ECSE 316 Assignment 1 Report
Rambod Azimi – 260911967
Saghar Sahebi – 260908343
Group 39
Date: January 26th, 2023

1. Introduction

In the first assignment of ECSE 316, we implemented a *domain name system (DNS)* client using sockets in *Python*. DNS is a distributed hierarchical database with several important features, but domain-to-IP mapping is one of its most essential. We used *IPv4 and UDP* (user datagram protocol) to implement the DNS client application. UDP is a *connectionless* protocol, meaning we can transmit packets without knowing any information about the destinations. The DNS client can be invoked from the command line using optional and required arguments (such as timeout, port, server, name, …). It sends a query to the server using the UDP protocol and waits for the response to be returned. Finally, it displays the result to the command line display. There are several types of queries which should be handled individually. The query of type A is the standard query (IP address); however, queries of types MX and NS are used for the mail server and name server, respectively. The responses can contain A records or CNAME records.

The main challenge in this lab was handling the *DNS packet structure*, which contains a Header, Question, Answer, Authority, and Additional. We had to build a structure used for sending a request to the server as well as unbuild the structure received from the server to extract helpful information from the packet received and to be able to interpret it to the user. The main result of the DNS client program is that the app will send a request to the DNS server, receive the server's response, and interpret the answer to the client. In other words, the client can query a hostname and an IP address, and the response contains valuable information about the host. Also, the program will generate an error message in case of facing a problem.

2. DNS Client Program Design

The design of the DNS Client program consists of two primary sections: 1) sending a *request* from the client to the server and 2) displaying an *output* received from the server to the client. First, we had to deal with the syntax for our application. The application should be invoked from the command line with the following syntax:

python3 DnsClient [-t timeout] [-r max-retries] [-p port] [-mx|-ns] @server name

To do so, we used the *argparse* module in Python to parse the arguments in the command line. Some of the arguments are optional, and some of them are mutually exclusive. For example, the user cannot run the application with both -mx and -ns arguments. Also, if the user does not specify any value for the timeout, max-retries, and port, they will automatically be set to 5, 3, and 53, respectively, using their default values. After setting the values and saving them into variables, we checked the query type (A, MX, NS) and put the correct value for each query type. For example, query type A has a value of 1, query type NS has a value of 2, and MX has a value of 15. Then, we *summarized* the query to the command line. An example is shown in figure 1.

```
rambodazimi@Rambods-MacBook-Pro Desktop % python3 Client.py @8.8.8.8 www.mcgill.ca
DnsClient sending request for: www.mcgill.ca
Server: 8.8.8.8
Request type: A
```

Figure 1. An example of summarizing a query of type A to the command line

After outlining the query to the command line, we implemented the *packet_builder()* routine, which builds request packets. As mentioned in the dnsprimer document, DNS packets have a particular structure. Figure 2 demonstrates the different sections of the DNS packet, namely Header, Question, Answer, Authority, and Additional. The design of the packet_builder() routine design will be explained in more depth in the Test section of the report.

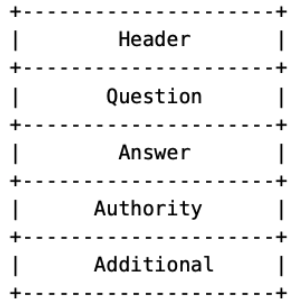


Figure 2. DNS packet structure

When the user starts the program to ask the DNS server for some information, it specifies the number of times to retransmit an unanswered query before giving up. If the user does not determine the *max_retries*, the program will set the default value of 3 to that. That is why we tried max_retries number of times to send a packet request to the DNS server (using the for loop). Then, we created a client socket object using the *socket* module in Python with two parameters *socket.AF_INET, socket.SOCK_DGRAM*. The first parameter indicates that we want to use the IPv4 protocol to send the packets to the server. The second parameter indicates that we want to use the User Datagram Protocol, a connectionless, unreliable, and unordered protocol to send the packets to the DNS server.

