



UNIVERSITY OF DHAKA

Department of Computer Science and Engineering

CSE-3111 : Computer Networking Lab

Lab Report 4: **Distributed Database Management, Implementation of Iterative, and Recursive Queries of DNS Records**

Submitted By:

Joty Saha (Roll-51)
Ahona Rahman (Roll-59)

Submitted To:

Dr. Md. Abdur Razzaque
Md. Mahmudur Rahman
Md. Ashraful Islam
Md. Fahim Arefin

Submitted On: February 14, 2024

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 3 |
| 1.1 | Objectives | 3 |
| 2 | Theory | 4 |
| 2.1 | Domain Name Structure: | 4 |
| 2.2 | DNS Resolution Process: | 4 |
| 3 | Methodology | 6 |
| 3.1 | Part 1: Setting up the DNS server | 6 |
| 3.2 | Part 2: Iterative DNS resolution | 11 |
| 3.3 | Part 3: Recursive DNS resolution | 21 |
| 3.4 | Part 4: Extending the System | 31 |
| 3.4.1 | Use a short TTL value and try Deleting resource record based on TTL value: | 31 |
| 3.4.2 | Implement DNS caching in local and TLD servers: | 33 |
| 3.4.3 | Test failure of a DNS server process: | 37 |
| 4 | Experimental result | 41 |
| 4.1 | Task 1- Setting up the DNS server | 41 |
| 4.1.1 | 'dns records.txt' File: | 41 |
| 4.1.2 | After Running Server Code: | 41 |
| 4.1.3 | After Running Client Code: | 42 |
| 4.1.4 | After entering target address, in client: | 42 |
| 4.1.5 | After entering target address, in server: : | 43 |
| 4.1.6 | Error for server | 43 |
| 4.1.7 | Error for client | 44 |
| 4.2 | Task 2- Iterative DNS resolution | 45 |
| 4.2.1 | After Running rootSever Code: | 45 |
| 4.2.2 | After Running tld Code: | 45 |
| 4.2.3 | After Running authoritative Code: | 45 |
| 4.2.4 | After Running localDnsSever Code: | 45 |
| 4.2.5 | After Running client Code: | 46 |
| 4.2.6 | After client entering address: | 46 |
| 4.2.7 | Response from Root server: | 46 |
| 4.2.8 | Response from tld server: | 47 |
| 4.2.9 | Response from authoratitive server: | 47 |
| 4.2.10 | Error | 48 |
| 4.3 | Task 3- Recursive DNS resolution | 49 |
| 4.3.1 | After client entering address: | 49 |
| 4.3.2 | Response from Root server: | 49 |
| 4.3.3 | Response from tld server: | 50 |
| 4.3.4 | Response from authoratitive server: | 50 |
| 4.3.5 | Error | 50 |
| 4.4 | Task 4- Extending the System | 51 |
| 4.4.1 | After running ttl server code: | 51 |
| 4.4.2 | After running ttl client code: | 51 |
| 4.4.3 | After running cached client code: | 52 |
| 4.4.4 | Error cached client code: | 52 |
| 4.4.5 | After running client code: | 53 |
| 4.4.6 | After responding server code: | 53 |

1 Introduction

In the realm of computer networks and distributed systems, the Domain Name System (DNS) plays a pivotal role in translating human-readable domain names into IP addresses, facilitating the efficient routing of data across the internet. The management of DNS records is critical for maintaining the integrity and accessibility of web services, and it often involves the use of distributed database management systems.

Distributed Database Management Systems (DDBMS) are designed to handle large volumes of data spread across multiple locations, providing scalability, fault tolerance, and high availability. These systems are employed in various applications, including DNS management, to ensure efficient data storage, retrieval, and synchronization across distributed environments.

This experiment focuses on the implementation of iterative and recursive queries for DNS records within a distributed database management context. Iterative and recursive queries are two fundamental approaches used by DNS resolvers to resolve domain names into IP addresses.

1.1 Objectives

Here are some objectives for the experiment:

1. **Understanding Distributed Database Management Principles:** Gain insights into the fundamental concepts and principles of distributed database management systems (DDBMS), including data distribution, replication, consistency, and fault tolerance.
2. **Implementation of DNS Record Management:** Implement mechanisms for storing, updating, and retrieving DNS records within a distributed database management system.
3. **Iterative Query Implementation:** Develop algorithms and mechanisms to perform iterative DNS queries within the distributed database environment, simulating the iterative resolution process used by DNS resolvers.
4. **Recursive Query Implementation:** Implement algorithms and mechanisms for performing recursive DNS queries within the distributed database system, mimicking the recursive resolution process employed by DNS resolvers.
5. **Concurrency Control and Transaction Management:** Explore concurrency control mechanisms and transaction management strategies within the distributed database system to ensure data consistency and integrity during DNS record updates and queries.
6. **Fault Tolerance and Replication:** Investigate fault tolerance mechanisms such as data replication and redundancy to ensure continuous availability and reliability of DNS records, even in the presence of server failures or network partitions.
7. **Security and Access Control:** Implement security measures and access control mechanisms to protect DNS records from unauthorized access, tampering, or malicious attacks, ensuring the confidentiality, integrity, and authenticity of the DNS data.
8. **Integration with DNS Infrastructure:** Integrate the distributed database management system with existing DNS infrastructure, such as authoritative DNS servers and caching resolvers, to demonstrate interoperability and compatibility with real-world DNS operations.

These objectives aim to provide a structured approach to exploring distributed database management principles and their application in the context of DNS record management and resolution.

2 Theory

The Domain Name System (DNS) is a hierarchical decentralized naming system for computers, services, or any resource connected to the Internet or a private network. It translates more readily memorized domain names to the numerical IP addresses needed for locating and identifying computer services and devices with the underlying network protocols.

2.1 Domain Name Structure:

DNS names are organized in a hierarchical structure, with each level separated by a dot (.), forming a domain name. For example, in the domain name "www.example.com":

- "www" is a **hostname**.
- "example" is the **second-level domain (SLD)**.
- "com" is the **top-level domain (TLD)**.

2.2 DNS Resolution Process:

A DNS query begins when a user types a domain name into a web browser or clicks on a hyperlink. The user's computer sends a request to a DNS resolver, which is usually provided by the user's Internet service provider (ISP). The resolver then forwards the request to a series of DNS servers, starting with the root DNS servers and working its way down the hierarchy until it finds the server that is authoritative for the domain name in question. The authoritative server returns the IP address corresponding to the domain name, which the resolver then passes back to the user's computer.

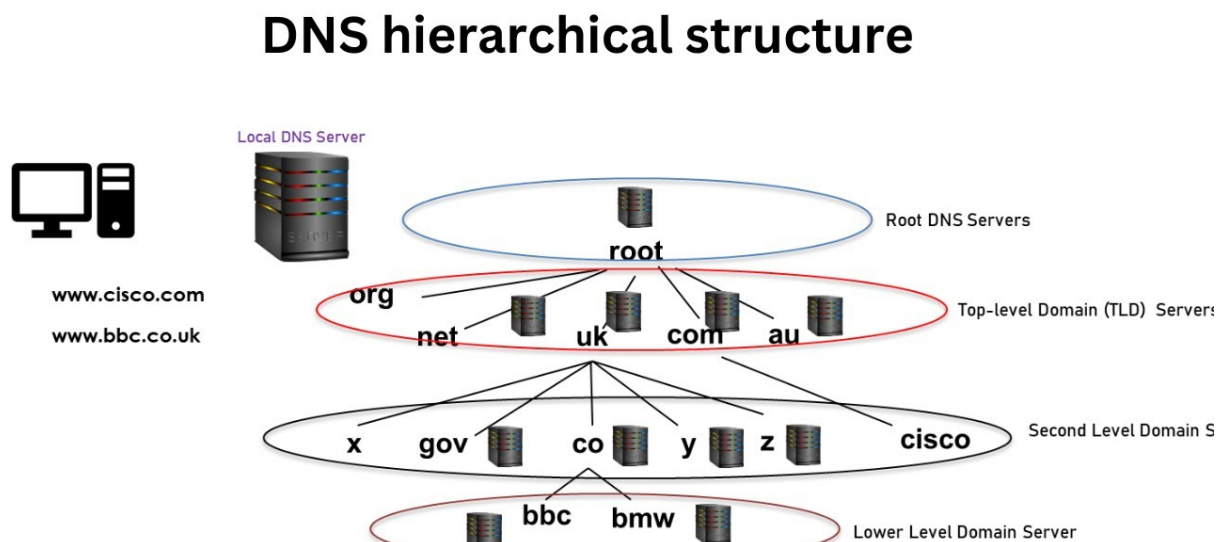


Figure 1: Hiercharchy of DNS

When a user attempts to access a website using a domain name, their device initiates a DNS resolution process to translate the domain name into an IP address. This process typically involves the following steps:

1. **DNS Cache Search:** Initially, when a domain name like *google.com* is queried, the local computer checks its DNS cache to see if it already has the corresponding IP address stored. This step emphasizes the importance of caching in improving DNS query performance and reducing network traffic.
2. **ISP's DNS Servers:** If the information is not found in the local cache, the computer queries the DNS servers provided by its Internet Service Provider (ISP). These servers are responsible for resolving DNS queries on behalf of their clients and may cache frequently accessed records to improve performance.
3. **Root Nameservers:** If the ISP's DNS servers do not have the requested information, they query the root nameservers. These root servers maintain a global directory of authoritative DNS servers responsible for top-level domains (TLDs) like .com, .org, etc. This step highlights the hierarchical nature of DNS resolution and the distributed nature of the DNS infrastructure.
4. **Top-Level Domain (TLD) Nameservers:** The root nameservers direct the query to the TLD nameservers responsible for the specific TLD of the domain being queried (e.g., .com for *google.com*). Each TLD has its own set of authoritative nameservers, which manage DNS records for domains within that TLD.
5. **Authoritative DNS Servers:** The TLD nameservers then forward the query to the authoritative nameservers for the specific domain (e.g., *google.com*). These authoritative servers store the most up-to-date DNS records for the domain, including the mapping of domain names to IP addresses.
6. **Record Retrieval and Caching:** The authoritative DNS servers retrieve the requested DNS record (e.g., IP address for *google.com*) and return it to the ISP's DNS server. The ISP's server caches this information locally to expedite future queries for the same domain. However, DNS records have expiration times to ensure that outdated information is not used, necessitating periodic re-querying for updated records.
7. **Answer Reception:** Finally, the ISP's DNS server returns the DNS record to the requesting computer, which then stores it in its cache. The computer extracts the IP address from the record and passes it to the web browser, enabling the browser to establish a connection with the webserver hosting the requested website.

This experiment will focus on implementing iterative and recursive querying mechanisms within a distributed database management system, simulating the steps involved in DNS resolution. By studying these processes, participants will gain insights into distributed database management principles and their application in the context of DNS record management and resolution.

3 Methodology

Here's the methodology for the experiment on Distributed Database Management and Implementation of Iterative and Recursive Queries of DNS Records:

- 1. Thread Creation for DNS Servers:** Begin by creating a separate thread for each DNS server in the distributed system. These threads will handle incoming DNS queries from clients asynchronously, allowing the servers to handle multiple requests concurrently.
- 2. Tree Structure of DNS Servers:** Establish a hierarchical structure of DNS servers resembling a tree, with a root node representing the highest level of the DNS hierarchy. Each DNS server in the tree will have references to its parent server (if any) and child servers, enabling the servers to forward DNS queries as needed.
- 3. Parent-Child Relationships:** Assign each DNS server its parent server and child servers based on the hierarchical structure. This relationship will facilitate the iterative or recursive resolution process, where DNS queries are forwarded from child servers to parent servers as necessary until a resolution is achieved.
- 4. Awaiting Client Requests:** Each DNS server awaits DNS query requests from clients. Upon receiving a request, the server initiates the resolution process by searching for the requested domain name within its own database.
- 5. Domain Name Resolution:** If the requested domain name is found in the current DNS server's database, the server returns the corresponding IP address to the client. If not found, the server checks its child servers (if any) for the domain name. If still not found, the server forwards the query to its parent server.
- 6. Iterative or Recursive Resolution:** Depending on the experimental setup, the resolution process can be implemented iteratively or recursively. In iterative resolution, each DNS server forwards the query to its parent server and awaits a response. In recursive resolution, DNS servers recursively query their parent servers until a resolution is obtained.
- 7. Error Handling:** If the domain name cannot be found in any of the DNS servers in the distributed system, the client is notified with an error message indicating that the requested domain does not exist.
- 8. Successful Resolution:** If the domain name is found in any of the DNS servers during the resolution process, the corresponding IP address is returned to the requesting client, completing the DNS query process.

3.1 Part 1: Setting up the DNS server

1. Configure the DNS server to act as an authoritative server for a domain (e.g., cse.du.ac.bd).
2. Add A, AAAA, CNAME, and MX records for the domain in a file.
 - Use IP address of your friends' PC as the IP address and their name as the domain name.
 - We have added an example file dnsRecords.txt. This file represents a simple DNS zone file for the domain "cse.du.ac.bd". It includes DNS records for the domain in (Name, Value, Type, TTL) format.
3. Start the DNS server.

4. Verify that the DNS server is running and that it can resolve queries for the domain when requested by a client via UDP socket [Hint: ‘DatagramSocket’ and ‘DatagramPacket’ classes provided by the ‘java.net’ package]
5. For exchanging message between DNS server and client, use the following format:
6. Modify the DNS server to perform as root and TLD server.

