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

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.

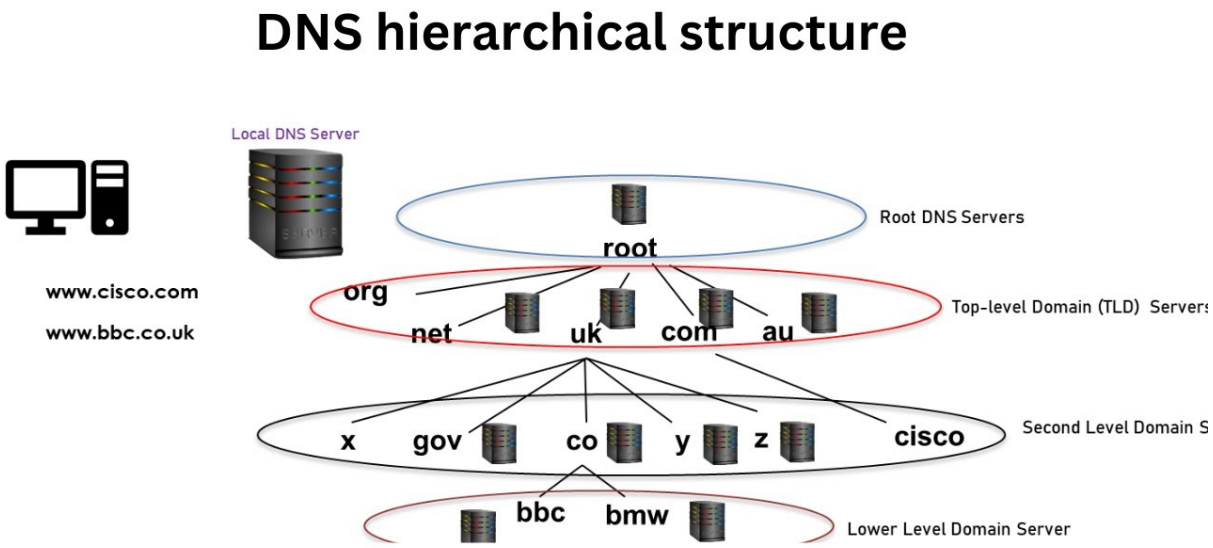

Figure 1: Hiercharchy of DNS

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.

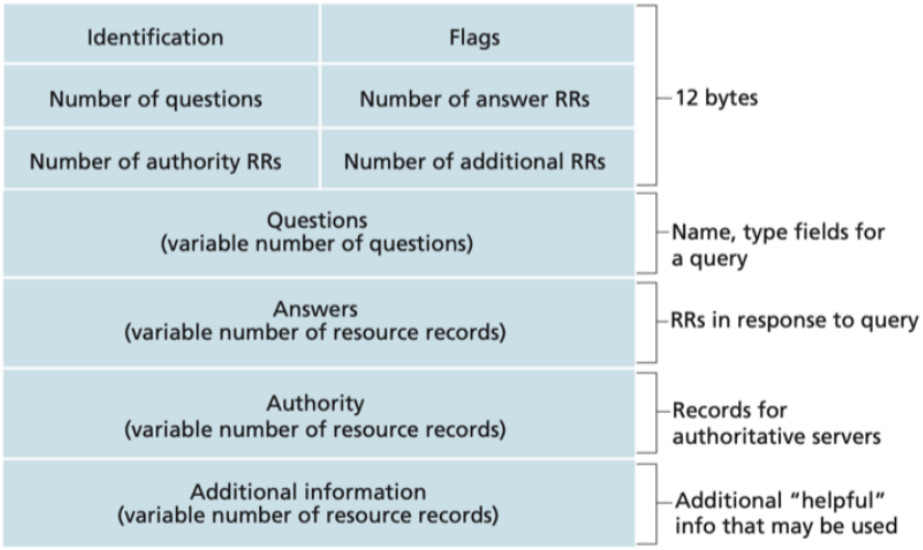

Figure 2: DNS Message Format

Listing 1: DNS setting Server Code

```
import os
import socket
import threading
import struct
import time

IP = '192.168.1.194'
PORT = 4487
ADDR = (IP, PORT)
SIZE = 1024
FORMAT = "utf-8"
SERVER_DATA_PATH = "server_data"
dic = {}


def handle_client(data, addr, server):
    try:
        request_time = time.strftime("%Y-%m-%d %H:%M:%S",
           time.localtime())
        print(f"[RECEIVED MESSAGE] {data} from {addr} at {
           request_time}.")

        data = data.split()
        print("Request Time:", request_time)
        domain_name = data[0]
        print("Domain Name:", domain_name)

        file1 = open('dns_records.txt', 'r')
        found = False
        for line in file1:
            line = line.split()
            name = line[0]
            value = line[1]
            type = line[2]
            ttl = line[3]
            if name == domain_name and type == data[1]:
                print('Found DNS Record')
                flag = 0
                q = 0
                a = 1
                auth_rr = 0
                add_rr = 0

                # Pack DNS header fields and message into the
                    same buffer
                ms = (name + ' ' + value + ' ' + type + ' ' +
                    ttl).encode('utf-8')
                packed_data = struct.pack(f"6H{len(ms)}s",
                   50, flag, q, a, auth_rr, add_rr, ms)
```

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

Listing 2: DNS Setting Client Code

```
import socket
import struct
import time

ADDR = ('192.168.1.194', 4487)
SIZE = 1024
FORMAT = 'utf-8'


def main():
    try:
        client = socket.socket(socket.AF_INET, socket.
           SOCK_DGRAM)

        message = input("Enter a message to send to the
           server: ")
        client.sendto(message.encode(FORMAT), ADDR)
        request_time = time.strftime("%Y-%m-%d %H:%M:%S",
           time.localtime())

        msg, addr = client.recvfrom(SIZE)
        print('Received DNS Response Message in bytes:')
        print(msg)

        header = struct.unpack("6H", msg[:12])
        ms = msg[12:].decode('utf-8').split()
        print('\nDecoded DNS Response:')
        print("Header:", header)
        print("Request Time:", request_time)
        print("Domain Name:", ms[0])
        if len(ms) > 1:
            print("DNS Record:", ms[1])
            if len(ms) > 2:
                print("Additional Information:", ms[2])

    except Exception as e:
        print("Error occurred while communicating with the
           server:", e)


if __name__ == '__main__':
    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.

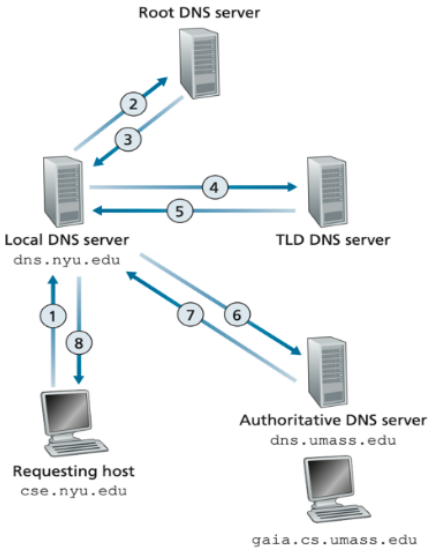

Figure 3: Iterative DNS Resolution

Listing 3: Iterative Local DNS Server Code

