CSE351

Computer Networks Project

DNS Server Agent

Phase 1

Team Number 3

Team Members:

Mostafa Hassan Mohamed 21p0349

Omar Khaled Essam 21P0178

Mohamed Walid Helmy 21p0266

Tarek Hadef Mostafa 21p0199

Table of Contents

Introduction

This document serves as an in-depth guide for planning and designing a DNS server agent. The Domain Name System (DNS) protocol was selected due to its critical role in the infrastructure of the internet, converting human-readable domain names into machine-friendly IP addresses. This project focuses on implementing a DNS server that adheres to RFC 1034, RFC 1035, and RFC 2181 standards. Each of these RFCs defines crucial aspects of DNS:

RFC 1034: Concepts and facilities for DNS architecture.

RFC 1035: Implementation and specification of DNS operations.

RFC 2181: Clarifications and additional requirements for DNS behavior and data consistency.

The implementation will offer insights into networking concepts, message parsing, error handling, and protocol compliance. It will also lay the groundwork for potential future enhancements like DNSSEC for security improvements.

Objective

The objectives of this phase are:

1. Define the precise scope and goals of the DNS server project.
2. Establish a clear design framework detailing the system's architecture and components.
3. Ensure adherence to DNS standards as outlined in the relevant RFCs.
4. Provide the foundation for implementation and subsequent optimization phases.

The DNS server will prioritize modularity, scalability, and protocol compliance while enabling straightforward extensibility for future requirements.

Deliverables

1. A comprehensive design document detailing system architecture, components, and their interactions.
2. Well-defined communication protocols based on DNS message formats.
3. A testing framework with scenarios to validate functionality and compliance.
4. Initial implementation of modular components for the DNS server.

1. User Authentication Mechanisms

- DNS, as a protocol, is designed to be lightweight and efficient, which is why user authentication is not inherently part of its operations. However, as cybersecurity concerns grow, additional security measures like DNSSEC (Domain Name System Security Extensions) can be integrated in future phases.
- **DNSSEC Overview:**
- DNSSEC protects against attacks like cache poisoning by:
    - Adding digital signatures to DNS records.
    - Using public-key cryptography to validate DNS responses.
    - Ensuring authenticity and integrity of DNS data.
- **Benefits of DNSSEC:**
    - Prevents man-in-the-middle attacks.
    - Enhances trust in DNS communications.
    - Protects end users from malicious redirections.

- For future integration, DNSSEC implementation would require:
    - Generating DNSKEY and RRSIG records.
    - Configuring the DNS server to sign zones and validate signatures.
    - Extending the logger to capture DNSSEC-related data for debugging and auditing
- .

2. Project Scope

The DNS server aims to support core functionalities while maintaining compliance with relevant standards.

Key Functionalities:

1. Query Processing: Handle DNS queries for record types A (IPv4), AAAA (IPv6), and CNAME (canonical names).
2. Response Generation: Provide accurate and efficient responses adhering to DNS message formats.
3. Error Handling: Address malformed queries or unsupported requests with appropriate error codes (e.g., NXDOMAIN).
4. Logging: Maintain detailed logs of all queries, responses, and errors for debugging and analysis.

Why These Functionalities Matter:

- **Query Processing** ensures the server meets the fundamental purpose of DNS.
- **Error Handling** improves reliability and user trust.
- **Logging** provides a mechanism for tracking issues and monitoring server performance.

3. System Architecture

The DNS server's architecture is designed for modularity and scalability, ensuring that individual components can be developed, tested, and maintained independently.

3.1Components:

1. DNS Server Core:

- **Objective** : Act as the central controller for DNS operations.
- **Responsibilities**:
    - Listens on UDP port 53 for incoming DNS requests.
    - Routes queries to the Query Parser module.
    - Manages timeouts and retries.

2. Query Parser:

- **Objective** : Parse and validate incoming DNS queries.
- **Responsibilities**:
    - Validates query structure and formats (RFC 1035).
    - Extracts domain names and query types (e.g., A, AAAA).
    - Ensures compliance with DNS message format.

3. Response Generator:

- **Objective** : Create DNS responses based on query results.
- **Responsibilities**:
    - Fetches resource records from the database.
    - Constructs responses according to DNS message standards (RFC 1035).
    - Adds authoritative and additional section data as necessary.

4. Error Handler:

- **Objective** : Manage errors and provide appropriate responses.
- **Responsibilities**:
    - Generates error messages (e.g., NXDOMAIN, SERVFAIL).
    - Logs all errors for debugging and operational insights.

5. Logger:

- **Objective** : Record and analyze server operations.
- **Responsibilities**:
    - Logs queries, responses, and errors with timestamps.
    - Assists in debugging and compliance auditing.

6.Database

- **Objective :** Store and retrieve DNS records.
- **Responsibilities:**
    - Contains entries for domain names, record types, TTLs, and values:
        - ✓ **Domain Name**: Primary key for lookups.
        - ✓ **Record Type**: A, AAAA, or CNAME.
        - ✓ **TTL (Time to Live)**: Defines how long the record is valid.
        - ✓ **Value**: Corresponding IP address or canonical name.

    - Provides a scalable backend for quick lookups.

7.Client :

- **Objective :** External component that sends queries to the DNS server.
- **Responsibilities:**
    - Generates DNS query messages following RFC standards.
    - Receives and processes responses or errors from the DNS server.

8.Network(internet)

- **Objective :** The medium through which DNS messages are transmitted.
- **Responsibilities:**
    - Facilitates communication between the client and the DNS server.
    - Ensures message integrity and timely delivery.

3.2Interaction between components:

1. **DNS Server Core**:
    - Receives a query over UDP (or TCP if necessary) from the Network and forwards it to the Query Parser.
    - Receives parsed query results and routes them to the Response Generator or Error Handler.
2. **Query Parser**:
    - Decodes the DNS query header and extracts the domain name and query type.
    - Sends the extracted information to the Response Generator for processing.

