ECSE 316 Assignment 1 report
February 14th, 2020

Programming 36
260781081 Liang Zhao
260712639 Yudi Xie

1. Introduction

1.1. The objective of this assignment:

The main objectives of this assignment is to implement a DNS query system with UDP protocol, that can send query requests to a specific domain, and get responses from the domain.

1.2. The main result:

The results of each query are exactly what we expected. From the experiments we have so far, all of them can send and receive and display the results as described in the assignment instruction.

1.3. The main challenges:

We have two main challenges when implementing this assignment: 1. when parsing data from an answer, there might be more than one record that we need to print out, and since the domain names are compressed with pointers. therefore we need to recursively parse domain names. 2. We also had difficulties implementing the domain name algorithm. At the beginning, we implemented it with a while loop, but later we found that by using while loop, it can only get part of the domain name. Therefore, we used recursion rather than while loops.

2. DNS Client Program Design

2.1. The design of our DNS Client Program:

It is a command line program that parse arguments, includes timeout, port number, max retry, DNS host, and the domain name we are query for. We write the information to a bytebuffer, pack it into a datagrampacket and send it through a datagramsocket that we open up with our local server. This DNS Client will receive the response packet, and interpret it to the data we needed. DNS Client will first interpret the header, and get the count number of records. According to the number of records, we interpret the bytes after the header bytes to get results by checking with the structure of response given in dnsprimer. pdf in the assignment attachment.
We choose to put everything from this program to one single class, because this program is relatively small in size, and implementing it in one class makes it easier to compile in command line.

2.2. How can our Program handle errors:

We have two ways to handle errors: 1. We check the Rcode for getting the error message showing if there is something wrong with the response. 2. We use different exceptions that java provides, such as sockettimeoutexception. We can catch these exceptions when there are errors.

2.3. How can our program handle compressed domain names in DNS responses:

For parsing data from an answer, there might be more than one record that we need to print out, and since the domain names are compressed with pointers, we used a recursive algorithm to handle the compressed domain names problem.

3. Testing

3.1. Testing the different features of our client application:

In order to test the different features of our client application, we have created a list of test cases. The test cases may involve different query types, that is test the client with all A, NS, MX queries. And may also involve various timeout or max-tries arguments, for this feature, we have different test cases with different timeout or max-tries. We also made test cases for the request that may cause errors.

To validate the result of our test, we use an online DNS client *technitium.com*, to get an expected response from it, and then send the same request with our Java program, then compare both results.

3.2. Validating the formatting of the DNS request packet.

We followed the structure of the packet that is mentioned in *dnsprimer.pdf*, and in our application, we write the packet content byte by byte using mainly nio.ByteBuffer. We then use an integer as a pointer and bit calculation to parse the bits that we need for the result.

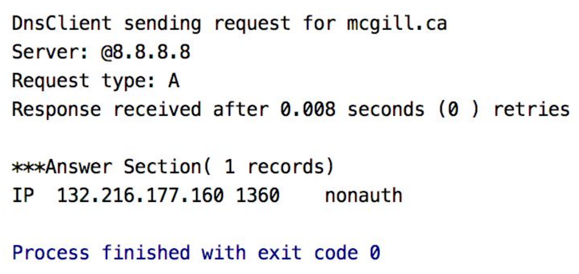
We used a naive method to validate the packet-making and packet-interpreting algorithm. The way is to print out the hex dump of both the response and request to the console, and we manually check the hex dump byte by byte to see if it is correct by checking with *dnsprimer.pdf*.

3.3: Not tested/worked features:

The additional record section reading is not tested in our application, the reason is that we cannot find a domain that sends the response with a record in the additional record section, therefore there is no way that we can interpret the additional record. The way we implement the additional record section reading is to let the pointer skip through the NS section, and move the pointer to the beginning of the additional record section, and then read the additional section the same as for the answer section. A naive way to test for the additional record section reading is to get a response that contains multiple answer records, and set one of the last records as an additional record, then simulate the case that the response has one additional record. In this simulated case, our application works perfectly. Other than this, we have tested everything, and everything works.