```
import socket
import struct

ROOT_DNS_ADDR = ('10.33.3.11', 4487)
TLD_DNS_ADDR = ('10.33.3.11', 4488)
AUTH_DNS_ADDR = ('10.33.3.11', 4489)

def encode_msg(message):
    data = message.split()
    name = data[0]
    type = data[1]
    flag = 0
    q = 0
    a = 1
    auth_rr = 0
    add_rr = 0
    ms = (name + ' ' + type + ' ' + data[2] + ' ' + data[3]).
       encode('utf-8')
    packed_data = struct.pack(f"6H{len(ms)}s", 50, flag, q, a
       , auth_rr, add_rr, ms)
    return packed_data

def resolve_domain(domain_name):
    try:
        # Query Root DNS Server
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            as root_server:
            root_server.sendto(domain_name.encode(),
               ROOT_DNS_ADDR)
            root_response, _ = root_server.recvfrom(1024)
            tld_addr = root_response.decode()

        # Query TLD DNS Server
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            as tld_server:
            tld_server.sendto(domain_name.encode(),
               TLD_DNS_ADDR)
            tld_response, _ = tld_server.recvfrom(1024)
            auth_addr = tld_response.decode()

        # Query Authoritative DNS Server
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            as auth_server:
            auth_server.sendto(domain_name.encode(),
               AUTH_DNS_ADDR)
            auth_response, _ = auth_server.recvfrom(1024)
            ip_address = auth_response.decode()

        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.
                SOCK_DGRAM)
50          server.bind(('localhost', 5000))
51
52          print("[LISTENING] Local DNS Server is listening on
                port 5000...")
53
54          while True:
55              data, addr = server.recvfrom(1024)
56              domain_name = data.decode()
57              print(f"[RECEIVED QUERY] {domain_name} from {addr
                    }")
58
59              # Resolve domain iteratively
60              ip_address = resolve_domain(domain_name)
61
62              if ip_address:
63                  # Send IP address to client
64                  server.sendto(ip_address.encode(), addr)
65                  print(f"Sent IP address of {domain_name} to {
                        addr}")
66
67      except Exception as e:
68          print("Error occurred in main:", e)
69
70  if __name__ == "__main__":
71      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"
```

```
dic = {
    "www.google.com": ('100.20.8.1', 'A', 86400),
    "www.cse.du.ac.bd": ('4488', 'NS', 86400),
    "www.yahoo.com": ('4488', "NS", 86400)
}


def handle_client(data, addr, server):
    try:
        print(f"[RECEIVED MESSAGE] {data} from {addr}.")
        msg = encode_msg(str(data + ' ' + dic[data][0] + ' '
           + dic[data][1] + ' ' + str(dic[data][2])))
        server.sendto(msg, addr)


    except Exception as e:
        print("ERROR: ", str(e))


def encode_msg(message):
    data = message.split()
    name = data[0]
    type = data[1]
    print(message)
    flag = 0
    q = 0
    a = 1
    auth_rr = 0
    add_rr = 0

    ms = (name + ' ' + type + ' ' + data[2] + ' ' + data[3]).
       encode('utf-8')
    packed_data = struct.pack(f"6H{len(ms)}s", 50, flag, q, a
       , auth_rr, add_rr, ms)
    return packed_data


def decode_msg(msg):
    header = struct.unpack("6H", msg[:12])
    ms = msg[12:].decode('utf-8')
    print('\n After Decoding')
    print({header}, {ms})
    ms = ms.split()
    return ms[1], ms[4]


def main():
    print("[STARTING] ROOT Server is starting")
    server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    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 Authoratitive 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]).
          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
37 def decode_msg(msg):
38     header = struct.unpack("6H", msg[:12])
39     ms = msg[12:].decode('utf-8')
40     print('\n After Decoding')
41     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] + ' '
                + dic[data][1] + ' ' + str(dic[data][2])))
49          server.sendto(msg, addr)
50
51          # if dic[data][1]=='A' or dic[data][1]=='AAAA':
52          #     print('sending')
53          #     server.sendto(str(data+' '+dic[data][0]+' '+dic
                [data][1]+' '+str(dic[data][2])).encode(FORMAT),
                addr)
54          # elif  dic[data][1]=='NS':
55          #     server.sendto(data.encode(FORMAT),authoratitive
                )
56          #     ans, auth_adr = server.recvfrom(SIZE)
57          #     server.sendto(ans,addr)
58      except Exception as e:
59          server.sendto(data.encode(FORMAT), ('', 4487))
60
61          print("ERROR: ", str(e))
62
63
64  def main():
65      print("[STARTING] TLD Server is starting")
66      server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
67      server.bind(ADDR)
68      print(f"[LISTENING] TLD Server is listening on {IP}:{PORT
            }.")
69
70      while True:
71          data, addr = server.recvfrom(SIZE)
72          data = data.decode(FORMAT)
73          # thread = threading.Thread(target=handle_client,
                args=(data, addr,server))
74          # thread.start()
75          handle_client(data, addr, server)
76          print(f"[ACTIVE CONNECTIONS] {threading.active_count
                () - 1}")
77
78
79  if __name__ == "__main__":
80      main()
```

Listing 7: Iterative Client Code

```
import socket
import struct

IP = ''
PORT = 4487
ADDR = (IP, PORT)
SIZE = 1024
FORMAT = "utf-8"


def encode_msg(message):
    data = message.split()
    name = data[0]
    type = data[1]

    flag = 0
    q = 0
    a = 1
    auth_rr = 0
    add_rr = 0

    ms = (name + ' ' + type).encode('utf-8')
    packed_data = struct.pack(f"6H{len(ms)}s", 50, flag, q, a
       , auth_rr, add_rr, ms)
    return packed_data


def decode_msg(msg):
    header = struct.unpack("6H", msg[:12])
    ms = msg[12:].decode('utf-8')

    print('\n Before Decoding')
    print(msg)

    print('\n After Decoding')
    print({header}, {ms})

    return ms


def main():
    client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    message = input("Enter an address: ")
    client.sendto(message.encode(FORMAT), ADDR)
    msg, addr = client.recvfrom(SIZE)
    msg1 = decode_msg(msg)
    print(msg1)
    data = msg1.split()
```

```
49      while data[2] == "NS":
50          # print(data[1])
51          new_adr = ('', int(data[1]))
52          print('Connecting to port', data[1])
53          client.sendto(message.encode(FORMAT), new_adr)
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.