3. **Response Generator**:
    - Queries the Database for a matching record.
    - Constructs a complete DNS response, adhering to RFC 1035.
    - Returns the response to the DNS Server Core for transmission.
4. **Error Handler**:
    - Engaged if the Query Parser detects a malformed query or the Database returns no matching record.
    - Generates appropriate error responses, following RFC 1035 error codes.
    - Passes error messages to the Logger and back to the Server Core.
5. **Logger**:
    - Receives data from all components to create a detailed operational log.
    - Stores logs for error resolution and performance analysis.
6. **Database**:
    - Interacts with the Response Generator to supply queried records.
    - Notifies the Error Handler if no record matches a query.
7. **Client**:
    - Sends queries via the Network to the DNS Server Core.
    - Receives responses or error messages for further processing.

8. **Network**:
    - Transmits DNS queries and responses between the Client and DNS Server Core.
    - Handles packet loss and retries in case of failed transmissions.

3.3 Compliance with RFCs

- RFC 1034 and RFC 1035: Used to define query and response format.
- RFC 2181: Ensures data consistency and caching rules.
- Error Handling: Implements standard error codes (NXDOMAIN, SERVFAIL).

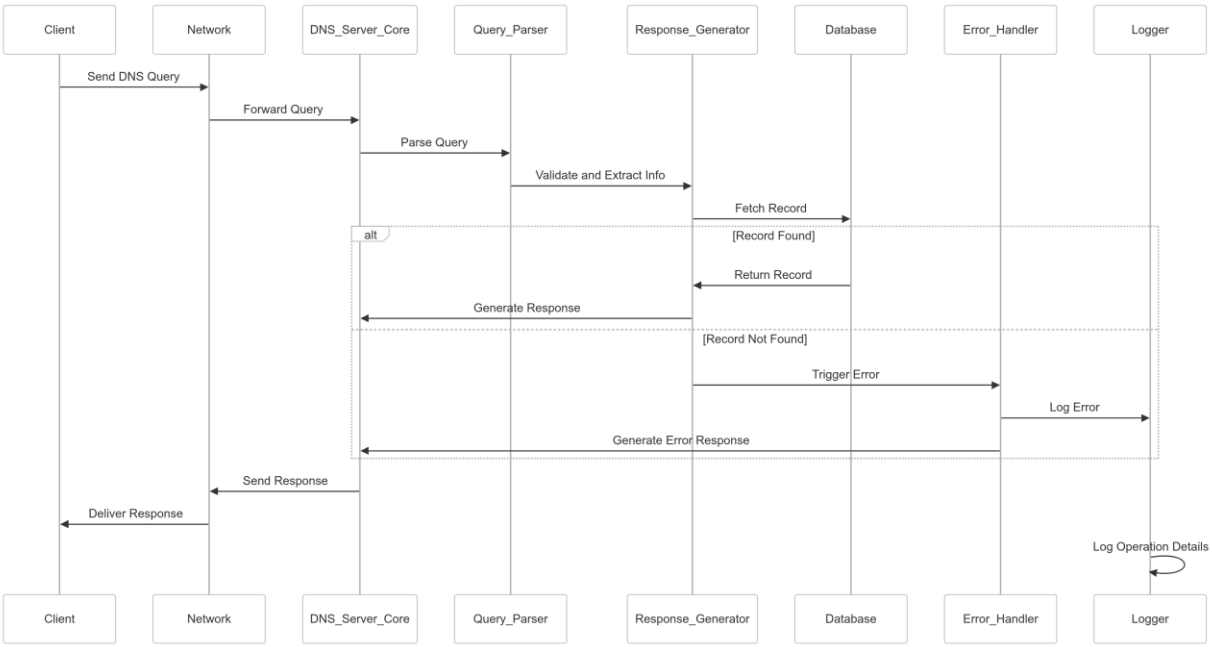
3.4 Example Interaction Flow

1. A client sends a query for example.com (type A).
2. The Network transmits the query to the DNS Server Core.
3. The DNS Server Core receives the query and passes it to the Query Parser.
4. The Query Parser validates the query structure and extracts example.com.
5. The Response Generator queries the Database for example.com and finds an A record.
6. The Response Generator constructs a response with the IP address.
7. The DNS Server Core sends the response back to the client via the Network.

If no record exists:

1. The Response Generator notifies the Error Handler.
2. The Error Handler generates an NXDOMAIN error.
3. The DNS Server Core transmits the error to the client via the Network.

3.5 Components Interaction Diagram:



Link of mermaid for edit Diagram :

https://www.mermaidchart.com/app/projects/b4e4e862-89e7-46a5-90c1-8b5a8e7b8d37/diagrams/0803d2ce-5b9a-4e2f-a61b-fa23124ceb44/version/v0.1/edit

4. Communication and Message Protocols

4.1 Protocol Details and Header Format

4.1.1 DNS Message Format (RFC1035):

Each DNS message consists of:

1. **Header**:

- **ID (16 bits)**: Identifies the query uniquely for matching responses.
- **Flags (16 bits)**:
  - QR (1 bit): Query (0) or Response (1).
  - Opcode (4 bits): Query type (e.g., standard query = 0).
  - AA (1 bit): Authoritative answer.
  - TC (1 bit): Truncated message.
  - RD (1 bit): Recursion desired.
  - RA (1 bit): Recursion available.
  - Z (3 bits): Reserved, must be zero.
  - RCODE (4 bits): Response code (e.g., NOERROR, NXDOMAIN).
- **QDCOUNT (16 bits)**: Number of queries.
- **ANCOUNT (16 bits)**: Number of answers.
- **NSCOUNT (16 bits)**: Number of authoritative records.
- **ARCOUNT (16 bits)**: Number of additional records.

2. **Question Section**:
   - **QNAME**: Domain name being queried.
     - **Example:** For the query example.com, QNAME will be example.com
   - **QTYPE**: Query type
     - Common Types:
       - ✓ A (1): IPv4 address record.
       - ✓ AAAA (28): IPv6 address record.

- ✓ MX (15): Mail exchange record.

- **QCLASS**: Query class (usually IN for internet).
  - ➢ **Example:** QCLASS is usually **IN** (internet), though other classes like **CH** (CHAOS) and **HS** (Hesiod) exist but are rarely used.

3. **Answer, Authority, Additional Sections**:

These sections contain the resource records that the DNS server returns in response to the query. The content varies depending on the query type.

- **Answer Section:** Contains the record(s) corresponding to the query. For example, if you query for an A record, this section will contain the IP address of the domain.
- **Authority Section:** Lists authoritative DNS servers that could be queried further for information.
- **Additional Section:** Provides additional records that might be useful (e.g., if a record points to a mail server, the additional section might contain MX records).