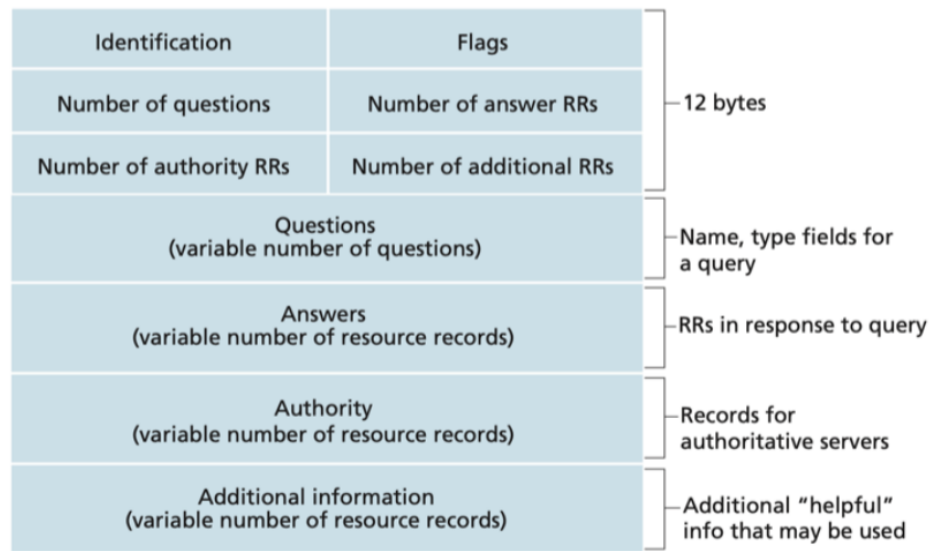


Figure 2: DNS Message Format

Listing 1: DNS setting Server Code

```

1 import os
2 import socket
3 import threading
4 import struct
5 import time
6
7 IP = '192.168.1.194'
8 PORT = 4487
9 ADDR = (IP, PORT)
10 SIZE = 1024
11 FORMAT = "utf-8"
12 SERVER_DATA_PATH = "server_data"
13 dic = {}
14
15
16 def handle_client(data, addr, server):
17     try:
18         request_time = time.strftime("%Y-%m-%d %H:%M:%S",
19                                     time.localtime())
20         print(f"[RECEIVED MESSAGE] {data} from {addr} at {
21               request_time}.")
22
23         data = data.split()
24         print("Request Time:", request_time)
25         domain_name = data[0]
26         print("Domain Name:", domain_name)
27
28         file1 = open('dns_records.txt', 'r')
29         found = False
30         for line in file1:
31             line = line.split()
32             name = line[0]
33             value = line[1]
34             type = line[2]
35             ttl = line[3]
36             if name == domain_name and type == data[1]:
37                 print('Found DNS Record')
38                 flag = 0
39                 q = 0
40                 a = 1
41                 auth_rr = 0
42                 add_rr = 0
43
44                 # Pack DNS header fields and message into the
45                 same buffer
46                 ms = (name + ' ' + value + ' ' + type + ' ' +
47                       ttl).encode('utf-8')
48                 packed_data = struct.pack(f"6H{len(ms)}s",
49                                           50, flag, q, a, auth_rr, add_rr, ms)

```

```

45         server.sendto(packed_data, addr)
46         found = True
47         break
48
49
50     if not found:
51         # If no matching DNS record found, send an empty
52         response
53         print('No DNS Record Found')
54         server.sendto(b'', addr)
55
56 except Exception as e:
57     print("Error occurred while handling client request:"
58           , e)
59
60 def main():
61     print("[STARTING] Server is starting")
62     server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
63     #SOCK_DGRAM-> use UDP instead of TCP
64     server.bind(ADDR)
65     print(f"[LISTENING] Server is listening on {IP}:{PORT}.")
66
67     while True:
68         try:
69             data, addr = server.recvfrom(SIZE)
70             data = data.decode(FORMAT)
71             thread = threading.Thread(target=handle_client,
72                                       args=(data, addr, server))
73             thread.start()
74             print(f"[ACTIVE CONNECTIONS] {threading.
75                   active_count() - 1}")
76
77         except Exception as e:
78             print("Error occurred while accepting client
79                   connection:", e)
80
81 if __name__ == "__main__":
82     main()

```

Listing 2: DNS Setting Client Code

```

1 import socket
2 import struct
3 import time
4
5 ADDR = ('192.168.1.194', 4487)
6 SIZE = 1024
7 FORMAT = 'utf-8'
8
9
10 def main():
11     try:
12         client = socket.socket(socket.AF_INET, socket.
13                                SOCK_DGRAM)
14
15         message = input("Enter a message to send to the
16                          server: ")
17         client.sendto(message.encode(FORMAT), ADDR)
18         request_time = time.strftime("%Y-%m-%d %H:%M:%S",
19                                     time.localtime())
20
21         msg, addr = client.recvfrom(SIZE)
22         print('Received DNS Response Message in bytes:')
23         print(msg)
24
25         header = struct.unpack("6H", msg[:12])
26         ms = msg[12:].decode('utf-8').split()
27         print('\nDecoded DNS Response:')
28         print("Header:", header)
29         print("Request Time:", request_time)
30         print("Domain Name:", ms[0])
31         if len(ms) > 1:
32             print("DNS Record:", ms[1])
33             if len(ms) > 2:
34                 print("Additional Information:", ms[2])
35
36     except Exception as e:
37         print("Error occurred while communicating with the
38              server:", e)
39
40 if __name__ == '__main__':
41     main()

```

3.2 Part 2: Iterative DNS resolution

1. Write a script that sends a DNS query to the root DNS server.
2. The root DNS server will respond with a referral to a top-level domain (TLD) DNS server.
3. The script will then send the query to the TLD DNS server.
4. The TLD DNS server will respond with a referral to the authoritative DNS server for the domain.
5. The script will then send the query to the authoritative DNS server.
6. The authoritative DNS server will respond with the IP address for the domain.
7. Verify that the script correctly implements iterative DNS resolution.

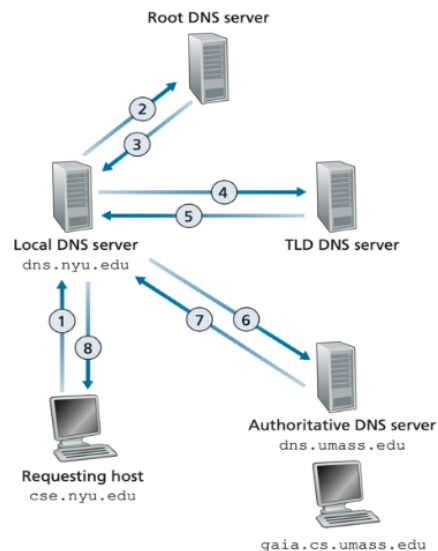


Figure 3: Iterative DNS Resolution

Listing 3: Iterative Local DNS Server Code

```

1 import socket
2 import struct
3
4 ROOT_DNS_ADDR = ('10.33.3.11', 4487)
5 TLD_DNS_ADDR = ('10.33.3.11', 4488)
6 AUTH_DNS_ADDR = ('10.33.3.11', 4489)
7
8 def encode_msg(message):
9     data = message.split()
10    name = data[0]
11    type = data[1]
12    flag = 0
13    q = 0
14    a = 1
15    auth_rr = 0
16    add_rr = 0
17    ms = (name + ' ' + type + ' ' + data[2] + ' ' + data[3]).
        encode('utf-8')
18    packed_data = struct.pack(f"6H{len(ms)}s", 50, flag, q, a
        , auth_rr, add_rr, ms)
19    return packed_data
20
21 def resolve_domain(domain_name):
22     try:
23         # Query Root DNS Server
24         with socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            as root_server:
25             root_server.sendto(domain_name.encode(),
                ROOT_DNS_ADDR)
26             root_response, _ = root_server.recvfrom(1024)
27             tld_addr = root_response.decode()
28
29         # Query TLD DNS Server
30         with socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            as tld_server:
31             tld_server.sendto(domain_name.encode(),
                TLD_DNS_ADDR)
32             tld_response, _ = tld_server.recvfrom(1024)
33             auth_addr = tld_response.decode()
34
35         # Query Authoritative DNS Server
36         with socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            as auth_server:
37             auth_server.sendto(domain_name.encode(),
                AUTH_DNS_ADDR)
38             auth_response, _ = auth_server.recvfrom(1024)
39             ip_address = auth_response.decode()
40
41     return ip_address

```

```

42
43     except Exception as e:
44         print("Error occurred while resolving domain:", e)
45         return None
46
47 def main():
48     try:
49         server = socket.socket(socket.AF_INET, socket.
50             SOCK_DGRAM)
51         server.bind(('localhost', 5000))
52
53         print("[LISTENING] Local DNS Server is listening on
54             port 5000...")
55
56         while True:
57             data, addr = server.recvfrom(1024)
58             domain_name = data.decode()
59             print(f"[RECEIVED QUERY] {domain_name} from {addr
60                 }")
61
62             # Resolve domain iteratively
63             ip_address = resolve_domain(domain_name)
64
65             if ip_address:
66                 # Send IP address to client
67                 server.sendto(ip_address.encode(), addr)
68                 print(f"Sent IP address of {domain_name} to {
69                     addr}")
70
71     except Exception as e:
72         print("Error occurred in main:", e)
73
74 if __name__ == "__main__":
75     main()

```

Listing 4: Iterative Root Server Code

```

1 import os
2 import socket
3 import threading
4 import struct
5
6 IP = '10.33.3.11'
7 PORT = 4487
8 ADDR = (IP, PORT)
9 tld = (IP, 4488)
10 SIZE = 1024
11 FORMAT = "utf-8"
12 SERVER_DATA_PATH = "server_data"

```

```

13 dic = {
14     "www.google.com": ('100.20.8.1', 'A', 86400),
15     "www.cse.du.ac.bd": ('4488', 'NS', 86400),
16     "www.yahoo.com": ('4488', "NS", 86400)
17 }
18
19
20 def handle_client(data, addr, server):
21     try:
22         print(f"[RECEIVED MESSAGE] {data} from {addr}.")
23         msg = encode_msg(str(data + ' ' + dic[data][0] + ' ' +
24                               + dic[data][1] + ' ' + str(dic[data][2])))
25         server.sendto(msg, addr)
26
27     except Exception as e:
28         print("ERROR: ", str(e))
29
30
31 def encode_msg(message):
32     data = message.split()
33     name = data[0]
34     type = data[1]
35     print(message)
36     flag = 0
37     q = 0
38     a = 1
39     auth_rr = 0
40     add_rr = 0
41
42     ms = (name + ' ' + type + ' ' + data[2] + ' ' + data[3]).
43         encode('utf-8')
44     packed_data = struct.pack(f"6H{len(ms)}s", 50, flag, q, a
45                               , auth_rr, add_rr, ms)
46     return packed_data
47
48
49 def decode_msg(msg):
50     header = struct.unpack("6H", msg[:12])
51     ms = msg[12:].decode('utf-8')
52     print('\n After Decoding')
53     print({header}, {ms})
54     ms = ms.split()
55     return ms[1], ms[4]
56
57
58 def main():
59     print("[STARTING] ROOT Server is starting")
60     server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
61     server.bind(ADDR)

```

```

60     print(f"[LISTENING] ROOT Server is listening on {IP}:{
        PORT}.")
61
62     while True:
63         data, addr = server.recvfrom(SIZE)
64         data = data.decode(FORMAT)
65         # thread = threading.Thread(target=handle_client,
            args=(data, addr, server))
66         # thread.start()
67         handle_client(data, addr, server)
68         print(f"[ACTIVE CONNECTIONS] {threading.active_count
            () - 1}")
69
70
71 if __name__ == "__main__":
72     main()

```

Listing 5: Iterative Authoritative Server Code

```

1  import os
2  import socket
3  import threading
4  import struct
5
6  IP = '10.33.3.11'
7  PORT = 4489
8  ADDR = (IP, PORT)
9  SIZE = 1024
10 FORMAT = "utf-8"
11 SERVER_DATA_PATH = "server_data"
12 dic = {
13     "www.google.com": ('100.20.8.1', 'A', 86400),
14     "www.cse.du.ac.bd": ('192.0.2.3', 'A', 86400),
15     "www.yahoo.com": ('1.2.3.9999', "A", 86400)
16 }
17
18
19 def encode_msg(message):
20     print(message)
21     data = message.split()
22     name = data[0]
23     type = data[1]
24     print(message)
25     flag = 0
26     q = 0
27     a = 1
28     auth_rr = 0
29     add_rr = 0
30

```



```

31     ms = (name + ' ' + type + ' ' + data[2] + ' ' + data[3]).
        encode('utf-8')
32     packed_data = struct.pack(f"6H{len(ms)}s", 50, flag, q, a
        , auth_rr, add_rr, ms)
33     return packed_data
34
35
36 def decode_msg(msg):
37     header = struct.unpack("6H", msg[:12])
38     ms = msg[12:].decode('utf-8')
39     print('\n After Decoding')
40     print({header}, {ms})
41     ms = ms.split()
42     return ms[1], ms[4]
43
44
45 def handle_client(data, addr, server):
46     try:
47         print(data)
48         msg = encode_msg(str(data + ' ' + dic[data][0] + ' '
        + dic[data][1] + ' ' + str(dic[data][2])))
49         server.sendto(msg, addr)
50
51         # print(f"[RECEIVED MESSAGE] {data} from {addr}.")
52         # if dic[data][1]=='A' or dic[data][1]=='AAAA':
53         #     print('sending')
54         #     server.sendto(str(data+' '+dic[data][0]+' '+dic
        [data][1]+' '+str(dic[data][2])).encode(FORMAT),
        addr)
55     except Exception as e:
56         print("ERROR: ", str(e))
57
58
59 def main():
60     print("[STARTING] auth Server is starting")
61     server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
62     server.bind(ADDR)
63     print(f"[LISTENING] auth Server is listening on {IP}:{
        PORT}.")
64
65     while True:
66         data, addr = server.recvfrom(SIZE)
67         data = data.decode(FORMAT)
68         # thread = threading.Thread(target=handle_client,
        args=(data, addr, server))
69         # thread.start()
70         handle_client(data, addr, server)
71         print(f"[ACTIVE CONNECTIONS] {threading.active_count
        () - 1}")
72

```

```

73
74 if __name__ == "__main__":
75     main()

```

Listing 6: Iterative TLD Server Code

```

1  import os
2  import socket
3  import threading
4  import struct
5  import time
6
7  IP = '10.33.3.11'
8  PORT = 4488
9  authoratitive = (IP, 4489)
10 ADDR = (IP, PORT)
11 SIZE = 1024
12 FORMAT = "utf-8"
13 SERVER_DATA_PATH = "server_data"
14 dic = {
15     "www.google.com": ('100.20.8.1', 'A', 86400),
16     "www.cse.du.ac.bd": ('192.0.2.3', 'A', 86400),
17     "www.yahoo.com": ('4489', "NS", 86400)
18 }
19
20
21 def encode_msg(message):
22     data = message.split()
23     name = data[0]
24     type = data[1]
25     print(message)
26     flag = 0
27     q = 0
28     a = 1
29     auth_rr = 0
30     add_rr = 0
31
32     ms = (name + ' ' + type + ' ' + data[2] + ' ' + data[3]).
33         encode('utf-8')
34     packed_data = struct.pack(f"6H{len(ms)}s", 50, flag, q, a
35         , auth_rr, add_rr, ms)
36     return packed_data
37
38 def decode_msg(msg):
39     header = struct.unpack("6H", msg[:12])
40     ms = msg[12:].decode('utf-8')
41     print('\n After Decoding')
42     print({header}, {ms})