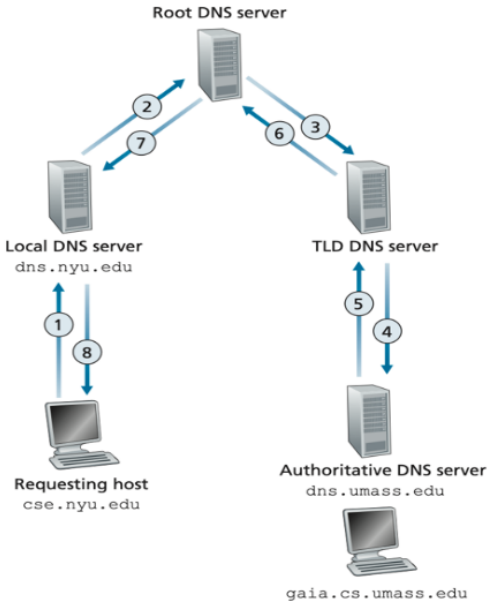

Figure 4: Recursive DNS Resolution

Listing 8: Recursive Local DNS Server Code

```
import socket
import struct

ROOT_DNS_ADDR = ('127.0.0.1', 4487)


def encode_msg(message):
    try:
        data = message.split()
        name = data[0]
        type = data[1]
        flag = 0
        q = 0
        a = 1
        auth_rr = 0
        add_rr = 0
        ms = (name + ' ' + type + ' ' + data[2] + ' ' + data
           [3]).encode('utf-8')
        packed_data = struct.pack(f"6H{len(ms)}s", 50, flag,
           q, a, auth_rr, add_rr, ms)
        return packed_data
    except Exception as e:
        print("Error occurred while encoding message:", e)
        return None


def resolve_domain(domain_name, server_addr):
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            as dns_server:
            dns_server.sendto(domain_name.encode(),
               server_addr)
            response, _ = dns_server.recvfrom(1024)
            return response.decode(), server_addr[1]

    except Exception as e:
        print("Error occurred while resolving domain:", e)
        return None, None


def main():
    try:
        server = socket.socket(socket.AF_INET, socket.
           SOCK_DGRAM)
        server.bind(('localhost', 5000))

        print("[LISTENING] Local DNS Server is listening on
           port 5000...")

```

```
        while True:
            try:
                data, addr = server.recvfrom(1024)
                domain_name = data.decode()
                print(f"[RECEIVED QUERY] {domain_name} from {addr[0]} for target {addr[1]}")

                ip_address, port = resolve_domain(domain_name, ROOT_DNS_ADDR)

                if ip_address and port:
                    server.sendto(ip_address.encode(), addr)
                    print(f"Sent IP address of {domain_name} to {addr}")

            except Exception as e:
                print("Error occurred in query processing:", e)

    except KeyboardInterrupt:
        print("Server terminated by user.")
    except Exception as e:
        print("Error occurred in main:", e)
    finally:
        server.close()


if __name__ == "__main__":
    main()
```

Listing 9: Recursive Root Server Code

```
import os
import socket
import threading
import struct

IP = '127.0.0.1'
PORT = 4487
ADDR = (IP, PORT)
tld = (IP, 4488)
SIZE = 1024
FORMAT = "utf-8"
SERVER_DATA_PATH = "server_data"
dic = {
    "www.google.com": ('100.20.8.1', 'A', 86400),
    "www.cse.du.ac.bd": ('4488', 'NS', 86400),
    "www.yahoo.com": ('4488', "NS", 86400)
}
```

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

```
def main():
    print("[STARTING] ROOT Server is starting")
    server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server.bind(ADDR)
    print(f"[LISTENING] ROOT Server is listening on {IP}:{
       PORT}.")

    while True:
        try:
            data, addr = server.recvfrom(SIZE)
            data = data.decode(FORMAT)
            handle_client(data, addr, server)
            print(f"[ACTIVE CONNECTIONS] {threading.
               active_count() - 1}")
        except KeyboardInterrupt:
            print("Server terminated by user.")
            break
        except Exception as e:
            print("Error occurred:", e)

    server.close()


if __name__ == "__main__":
    main()
```

Listing 10: Recursive Authoratitive Server Code

```
import os
import socket
import threading
import struct

IP = '127.0.0.1'
PORT = 4489
ADDR = (IP, PORT)
SIZE = 1024
FORMAT = "utf-8"
SERVER_DATA_PATH = "server_data"
dic = {
    "www.google.com": ('100.20.8.1', 'A', 86400),
    "www.cse.du.ac.bd": ('192.0.2.3', 'A', 86400),
    "www.yahoo.com": ('1.2.3.9999', "A", 86400)
}


def encode_msg(message):
    try:
        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
               [3]).encode('utf-8')
30          packed_data = struct.pack(f"6H{len(ms)}s", 50, flag,
               q, a, auth_rr, add_rr, ms)
31          return packed_data
32      except Exception as e:
33          print("Error occurred while encoding message:", e)
34          return None
35
36
37  def decode_msg(msg):
38      try:
39          header = struct.unpack("6H", msg[:12])
40          ms = msg[12:].decode('utf-8')
41          print('\n After Decoding')
42          print({header}, {ms})
43          ms = ms.split()
44          return ms[1], ms[4]
45      except Exception as e:
46          print("Error occurred while decoding message:", e)
47          return None, None
48
49
50  def handle_client(data, addr, server):
51      try:
52          print(data)
53          msg = encode_msg(str(data + ' ' + dic[data][0] + ' '
               + dic[data][1] + ' ' + str(dic[data][2])))
54          if msg:
55              server.sendto(msg, addr)
56      except KeyError:
57          print(f"Requested domain '{data}' not found in
               authoritative DNS.")
58      except Exception as e:
59          print("ERROR: ", str(e))
60
61
62  def main():
63      print("[STARTING] auth Server is starting")
64      server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
65      server.bind(ADDR)
66      print(f"[LISTENING] auth Server is listening on {IP}:{
           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
               [3]).encode('utf-8')
34          packed_data = struct.pack(f"6H{len(ms)}s", 50, flag,
               q, a, auth_rr, add_rr, ms)
35          return packed_data
36      except Exception as e:
37          print("Error occurred while encoding message:", e)
38          return None
39
40
41  def decode_msg(msg):
42      try:
43          header = struct.unpack("6H", msg[:12])
44          ms = msg[12:].decode('utf-8')
45          print('\n After Decoding')
46          print({header}, {ms})
47          return ms
48      except Exception as e:
49          print("Error occurred while decoding message:", e)
50          return None
51
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] + ' '
               + dic[data][1] + ' ' + str(dic[data][2])))
57          server.sendto(msg, addr)
58      except KeyError:
59          print(f"Requested domain '{data}' not found in TLD
               DNS.")
60      except Exception as e:
61          server.sendto(data.encode(FORMAT), ('', 4487))
62          print("ERROR: ", str(e))
63
64
65  def main():
66      print("[STARTING] TLD Server is starting")
67      server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
68      server.bind(ADDR)
69      print(f"[LISTENING] TLD Server is listening on {IP}:{PORT
           }.")
70
71      while True:
72          try:
```

```
            data, addr = server.recvfrom(SIZE)
            data = data.decode(FORMAT)
            handle_client(data, addr, server)
            print(f"[ACTIVE CONNECTIONS] {threading.
               active_count() - 1}")
        except KeyboardInterrupt:
            print("Server terminated by user.")
            break
        except Exception as e:
            print("Error occurred:", e)

    server.close()


if __name__ == "__main__":
    main()
```

Listing 12: Recursive Client Code

```
import socket
import struct

IP = ''
PORT = 4487
ADDR = (IP, PORT)
SIZE = 1024
FORMAT = "utf-8"


def encode_msg(message):
    data = message.split()
    name = data[0]
    type = data[1]

    flag = 0
    q = 0
    a = 1
    auth_rr = 0
    add_rr = 0

    ms = (name + ' ' + type).encode('utf-8')
    packed_data = struct.pack(f"6H{len(ms)}s", 50, flag, q, a
       , auth_rr, add_rr, ms)
    return packed_data


def decode_msg(msg):
    header = struct.unpack("6H", msg[:12])
    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.

Listing 13: TTL value Server Code

