89 (*Figure 13*). We tested this using Dig (*Figure 14*).

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ python dnsClient.py @8.8.8.8 www.jetbrains.com
DnsClient sending request for www.jetbrains.com
Server: 8.8.8.8
Request type: A
Response received after 0.064434542 seconds (0 retries)
*** Answer Section (4 records) ***
IP   3.162.3.74   60   nonauth
IP   3.162.3.22   60   nonauth
IP   3.162.3.113   60   nonauth
IP   3.162.3.89   60   nonauth
NOTFOUND - Additional Section
```

Figure 13: Query for www.jetbrains.com

```
(base) MacBook-Pro:ECSE316-Assignments ameliebarsoum$ dig @8.8.8.8 www.jetbrains.com

; <<>> DiG 9.10.6 <<>> @8.8.8.8 www.jetbrains.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 25672
;; flags: qr rd ra; QUERY: 1, ANSWER: 4, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;www.jetbrains.com.             IN      A

;; ANSWER SECTION:
www.jetbrains.com.      60      IN      A       3.162.3.89
www.jetbrains.com.      60      IN      A       3.162.3.74
www.jetbrains.com.      60      IN      A       3.162.3.22
www.jetbrains.com.      60      IN      A       3.162.3.113

;; Query time: 105 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Mon Oct 02 17:05:16 EDT 2023
;; MSG SIZE  rcvd: 110
```

Figure 14: Output validation using Dig

c. Briefly explain what a DNS server does and how a query is carried out. Modern web browsers are designed to cache DNS records for a set amount of time. Explain how caching DNS records can speed up the process of resolving an IP address. You can draw a diagram to help clarify your answer.

A DNS server's main purpose is to translate user-friendly domain addresses to their corresponding IP address. While computers interact with the IP addresses, it is easier for humans to remember domain names, hence the need for translation. A DNS query happens as follows: when the user enters a domain name, before querying the server, the browser first checks its local cache to see if the IP address of this domain is already saved. If it is not in the cache, we enter the server hierarchy organized by level of authority, and the browser begins a recursive query to get the IP address of the domain name, which will first ask the local DNS server, then ask a root server to point it to the appropriate TLD server. The TLD server will guide us to the authoritative name server for our domain name. From the authoritative server, we get the IP address, which is sent back to the browser and can be used by the browser to generate the website content using the appropriate web servers. The flowchart in *Figure 15* visually represents this process.

Caching DNS records can greatly improve performance by eliminating computationally heaving DNS queries when the website has been recently/frequently visited, depending on the type of cache. Instead, we can skip straight to the step where the browser generates the website by querying the appropriate web server, and bypass the DNS query and the website will load much faster.

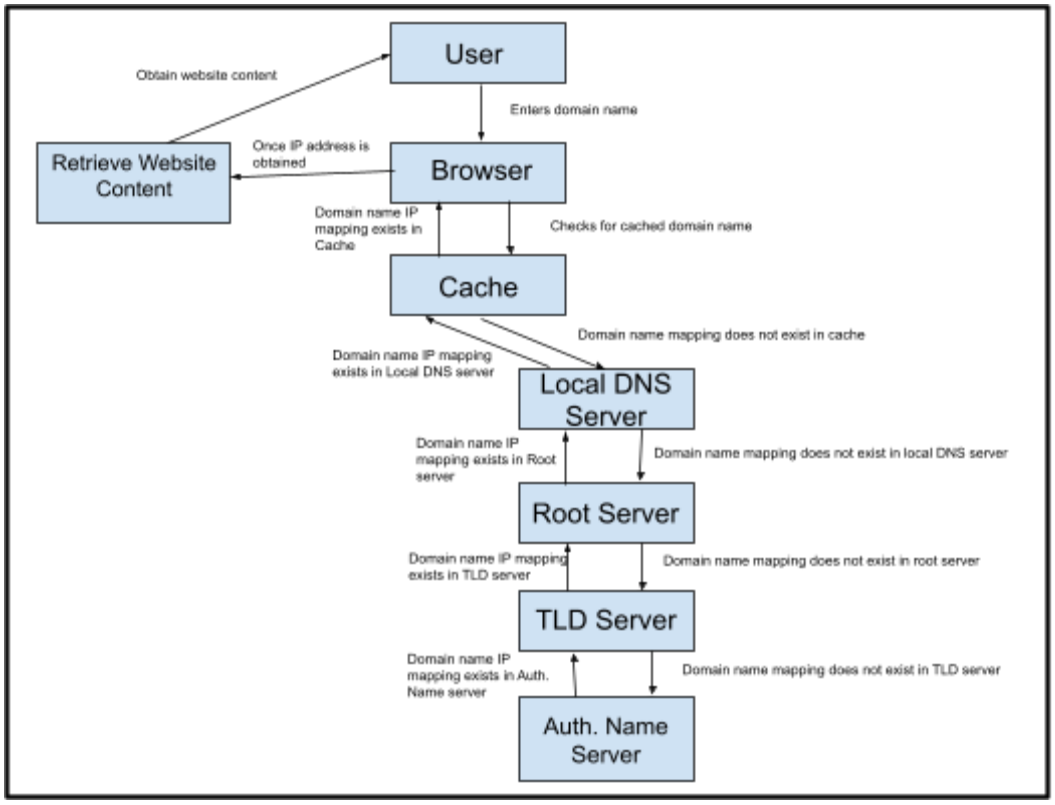

Figure 15: Flowchart of DNS query process

V. Discussion

Throughout this assignment, we were able to make some key observations regarding the intricacies of the DNS resolution process. We were able to learn about the usefulness of caching, and how much it contributes to improving browser performance. We also developed a robust error detection mechanism, with the ability to detect errors in IP address/domain names, and which also included using timeouts and retries to deal with unresponsive or non-existent servers. By querying mcgill.ca as an NS query, we learnt that the DNS server for a network is not necessarily authoritative for related networks, since we obtained different IP addresses for mcgill.ca, which did not match the McGill DNS Server.

As discussed in the introduction paragraph, the main challenges for us were executing proper packet parsing and avoiding formatting errors which would lead to bigger errors. We also found it difficult to implement and test all of the record types, such as MX, NS and A type DNS records. We also found that certain queries could have a very large latency with certain servers, and we had to experiment in order to determine appropriate timeout times and number of retries to avoid waiting for a long time. Finally, the recursive elements in the assignment were difficult to implement, especially because we had to consider the fact that some elements we encounter may be pointers.

To help with the challenge of implementing the proper packet parsing, we could use a DNS library, like dnspython or Scapy, which would help to deal with the nitty gritty details of the parsing. To deal with the latency problem, we could try sending multiple queries in parallel but to different servers, and accept the fastest answer, which would reduce waiting time when a server is too slow or even unresponsive. Finally, the recursive elements could be easier to implement also with the use of external libraries, as well as a sort of state management mechanism, like a stack, to retain the state between recursive calls.

VI. Resources

[1] https://www.ibm.com/docs/en/ztpf/2020?topic=sockets-types-supported-by-tcpip
[2] https://docs.python.org/3/library/socket.html#module-socket
[3] https://docs.python.org/3/howto/sockets.html#creating-a-socket
[4] https://wiki.python.org/moin/UdpCommunication
[5] https://pythontic.com/modules/socket/udp-client-server-example
[6] https://stackoverflow.com/questions/18311338/python-udp-client-time-out-machinsm
[7] https://dnslookup.online/?fbclid=IwAR0vYfeNnKm4f7l_2YE5DmmdRGnHQgTm5C86DGnq3W942fKmDGE3LOyNmYs