ECSE 316 Assignment 1 Report

Mark Coates

January 30, 2023

Group 17:

Joey Koay (260956108)

Mihail Calitoiu (260972537)

Introduction:

The purpose of this assignment is to design a Java or Python application that allows users to connect to a user-specified domain name system (DNS) client through sockets. We decided to implement this system via Python. We choose Python because it often allows for more eloquent code implementations compared to Java. The final application constructs a DNS request packet using the user's input and sends the packet to the DNS server. The received response is unpacked, parsed and printed to the output.

The main challenge in implementing the program was to understand the many steps both in constructing the packets and interpreting the response. For example, the format of the compressed packets suggested that reading byte by byte would be a correct approach, but we had to experiment many times until we retrieved the data correctly. Additionally, we found working with packets and sockets unforgiving, and we had to debug many times to ensure that the code was working as expected. Overall, although we encountered many issues, we are satisfied with the end result and tried our best to create the shortest, cleanest python implementation of the requirements.

Design:

To receive a summary of the performance and content of the DNS response, the user must provide the IPv4 address and domain name to the program, with the option of specifying the amount of time (in seconds) that the program waits before retransmitting the query (timeout), the maximum number of time the program should send the query before terminating (max-retries), the UDP port number of the DNS server (port), and whether it is a mail server or name server type of queries (the default is type A - IP address). To run the program, the user should send in the command line through the following syntax.

```
Python DnsClient [-t timeout] [-r max-retries] [-p port] [-mx|-ns] @server name
```

In our implementation of the application, we created one file with four functions: main, print_line, deepFlatten and decompress. The program follows a very linear layout, below is a high overview of the general structure.

1. Variable declarations: The variables xtimeout, xmaxretries, xport, xRecordT, xip, xname, and q_type are declared. These variables store the various options for the DNS request such as the server IP, the type of record requested, the maximum number of retries, etc.
2. Input parsing: A loop is used to parse the command line arguments. The options given by the user are processed, and the respective variables are updated accordingly.
3. Packet construction: The DNS request packet is constructed in this section. The parts of the requested name are extracted and used to build the packet. The packet has five sections, such as the header, question, answer, authority, and additional sections.
4. Socket construction: A socket is created using the Python socket library. The socket is configured to use an IPv4 address, a datagram socket type, and has a timeout value set.
5. Request sending: A loop is used to send the DNS request packet to the server. If the response is not received within the specified timeout time, the request is retried xmaxretries times. If the maximum number of retries is exceeded, an error message is displayed.
6. Response processing: The response from the server is unpacked and the information is extracted and printed. The number of answers, flags, and additional record counts are extracted from the response and used to process the data.

Main function:
The main function takes in one argument which is the user's input that is given via the command line. In this function, we first determine whether the optional fields are given by the user (timeout, max-retries, port and mx/ns). If the optional fields are not specified, default values are used. After the input has been parsed, we print a message to the user confirming the request information.

Then, the program builds the packets. Since the packets work in bytes, an empty byte object is first created, then the required sections (header, questions, answers, authorities, additional information) are added to the packet. Following the creation of the byte object, the domain name of the program is converted into bytes and added to the packet along with the type (A, MX, NS).

After the packets have been created, the program builds the socket that will be used to send the newly built packet by using Python's built-in socket libraries. When creating the socket, information such as timeout is specified using the user's preference or the default value. After the socket has been established, the DNS request packet is sent to the server via the socket.

While sending the packet, we implemented error handlings for several common errors. If no response is received in the indicated time, a message is sent to the user via the terminal showcasing that the socket has a timeout. This is repeated for a given number of times that the user has determined in the initial input; the default value (3) is used otherwise. If any errors have occurred (other than timeout) during the execution, the specific error will be shown to the user, and the program will terminate. After the maximum number of tries has occurred and no response is received, the program will terminate and notify the user that the maximum number of retries has been exceeded.