```

```

42     return ms
43
44
45 def handle_client(data, addr, server):
46     try:
47         print(f"[RECEIVED MESSAGE] {data} from {addr}.")
48         msg = encode_msg(str(data + ' ' + dic[data][0] + ' ' +
49                               + dic[data][1] + ' ' + str(dic[data][2])))
50         server.sendto(msg, addr)
51
52         # if dic[data][1]=='A' or dic[data][1]=='AAAA':
53         #     print('sending')
54         #     server.sendto(str(data+' '+dic[data][0]+' '+dic
55                               [data][1]+' '+str(dic[data][2])).encode(FORMAT),
56                               addr)
57         # elif dic[data][1]=='NS':
58         #     server.sendto(data.encode(FORMAT),authoritative
59                               )
60         #     ans, auth_addr = server.recvfrom(SIZE)
61         #     server.sendto(ans,addr)
62     except Exception as e:
63         server.sendto(data.encode(FORMAT), ('', 4487))
64
65         print("ERROR: ", str(e))
66
67
68 def main():
69     print("[STARTING] TLD Server is starting")
70     server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
71     server.bind(ADDR)
72     print(f"[LISTENING] TLD Server is listening on {IP}:{PORT
73           }.")
74
75     while True:
76         data, addr = server.recvfrom(SIZE)
77         data = data.decode(FORMAT)
78         # thread = threading.Thread(target=handle_client,
79           args=(data, addr,server))
80         # thread.start()
81         handle_client(data, addr, server)
82         print(f"[ACTIVE CONNECTIONS] {threading.active_count
83               () - 1}")
84
85
86 if __name__ == "__main__":
87     main()

```

Listing 7: Iterative Client Code

```

1 import socket
2 import struct
3
4 IP = ''
5 PORT = 4487
6 ADDR = (IP, PORT)
7 SIZE = 1024
8 FORMAT = "utf-8"
9
10
11 def encode_msg(message):
12     data = message.split()
13     name = data[0]
14     type = data[1]
15
16     flag = 0
17     q = 0
18     a = 1
19     auth_rr = 0
20     add_rr = 0
21
22     ms = (name + ' ' + type).encode('utf-8')
23     packed_data = struct.pack(f"6H{len(ms)}s", 50, flag, q, a
24                               , auth_rr, add_rr, ms)
25     return packed_data
26
27 def decode_msg(msg):
28     header = struct.unpack("6H", msg[:12])
29     ms = msg[12:].decode('utf-8')
30
31     print('\n Before Decoding')
32     print(msg)
33
34     print('\n After Decoding')
35     print({header}, {ms})
36
37     return ms
38
39
40 def main():
41     client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
42
43     message = input("Enter an address: ")
44     client.sendto(message.encode(FORMAT), ADDR)
45     msg, addr = client.recvfrom(SIZE)
46     msg1 = decode_msg(msg)
47     print(msg1)
48     data = msg1.split()

```

```

49     while data[2] == "NS":
50         # print(data[1])
51         new_addr = ('', int(data[1]))
52         print('Connecting to port', data[1])
53         client.sendto(message.encode(FORMAT), new_addr)
54         msg, addr = client.recvfrom(SIZE)
55         msg1 = decode_msg(msg)
56         print(msg1)
57         data = msg1.split()
58
59         # print(struct.unpack("6H",msg))
60         # msg,addr=client.recvfrom(SIZE)
61
62
63 if __name__ == "__main__":
64     main()

```

3.3 Part 3: Recursive DNS resolution

1. Modify the script from Part 2 to send a recursive DNS query to a recursive DNS resolver.
2. The recursive DNS resolver will send queries to the root, TLD, and authoritative DNS servers on behalf of the script.
3. The recursive DNS resolver will respond with the IP address for the domain.
4. Verify that the script correctly implements recursive DNS resolution.

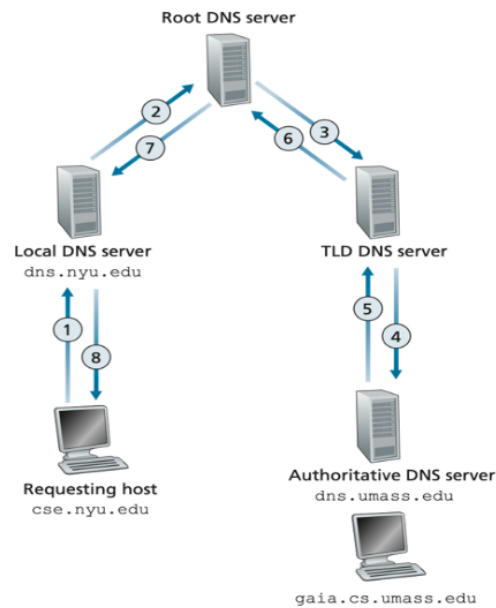


Figure 4: Recursive DNS Resolution

Listing 8: Recursive Local DNS Server Code

```

1 import socket
2 import struct
3
4 ROOT_DNS_ADDR = ('127.0.0.1', 4487)
5
6
7 def encode_msg(message):
8     try:
9         data = message.split()
10        name = data[0]
11        type = data[1]
12        flag = 0
13        q = 0
14        a = 1
15        auth_rr = 0
16        add_rr = 0
17        ms = (name + ' ' + type + ' ' + data[2] + ' ' + data
18              [3]).encode('utf-8')
19        packed_data = struct.pack(f"6H{len(ms)}s", 50, flag,
20                                q, a, auth_rr, add_rr, ms)
21        return packed_data
22    except Exception as e:
23        print("Error occurred while encoding message:", e)
24        return None
25
26 def resolve_domain(domain_name, server_addr):
27     try:
28         with socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
29             as dns_server:
30             dns_server.sendto(domain_name.encode(),
31                               server_addr)
32             response, _ = dns_server.recvfrom(1024)
33             return response.decode(), server_addr[1]
34
35     except Exception as e:
36         print("Error occurred while resolving domain:", e)
37         return None, None
38
39 def main():
40     try:
41         server = socket.socket(socket.AF_INET, socket.
42                                SOCK_DGRAM)
43         server.bind(('localhost', 5000))
44
45         print("[LISTENING] Local DNS Server is listening on
46               port 5000...")
47

```

```

44     while True:
45         try:
46             data, addr = server.recvfrom(1024)
47             domain_name = data.decode()
48             print(f"[RECEIVED QUERY] {domain_name} from {
               addr[0]} for target {addr[1]}")
49
50             ip_address, port = resolve_domain(domain_name
               , ROOT_DNS_ADDR)
51
52             if ip_address and port:
53                 server.sendto(ip_address.encode(), addr)
54                 print(f"Sent IP address of {domain_name}
               to {addr}")
55
56             except Exception as e:
57                 print("Error occurred in query processing:",
               e)
58
59     except KeyboardInterrupt:
60         print("Server terminated by user.")
61     except Exception as e:
62         print("Error occurred in main:", e)
63     finally:
64         server.close()
65
66
67 if __name__ == "__main__":
68     main()

```

Listing 9: Recursive Root Server Code

```

1  import os
2  import socket
3  import threading
4  import struct
5
6  IP = '127.0.0.1'
7  PORT = 4487
8  ADDR = (IP, PORT)
9  tld = (IP, 4488)
10 SIZE = 1024
11 FORMAT = "utf-8"
12 SERVER_DATA_PATH = "server_data"
13 dic = {
14     "www.google.com": ('100.20.8.1', 'A', 86400),
15     "www.cse.du.ac.bd": ('4488', 'NS', 86400),
16     "www.yahoo.com": ('4488', "NS", 86400)
17 }

```



```

18
19
20 def handle_client(data, addr, server):
21     try:
22         print(f"[RECEIVED MESSAGE] {data} from {addr}.")
23         msg = encode_msg(str(data + ' ' + dic[data][0] + ' ' +
24                               + dic[data][1] + ' ' + str(dic[data][2])))
25         server.sendto(msg, addr)
26     except KeyError:
27         print(f"Requested domain '{data}' not found in root
28               DNS.")
29     except Exception as e:
30         print("ERROR: ", str(e))
31
32 def encode_msg(message):
33     try:
34         data = message.split()
35         name = data[0]
36         type = data[1]
37         print(message)
38         flag = 0
39         q = 0
40         a = 1
41         auth_rr = 0
42         add_rr = 0
43
44         ms = (name + ' ' + type + ' ' + data[2] + ' ' + data
45               [3]).encode('utf-8')
46         packed_data = struct.pack(f"6H{len(ms)}s", 50, flag,
47                                   q, a, auth_rr, add_rr, ms)
48         return packed_data
49     except Exception as e:
50         print("Error occurred while encoding message:", e)
51         return None
52
53 def decode_msg(msg):
54     try:
55         header = struct.unpack("6H", msg[:12])
56         ms = msg[12:].decode('utf-8')
57         print('\n After Decoding')
58         print({header}, {ms})
59         ms = ms.split()
60         return ms[1], ms[4]
61     except Exception as e:
62         print("Error occurred while decoding message:", e)
63         return None, None

```

```

64 def main():
65     print("[STARTING] ROOT Server is starting")
66     server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
67     server.bind(ADDR)
68     print(f"[LISTENING] ROOT Server is listening on {IP}:{PORT}.")
69
70     while True:
71         try:
72             data, addr = server.recvfrom(SIZE)
73             data = data.decode(FORMAT)
74             handle_client(data, addr, server)
75             print(f"[ACTIVE CONNECTIONS] {threading.
                active_count() - 1}")
76         except KeyboardInterrupt:
77             print("Server terminated by user.")
78             break
79         except Exception as e:
80             print("Error occurred:", e)
81
82     server.close()
83
84
85 if __name__ == "__main__":
86     main()

```

Listing 10: Recursive Authoritative Server Code

```

1  import os
2  import socket
3  import threading
4  import struct
5
6  IP = '127.0.0.1'
7  PORT = 4489
8  ADDR = (IP, PORT)
9  SIZE = 1024
10 FORMAT = "utf-8"
11 SERVER_DATA_PATH = "server_data"
12 dic = {
13     "www.google.com": ('100.20.8.1', 'A', 86400),
14     "www.cse.du.ac.bd": ('192.0.2.3', 'A', 86400),
15     "www.yahoo.com": ('1.2.3.9999', "A", 86400)
16 }
17
18
19 def encode_msg(message):
20     try:
21         data = message.split()

```

```

22     name = data[0]
23     type = data[1]
24     flag = 0
25     q = 0
26     a = 1
27     auth_rr = 0
28     add_rr = 0
29     ms = (name + ' ' + type + ' ' + data[2] + ' ' + data
30           [3]).encode('utf-8')
31     packed_data = struct.pack(f"6H{len(ms)}s", 50, flag,
32                               q, a, auth_rr, add_rr, ms)
33     return packed_data
34 except Exception as e:
35     print("Error occurred while encoding message:", e)
36     return None
37
38 def decode_msg(msg):
39     try:
40         header = struct.unpack("6H", msg[:12])
41         ms = msg[12:].decode('utf-8')
42         print('\n After Decoding')
43         print({header}, {ms})
44         ms = ms.split()
45         return ms[1], ms[4]
46     except Exception as e:
47         print("Error occurred while decoding message:", e)
48         return None, None
49
50 def handle_client(data, addr, server):
51     try:
52         print(data)
53         msg = encode_msg(str(data + ' ' + dic[data][0] + ' ' +
54                               + dic[data][1] + ' ' + str(dic[data][2])))
55         if msg:
56             server.sendto(msg, addr)
57     except KeyError:
58         print(f"Requested domain '{data}' not found in
59               authoritative DNS.")
60     except Exception as e:
61         print("ERROR: ", str(e))
62
63 def main():
64     print("[STARTING] auth Server is starting")
65     server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
66     server.bind(ADDR)
67     print(f"[LISTENING] auth Server is listening on {IP}:{
68           PORT}.")