Example Explanation:

- When a query with ID=1234 is sent asking for example.com, the Header will have QDCOUNT=1, QR=0 (query), and appropriate flags for recursion (RD=1).
- The response Header will have QR=1, AA=1 (if authoritative), and possibly RCODE=NOERROR or NXDOMAIN.

4.1.2 Protocol Details:

- **UDP**:
  - ✓ Default protocol for DNS queries.

- ✓ Lightweight and efficient for most scenarios.
- ✓ Limited to 512 bytes (extended with EDNS0).

- **TCP**:
  - ✓ Used for zone transfers (AXFR) or large messages.
  - ✓ Reliable transport ensures delivery of all fragments.

5. Compliance with RFCs

To ensure compliance with DNS protocol standards, the DNS server will adhere to the following RFCs:

- **RFC 1035** - *Domain Names - Implementation and Specification*: This RFC provides the foundational specifications for DNS, including message format, query and response structure, and the behavior of DNS servers.
  - **Message Format Compliance:** The DNS server will ensure that message formats, including headers, flags, and sections, conform to the structure defined in RFC 1035.
  - **Caching Rules:** The DNS server will implement caching mechanisms as per RFC 1035, ensuring that resolved records are cached with appropriate time-to-live (TTL) values to improve performance and reduce query load.
- **RFC 1034** - *Domain Names - Concepts and Facilities*: This RFC defines the overall architecture of the Domain Name System, including the structure of the domain names and the responsibilities of DNS servers.
  - **Message Validity:** The DNS server will validate domain names and ensure they follow the syntax rules defined in RFC 1034.

- **RFC 2181** - *Clarifications to the DNS Specification*: This RFC provides clarifications to DNS operations and ensures that implementations follow best practices for handling queries and responses.
    - **Query Handling:** The DNS server will reject queries with invalid domain names or malformed messages as defined in RFC 2181, ensuring robustness against malformed requests.
- **Error Handling:** When a DNS query is invalid, the server will respond with appropriate error codes defined in **RFC 1035** (e.g., RCODE = FORMERR for format errors, RCODE = NXDOMAIN for non-existent domain errors).
    - **Example:** If a malformed query is received, the DNS server will return an error with **RCODE = FORMERR** to indicate the query format is incorrect, in compliance with RFC 1035.

By adhering to these RFCs, the DNS server will maintain protocol compliance and provide reliable and consistent resolution of DNS queries.

CSE351

Computer Networks Project

DNS Server Agent

Phase 2

Team Number 3

Team Members:

Mostafa Hassan Mohamed 21p0349

Omar Khaled Essam 21P0178

Mohamed Walid Helmy 21p0266

Tarek Hadef Mostafa 21p0199

File Server.py:

```python
# Function to build DNS response
def build_dns_response(query, ip_addresses):
    transaction_id = query[:2]
    flags = struct.pack("!H", 0x8180)  # Standard query response, no error
    questions = struct.pack("!H", 1)  # One question
    answer_rrs = struct.pack("!H", len(ip_addresses))  # Number of answer records
    authority_rrs = struct.pack("!H", 0)
    additional_rrs = struct.pack("!H", 0)
    question = query[12:]

    # Build the answer section for multiple IPs
    answers = b""
    for ip in ip_addresses:
        answer_name = b'\xc0\x0c'  # Pointer to the domain name in the question
        answer_type = struct.pack("!H", 1)  # Type A (IPv4 address)
        answer_class = struct.pack("!H", 1)  # Class IN (Internet)
        answer_ttl = struct.pack("!I", 3600)  # TTL (time to live)
        answer_length = struct.pack("!H", 4)  # Length of the IP address
        answer_ip = socket.inet_aton(ip)  # Convert IP to 4-byte format
        answers += answer_name + answer_type + answer_class + answer_ttl + answer_length + answer_ip

    response = transaction_id + flags + questions + answer_rrs + authority_rrs + additional_rrs
    response += question + answers
    return response
```

This function constructs a DNS response packet for a query received from a client. It starts by extracting the transaction ID from the query, which helps match the response to the original query. Flags are set to indicate that the response is a standard DNS reply without errors. The response includes counts for the number of questions, answers, and other sections, with only the answers containing data.

The original question from the query is included unchanged in the response. For each IP address provided, the function creates an answer record with details such as the type (IPv4), class (Internet), time-to-live (TTL), and the IP address in a suitable binary format. These answer records are then combined into the final response packet, which is a complete DNS response ready to be sent back to the client.

```python
# Function to handle incoming DNS queries
def handle_dns_query(data, addr, server_socket):
    try:
        # Extract the domain name from the query
        domain_name = ""
        i = 12  # Skip the header

        while data[i] != 0:
            length = data[i]
            domain_name += data[i+1:i+1+length].decode() + "."
            i += length + 1

        domain_name = domain_name[:-1]  # Remove the trailing dot

        print(f"Received query for domain: {domain_name}")

        # Check if the domain exists in the DNS records
        if domain_name in DNS_RECORDS:
            ip_addresses = DNS_RECORDS[domain_name]
            if isinstance(ip_addresses, list):
                ip_to_return = random.choice(ip_addresses)  # Randomly pick one IP for this query
            else:
                ip_to_return = ip_addresses
            print(f"Resolved {domain_name} to {ip_to_return}")
            response = build_dns_response(data, [ip_to_return])
        else:
            print(f"Domain {domain_name} not found")
            response = build_dns_response(data, ["0.0.0.0"])  # Respond with an empty IP (indicating error)

        # Send the response back to the client
        server_socket.sendto(response, addr)

    except Exception as e:
        print(f"Error handling DNS query: {e}")
```

This function processes incoming DNS queries and generates appropriate responses. It begins by extracting the domain name from the query packet by decoding its structure. Once the domain name is identified, it logs the query for debugging purposes.

The function then checks if the domain exists in a predefined list of DNS records. If the domain is found, it resolves it to the corresponding IP address or randomly selects one if multiple IPs are available. If the domain is not found, it prepares a response with a default "not found" IP address (0.0.0.0).