The user can also specify the timeout value, which gives how long to wait, in seconds, before retransmitting an unanswered query. The default value for the timeout argument is five if the user does not indicate it to the program. To measure the time it takes to send the packet to the server and wait for a response, we used the *time* module in Python and calculated the interval time by subtracting the start time from the end time. In case of a timeout error, we print an appropriate error message indicating that the time has exceeded the threshold. Finally, we used the *sendto()* routine to send the request packet to the DNS server with two arguments: 1) a packet which has been built by the *packet_builder()* routine, and 2) a *tuple* containing the IP address and the port. After sending the request packet to the server, we used *recvfrom()* routine. This method is a blocking method, meaning that the client will wait until the server sends a response to the client. We saved the result into the *answer* variable and returned from the routine to perform the display routine. Figure 3 displays the send_request() routine implemented in Python.

```python
def send_request(max_retries, timeout, domain_name, queryNumber, ip_address, port):
    finished = False
    for i in range(max_retries): # try sending request max_retries number of times
        client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # create a client socket object using IPV4 and UDP protocols
        client_socket.settimeout(timeout) # set the timeout for the client socket
        try:
            start_time = time.time()
            # without connecting to any server (TCP), with UDP we can send the request packet to the tuple (ip address, port number)
            client_socket.sendto(packet_builder(domain_name, queryNumber), (ip_address, port))
            answer, _ = client_socket.recvfrom(512) # wait until get an reply from the server
            finished = True
            end_time = time.time()
            time_taken = end_time - start_time # compute the length of time taken to send the request and receive a response from the server
            print(f"Response received after {time_taken} seconds ({i} retries)")
            if (finished): # if a response is received, break from the for loop
                break
        except socket.timeout:
            print(f"ERROR    Time Out Error. You set {timeout} seconds to wait")
            if(i >= max_retries):
                print(f"ERROR   Maximum number of retries {max_retries} exceeded")

    return answer # return the response received from the server

#same as build packets but unbuild them to be able to decode them later
```

Figure 3. Implementation of send_request() routine in Python

After sending the packet request to the DNS server, it is time to interpret the result to the command line using STDOUT. The *display_output()* routine accepts two arguments: the server response and the domain name. We implemented another method named *unbuild_packet()* which is very similar to what we implemented in the build_packet(), but this time, we unpack the result received as a reply to extract each important field (such as flags, qdcount, ancount, nscount, and arcount). Each value is 2 bytes away, so the idea is to unpack and increase the offset by 2.
There may be more than one recourse record in the result. Therefore, we iterated through each record to display the output for each record separately. The first task is to check the RCODE in the header section of the DNS packet to see if there is any error in the response. To do so, we implemented a helper method called *display_request_error_handler()*. Each value for RCODE means a particular error. For example, if RCODE is one, it means the name server could not interpret the query.
To interpret the output in the Answer section, we used the *change_pointer_position()* routine, a helper function to skip the names of fields we do not need. Also, to get the required information from the response, the *get_answer_info()* routine checks for the type of the response (A, MX, NS, CNAME) and extracts data from each response type depending on the type. For example, for type A (IP), we converted each label to a string and combined them with a dot between each label. The implementation of the get_answer_info() routine is shown in Figure 4.

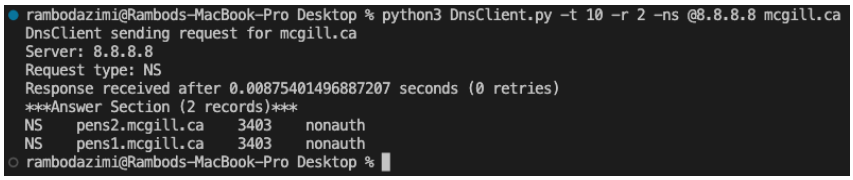
```
def get_answer_info(result, pointer_position, anType, rdLength):

    dataValue = ""
    if (anType == 0x0001):
        requestType= "IP"
        an_r_data = struct.unpack_from(">"+"B" * rdLength, result, pointer_position)
        for i in an_r_data:
            dataValue += str(i) + "."
        dataValue = dataValue[:-1]
        pointer_position += rdLength
    elif (anType == 0x0002):
        requestType = "NS"
        an_r_data = decode_label(result, pointer_position)
        for i in an_r_data:
            dataValue += str(i.decode("utf-8")) + '.'
        dataValue = dataValue[:-1]
        pointer_position += rdLength
    elif (anType == 0x0005):
        requestType = "CNAME"
        an_r_data = decode_label(result, pointer_position)
        for i in an_r_data:
            dataValue += str(i.decode("utf-8"))  + '.'
        dataValue = dataValue[:-1]
        pointer_position += rdLength
    elif (anType == 0x000f):
        requestType = "MX"
        attribute = struct.unpack_from(">H", result, pointer_position)[0]
        pointer_position += 2
        an_r_data = decode_label(result, pointer_position)
        for i in an_r_data:
            dataValue += str(i.decode("utf-8"))  + '.'
        dataValue = dataValue[:-1]
        dataValue += "," + str(attribute)
        pointer_position += rdLength
    else:
        raise Exception("ERROR   unvalid response")

    return dataValue, pointer_position, requestType
```