```

```

67
68     while True:
69         try:
70             data, addr = server.recvfrom(SIZE)
71             data = data.decode(FORMAT)
72             handle_client(data, addr, server)
73             print(f"[ACTIVE CONNECTIONS] {threading.
                active_count() - 1}")
74         except KeyboardInterrupt:
75             print("Server terminated by user.")
76             break
77         except Exception as e:
78             print("Error occurred:", e)
79
80     server.close()
81
82
83 if __name__ == "__main__":
84     main()

```

Listing 11: Recursive TLD Server Code

```

1  import os
2  import socket
3  import threading
4  import struct
5  import time
6
7  IP = '127.0.0.1'
8  PORT = 4488
9  authoritative = (IP, 4489)
10 ADDR = (IP, PORT)
11 SIZE = 1024
12 FORMAT = "utf-8"
13 SERVER_DATA_PATH = "server_data"
14 dic = {
15     "www.google.com": ('100.20.8.1', 'A', 86400),
16     "www.cse.du.ac.bd": ('192.0.2.3', 'A', 86400),
17     "www.yahoo.com": ('4489', "NS", 86400)
18 }
19
20
21 def encode_msg(message):
22     try:
23         data = message.split()
24         name = data[0]
25         type = data[1]
26         print(message)
27         flag = 0

```

```

28         q = 0
29         a = 1
30         auth_rr = 0
31         add_rr = 0
32
33         ms = (name + ' ' + type + ' ' + data[2] + ' ' + data
34               [3]).encode('utf-8')
35         packed_data = struct.pack(f"6H{len(ms)}s", 50, flag,
36                                   q, a, auth_rr, add_rr, ms)
37         return packed_data
38     except Exception as e:
39         print("Error occurred while encoding message:", e)
40         return None
41
42 def decode_msg(msg):
43     try:
44         header = struct.unpack("6H", msg[:12])
45         ms = msg[12:].decode('utf-8')
46         print('\n After Decoding')
47         print({header}, {ms})
48         return ms
49     except Exception as e:
50         print("Error occurred while decoding message:", e)
51         return None
52
53 def handle_client(data, addr, server):
54     try:
55         print(f"[RECEIVED MESSAGE] {data} from {addr}.")
56         msg = encode_msg(str(data + ' ' + dic[data][0] + ' ' +
57                               + dic[data][1] + ' ' + str(dic[data][2])))
58         server.sendto(msg, addr)
59     except KeyError:
60         print(f"Requested domain '{data}' not found in TLD
61               DNS.")
62     except Exception as e:
63         server.sendto(data.encode(FORMAT), ('', 4487))
64         print("ERROR: ", str(e))
65
66 def main():
67     print("[STARTING] TLD Server is starting")
68     server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
69     server.bind(ADDR)
70     print(f"[LISTENING] TLD Server is listening on {IP}:{PORT}
71           .")
72
73     while True:
74         try:

```

```

73         data, addr = server.recvfrom(SIZE)
74         data = data.decode(FORMAT)
75         handle_client(data, addr, server)
76         print(f"[ACTIVE CONNECTIONS] {threading.
            active_count() - 1}")
77     except KeyboardInterrupt:
78         print("Server terminated by user.")
79         break
80     except Exception as e:
81         print("Error occurred:", e)
82
83     server.close()
84
85
86 if __name__ == "__main__":
87     main()

```

Listing 12: Recursive Client Code

```

1  import socket
2  import struct
3
4  IP = ''
5  PORT = 4487
6  ADDR = (IP, PORT)
7  SIZE = 1024
8  FORMAT = "utf-8"
9
10
11 def encode_msg(message):
12     data = message.split()
13     name = data[0]
14     type = data[1]
15
16     flag = 0
17     q = 0
18     a = 1
19     auth_rr = 0
20     add_rr = 0
21
22     ms = (name + ' ' + type).encode('utf-8')
23     packed_data = struct.pack(f"6H{len(ms)}s", 50, flag, q, a
        , auth_rr, add_rr, ms)
24     return packed_data
25
26
27 def decode_msg(msg):
28     header = struct.unpack("6H", msg[:12])
29     ms = msg[12:].decode('utf-8')

```

```

30
31     print('\n Before Decoding')
32     print(msg)
33
34     print('\n After Decoding')
35     print({header}, {ms})
36
37     return ms
38
39
40 def main():
41     try:
42         client = socket.socket(socket.AF_INET, socket.
            SOCK_DGRAM)
43
44         message = input("Enter an address: ")
45         client.sendto(message.encode(FORMAT), ADDR)
46         msg, addr = client.recvfrom(SIZE)
47         msg1 = decode_msg(msg)
48         print(msg1)
49         data = msg1.split()
50         while data[2] == "NS":
51             # print(data[1])
52             new_adr = ('', int(data[1]))
53             print('Connecting to port', data[1])
54             client.sendto(message.encode(FORMAT), new_adr)
55             msg, addr = client.recvfrom(SIZE)
56             msg1 = decode_msg(msg)
57             print(msg1)
58             data = msg1.split()
59
60             # print(struct.unpack("6H",msg))
61             # msg,addr=client.recvfrom(SIZE)
62         except KeyboardInterrupt:
63             print("Client terminated by user.")
64         except Exception as e:
65             print("An error occurred:", e)
66         finally:
67             client.close()
68
69
70 if __name__ == "__main__":
71     main()

```

3.4 Part 4: Extending the System

1. Use a short TTL value and try Deleting resource record based on TTL value.
2. Implement DNS caching in local and TLD servers
3. Test failure of a DNS server process.

3.4.1 Use a short TTL value and try Deleting resource record based on TTL value:

Listing 13: TTL value Server Code

```
1 import socket
2 import threading
3 import time
4
5 class DNSRecord:
6     def __init__(self, name, value, record_type, ttl):
7         self.name = name
8         self.value = value
9         self.record_type = record_type
10        self.ttl = ttl
11        self.creation_time = time.time()
12
13    def is_expired(self):
14        return (time.time() - self.creation_time) > self.ttl
15
16 class DNSServer:
17     def __init__(self):
18         self.records = {}
19         self.deleted_records = {} # To store deleted records
20         self.cleanup_interval = 10 # Cleanup interval in
21                                     seconds
22         self.cleanup_thread = threading.Thread(target=self.
23                                                 cleanup_records)
24         self.cleanup_thread.daemon = True
25         self.cleanup_thread.start()
26
27     def add_record(self, name, value, record_type, ttl):
28         self.records[name] = DNSRecord(name, value,
29                                         record_type, ttl)
27
28     def get_record(self, name):
29         record = self.records.get(name)
30         if record and not record.is_expired():
31             return record.value
32         else:
33             return None
34
```



```

35     def cleanup_records(self):
36         while True:
37             expired_records = [name for name, record in self.
38                               records.items() if record.is_expired()]
39             for name in expired_records:
40                 # Move expired record to deleted_records
41                 self.deleted_records[name] = self.records.pop
42                 (name)
43                 print(f"Record {name} deleted due to TTL
44                       expiration")
45                 time.sleep(self.cleanup_interval)
46
47 def handle_client(client_socket, dns_server):
48     while True:
49         request = client_socket.recv(1024).decode('utf-8')
50         if not request:
51             break
52         response = dns_server.get_record(request)
53         if response:
54             client_socket.send(response.encode('utf-8'))
55         else:
56             client_socket.send(b'Record not found')
57     client_socket.close()
58
59 def main():
60     dns_server = DNSServer()
61     dns_server.add_record("www.example.com", "192.168.1.100",
62                           "A", ttl=5)
63
64     server_socket = socket.socket(socket.AF_INET, socket.
65                                  SOCK_STREAM)
66     server_socket.bind(('127.0.0.1', 12345))
67     server_socket.listen(5)
68
69     print("DNS Server running...")
70
71     while True:
72         client_socket, addr = server_socket.accept()
73         print("Connected to", addr)
74         threading.Thread(target=handle_client, args=(
75             client_socket, dns_server)).start()
76
77 if __name__ == "__main__":
78     main()

```

Listing 14: TTL value Client Code

```

1 import socket
2
3 def main():
4     client_socket = socket.socket(socket.AF_INET, socket.
5         SOCK_STREAM)
6     client_socket.connect(('127.0.0.1', 12345))
7
8     query = "www.example.com"
9     client_socket.send(query.encode('utf-8'))
10    response = client_socket.recv(1024).decode('utf-8')
11    print("Response:", response)
12
13    client_socket.close()
14
15 if __name__ == "__main__":
16     main()

```

3.4.2 Implement DNS caching in local and TLD servers:

Listing 15: Cache Code

```

1 import socket
2 import struct
3 import time
4
5 IP = ''
6 PORT = 4487
7 ADDR = (IP, PORT)
8 SIZE = 1024
9 FORMAT = "utf-8"
10 cached = {}
11
12
13 def encode_msg(message):
14     data = message.split()
15     name = data[0]
16     type = data[1]
17
18     flag = 0
19     q = 0
20     a = 1
21     auth_rr = 0
22     add_rr = 0
23
24     ms = (name + ' ' + type).encode('utf-8')
25     packed_data = struct.pack(f"6H{len(ms)}s", 50, flag, q, a
        , auth_rr, add_rr, ms)

```

```

26     return packed_data
27
28
29 def decode_msg(msg):
30     header = struct.unpack("6H", msg[:12])
31     ms = msg[12:].decode('utf-8')
32
33     print('\n Before Decoding')
34     print(msg)
35
36     print('\n After Decoding')
37     print({header}, {ms})
38
39     return ms
40
41
42 def main():
43     client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
44     while True:
45         message = input("Enter an address or enter 'data' to
46             see the cached data: ")
47         if message == "data":
48             print('\n -----Cached Data-----\n')
49             for i in cached:
50                 print(i + ' ' + cached[i][0])
51             print('\n -----End-----\n')
52         else:
53             try:
54                 client.sendto(message.encode(FORMAT), ADDR)
55                 msg, addr = client.recvfrom(SIZE)
56                 msg = decode_msg(msg)
57                 print('Received response:', msg)
58                 qu = msg.split()
59                 if qu[0] == 'error':
60                     print('Error:', qu[1])
61                 else:
62                     cached[qu[0]] = (qu[1], qu[2], qu[3])
63             except Exception as e:
64                 print("An error occurred:", e)
65
66 if __name__ == "__main__":
67     main()

```

Listing 16: Cache Server Code

```

1 import os
2 import socket
3 import threading
4 import struct
5 import time
6
7 IP = ''
8 PORT = 4487
9 ADDR = (IP, PORT)
10 SIZE = 1024
11 FORMAT = "utf-8"
12 SERVER_DATA_PATH = "server_data"
13 dic = {
14     "www.google.com": ('100.20.8.1', 'A', 86400),
15     "www.cse.du.ac.bd": ('100.20.55.2', 'A', 86400),
16     "www.yahoo.com": ('100.20.89.7', "A", 86400)
17 }
18
19
20 def encode_msg(message):
21     data = message.split()
22     name = data[0]
23     type = data[1]
24
25     flag = 0
26     q = 0
27     a = 1
28     auth_rr = 0
29     add_rr = 0
30
31     ms = (name + ' ' + type + ' ' + data[2] + ' ' + data[3]).
32         encode('utf-8')
33     packed_data = struct.pack(f"6H{len(ms)}s", 50, flag, q, a
34         , auth_rr, add_rr, ms)
35     return packed_data
36
37 def decode_msg(msg):
38     header = struct.unpack("6H", msg[:12])
39     ms = msg[12:].decode('utf-8')
40
41     print('\n Before Decoding')
42     print(msg)
43
44     print('\n After Decoding')
45     print({header}, {ms})
46
47     return ms

```

```

48
49 def handle_client(data, addr, server):
50     try:
51         print(f"[{time.ctime()}] [RECEIVED MESSAGE] {data}
52             from {addr}.")
53         if data in dic:
54             if dic[data][1] == 'A' or dic[data][1] == 'AAAA':
55                 msg = encode_msg(str(data + ' ' + dic[data]
56                                     ][0] + ' ' + dic[data][1] + ' ' + str(dic[
57                                     data][2])))
58                 server.sendto(msg, addr)
59             else:
60                 server.sendto(('error Server could not find the
61                     requested domain').encode(FORMAT), addr)
62         except Exception as e:
63             print("ERROR: ", str(e))
64             server.sendto(('error ' + str(e)).encode(FORMAT),
65                             addr)
66
67
68
69 def main():
70     print("[STARTING] ROOT Server is starting")
71     server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
72     server.bind(ADDR)
73     print(f"[LISTENING] ROOT Server is listening on {IP}:{
74         PORT}.")
75
76     while True:
77         data, addr = server.recvfrom(SIZE)
78         data = data.decode(FORMAT)
79         handle_client(data, addr, server)
80         print(f"[ACTIVE CONNECTIONS] {threading.active_count
81             () - 1}")
82
83
84 if __name__ == "__main__":
85     main()

```

3.4.3 Test failure of a DNS server process:

Listing 17: Test failure Cache Code

```
1 import socket
2 import struct
3 import time
4
5 IP = ''
6 PORT = 4487
7 ADDR = (IP, PORT)
8 SIZE = 1024
9 FORMAT = "utf-8"
10 cached = {}
11
12 # Timeout value in seconds
13 TIMEOUT = 5
14
15 def encode_msg(message):
16     data = message.split()
17     name = data[0]
18     type = data[1]
19
20     flag = 0
21     q = 0
22     a = 1
23     auth_rr = 0
24     add_rr = 0
25
26     ms = (name + ' ' + type).encode('utf-8')
27     packed_data = struct.pack(f"6H{len(ms)}s", 50, flag, q, a,
28                               , auth_rr, add_rr, ms)
29     return packed_data
30
31 def decode_msg(msg):
32     header = struct.unpack("6H", msg[:12])
33     ms = msg[12:].decode('utf-8')
34
35     print('\n Before Decoding')
36     print(msg)
37
38     print('\n After Decoding')
39     print({header}, {ms})
40
41     return ms
42
43
44 def main():
45     client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
46     client.settimeout(TIMEOUT) # Set socket timeout
```

```

47
48     while True:
49         message = input("Enter 'simulate_recovery' to
        simulate the recovery of the server process or
        enter 'simulate_failure' to server to simulate the
        failure: ")
50         if message == "data":
51             print('\n _____Cached Data_____ \n')
52             for i in cached:
53                 print(i + ' ' + cached[i][0])
54             print('\n _____End_____ \n')
55         else:
56             try:
57                 client.sendto(message.encode(FORMAT), ADDR)
58                 msg, addr = client.recvfrom(SIZE)
59                 msg = decode_msg(msg)
60                 print('Received response:', msg)
61                 qu = msg.split()
62                 if qu[0] == 'error':
63                     print('Error:', qu[1])
64                 else:
65                     cached[qu[0]] = (qu[1], qu[2], qu[3])
66             except socket.timeout:
67                 print("Server is not responding. DNS server
        process might have failed.")
68             except Exception as e:
69                 print("An error occurred:", e)
70
71
72 if __name__ == "__main__":
73     main()

```

Listing 18: Test failure Server Code

```

1  import os
2  import socket
3  import threading
4  import struct
5
6  IP = ''
7  PORT = 4487
8  ADDR = (IP, PORT)
9  SIZE = 1024
10 FORMAT = "utf-8"
11 SERVER_DATA_PATH = "server_data"
12 dic = {
13     "www.google.com": ('100.20.8.1', 'A', 86400),
14     "www.cse.du.ac.bd": ('100.20.55.2', 'A', 86400),
15     "www.yahoo.com": ('100.20.89.7', "A", 86400)

```

```

16 }
17
18 # Variable to indicate whether the server is active or not
19 server_active = True
20
21 def encode_msg(message):
22     data = message.split()
23     name = data[0]
24     type = data[1]
25
26     flag = 0
27     q = 0
28     a = 1
29     auth_rr = 0
30     add_rr = 0
31
32     ms = (name + ' ' + type + ' ' + data[2] + ' ' + data[3]).
        encode('utf-8')
33     packed_data = struct.pack(f"6H{len(ms)}s", 50, flag, q, a
        , auth_rr, add_rr, ms)
34     return packed_data
35
36 def decode_msg(msg):
37     header = struct.unpack("6H", msg[:12])
38     ms = msg[12:].decode('utf-8')
39     return ms
40
41 def handle_client(data, addr, server):
42     global server_active
43     try:
44         print(f"[RECEIVED MESSAGE] {data} from {addr}.")
45         if data == "simulate_failure":
46             print("Simulating server failure...")
47             server_active = False
48         elif data == "simulate_recovery":
49             print("Simulating server recovery...")
50             server_active = True
51         else:
52             if server_active: # Check if the server is
                active
53                 if data in dic:
54                     if dic[data][1] == 'A' or dic[data][1] ==
                        'AAAA':
55                         msg = encode_msg(str(data + ' ' + dic
                            [data][0] + ' ' + dic[data][1] + '
                                ' + str(dic[data][2])))
56                         server.sendto(msg, addr)
57                     else:
58                         server.sendto(('error Server could not
                            find the requested domain')).encode(