```
import socket
import threading
import time

class DNSRecord:
    def __init__(self, name, value, record_type, ttl):
        self.name = name
        self.value = value
        self.record_type = record_type
        self.ttl = ttl
        self.creation_time = time.time()

    def is_expired(self):
        return (time.time() - self.creation_time) > self.ttl

class DNSServer:
    def __init__(self):
        self.records = {}
        self.deleted_records = {}  # To store deleted records
        self.cleanup_interval = 10  # Cleanup interval in
            seconds
        self.cleanup_thread = threading.Thread(target=self.
            cleanup_records)
        self.cleanup_thread.daemon = True
        self.cleanup_thread.start()

    def add_record(self, name, value, record_type, ttl):
        self.records[name] = DNSRecord(name, value,
            record_type, ttl)

    def get_record(self, name):
        record = self.records.get(name)
        if record and not record.is_expired():
            return record.value
        else:
            return None

    def cleanup_records(self):
        while True:
            expired_records = [name for name, record in self.
                records.items() if record.is_expired()]
            for name in expired_records:
```

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

Listing 14: TTL value Client Code

```
import socket

def main():
    client_socket = socket.socket(socket.AF_INET, socket.
       SOCK_STREAM)
    client_socket.connect(('127.0.0.1', 12345))

    query = "www.example.com"
    client_socket.send(query.encode('utf-8'))
    response = client_socket.recv(1024).decode('utf-8')
    print("Response:", response)

    client_socket.close()

if __name__ == "__main__":
    main()
```

4 Experimental result

4.1 Task 1- Setting up the DNS server

4.1.1 'dns records.txt' File:

server.py | dns_records.txt | client.py

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

Figure 5: Content of dns records.txt

4.1.2 After Running Server Code:

server | client

```
/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:

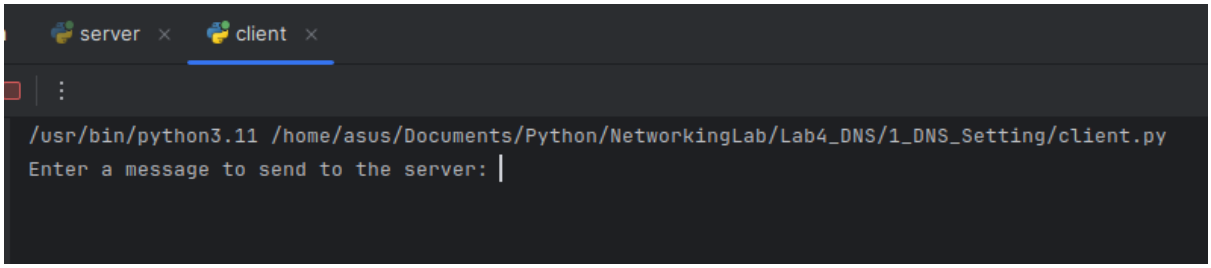

Figure 7: Server Connect to Client

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.

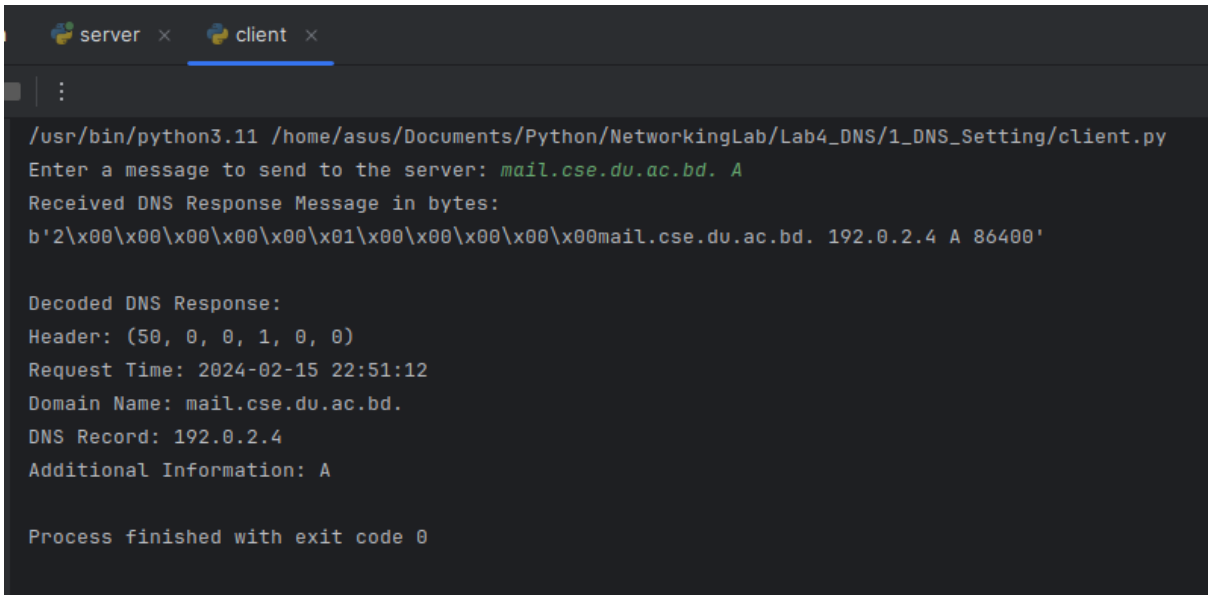

Figure 8: Get information of target address

4.1.5 After entering target address, in server: :

Here, we can see the file we just downloaded.