Figure 4. Implementation of get_answer_info() routine in Python

Finally, depending on the type of response, we printed the output to the terminal along with appropriate lines for each additional record that matches one of the types A, CNAME, MX, or NS. Then, to interpret the additional section, we iterated over each resource record in the additional section and displayed the appropriate messages. Figure 5 shows an example of running the application with the output displayed in the terminal.



```
rambodazimi@Rambods-MacBook-Pro Desktop % python3 DnsClient.py -t 10 -r 2 -ns @8.8.8.8 mcgill.ca
DnsClient sending request for mcgill.ca
Server: 8.8.8.8
Request type: NS
Response received after 0.00875401496887207 seconds (0 retries)
***Answer Section (2 records)***
NS    pens2.mcgill.ca    3403    nonauth
NS    pens1.mcgill.ca    3403    nonauth
rambodazimi@Rambods-MacBook-Pro Desktop %
```

Figure 5. An example of the DnsClient with specified arguments and the result (timeout = 10, max retries = 2, -ns type google dns server at 8.8.8.8 and the domain name of mcgill.ca)

3. Testing

Users can also set arguments when running the DnsClient application using the terminal. As discussed in the previous section, we used the *argparse* module in Python to parse the arguments in the command line. Thus, the user can set timeout using -t or --timeout to specify how long to wait, in seconds, before retransmitting an unanswered query, as well as -r or --maxretries to indicate the maximum number of times to retransmit an unanswered packet. The user can also use -p or --port to set the port number when sending a request to the DNS server.

One of the most important tasks in this assignment was understanding each DNS packet's structure. As discussed in the DNS Client Program Design section and Figure 2, each packet has different sections with different values. We used *packet_builder()* and *unbuild_packet()* methods to set the correct values for each section and parse them to display the result to the user.
In order to build request packets, we used two functions in the struct module in Python: *pack ()* and *unpack_from()*. Each query has a unique ID which is in the Header section. We used a 16-bit random number to generate a random identifier for each packet to be sent. We set the flag to 256 (0x0100) because, in the query, only the RD bit flag is zero to indicate that we desire recursion. After completing the Header section, we separated the domain name into a sequence of labels to put the characters into the QNAME field in the Question section. As mentioned in the description, there are different query types, such as IP (type A), mail server (type MX), and name server (type NS). Then, we set QTYPE to 0x0001 to represent an Internet access.

We tested the application with different arguments, and communicating with the server and interpreting the response from the server worked without error. We also tried to handle the various errors that can occur during the process. For example, if there is a problem with the query type, the program will display an error message indicating that the query type is invalid. Also, the RCODE in the header section of each packet can distinguish different types of errors. In addition, when the time exceeds the threshold or the maximum number of retries reaches the end, the related error message will be printed on the terminal display. Figure 6 shows an example when the server tries to access an invalid hostname, and the program displays an error message.

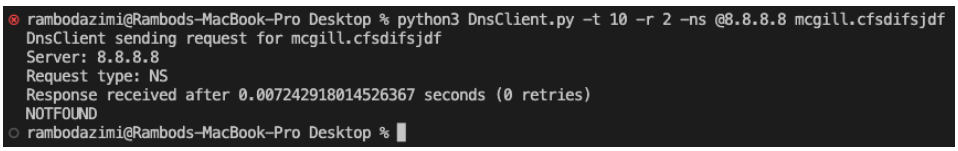


Figure 6. An example of a case when the user tries to get information from an invalid hostname

There are also some other tests which have yet to be covered by our design. For example, if the response received from the server to the client exceeds the limit of bytes (we set it to 512 bytes), there might be a problem. The second issue we did not cover in our implementation is that an error will be generated when the user runs the DnsClient program and tries to get some information at an invalid IP address. Figure 7 illustrates this problem on the command line.

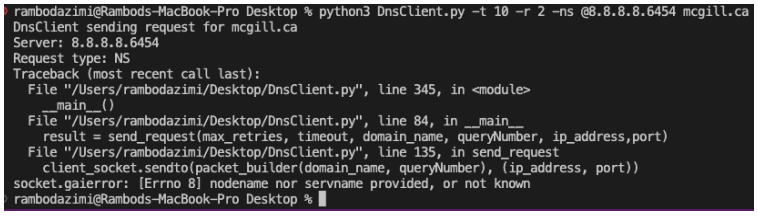


Figure 7. Error is generated when the user puts an invalid IP address as an argument

4. Experiment

a. What are the IP addresses of McGill's DNS servers? Use the Google public DNS server (8.8.8.8) to perform an 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?

```
rambodazimi@Rambods-MacBook-Pro Desktop % 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 0.010915040969848633 seconds (0 retries)
***Answer Section (2 records)***
NS    pens2.mcgill.ca    3600    nonauth
NS    pens1.mcgill.ca    3600    nonauth
rambodazimi@Rambods-MacBook-Pro Desktop %
```