```



```

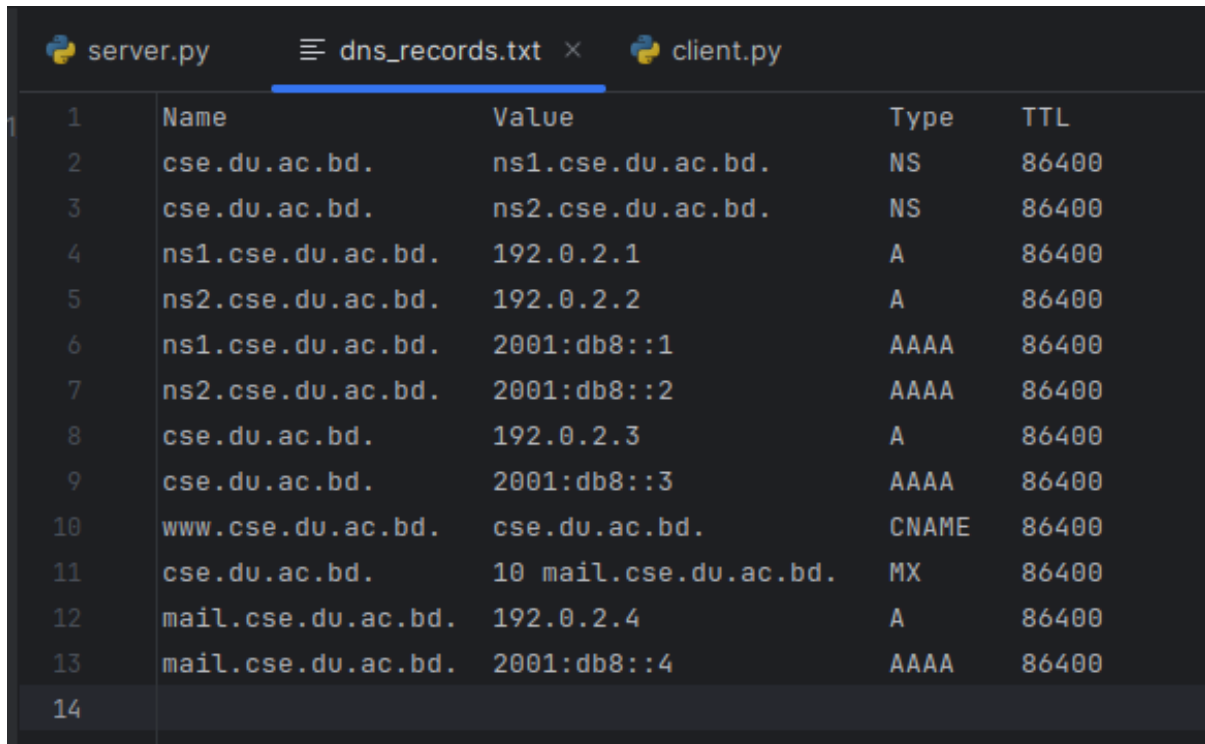
59             FORMAT), addr)
60         else:
61             print("Server is currently inactive. Cannot
62                 process requests.")
63     except Exception as e:
64         print("ERROR: ", str(e))
65         server.sendto(('error ' + str(e)).encode(FORMAT),
66                     addr)
67
68 def main():
69     global server_active
70     print("[STARTING] ROOT Server is starting")
71     server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
72     server.bind(ADDR)
73     print(f"[LISTENING] ROOT Server is listening on {IP}:{
74         PORT}.")
75
76     while True:
77         data, addr = server.recvfrom(SIZE)
78         data = data.decode(FORMAT)
79         handle_client(data, addr, server)
80         print(f"[ACTIVE CONNECTIONS] {threading.active_count
81             () - 1}")
82
83 if __name__ == "__main__":
84     main()

```

4 Experimental result

4.1 Task 1- Setting up the DNS server

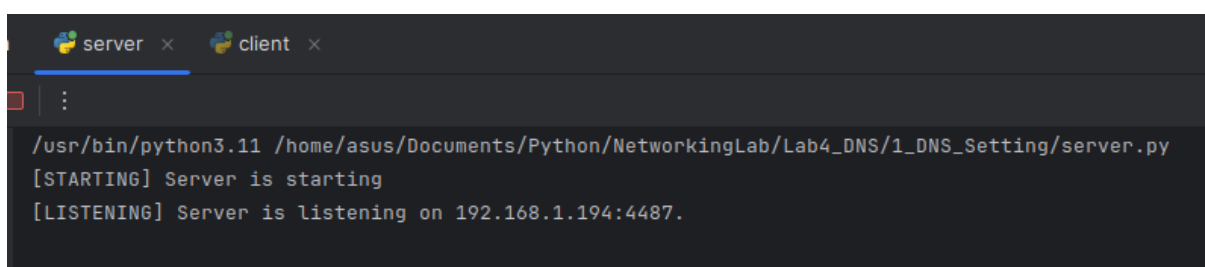
4.1.1 'dns records.txt' File:



| | Name | Value | Type | TTL |
|----|--------------------|-----------------------|-------|-------|
| 1 | cse.du.ac.bd. | ns1.cse.du.ac.bd. | NS | 86400 |
| 2 | cse.du.ac.bd. | ns2.cse.du.ac.bd. | NS | 86400 |
| 3 | ns1.cse.du.ac.bd. | 192.0.2.1 | A | 86400 |
| 4 | ns2.cse.du.ac.bd. | 192.0.2.2 | A | 86400 |
| 5 | ns1.cse.du.ac.bd. | 2001:db8::1 | AAAA | 86400 |
| 6 | ns2.cse.du.ac.bd. | 2001:db8::2 | AAAA | 86400 |
| 7 | cse.du.ac.bd. | 192.0.2.3 | A | 86400 |
| 8 | cse.du.ac.bd. | 2001:db8::3 | AAAA | 86400 |
| 9 | www.cse.du.ac.bd. | cse.du.ac.bd. | CNAME | 86400 |
| 10 | cse.du.ac.bd. | 10 mail.cse.du.ac.bd. | MX | 86400 |
| 11 | mail.cse.du.ac.bd. | 192.0.2.4 | A | 86400 |
| 12 | mail.cse.du.ac.bd. | 2001:db8::4 | AAAA | 86400 |
| 13 | | | | |
| 14 | | | | |

Figure 5: Content of dns records.txt

4.1.2 After Running Server Code:



```
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/1_DNS_Setting/server.py
[STARTING] Server is starting
[LISTENING] Server is listening on 192.168.1.194:4487.
```

Figure 6: Content of Server

4.1.3 After Running Client Code:

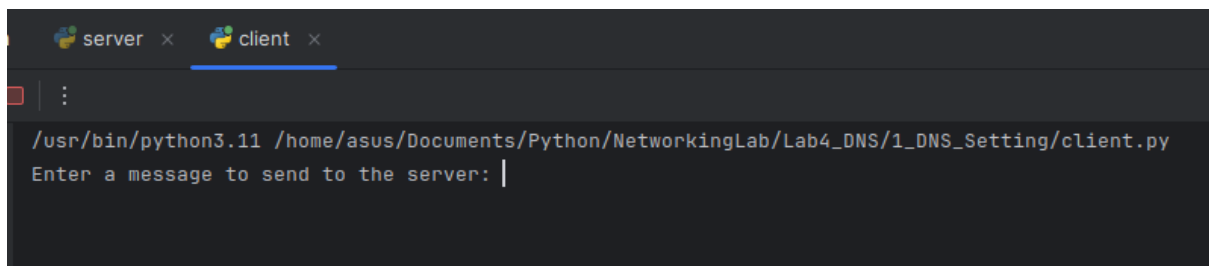
A terminal window with two tabs: 'server' and 'client'. The 'client' tab is active. The terminal shows the command `/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/1_DNS_Setting/client.py` and a prompt 'Enter a message to send to the server: |'.

Figure 7: Server Connect to Client

4.1.4 After entering target address, in client:

File transfer successfully occurs via socket programming; the file size is also mentioned to the clients.

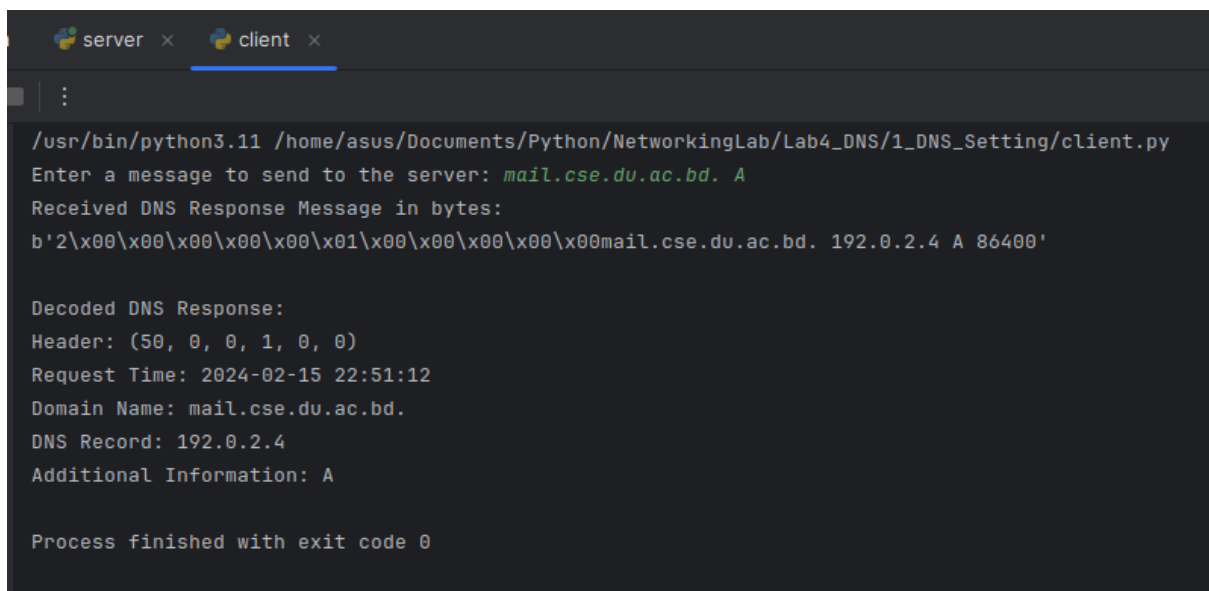
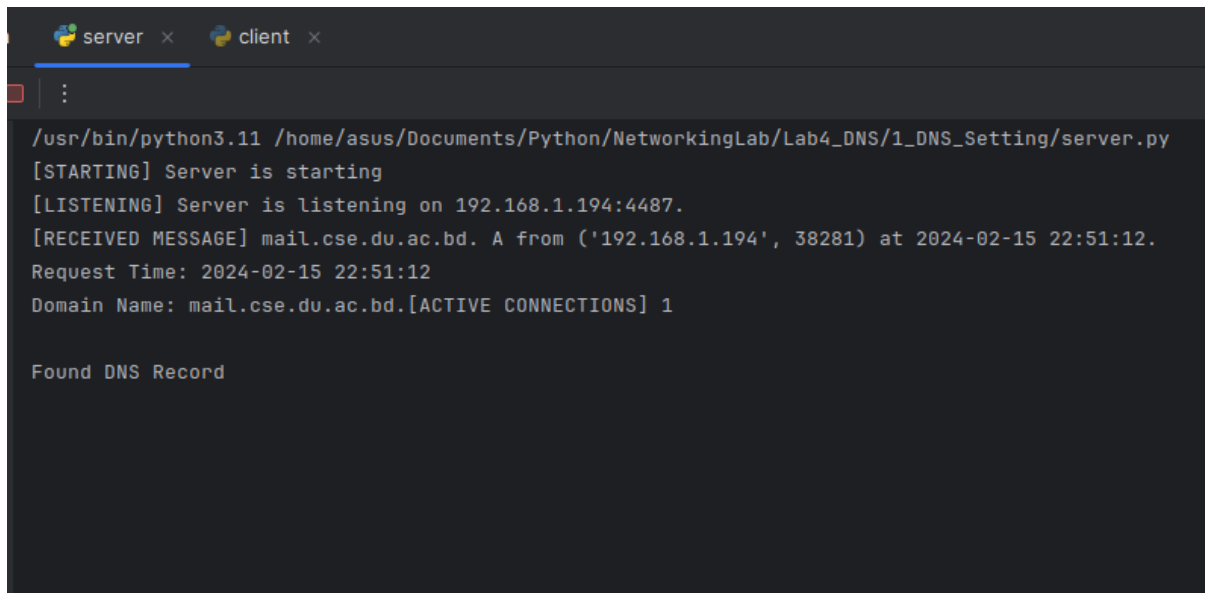
A terminal window with two tabs: 'server' and 'client'. The 'client' tab is active. The terminal shows the command `/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/1_DNS_Setting/client.py` and a prompt 'Enter a message to send to the server: mail.cse.du.ac.bd. A'. Below this, it shows the received DNS response in bytes: `b'2\x00\x00\x00\x00\x01\x00\x00\x00\x00mail.cse.du.ac.bd. 192.0.2.4 A 86400'`. Then, it shows the decoded DNS response: `Header: (50, 0, 0, 1, 0, 0)`, `Request Time: 2024-02-15 22:51:12`, `Domain Name: mail.cse.du.ac.bd.`, `DNS Record: 192.0.2.4`, and `Additional Information: A`. Finally, it shows `Process finished with exit code 0`.

Figure 8: Get information of target address

4.1.5 After entering target address, in server: :

Here, we can see the file we just downloaded.