Finally, it constructs the DNS response using the resolved IP address or the default IP and sends the response back to the client using the server's socket. This process allows the server to handle and respond to DNS requests dynamically.

```python
# A simple database of domain -> IP mappings
DNS_RECORDS = {
    "example.com": "93.184.216.34",  # example.com resolves to this IP
    "localhost": "127.0.0.1",
    "netflix.com": ["54.74.73.31", "54.155.178.5", "3.251.50.149"]
}
```

DNS_RECORDS is a dictionary mapping domain names to their IP addresses. Some domains, like example.com, resolve to a single IP, while others, like netflix.com, have multiple IPs listed to support load balancing or redundancy. It serves as a lookup table for resolving domain names in the DNS server.

```python
# DNS Server
def start_dns_server(host, port):
    try:
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        server_socket.bind((host, port))
        print(f"DNS Server started on {host}:{port}")

        while True:
            # Receive DNS query from a client
            data, addr = server_socket.recvfrom(512)  # DNS queries are typically < 512 bytes
            print(f"Received data from {addr}")
            handle_dns_query(data, addr, server_socket)
    except Exception as e:
        print(f"Error starting DNS server: {e}")
```

The function starts a UDP-based DNS server, binds it to the specified host and port, and continuously listens for queries. It processes each query using the handle_dns_query function and handles errors gracefully if they occur.

File Client.py:

```python
import socket
import struct

# Function to send DNS query and receive the response
def query_dns_server(domain, server_host, server_port):
    # Build DNS query (for A record type)
    transaction_id = struct.pack("!H", 0x1234)  # Transaction ID (random for simplicity)
    flags = struct.pack("!H", 0x0100)  # Standard query
    questions = struct.pack("!H", 1)  # One question
    answer_rrs = struct.pack("!H", 0)  # No answers yet
    authority_rrs = struct.pack("!H", 0)  # No authority records
    additional_rrs = struct.pack("!H", 0)  # No additional records

    # Encode the domain name in the query (fully qualified, e.g., "example.com")
    query_name = b''.join([bytes([len(label)]) + label.encode() for label in domain.split('.')]) + b'\x00'
    query_type = struct.pack("!H", 1)  # Type A (address record)
    query_class = struct.pack("!H", 1)  # Class IN (Internet)

    query = transaction_id + flags + questions + answer_rrs + authority_rrs + additional_rrs
    query += query_name + query_type + query_class

    # Send the DNS query to the server
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    client_socket.sendto(query, (server_host, server_port))

    # Receive the response from the server
    response, _ = client_socket.recvfrom(512)  # DNS response should fit in 512 bytes
    client_socket.close()

    # Extract the IP address from the response
    ip_address = '.'.join(map(str, response[-4:]))
    return ip_address
```

This function sends a DNS query for an A record to a specified server, receives the response, and extracts the resolved IP address from the server's reply. It returns the IP address of the queried domain.

```python
if __name__ == "__main__":
    # Define the server details
    server_host = '192.168.1.3'  # Change to the server's IP address
    server_port = 1053           # Use the port your server is listening on

    print("Type 'exit' to quit the program.")

    # Input validation for number of queries
    while True:
        try:
            num_queries = int(input("Enter the number of domain names to query: "))
            if num_queries <= 0:
                print("Please enter a positive number.")
                continue
            break
        except ValueError:
            print("Invalid input. Please enter a valid number.")

    for _ in range(num_queries):
        domain = input("Enter domain name to query (e.g., example.com): ")
        if domain.lower() == 'exit':
            print("Exiting the program.")
            break

        try:
            ip_address = query_dns_server(domain, server_host, server_port)
            print(f"{domain}: {ip_address}")
        except Exception as e:
            print(f"An error occurred while resolving {domain}: {e}")
```

This code allows the user to query a DNS server for multiple domains. It validates the number of queries and prompts the user for domain names. For each domain, it sends a DNS query, retrieves the resolved IP address, and prints it. The user can type "exit" to quit the program.

Result of a simple command-line interface showing the server status:

This is a snippet for the client server communication

Basic RFCs handled

RFC	Handled Feature in code
RFC 1034	Mapping domain names to IP addresses (DNS_RECORDS).
RFC 1035	Message format, domain name encoding, query/response handling, pointer compression, and UDP constraints
RFC 2181	Message format, domain name encoding, query/response handling, pointer compression, and UDP constraints

CSE351

Computer Networks Project

DNS Server Agent

Phase 3

Team Number 3

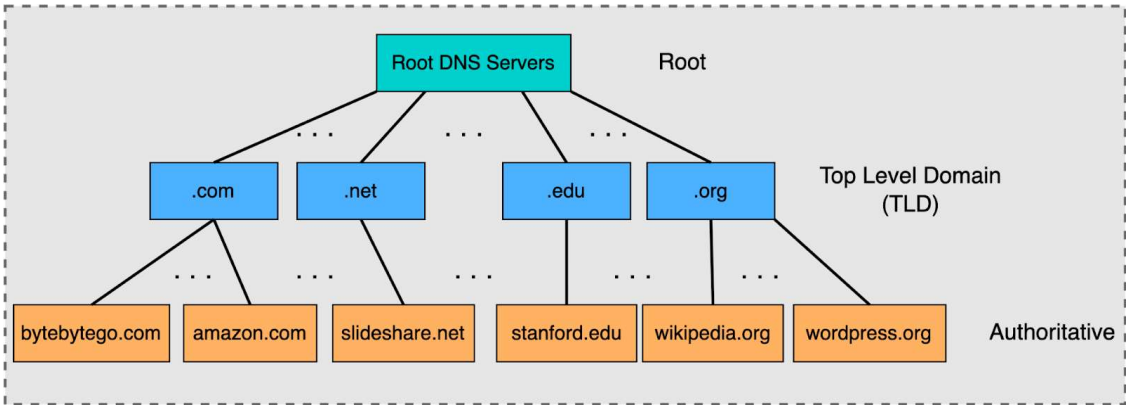
Team Members:

Mostafa Hassan Mohamed 21p0349

Omar Khaled Essam 21P0178

Mohamed Walid Helmy 21p0266

Tarek Hadef Mostafa 21p0199



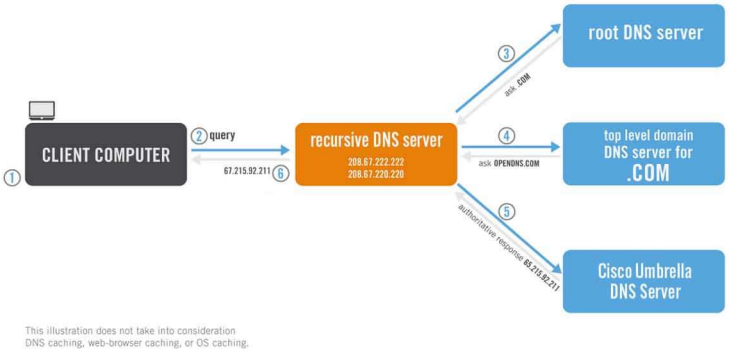
This image illustrates the hierarchical structure of the Domain Name System (DNS), which resolves human-readable domain names into IP addresses. Here's a breakdown:

1. **Root DNS Servers (Top Level)**:
    - At the top of the hierarchy, the root servers are the entry point for DNS resolution.
    - These servers handle queries for the top-level domains (TLDs) such as .com, .net, .edu, and .org.
2. **Top-Level Domain (TLD) Servers (Middle Level)**:
    - TLD servers are responsible for specific top-level domains.
    - For example, .com TLD servers handle domains like amazon.com and bytebytego.com.
3. **Authoritative Name Servers (Bottom Level)**:
    - These servers store the definitive DNS records for individual domains (e.g., bytebytego.com, amazon.com, wikipedia.org).
    - They provide the final answer, such as the IP address, for a query.

DNS Resolution Process:

1. **Step 1**: A client queries the root DNS server.
2. **Step 2**: The root server refers the query to the appropriate TLD server (e.g., .com).
3. **Step 3**: The TLD server refers the query to the authoritative name server for the specific domain.
4. **Step 4**: The authoritative server responds with the final IP address.

This structure ensures that DNS resolution is distributed, scalable, and efficient.



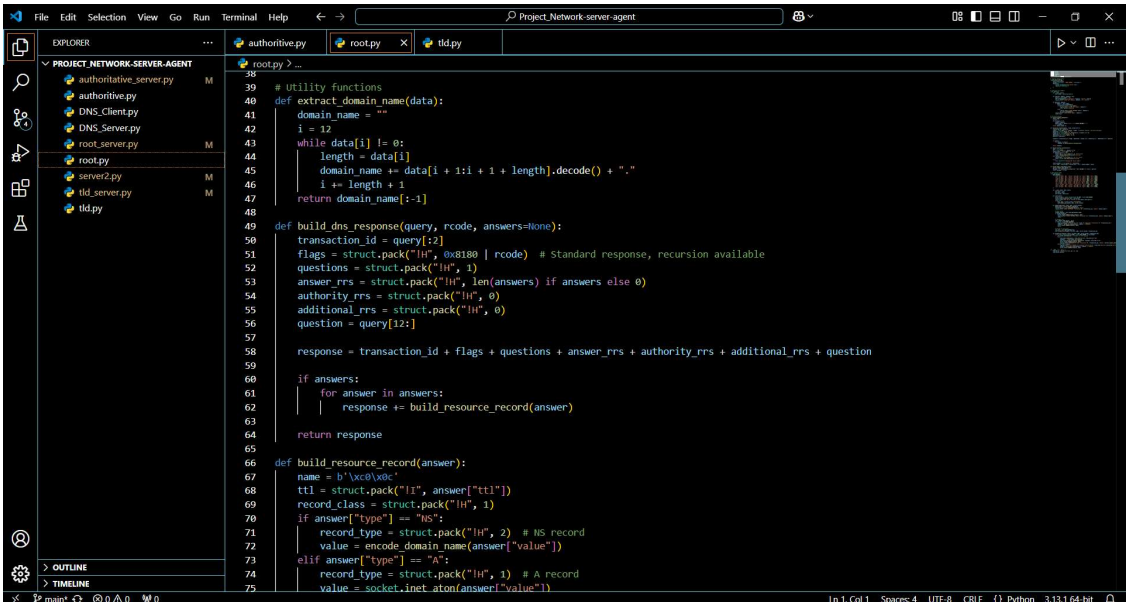
This image shows how a **recursive DNS resolution** process works when a client requests the IP address for a domain. Here's the step-by-step explanation:

Steps in Recursive DNS Resolution:

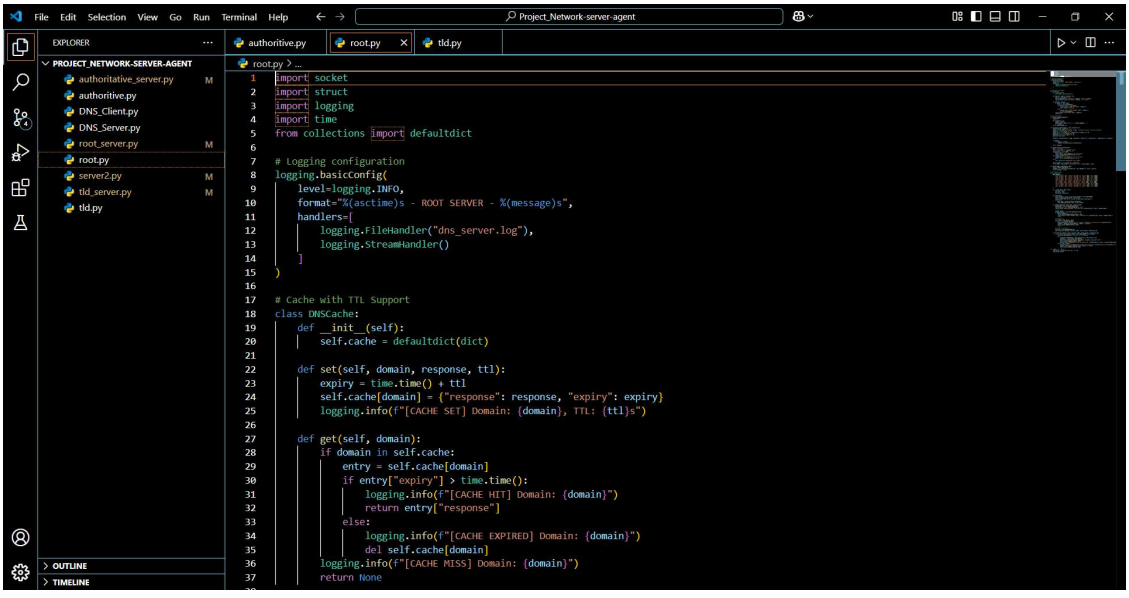
1. **Step 1: Client Query**:
    - The client computer sends a DNS query to the **recursive DNS server**. This is usually provided by the user's ISP or a third-party service (e.g., Google DNS or OpenDNS).
2. **Step 2: Recursive DNS Server**:
    - The recursive DNS server begins the process of resolving the query. If it doesn't already have the result cached, it sends the query to the **root DNS server**.
3. **Step 3: Root DNS Server**:
    - The root DNS server doesn't know the exact IP address but directs the recursive DNS server to the **TLD DNS server** for the domain's top-level domain (e.g., .com).
4. **Step 4: TLD DNS Server**:
    - The TLD server (for .com in this case) provides the recursive DNS server with the location of the **authoritative name server** responsible for the specific domain (e.g., opendns.com).
5. **Step 5: Authoritative DNS Server**:
    - The authoritative DNS server (in this example, a Cisco Umbrella DNS server) responds to the recursive server with the final IP address of the requested domain.