4. Experiment

Figure 4.1 shows the result for DnsClient @8.8.8.8 mcgill.ca

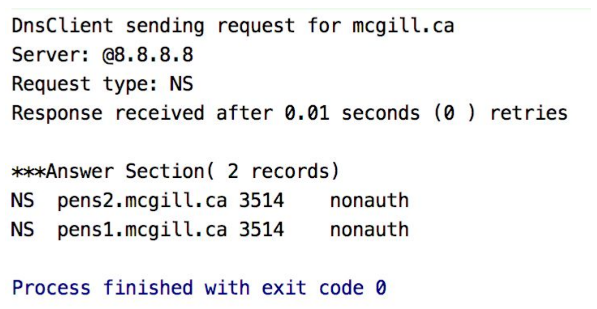


```
DnsClient sending request for mcgill.ca
Server: @8.8.8.8
Request type: A
Response received after 0.008 seconds (0 ) retries

***Answer Section( 1 records)
IP  132.216.177.160 1360    nonauth

Process finished with exit code 0
```

Figure 4.1

We can see that the result has one answer record, and it is A type. The IP address is 132.216.177.160.

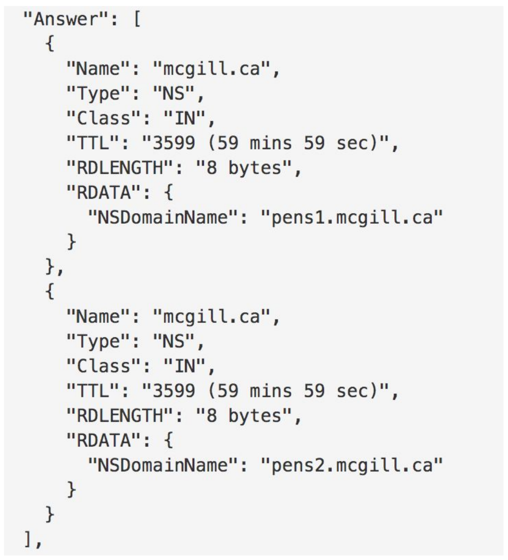
If we do a ns query, the result gives us two NS record, shows in Figure 4.2



```
DnsClient sending request for mcgill.ca
Server: @8.8.8.8
Request type: NS
Response received after 0.01 seconds (0 ) retries

***Answer Section( 2 records)
NS  pens2.mcgill.ca 3514    nonauth
NS  pens1.mcgill.ca 3514    nonauth

Process finished with exit code 0
```

Figure 4.2

We then used *technitium.com* to make the same request, the result is as figure 4.3:



```
"Answer": [
  {
    "Name": "mcgill.ca",
    "Type": "NS",
    "Class": "IN",
    "TTL": "3599 (59 mins 59 sec)",
    "RDLENGTH": "8 bytes",
    "RDATA": {
      "NSDomainName": "pens1.mcgill.ca"
    }
  },
  {
    "Name": "mcgill.ca",
    "Type": "NS",
    "Class": "IN",
    "TTL": "3599 (59 mins 59 sec)",
    "RDLENGTH": "8 bytes",
    "RDATA": {
      "NSDomainName": "pens2.mcgill.ca"
    }
  }
],
```

Figure 4.3

We can see that the RDATA matches with the results that we get from our client application.

For this part, we tried with seven common website, the result is shown in table 4.4:
In this table, we only show the number of answer records here.

Domain	Type	Expected	Recevied	Verified?
www.baidu.com	A	4	4	yes
www.bilibili.com	NS	1	1	yes
www.facebook.com	MX	1	1	yes
www.taobao.com	A	3	3	yes
www.github.com	NS	9	9	yes
www.twitter.com	MX	6	6	yes
www.qq.com	A	3	3	yes

Table 4.4

The results all matched.

We then move to the Mcgill campus to use the Mcgill DNS server to test, and the results are all matched as well.

5. Discussion

5.1: Key observations:

From the tests and experiments, we found the results are generally matched. The difference is that , if we have an A record, the IP address we get from the program is not the same as the IP address we found on technitium.com. Especially for large websites such as *google.com* or *baidu.com*. We found that it is normal because they do have float IP addresses that change when request coming.

5.2: Main results:

The results of each query are exactly what we expected. From the experiments we have so far, all of them can send and receive and display the results as described in the assignment instruction.

5.3: Main challenges in implementation:

We have two main challenges when implementing this assignment: 1. when parsing data from an answer, there might be more than one record that we need to print out, and since the domain names are compressed with pointers. therefore we need to recursively parse domain names. 2. We also had difficulties implementing the domain name algorithm. At the beginning, we implemented it with a while loop, but later we found that by using while loop, it can only get part of the domain name. Therefore, we used recursion rather than while loops.

5.4: possible different approach:

We used recursive functions to implement out DNSClient, but we can instead, using a stack to implement domain name parsing. By doing this, we could make our code more understandable and more flexible.