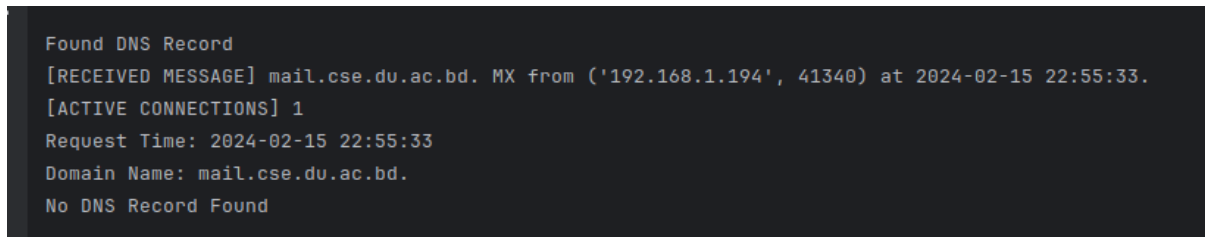
A screenshot of a terminal window with two tabs: 'server' and 'client'. The 'server' tab is active. The terminal shows the execution of a Python script. The output includes: the script path, a starting message, a listening message on 192.168.1.194:4487, a received message for mail.cse.du.ac.bd. A from ('192.168.1.194', 38281) at 2024-02-15 22:51:12, the request time, the domain name, the active connections count (1), and a message 'Found DNS Record'.

```
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/1_DNS_Setting/server.py
[STARTING] Server is starting
[LISTENING] Server is listening on 192.168.1.194:4487.
[RECEIVED MESSAGE] mail.cse.du.ac.bd. A from ('192.168.1.194', 38281) at 2024-02-15 22:51:12.
Request Time: 2024-02-15 22:51:12
Domain Name: mail.cse.du.ac.bd.[ACTIVE CONNECTIONS] 1

Found DNS Record
```

Figure 9: Get information of the client

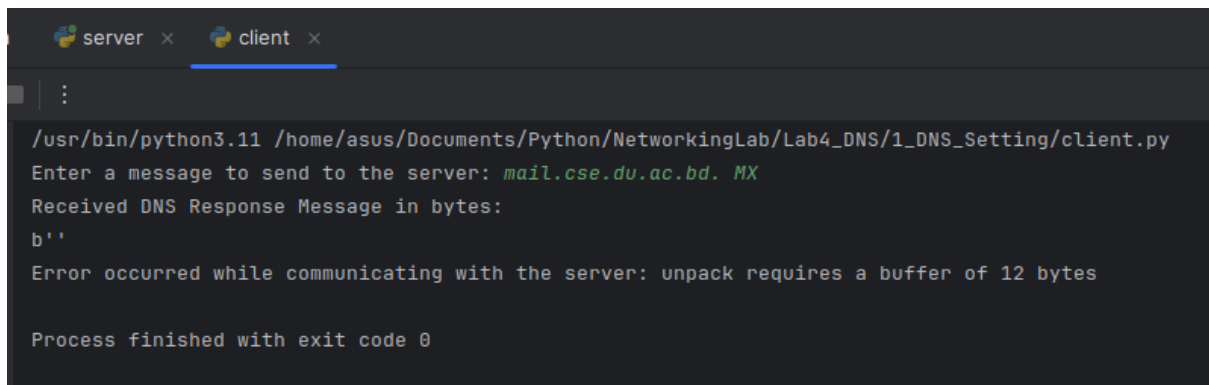
4.1.6 Error for server

A screenshot of a terminal window showing the continuation of the server logs. It shows: 'Found DNS Record', a received message for mail.cse.du.ac.bd. MX from ('192.168.1.194', 41340) at 2024-02-15 22:55:33, the active connections count (1), the request time, the domain name, and a message 'No DNS Record Found'.

```
Found DNS Record
[RECEIVED MESSAGE] mail.cse.du.ac.bd. MX from ('192.168.1.194', 41340) at 2024-02-15 22:55:33.
[ACTIVE CONNECTIONS] 1
Request Time: 2024-02-15 22:55:33
Domain Name: mail.cse.du.ac.bd.
No DNS Record Found
```

Figure 10: Error in server

4.1.7 Error for client

A terminal window with two tabs: 'server' and 'client'. The 'client' tab is active. The terminal shows the execution of a Python script. The user enters a message 'mail.cse.du.ac.bd. MX'. The script receives a DNS response message in bytes, which is shown as 'b''. An error message is displayed: 'Error occurred while communicating with the server: unpack requires a buffer of 12 bytes'. The process finishes with exit code 0.

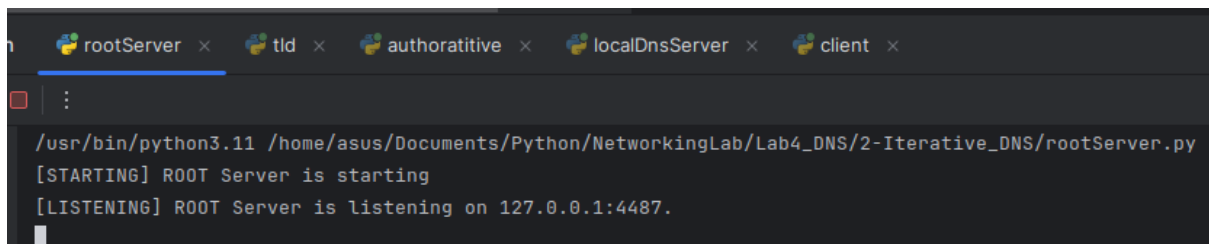
```
server x client x
| :
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/1_DNS_Setting/client.py
Enter a message to send to the server: mail.cse.du.ac.bd. MX
Received DNS Response Message in bytes:
b''
Error occurred while communicating with the server: unpack requires a buffer of 12 bytes

Process finished with exit code 0
```

Figure 11: Error in client

4.2 Task 2- Iterative DNS resolution

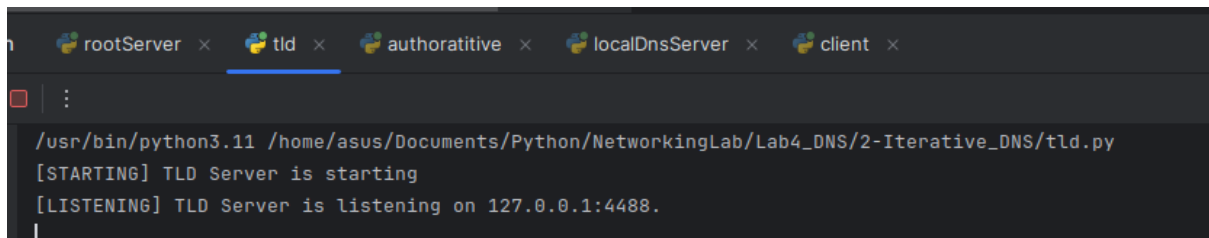
4.2.1 After Running rootSever Code:



A terminal window with five tabs: rootServer, tld, authoratitive, localDnsServer, and client. The rootServer tab is active. The terminal shows the command `/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/2-Iterative_DNS/rootServer.py` and its output: `[STARTING] ROOT Server is starting` and `[LISTENING] ROOT Server is listening on 127.0.0.1:4487.`

Figure 12: Root Server Connecting

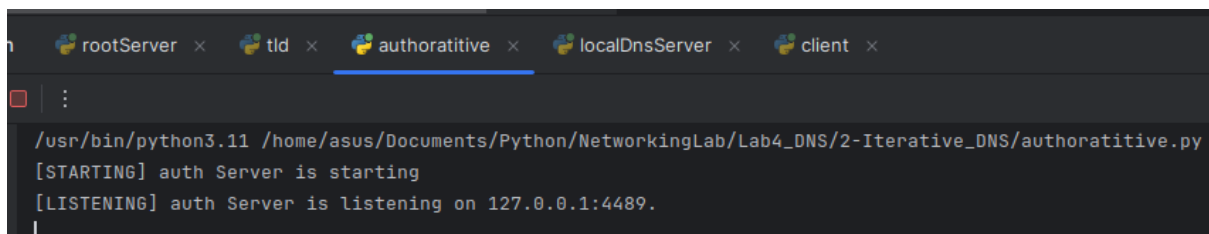
4.2.2 After Running tld Code:



A terminal window with five tabs: rootServer, tld, authoratitive, localDnsServer, and client. The tld tab is active. The terminal shows the command `/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/2-Iterative_DNS/tld.py` and its output: `[STARTING] TLD Server is starting` and `[LISTENING] TLD Server is listening on 127.0.0.1:4488.`

Figure 13: TLD Server Connecting

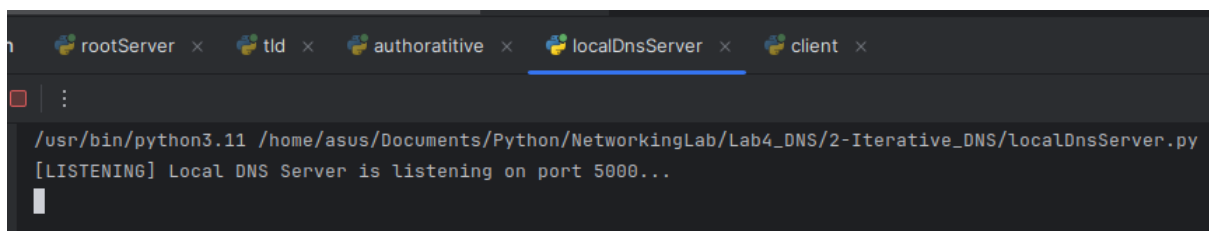
4.2.3 After Running authoritative Code:



A terminal window with five tabs: rootServer, tld, authoratitive, localDnsServer, and client. The authoratitive tab is active. The terminal shows the command `/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/2-Iterative_DNS/authoratitive.py` and its output: `[STARTING] auth Server is starting` and `[LISTENING] auth Server is listening on 127.0.0.1:4489.`

Figure 14: Authoritative Server Connecting

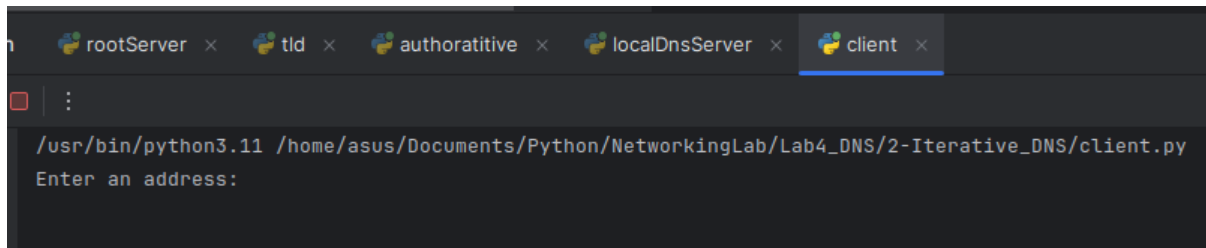
4.2.4 After Running localDnsSever Code:



A terminal window with five tabs: rootServer, tld, authoratitive, localDnsServer, and client. The localDnsServer tab is active. The terminal shows the command `/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/2-Iterative_DNS/localDnsServer.py` and its output: `[LISTENING] Local DNS Server is listening on port 5000...`

Figure 15: Local DNS Server Connecting

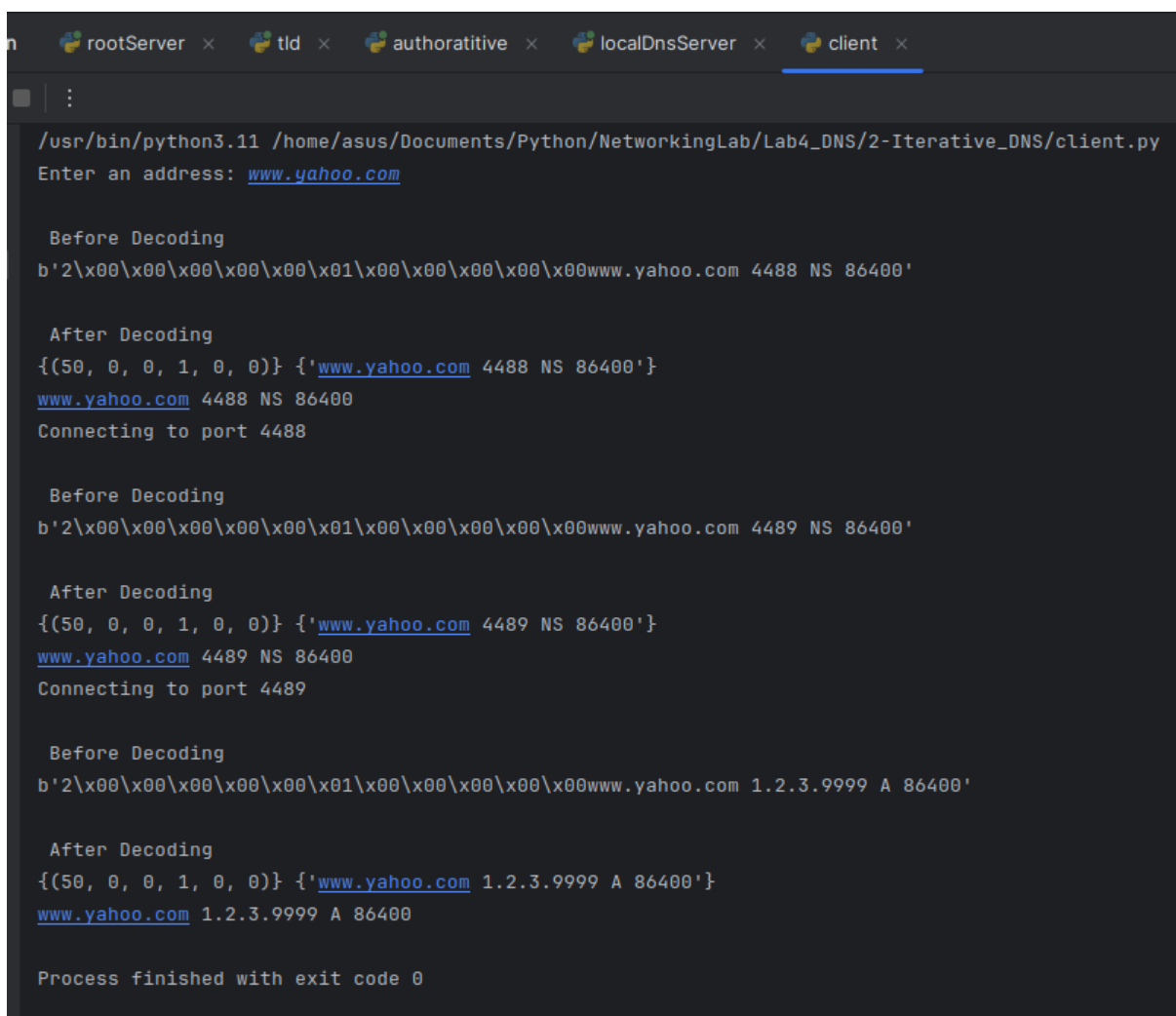
4.2.5 After Running client Code:



A terminal window with a dark background and light-colored text. The window has several tabs at the top: 'rootServer', 'tld', 'authoritative', 'localDnsServer', and 'client'. The 'client' tab is selected. The terminal shows the command `/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/2-Iterative_DNS/client.py` being executed. Below the command, it says 'Enter an address:'.