6. **Step 6: Response to Client**:
    - The recursive DNS server sends the resolved IP address back to the client computer, completing the process.

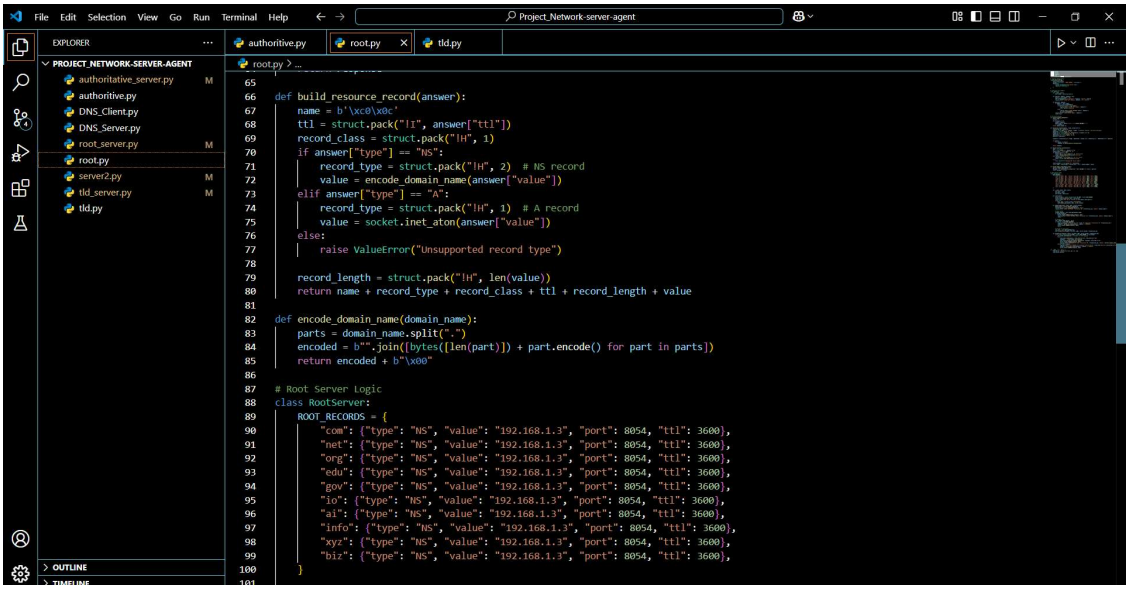
Root DNS



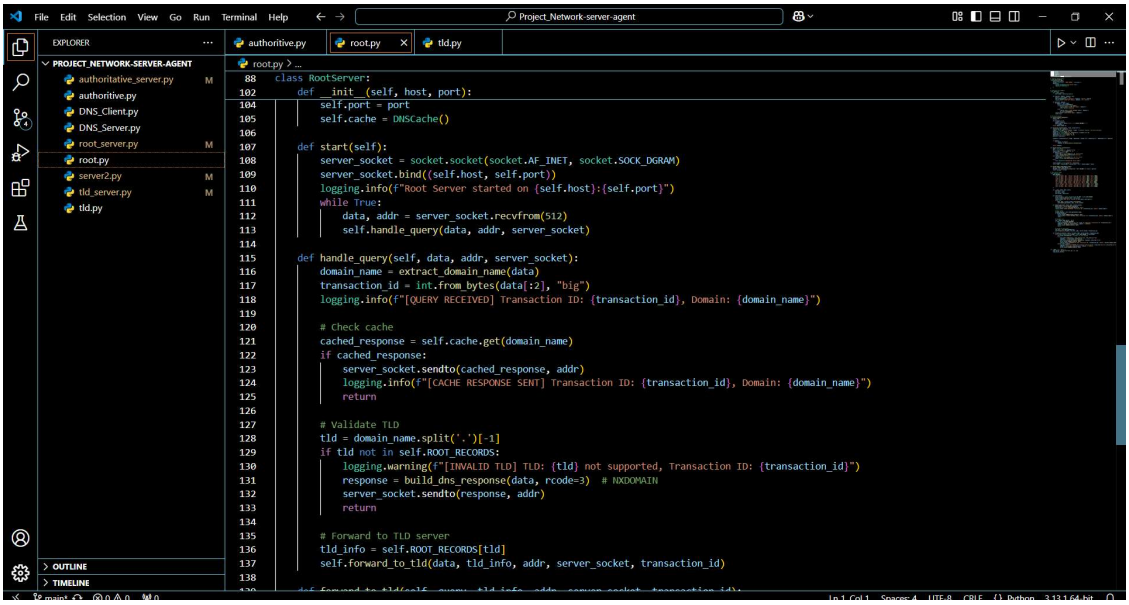
```python
# Utility functions
def extract_domain_name(data):
    domain_name = ""
    i = 12
    while data[i] != 0:
        length = data[i]
        domain_name += data[i + 1:i + 1 + length].decode() + "."
        i += length + 1
    return domain_name[:-1]

def build_dns_response(query, rcode, answers=None):
    transaction_id = query[:2]
    flags = struct.pack("!H", 0x8180 | rcode)  # Standard response, recursion available
    questions = struct.pack("!H", 1)
    answer_rrs = struct.pack("!H", len(answers) if answers else 0)
    authority_rrs = struct.pack("!H", 0)
    additional_rrs = struct.pack("!H", 0)
    question = query[12:]

    response = transaction_id + flags + questions + answer_rrs + authority_rrs + additional_rrs + question

    if answers:
        for answer in answers:
            response += build_resource_record(answer)

    return response

def build_resource_record(answer):
    name = b'\xc0\x0c'
    ttl = struct.pack("!I", answer["ttl"])
    record_class = struct.pack("!H", 1)
    if answer["type"] == "NS":
        record_type = struct.pack("!H", 2)  # NS record
        value = encode_domain_name(answer["value"])
    elif answer["type"] == "A":
        record_type = struct.pack("!H", 1)  # A record
        value = socket.inet_aton(answer["value"])
```