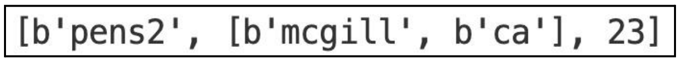
If a response is received, the program will begin unpacking the response to allow a formatted output to be shown to the user. In the DNS response, we are interested in the flags, the number of resource records in the answer section, and the number of resource records in the additional records section.

Since the sent message is compressed to help reduce the size of the message, a decompression of the message is performed. This is done by calling the decompressed helper function, which will be explained later in this report.

Following the decompression of the response, the print_line helper function (which will be explained later in the report) is used to identify the specific answers and additional sections of the response. The results will then be outputted in the command line with the indicated format shown in the assignment instruction. If no answer and additional information are found in the DNS response, the user will be notified that no information was found. The program is then terminated after returning the results.

print_line function:
This function takes in three arguments: the data, the offset and whether it is authoritative, and returns the formatted data and the amount of offset there is after the data. The purpose of this function is to retrieve the answer and additional information sent from the DNS response. We provide the decompress function with an offset which identifies where the current data starts at. Once we receive the data from the decompress function, we also receive an updated offset value which identifies the location of the next line (assuming it exists). For the NS and CNAME specifically, the data is processed by the deepFlatten function (explained below) and the result is printable information. Depending on the type, the corresponding messages will be saved into a list which the function will return. The returned string is then printed by the main function.

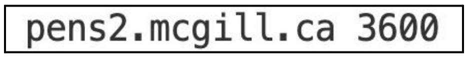
deepFlatten function:
The deepFlatten function formats the data returned by the decompress function by decoding and arranging it into an array of strings. Figure 1 below is an example of the returned data from the decompress function.



```
[b'pens2', [b'mcgill', b'ca'], 23]
```

Figure 1: Example data from decompress function.

Since the data is an array with elements that are bytes, numbers and arrays, we want to convert it to an array of same type elements, which is a process called "flattening." We do this by creating a new array and visiting the original array as deep as possible, meaning that we visit only elements. The resulting array will be able to be printed easily. Figure 2 below is the result of deep flattening the example data shown in Figure 1 above.



```
pens2.mcgill.ca 3600
```

Figure 2: Example data from decompress function.

We see that the data is being correctly decoded and printed.

decompress function:
There are two arguments required: the message and the offset. Two values that are returned by this function are the decompressed name and the offset. This is done by exploring byte by byte until a byte

with “11” as the most significant bits is reached or if a byte has a value of 0. We append all the explored bytes to an array and this is the data collected starting from the provided offset. We return the array and the new offset which marks the start of the next data (assuming there is any).

Testing:

To test our program, we create a set of test cases that allow us to ensure that most scenarios and edge cases are being taken care of. A tool that we used to help us find test cases is https://dnslookup.online/.

A few test cases that we created are as follows:

Test 1: -ns @132.206.44.69 mcgill.ca
Test 2: -mx @8.8.8.8 mcgill.ca
Test 3: @8.8.8.8 facebook.com
Test 4: -t 10 -r 2 -ns @8.8.8.8 mcgill.ca

These test cases allowed us to receive and verify the different types of responses.

To increase code robustness, we implemented several ways to check the information given and validate whether it is in the correct format. When entering the information in the command line in “Python DnsClient [-t timout] [-r max-retries] [-p port] [-mx|-ns] @server name” format, the arguments are already in a list itself. We simply looped through the elements in the list and checked whether “-t”, “-r”, “-p”, “-mx”, “-ns” appeared. If so, the corresponding values will overwrite the global default variable. If none were given, the default values would be used. If a token that is not recognized is parsed (i.e. none of the possible options), we print an error message. Since the required information (server and name) are the last arguments, the loop will stop checking once the “@” symbol is detected, as our program assumes the following two arguments are server and name. One improvement we could make in the future is to validate that the server is in the correct format. (a.b.c.d format).