Figure 16: Server Connect to Client

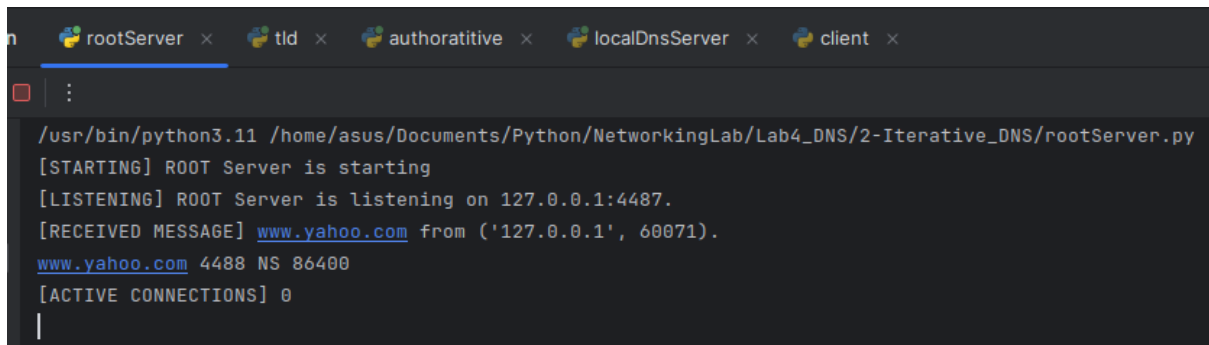
4.2.6 After client entering address:



A terminal window with a dark background and light-colored text. The window has several tabs at the top: 'rootServer', 'tld', 'authoritative', 'localDnsServer', and 'client'. The 'client' tab is selected. The terminal shows the command `/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/2-Iterative_DNS/client.py` being executed. Below the command, it says 'Enter an address: www.yahoo.com'. The output shows the process of decoding the address and connecting to the server. It displays the raw bytes before decoding and the decoded data after decoding. The decoded data shows the IP address 1.2.3.9999 and the record type A. The process finishes with exit code 0.

Figure 17: Client response

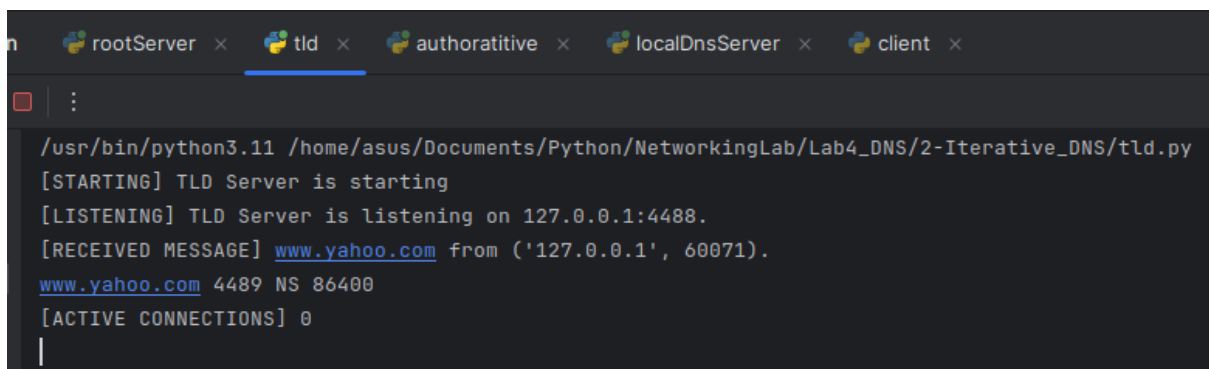
4.2.7 Response from Root server:

A terminal window with a dark background and light-colored text. The window has five tabs at the top: 'rootServer', 'tld', 'authoritative', 'localDnsServer', and 'client'. The 'rootServer' tab is selected. The terminal output shows the following:

```
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/2-Iterative_DNS/rootServer.py
[STARTING] ROOT Server is starting
[LISTENING] ROOT Server is listening on 127.0.0.1:4487.
[RECEIVED MESSAGE] www.yahoo.com from ('127.0.0.1', 60071).
www.yahoo.com 4488 NS 86400
[ACTIVE CONNECTIONS] 0
```

Figure 18: rootServer response

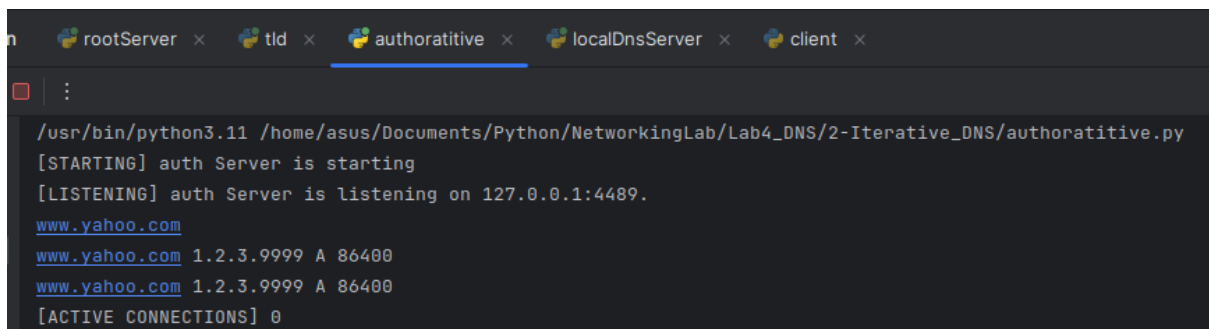
4.2.8 Response from tld server:

A terminal window with a dark background and light-colored text. The window has five tabs at the top: 'rootServer', 'tld', 'authoritative', 'localDnsServer', and 'client'. The 'tld' tab is selected. The terminal output shows the following:

```
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/2-Iterative_DNS/tld.py
[STARTING] TLD Server is starting
[LISTENING] TLD Server is listening on 127.0.0.1:4488.
[RECEIVED MESSAGE] www.yahoo.com from ('127.0.0.1', 60071).
www.yahoo.com 4489 NS 86400
[ACTIVE CONNECTIONS] 0
```

Figure 19: tld response

4.2.9 Response from authoritative server:

A terminal window with a dark background and light-colored text. The window has five tabs at the top: 'rootServer', 'tld', 'authoritative', 'localDnsServer', and 'client'. The 'authoritative' tab is selected. The terminal output shows the following:

```
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/2-Iterative_DNS/authoritative.py
[STARTING] auth Server is starting
[LISTENING] auth Server is listening on 127.0.0.1:4489.
www.yahoo.com
www.yahoo.com 1.2.3.9999 A 86400
www.yahoo.com 1.2.3.9999 A 86400
[ACTIVE CONNECTIONS] 0
```

Figure 20: authoritative response

4.2.10 Error

Entering addresses that are not in the Server-

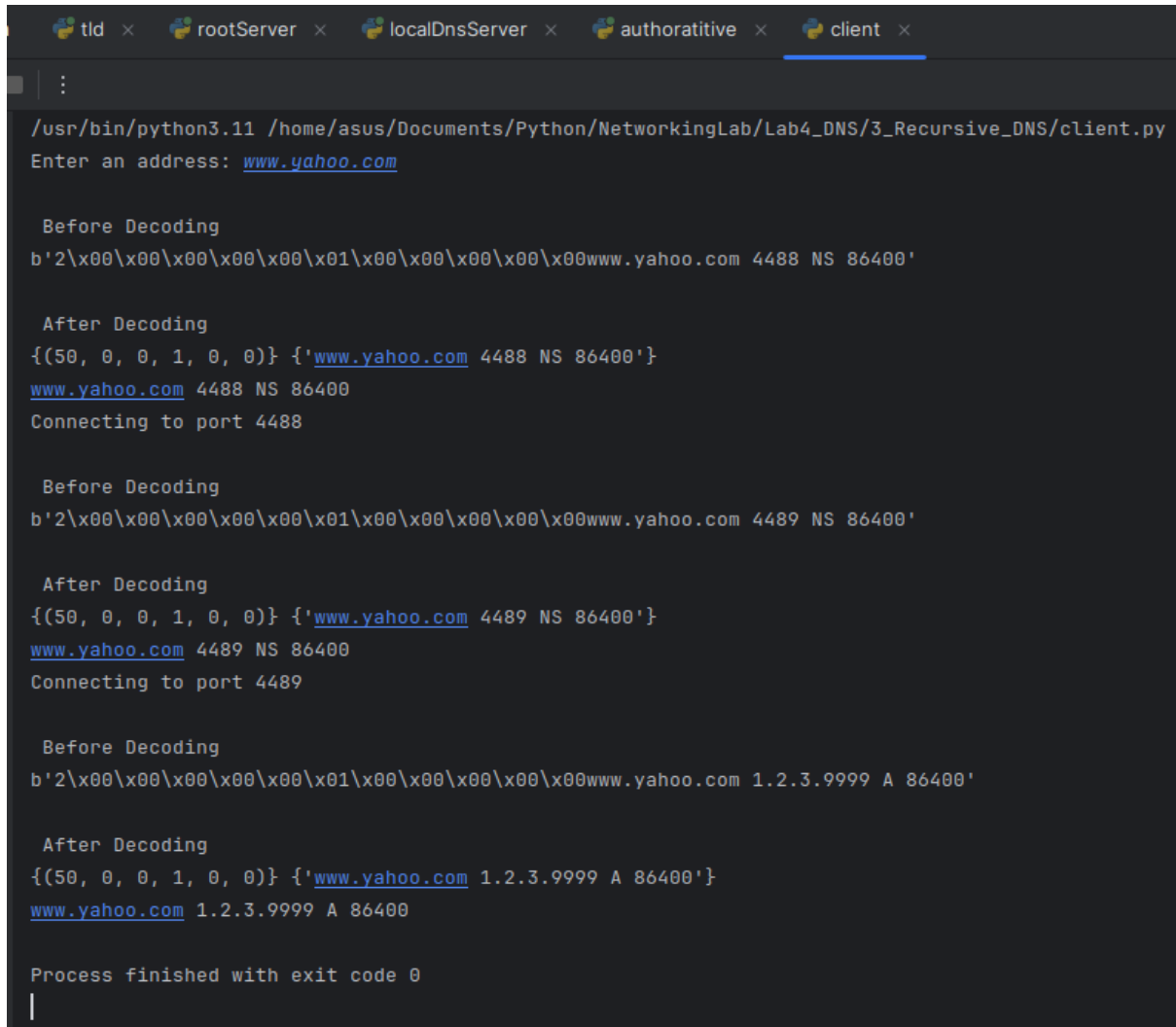
```
[RECEIVED MESSAGE] www.amazon.com from ('127.0.0.1', 59011).  
ERROR:  'www.amazon.com'  
[ACTIVE CONNECTIONS] 0  
|
```

Figure 21: rootserver response

4.3 Task 3- Recursive DNS resolution

Connecting to servers are same as Iterative servers.

4.3.1 After client entering address:



```
tld x rootServer x localDnsServer x authoritative x client x
:
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/3_Recursive_DNS/client.py
Enter an address: www.yahoo.com

Before Decoding
b'2\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00www.yahoo.com 4488 NS 86400'

After Decoding
{(50, 0, 0, 1, 0, 0)} {'www.yahoo.com 4488 NS 86400'}
www.yahoo.com 4488 NS 86400
Connecting to port 4488

Before Decoding
b'2\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00www.yahoo.com 4489 NS 86400'

After Decoding
{(50, 0, 0, 1, 0, 0)} {'www.yahoo.com 4489 NS 86400'}
www.yahoo.com 4489 NS 86400
Connecting to port 4489

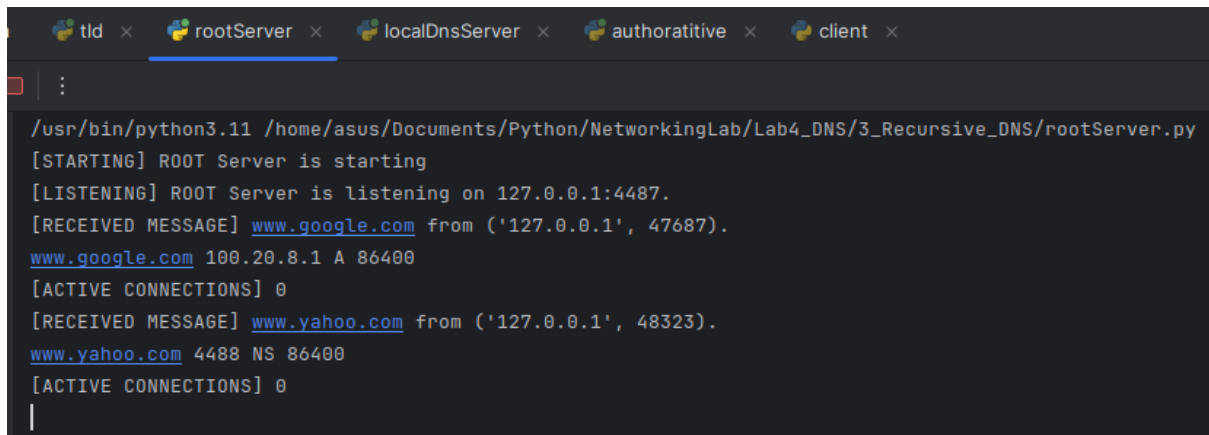
Before Decoding
b'2\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00www.yahoo.com 1.2.3.9999 A 86400'

After Decoding
{(50, 0, 0, 1, 0, 0)} {'www.yahoo.com 1.2.3.9999 A 86400'}
www.yahoo.com 1.2.3.9999 A 86400

Process finished with exit code 0
|
```