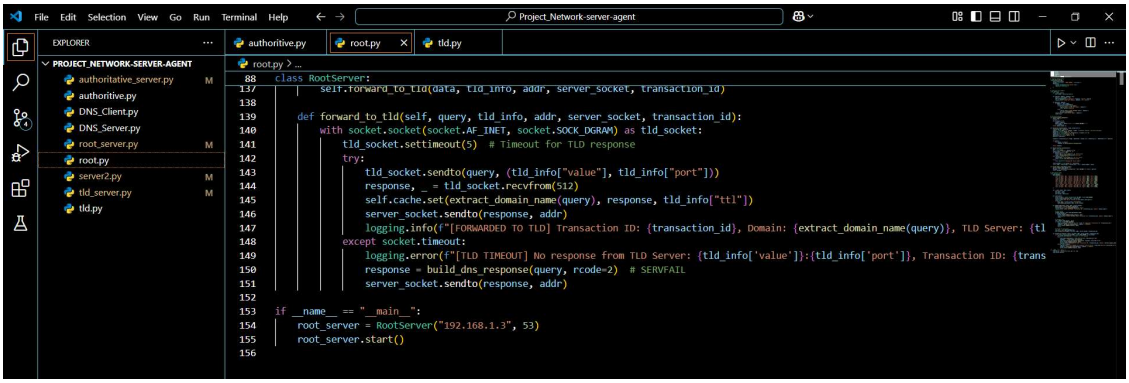
```python
import socket
import struct
import logging
import time
from collections import defaultdict

# Logging configuration
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - ROOT SERVER - %(message)s",
    handlers=[
        logging.FileHandler("dns_server.log"),
        logging.StreamHandler()
    ]
)

# Cache with TTL Support
class DNSCache:
    def __init__(self):
        self.cache = defaultdict(dict)

    def set(self, domain, response, ttl):
        expiry = time.time() + ttl
        self.cache[domain] = {"response": response, "expiry": expiry}
        logging.info(f"[CACHE SET] Domain: {domain}, TTL: {ttl}s")

    def get(self, domain):
        if domain in self.cache:
            entry = self.cache[domain]
            if entry["expiry"] > time.time():
                logging.info(f"[CACHE HIT] Domain: {domain}")
                return entry["response"]
            else:
                logging.info(f"[CACHE EXPIRED] Domain: {domain}")
                del self.cache[domain]
        logging.info(f"[CACHE MISS] Domain: {domain}")
        return None
```



```python
def build_resource_record(answer):
    name = b'\xc0\x0c'
    ttl = struct.pack("!I", answer["ttl"])
    record_class = struct.pack("!H", 1)
    if answer["type"] == "NS":
        record_type = struct.pack("!H", 2)  # NS record
        value = encode_domain_name(answer["value"])
    elif answer["type"] == "A":
        record_type = struct.pack("!H", 1)  # A record
        value = socket.inet_aton(answer["value"])
    else:
        raise ValueError("Unsupported record type")

    record_length = struct.pack("!H", len(value))
    return name + record_type + record_class + ttl + record_length + value

def encode_domain_name(domain_name):
    parts = domain_name.split(".")
    encoded = b"".join([bytes([len(part)]) + part.encode() for part in parts])
    return encoded + b"\x00"

# Root Server Logic
class RootServer:
    ROOT_RECORDS = {
        "com": {"type": "NS", "value": "192.168.1.3", "port": 8054, "ttl": 3600},
        "net": {"type": "NS", "value": "192.168.1.3", "port": 8054, "ttl": 3600},
        "org": {"type": "NS", "value": "192.168.1.3", "port": 8054, "ttl": 3600},
        "edu": {"type": "NS", "value": "192.168.1.3", "port": 8054, "ttl": 3600},
        "gov": {"type": "NS", "value": "192.168.1.3", "port": 8054, "ttl": 3600},
        "io": {"type": "NS", "value": "192.168.1.3", "port": 8054, "ttl": 3600},
        "ai": {"type": "NS", "value": "192.168.1.3", "port": 8054, "ttl": 3600},
        "info": {"type": "NS", "value": "192.168.1.3", "port": 8054, "ttl": 3600},
        "xyz": {"type": "NS", "value": "192.168.1.3", "port": 8054, "ttl": 3600},
        "biz": {"type": "NS", "value": "192.168.1.3", "port": 8054, "ttl": 3600},
    }
```



```python
class RootServer:
    def __init__(self, host, port):
        self.port = port
        self.cache = DNSCache()

    def start(self):
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        server_socket.bind((self.host, self.port))
        logging.info(f"Root Server started on {self.host}:{self.port}")
        while True:
            data, addr = server_socket.recvfrom(512)
            self.handle_query(data, addr, server_socket)

    def handle_query(self, data, addr, server_socket):
        domain_name = extract_domain_name(data)
        transaction_id = int.from_bytes(data[:2], "big")
        logging.info(f"[QUERY RECEIVED] Transaction ID: {transaction_id}, Domain: {domain_name}")

        # Check cache
        cached_response = self.cache.get(domain_name)
        if cached_response:
            server_socket.sendto(cached_response, addr)
            logging.info(f"[CACHE RESPONSE SENT] Transaction ID: {transaction_id}, Domain: {domain_name}")
            return

        # Validate TLD
        tld = domain_name.split('.')[-1]
        if tld not in self.ROOT_RECORDS:
            logging.warning(f"[INVALID TLD] TLD: {tld} not supported, Transaction ID: {transaction_id}")
            response = build_dns_response(data, rcode=3)  # NXDOMAIN
            server_socket.sendto(response, addr)
            return

        # Forward to TLD server
        tld_info = self.ROOT_RECORDS[tld]
        self.forward_to_tld(data, tld_info, addr, server_socket, transaction_id)
```