Figure 8. IP addresses of McGill's DNS servers using Google public DNS server

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).

```
rambodazimi@Rambods-MacBook-Pro Desktop % python3 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 0.008314132690429688 seconds (0 retries)
***Answer Section (1 records)***
IP    172.217.13.132    75    nonauth
rambodazimi@Rambods-MacBook-Pro Desktop % python3 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 0.023778915405273438 seconds (0 retries)
***Answer Section (3 records)***
CNAME    tp.47cf2c8c9-frontier.amazon.com    1148    nonauth
CNAME    d3ag4hukkh62yn.cloudfront.net    60    nonauth
IP    13.224.33.5    60    nonauth
rambodazimi@Rambods-MacBook-Pro Desktop % python3 DnsClient.py @8.8.8.8 www.facebook.com
DnsClient sending request for www.facebook.com
Server: 8.8.8.8
Request type: A
Response received after 0.006886959075927734 seconds (0 retries)
***Answer Section (2 records)***
CNAME    star-mini.c10r.facebook.com    1407    nonauth
IP    31.13.71.36    21    nonauth
rambodazimi@Rambods-MacBook-Pro Desktop % python3 DnsClient.py @8.8.8.8 www.twitter.com
DnsClient sending request for www.twitter.com
Server: 8.8.8.8
Request type: A
Response received after 0.008311033248901367 seconds (0 retries)
***Answer Section (3 records)***
CNAME    twitter.com    511    nonauth
IP    104.244.42.1    1711    nonauth
IP    104.244.42.129    1711    nonauth
```

```
rambodazimi@Rambods-MacBook-Pro Desktop % python3 DnsClient.py @8.8.8.8 www.youtube.com
DnsClient sending request for www.youtube.com
Server: 8.8.8.8
Request type: A
Response received after 0.029746055603027344 seconds (0 retries)
***Answer Section (5 records)***
CNAME    youtube-ui.l.google.com    19810    nonauth
IP    172.217.13.174    300    nonauth
IP    172.217.13.206    300    nonauth
IP    172.217.13.110    300    nonauth
IP    172.217.13.142    300    nonauth
rambodazimi@Rambods-MacBook-Pro Desktop % python3 DnsClient.py @8.8.8.8 www.canada.ca
DnsClient sending request for www.canada.ca
Server: 8.8.8.8
Request type: A
Response received after 0.02196812629699707 seconds (0 retries)
***Answer Section (3 records)***
CNAME    www.canada.ca.edgekey.net    201    nonauth
CNAME    e4073.dscb.akamaiedge.net    249    nonauth
IP    23.1.200.101    20    nonauth
rambodazimi@Rambods-MacBook-Pro Desktop % python3 DnsClient.py @8.8.8.8 www.apple.com
DnsClient sending request for www.apple.com
Server: 8.8.8.8
Request type: A
Response received after 0.0411074161529541 seconds (0 retries)
***Answer Section (4 records)***
CNAME    www.apple.com.edgekey.net    1132    nonauth
CNAME    www.apple.com.edgekey.net.globalredir.akadns.net    19551    nonauth
CNAME    e6858.dscx.akamaiedge.net    3600    nonauth
IP    184.24.144.214    20    nonauth
rambodazimi@Rambods-MacBook-Pro Desktop %
```

Figure 9. Testing seven different website addresses using the Google public DNS server and the output

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 has several tasks, but one of its most essential features is to translate the hostname, which is humanly readable, to a unique IP address which consists of numbers. The DNS server is a distributed database divided into many sub-servers to reduce traffic and delay. When we first type a website on the browser to access the website's information, the domain name must be resolved. Then, the browser will send a request to the DNS server using UDP protocol and wait for a response.

In order to reduce the time of the mapping, instead of sending a request to the root DNS server and waiting for a response, we can cache some of the most popular IP addresses in the local DNS server so that it is not necessary anymore to ask the root server to resolve a hostname for us. It translates faster with less traffic, as the local DNS server is near the end system.

5. Discussion

To summarize, the DnsClient application will query a request to the DNS server with the specified hostname and other arguments (such as port, IP address, …) and interpret the output to the user in the command line. The program first outputs a summary of the query that is going to be sent to the server. Then, it displays subsequent lines summarizing the performance and content of the response.

The main challenge in this assignment was handling the *DNS packet structure*, which contains different sections. We had to build a structure used for sending a request to the server as well as unbuild the structure received from the server to extract helpful information from the packet received and to be able to interpret it to the user. Another challenging part of the implementation and the design was error handling. Many cases can cause a query not to be able to get an appropriate response from the server. We had to handle each case independently and react to them separately.