Upon receiving the response packet, we parse the packet through a series of actions. First, we unpack the data into a tuple using struct and specifying that the data should be unpacked as six 16-bit unsigned integers in big-endian byte order. After unpacking, we only retrieve what is needed - the flags, the number of resource records in the answer section, and the number of resource records in the additional records section. With the flags, we specifically want to obtain the AA value to know whether the name server is an authority for the indicated domain name. After retrieving these three pieces of information, the program proceeds to show the specific answer and additional information through the response packet. To find the correct information, we need to find the amount of offset needed to locate the information. This is done through the decompressed function, as mentioned above.

The one feature that we were unable to test as extensively is the retrieving of the additional information part. We found a post on our class page’s discussion board, the teacher assistant suggested trying an NS-type test case with server 132.206.44.69 and domain name mcgill.ca; we originally noticed that the additional section was not being printed but quickly fixed the bug.

Experiment:

1. What are the IP addresses of McGill's DNS servers? Use the Google public DNS server (8.8.8.8) to perform a NS query for mcgill.ca and any subsequent follow-up queries that may be required. What response do you get? Does this match what you expected?

   The result we receive is:

   ```
   DnsClient sending request for mcgill.ca
   Server: 8.8.8.8
   Request type: NS
   Response received after 0.048 seconds (0 retries)
   ***Answer Section (2 records)***
   NS      pens2.mcgill.ca 3600    nonauth
   NS      pens1.mcgill.ca 3600    nonauth
   ```

   Yes, it is what we expect. Using https://dnslookup.online/, we receive the same response.

2. 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).

Table 1: Response from seven addresses

Web Addresses:	Client Result:
google.com	DnsClient sending request for google.com Server: 8.8.8.8 Request type: A Response received after 0.063 seconds (0 retries) ***Answer Section (1 records)*** IP 142.251.41.46 300 nonauth
amazon.com	DnsClient sending request for amazon.com Server: 8.8.8.8 Request type: A Response received after 0.270 seconds (0 retries) ***Answer Section (3 records)*** IP 54.239.28.85 353 nonauth IP 205.251.242.103 353 nonauth IP 52.94.236.248 353 nonauth
apple.com	DnsClient sending request for apple.com Server: 8.8.8.8 Request type: A

	Response received after 0.035 seconds (0 retries) ***Answer Section (1 records)*** IP 17.253.144.10 76 nonauth
facebook.com	DnsClient sending request for facebook.com Server: 8.8.8.8 Request type: A Response received after 0.183 seconds (0 retries) ***Answer Section (1 records)*** IP 157.240.249.35 300 nonauth
newyorktimes.com	DnsClient sending request for newyorktimes.com Server: 8.8.8.8 Request type: A Response received after 0.042 seconds (0 retries) ***Answer Section (4 records)*** IP 151.101.1.164 500 nonauth IP 151.101.193.164 500 nonauth IP 151.101.129.164 500 nonauth IP 151.101.65.164 500 nonauth
porsche.com	DnsClient sending request for porsche.com Server: 8.8.8.8 Request type: A Response received after 0.021 seconds (0 retries) ***Answer Section (1 records)*** IP 84.21.52.116 300 nonauth
yahoo.com	DnsClient sending request for yahoo.com Server: 8.8.8.8 Request type: A Response received after 0.037 seconds (0 retries) ***Answer Section (6 records)*** IP 98.137.11.164 999 nonauth IP 74.6.231.20 999 nonauth IP 74.6.143.25 999 nonauth IP 74.6.143.26 999 nonauth IP 74.6.231.21 999 nonauth IP 98.137.11.163 999 nonauth

The results above match what https://dnslookup.online/ returns, so the created client works as intended.

3. 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 (Domain Name System) server's main responsibility is to translate a human-readable domain name (i.e. facebook.com), into an IP address (i.e. 157.240.249.35). When a user requests a website, the browser sends a DNS query to a DNS server to convert the domain name into an IP address. The DNS server either resolves the query from its cache, which is fastest, or forwards the request to another DNS server to find the answer.