Figure 22: Client response

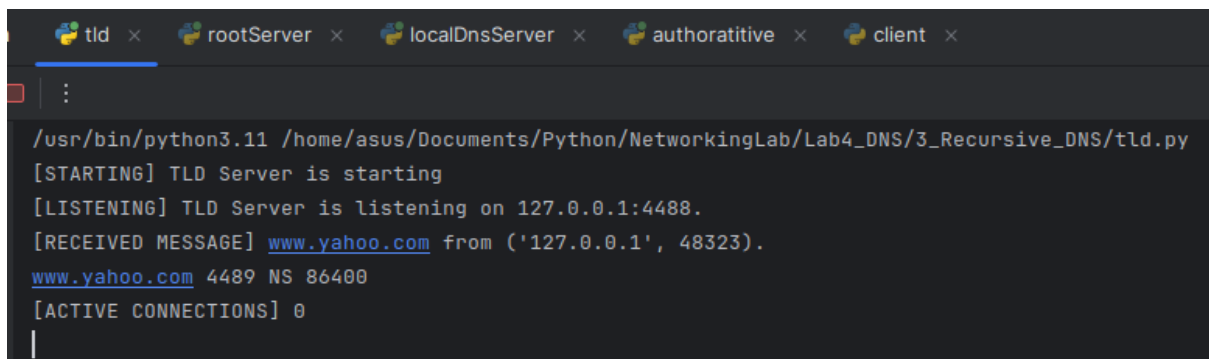
4.3.2 Response from Root server:

A terminal window with five tabs: tld, rootServer, localDnsServer, authoratitive, and client. The rootServer tab is active. The terminal shows the execution of rootServer.py, including startup messages, listening on 127.0.0.1:4487, and receiving two DNS queries for www.google.com and www.yahoo.com. The responses are 100.20.8.1 A 86400 and 4488 NS 86400 respectively. Active connections are shown as 0.

```
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/3_Recursive_DNS/rootServer.py
[STARTING] ROOT Server is starting
[LISTENING] ROOT Server is listening on 127.0.0.1:4487.
[RECEIVED MESSAGE] www.google.com from ('127.0.0.1', 47687).
www.google.com 100.20.8.1 A 86400
[ACTIVE CONNECTIONS] 0
[RECEIVED MESSAGE] www.yahoo.com from ('127.0.0.1', 48323).
www.yahoo.com 4488 NS 86400
[ACTIVE CONNECTIONS] 0
|
```

Figure 23: rootServer response

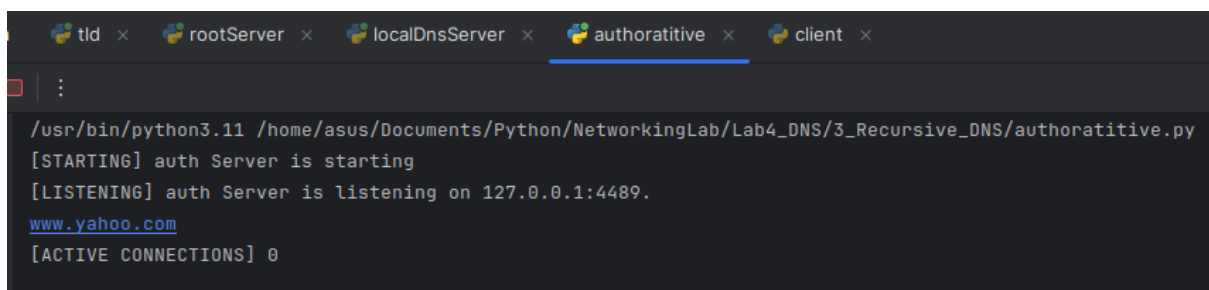
4.3.3 Responce from tld server:

A terminal window with five tabs: tld, rootServer, localDnsServer, authoratitive, and client. The tld tab is active. The terminal shows the execution of tld.py, including startup messages, listening on 127.0.0.1:4488, and receiving a DNS query for www.yahoo.com. The response is 4489 NS 86400. Active connections are shown as 0.

```
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/3_Recursive_DNS/tld.py
[STARTING] TLD Server is starting
[LISTENING] TLD Server is listening on 127.0.0.1:4488.
[RECEIVED MESSAGE] www.yahoo.com from ('127.0.0.1', 48323).
www.yahoo.com 4489 NS 86400
[ACTIVE CONNECTIONS] 0
|
```

Figure 24: tld response

4.3.4 Responce from authoratitive server:

A terminal window with five tabs: tld, rootServer, localDnsServer, authoratitive, and client. The authoratitive tab is active. The terminal shows the execution of authoratitive.py, including startup messages, listening on 127.0.0.1:4489, and receiving a DNS query for www.yahoo.com. The response is 4489 NS 86400. Active connections are shown as 0.

```
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/3_Recursive_DNS/authoratitive.py
[STARTING] auth Server is starting
[LISTENING] auth Server is listening on 127.0.0.1:4489.
www.yahoo.com
[ACTIVE CONNECTIONS] 0
```

Figure 25: authoratitive response

4.3.5 Error

Entering addresses that are not in the Server-

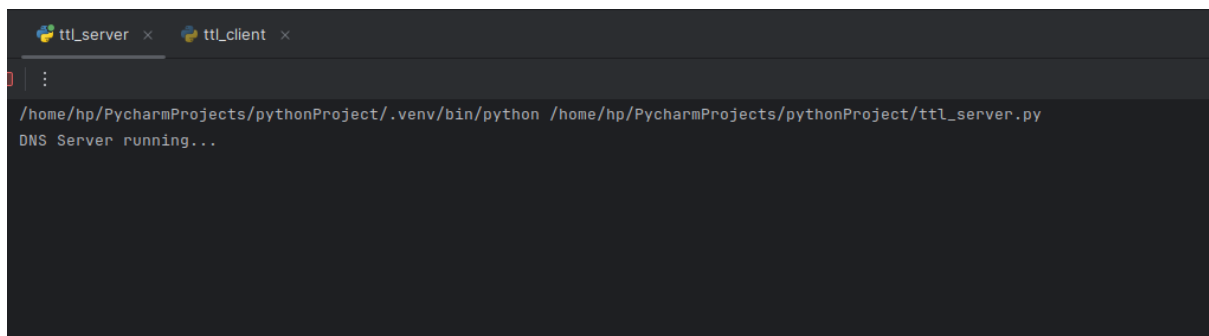
```
[RECEIVED MESSAGE] www.amazon.com from ('127.0.0.1', 54027).  
Requested domain 'www.amazon.com' not found in root DNS.  
[ACTIVE CONNECTIONS] 0  
|
```

Figure 26: rootserver response

4.4 Task 4- Extending the System

Use a short TTL value and try Deleting resource record based on TTL value

4.4.1 After running ttl server code:

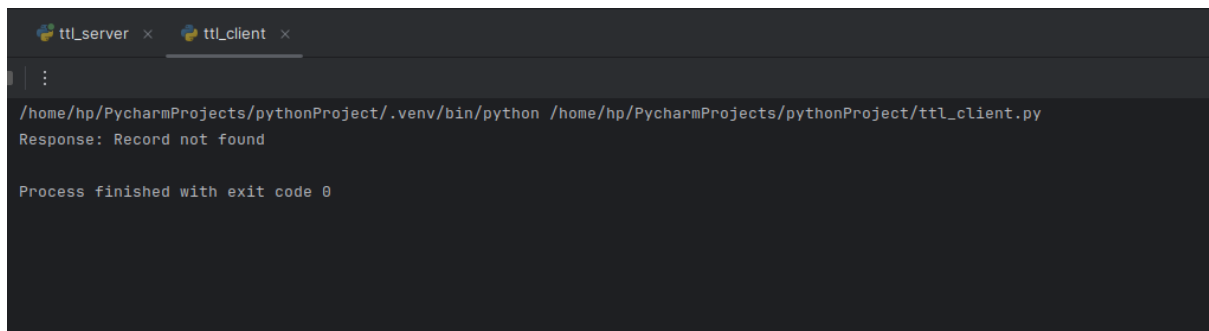


```
ttl_server x ttl_client x  
:  
/home/hp/PycharmProjects/pythonProject/.venv/bin/python /home/hp/PycharmProjects/pythonProject/ttl_server.py  
DNS Server running...
```

Figure 27: ttl value server

4.4.2 After running ttl client code:

Resources delete successfully.



```
ttl_server x ttl_client x  
:  
/home/hp/PycharmProjects/pythonProject/.venv/bin/python /home/hp/PycharmProjects/pythonProject/ttl_client.py  
Response: Record not found  
  
Process finished with exit code 0
```

Figure 28: ttl value client

Implement DNS caching in local and TLD servers

4.4.3 After running cached client code:

```
server x cache x
:
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/4_Extending_System/4.2-DNS cache/cache.py
Enter an address or enter 'data' to see the cached data: www.cse.du.ac.bd

Before Decoding
b'2\x00\x00\x00\x00\x01\x00\x00\x00\x00www.cse.du.ac.bd 100.20.55.2 A 86400'

After Decoding
{(50, 0, 0, 1, 0, 0)} {'www.cse.du.ac.bd 100.20.55.2 A 86400'}
Received response: www.cse.du.ac.bd 100.20.55.2 A 86400
Enter an address or enter 'data' to see the cached data: www.google.com

Before Decoding
b'2\x00\x00\x00\x00\x01\x00\x00\x00\x00www.google.com 100.20.8.1 A 86400'

After Decoding
{(50, 0, 0, 1, 0, 0)} {'www.google.com 100.20.8.1 A 86400'}
Received response: www.google.com 100.20.8.1 A 86400
Enter an address or enter 'data' to see the cached data: data

-----Cached Data-----

www.cse.du.ac.bd 100.20.55.2
www.google.com 100.20.8.1

-----End-----
```

Figure 29: cached client

4.4.4 Error cached client code:

```
Enter an address or enter 'data' to see the cached data: www.amazon.com

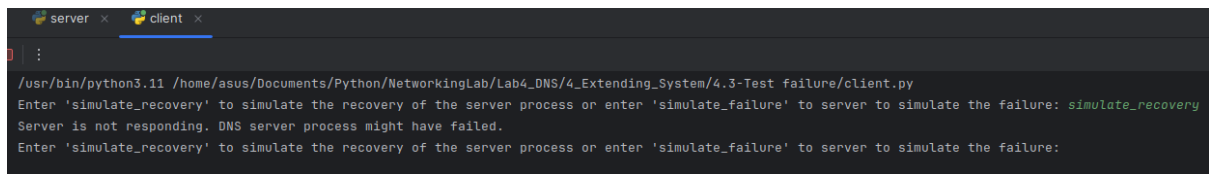
Before Decoding
b'error Server could not find the requested domain'

After Decoding
{(29285, 28530, 8306, 25939, 30322, 29285)} {' could not find the requested domain'}
Received response: could not find the requested domain
```

Figure 30: show error in cached client

Test failure of a DNS server process.

4.4.5 After running client code:

A terminal window with two tabs: 'server' and 'client'. The 'client' tab is active. The terminal shows the execution of a Python script. The prompt is a colon ':'. The first line of output is the file path: /usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/4_Extending_System/4.3-Test failure/client.py. The next line is a prompt: Enter 'simulate_recovery' to simulate the recovery of the server process or enter 'simulate_failure' to server to simulate the failure: simulate_recovery. The following line is a message: Server is not responding. DNS server process might have failed. The final line is another prompt: Enter 'simulate_recovery' to simulate the recovery of the server process or enter 'simulate_failure' to server to simulate the failure:

```
server x client x
:
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/4_Extending_System/4.3-Test failure/client.py
Enter 'simulate_recovery' to simulate the recovery of the server process or enter 'simulate_failure' to server to simulate the failure: simulate_recovery
Server is not responding. DNS server process might have failed.
Enter 'simulate_recovery' to simulate the recovery of the server process or enter 'simulate_failure' to server to simulate the failure:
```

Figure 31: error in client

4.4.6 After responding server code:

A terminal window with two tabs: 'server' and 'client'. The 'server' tab is active. The terminal shows the execution of a Python script. The prompt is a colon ':'. The first line of output is the file path: /usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/4_Extending_System/4.3-Test failure/server.py. The next line is a message: [STARTING] ROOT Server is starting. The following line is a message: [LISTENING] ROOT Server is listening on :4487. The next line is a message: [RECEIVED MESSAGE] simulate_recovery from ('127.0.0.1', 52485). The following line is a message: Simulating server recovery... The final line is a message: [ACTIVE CONNECTIONS] 0

```
server x client x
:
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab4_DNS/4_Extending_System/4.3-Test failure/server.py
[STARTING] ROOT Server is starting
[LISTENING] ROOT Server is listening on :4487.
[RECEIVED MESSAGE] simulate_recovery from ('127.0.0.1', 52485).
Simulating server recovery...
[ACTIVE CONNECTIONS] 0
```

Figure 32: response server

5 Experience

1. **Experiment Setup:** Configured DNS server as authoritative for "cse.du.ac.bd".
2. **Record Addition:** Added A, AAAA, CNAME, and MX records for friends' PCs to a zone file.
3. **Server Initialization:** Started DNS server to handle queries.
4. **Verification:** Ensured server operation and query resolution via UDP sockets.
5. **Communication Format:** Established message exchange format for DNS server-client interaction.
6. **Resolution Processes:** Implemented iterative and recursive resolution mechanisms within the DNS server.
7. **Challenges Faced:** Addressed issues with record configuration and communication troubleshooting.
8. **Key Learnings:** Deepened understanding of DNS fundamentals and distributed database management principles.
9. **Practical Skills:** Gained experience in managing complex distributed systems.

References

- [1] <https://aws.amazon.com/route53/what-is-dns>
- [2] <https://constellix.com/news/dns-record-types>
- [3] DigitalOcean. (n.d.). Java HttpURLConnection Example: Java HTTP Request GET/POST. Retrieved from <https://www.digitalocean.com/community/tutorials/java-httpurlconnection-example-java-http-request-get-post>Record Types -
- [4] JavaTpoint. (n.d.). Java GET and POST. Retrieved from <https://www.javatpoint.com/java-get-post>