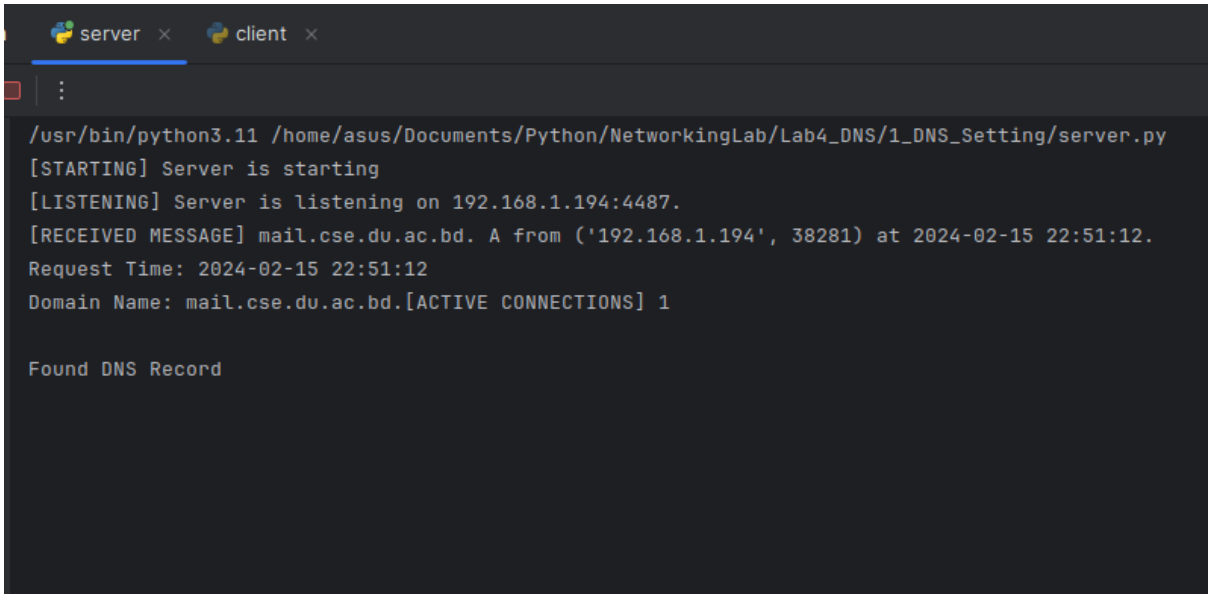

Figure 9: Get information of the client

4.1.6 Error for server

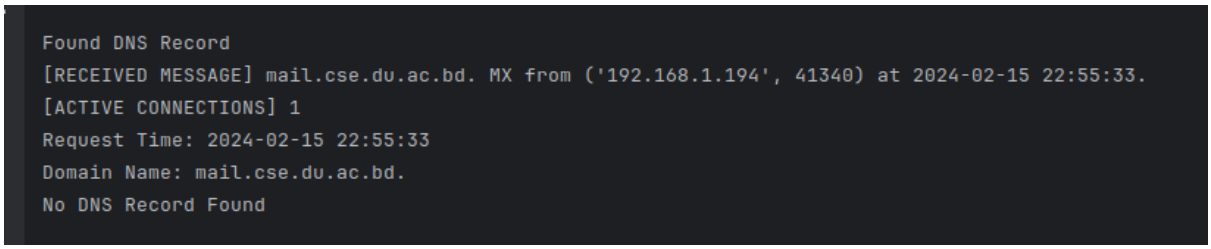

Figure 10: Error in server

4.1.7 Error for client

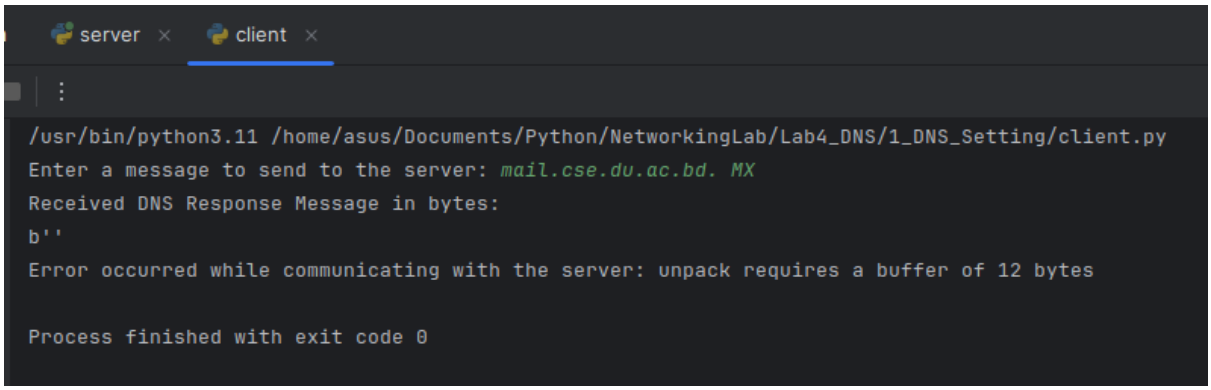

Figure 11: Error in client

4.2 Task 2- Iterative DNS resolution

4.2.1 After Running rootSever Code:

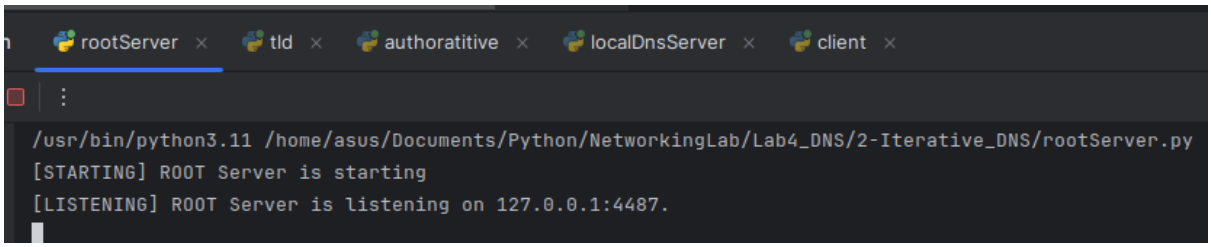

Figure 12: Root Server Connecting

4.2.2 After Running tld Code:

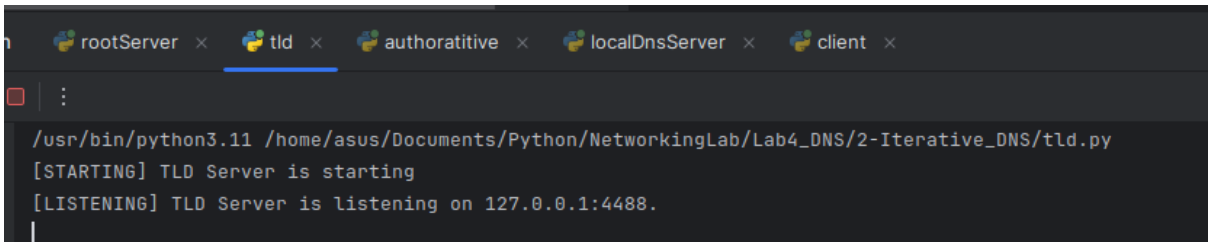

Figure 13: TLD Server Connecting

4.2.3 After Running authoritative Code:

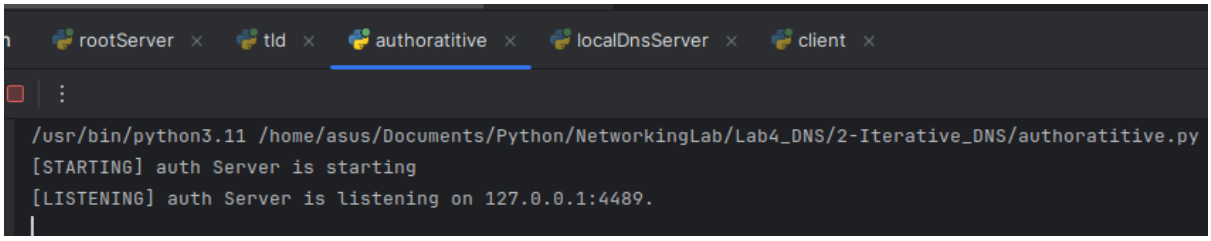

Figure 14: Authoritative Server Connecting

4.2.4 After Running localDnsSever Code:

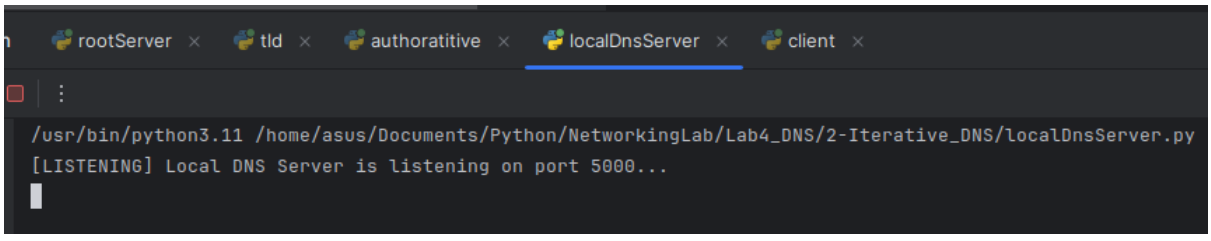

Figure 15: Local DNS Server Connectimg

4.2.5 After Running client Code:

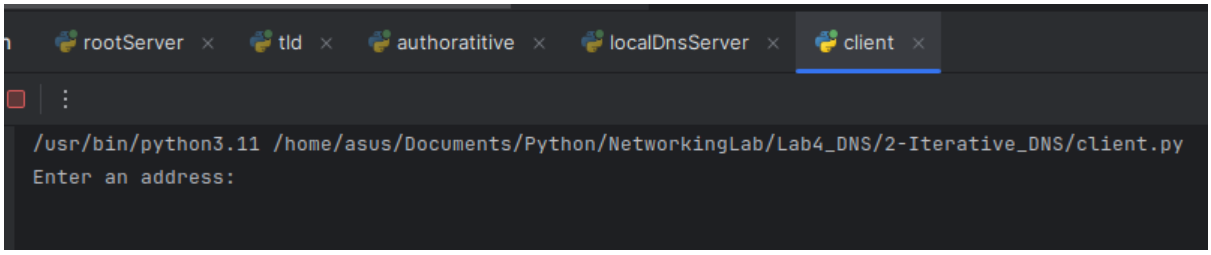

Figure 16: Server Connect to Client

4.2.6 After client entering address:

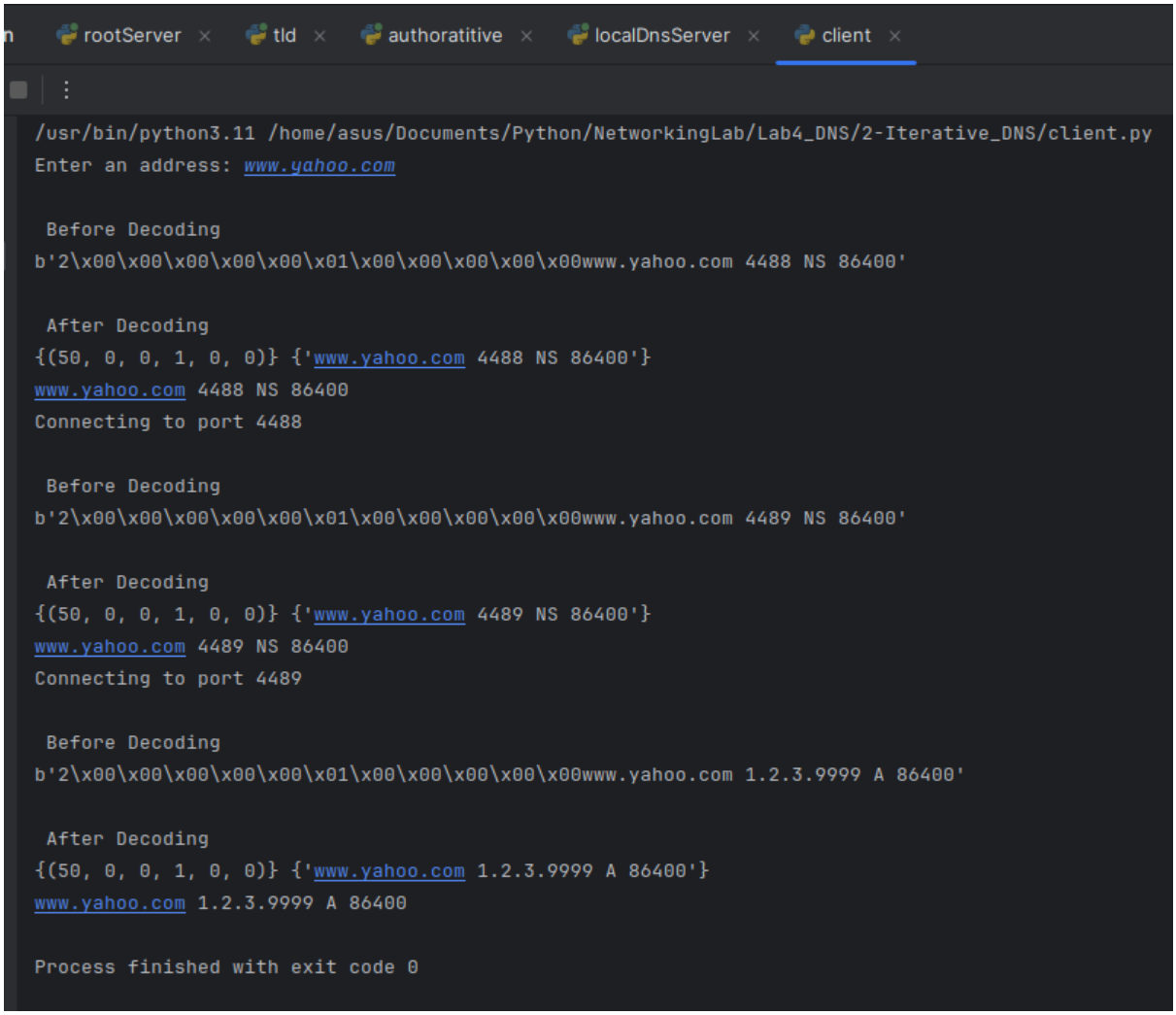

Figure 17: Client response

4.2.7 Respone from Root server:

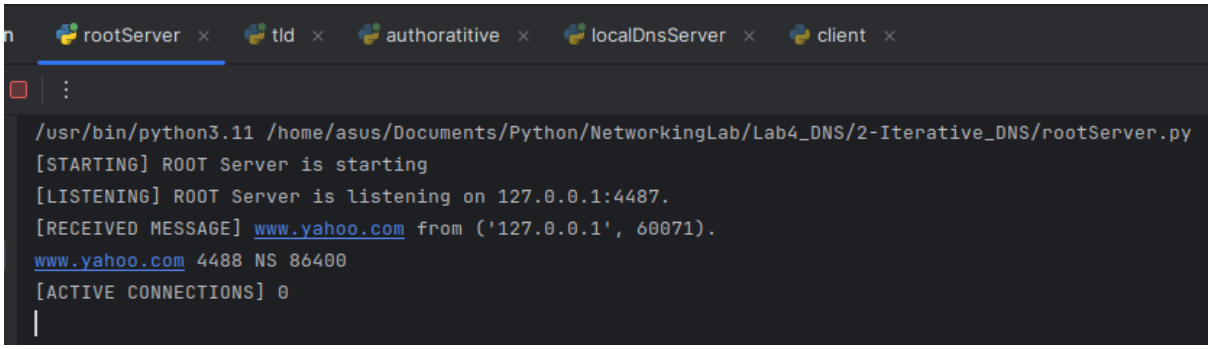

Figure 18: rootServer response

4.2.8 Respone from tld server:

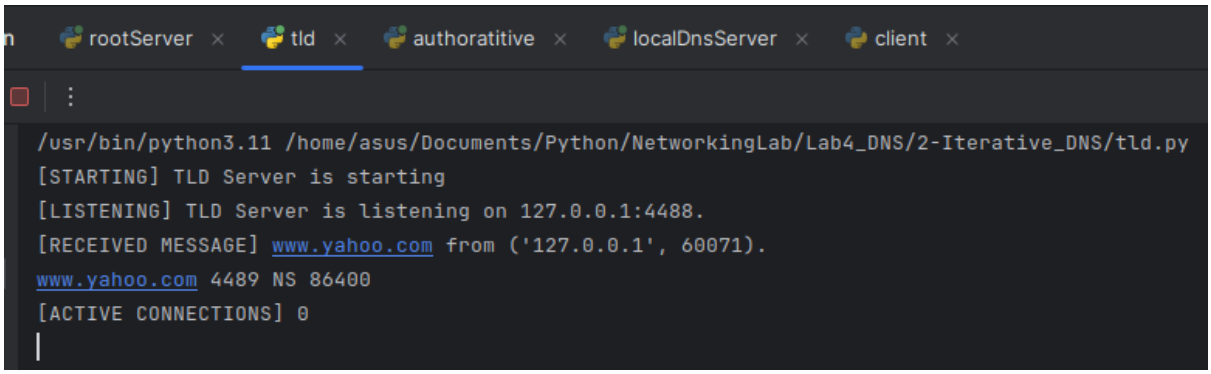

Figure 19: tld response

4.2.9 Respone from authoratitive server:

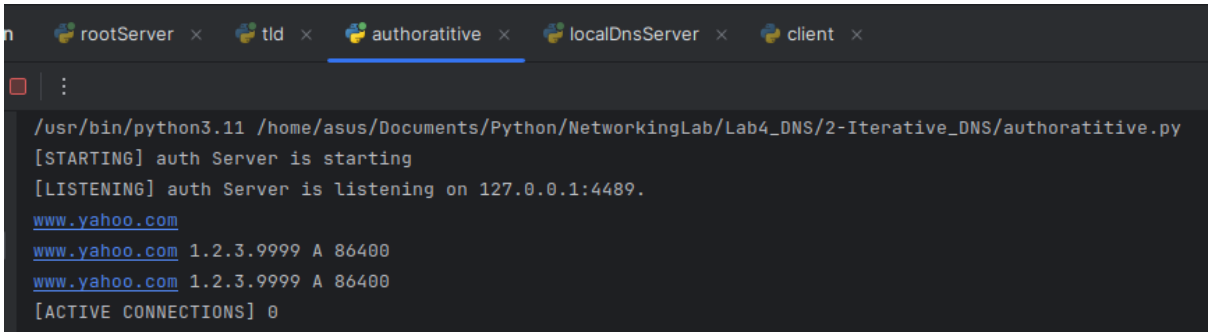

Figure 20: authoratitive response

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:

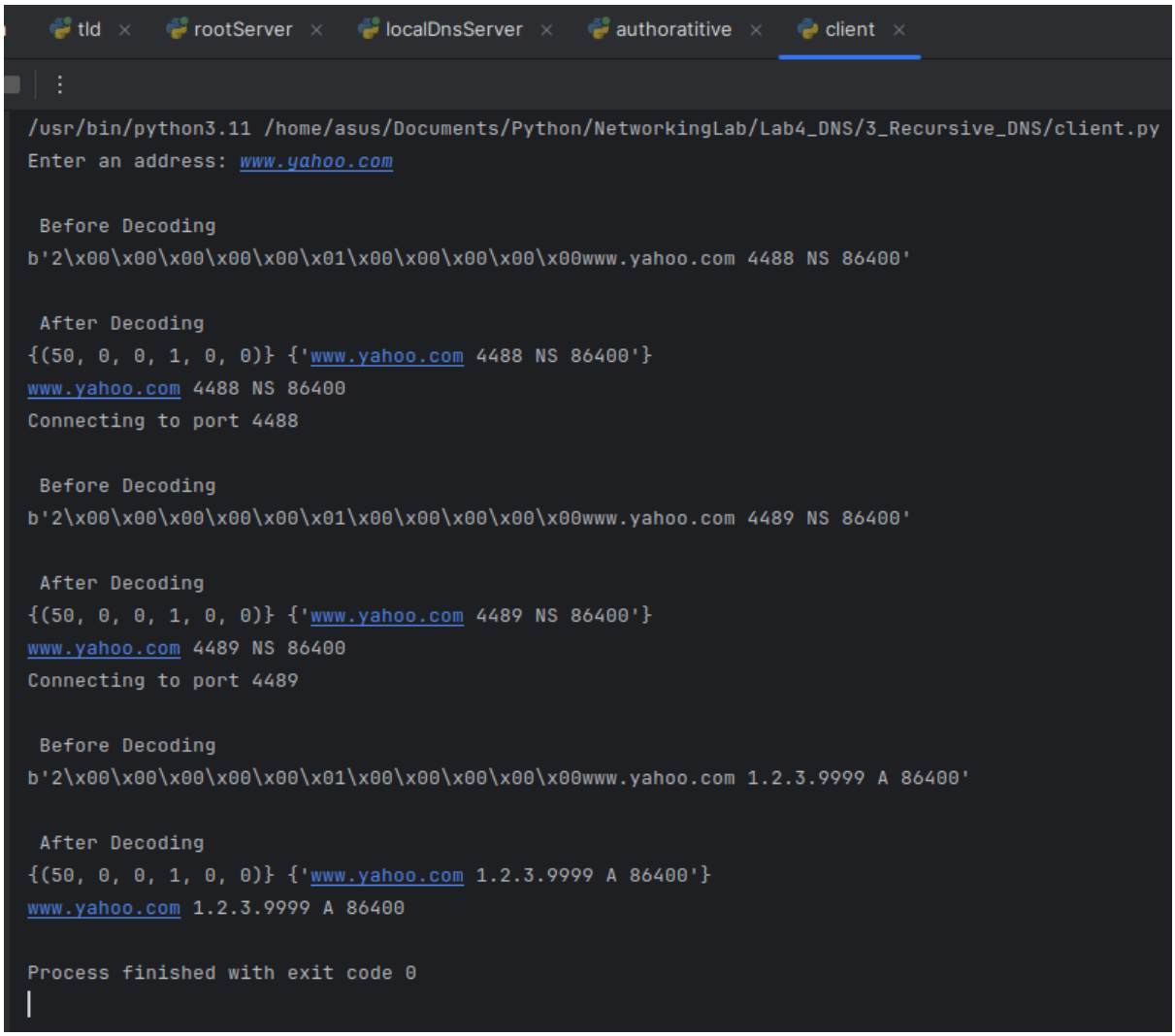

Figure 22: Client response

4.3.2 Respone from Root server:

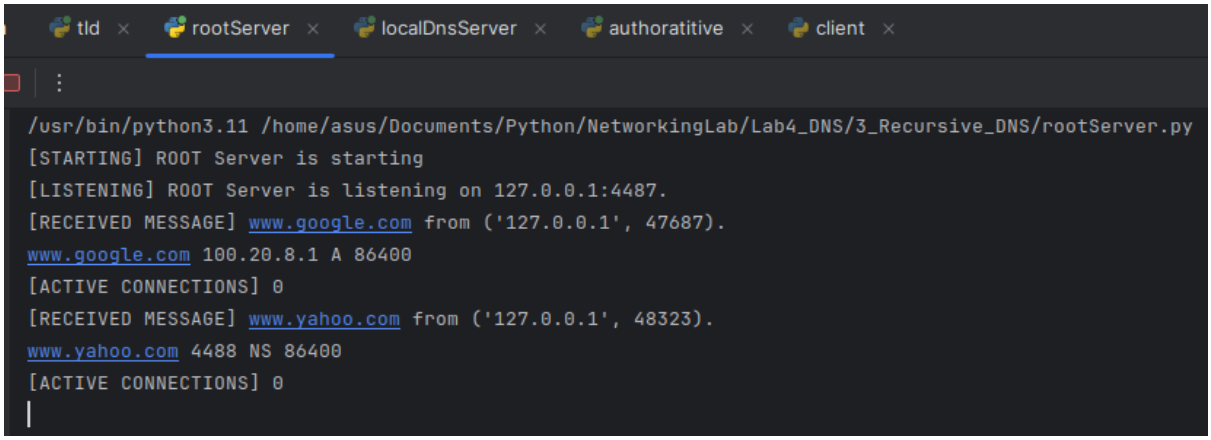

Figure 23: rootServer response

4.3.3 Respone from tld server:

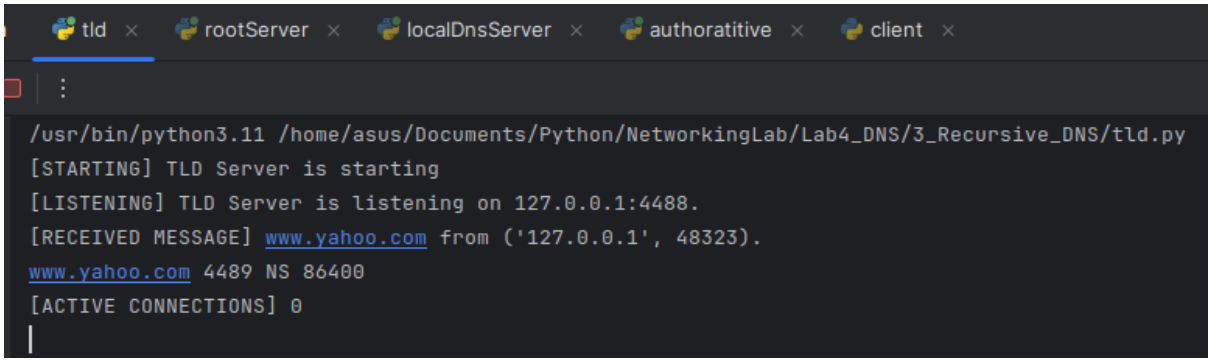

Figure 24: tld response

4.3.4 Respone from authoratitive server:

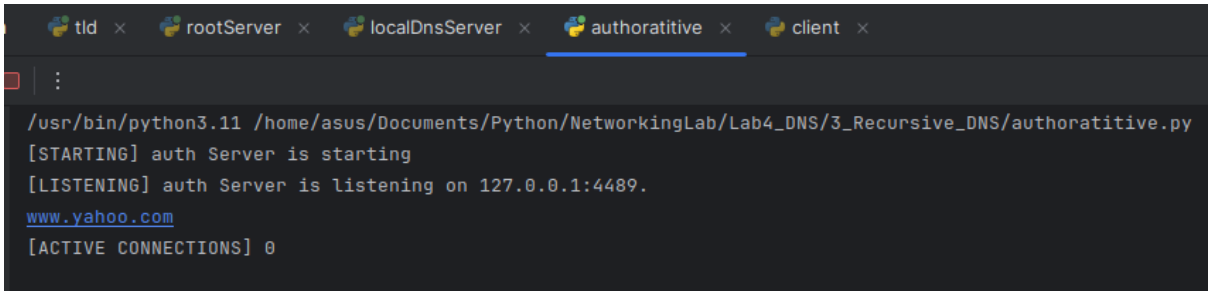

Figure 25: authoratitive response

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:

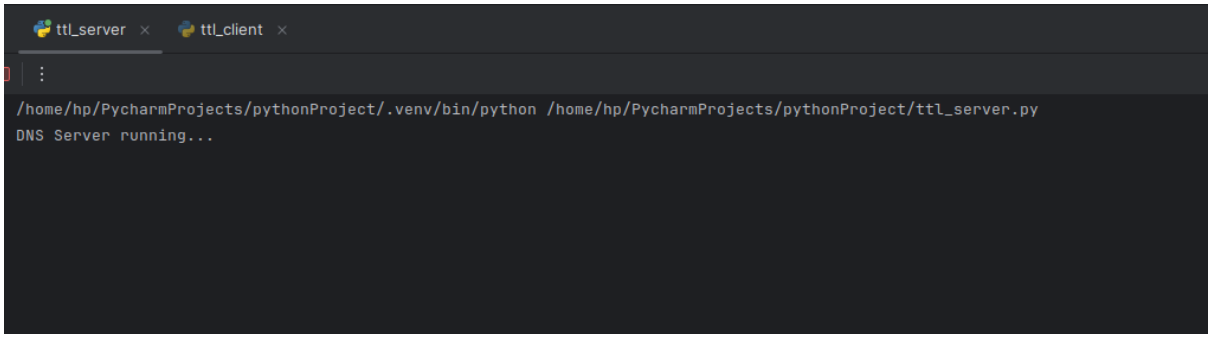

Figure 27: ttl value server

4.4.2 After running ttl client code:

Resources delete sucessfully.

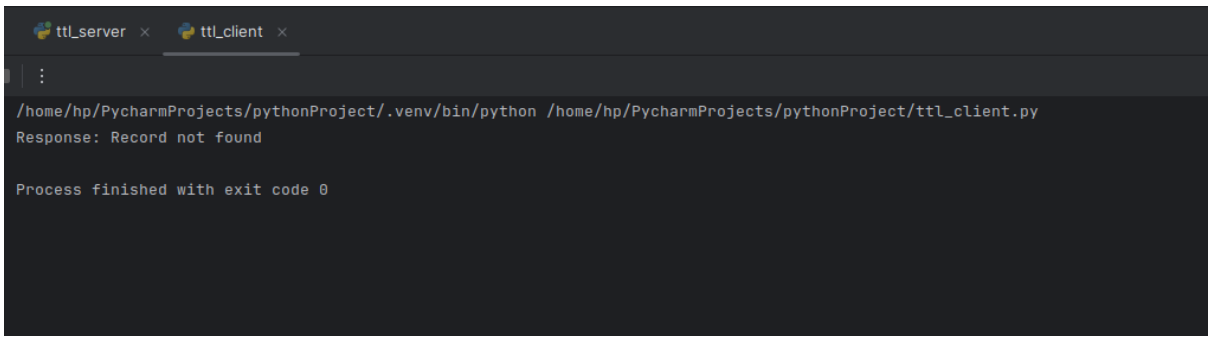

Figure 28: ttl value client

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-postRecord Types -

[4] JavaTpoint. (n.d.). Java GET and POST. Retrieved from https://www.javatpoint.com/java-get-post