Caching DNS records decreases the time needed to resolve an IP address by storing all the recently looked up IP addresses. This allows the browser to quickly retrieve the IP address without having to send another DNS query.

The following diagram illustrates this process:

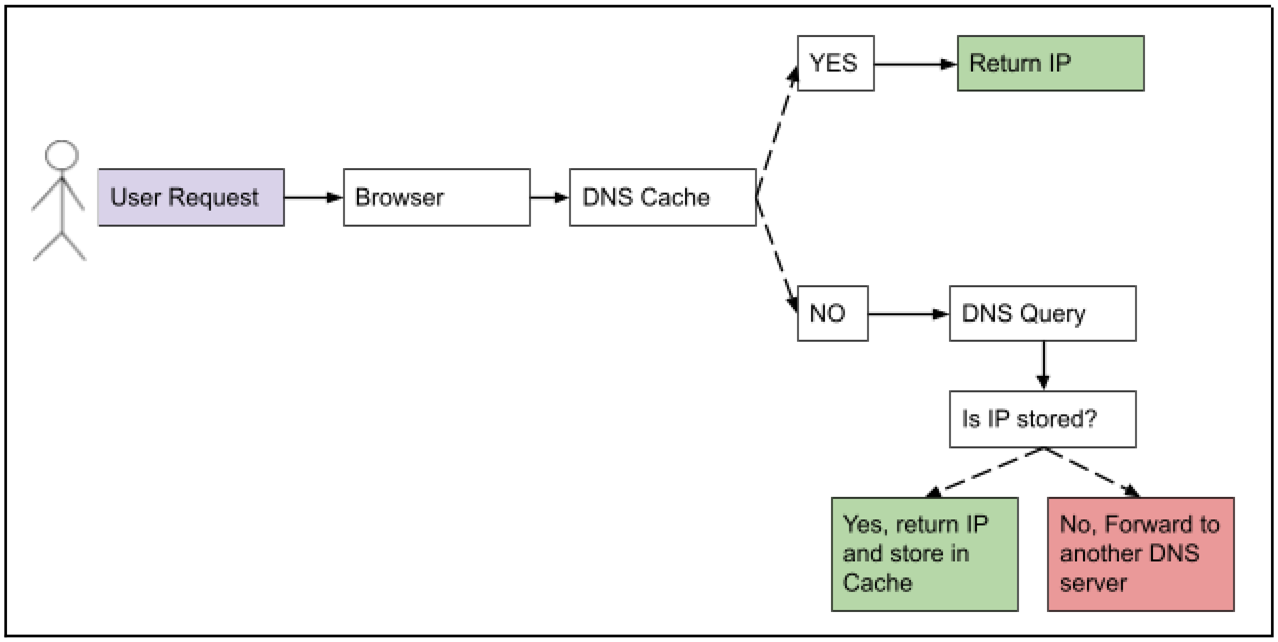

User Request
Browser
DNS Cache
YES
Return IP
NO
DNS Query
Is IP stored?
Yes, return IP and store in Cache
No, Forward to another DNS server

Figure 2: DNS caching overview

Via this method, the subsequent requests for the same domain name can be resolved faster, as the browser only needs to check the cache for the IP address, instead of sending another DNS query.

Discussion:

During this experiment, we were able to allow users to connect to a user-specified DNS client through sockets via a Python application. We observed that there are many procedures in both sending and receiving the packets to maximize efficiency (i.e. compressing packets to minimize repeated information). As seen in the testing and experiment section of this document, the output produced by our application is as expected.

The main challenge for implementing our application is in both building the packets and interpreting the response. Since the structure of the packet operates in bytes, we had to run the code and check whether we are converting the user's input into the packet's desired style correctly. Similarly, when receiving the DNS response, we had to run the code numerous times when new data is retrieved to ensure that the code is functioning correctly.

A better approach would have been to write pseudocode in the beginning instead of programming in a trial and error method. We tried many things and modified them until we received the expected output, which took a long time. If we wrote the code after understanding all the theory, the implementation time would have been lower.