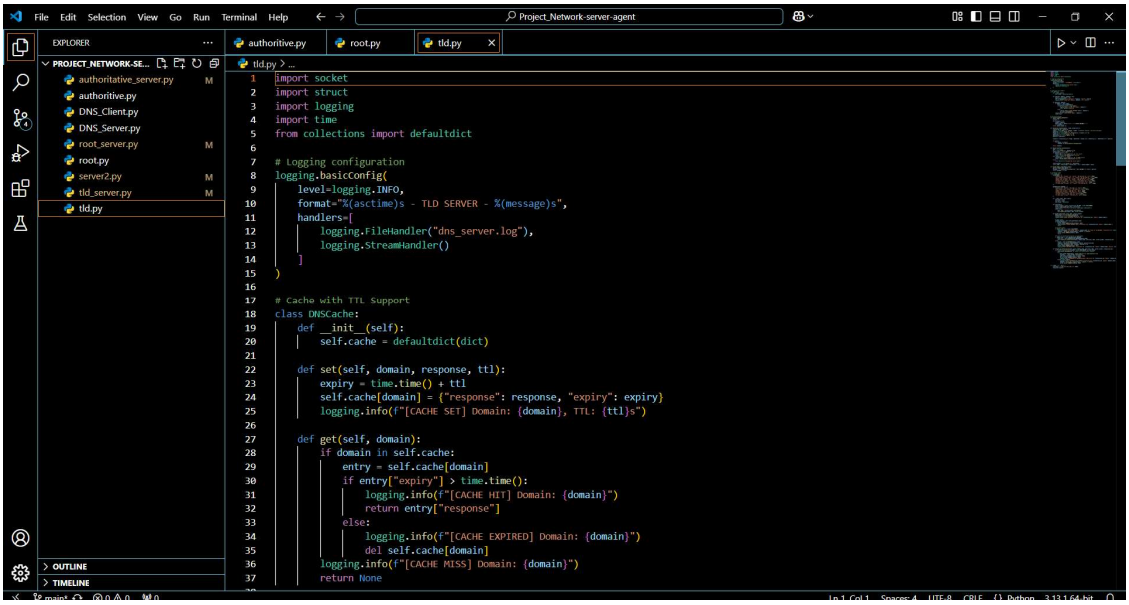


```python
class RootServer:
        self.forward_to_tld(data, tld_info, addr, server_socket, transaction_id)

    def forward_to_tld(self, query, tld_info, addr, server_socket, transaction_id):
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as tld_socket:
            tld_socket.settimeout(5)  # Timeout for TLD response
            try:
                tld_socket.sendto(query, (tld_info["value"], tld_info["port"]))
                response, _ = tld_socket.recvfrom(512)
                self.cache.set(extract_domain_name(query), response, tld_info["ttl"])
                server_socket.sendto(response, addr)
                logging.info(f"[FORWARDED TO TLD] Transaction ID: {transaction_id}, Domain: {extract_domain_name(query)}, TLD Server: {tl
            except socket.timeout:
                logging.error(f"[TLD TIMEOUT] No response from TLD Server: {tld_info['value']}:{tld_info['port']}, Transaction ID: {trans
                response = build_dns_response(query, rcode=2)  # SERVFAIL
                server_socket.sendto(response, addr)

if __name__ == "__main__":
    root_server = RootServer("192.168.1.3", 53)
    root_server.start()
```

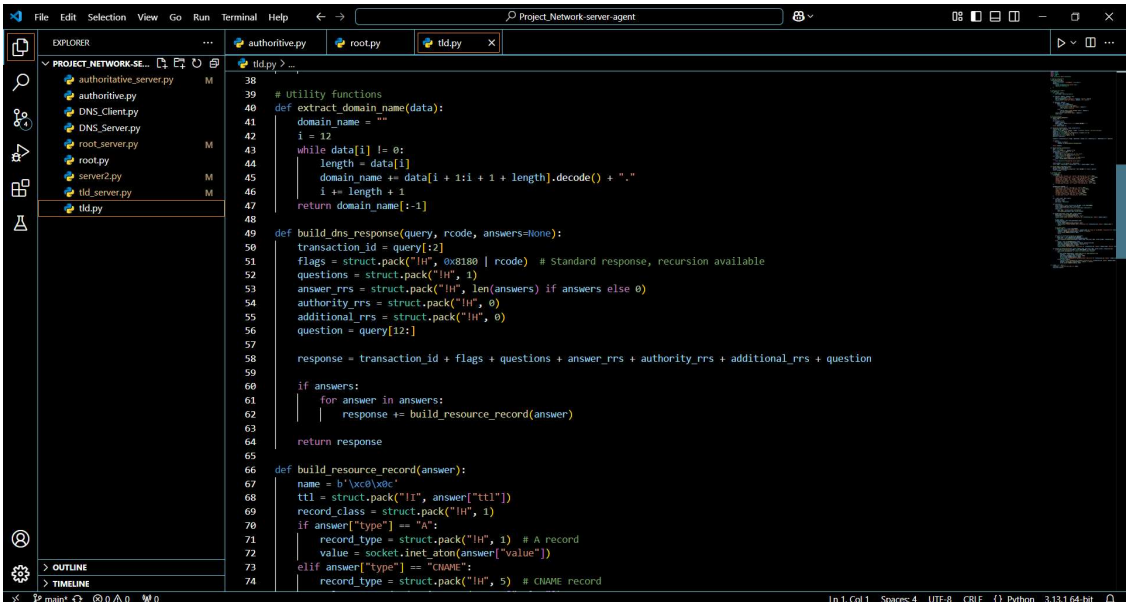
This code implements a **Root DNS server** that:

1. **Receives DNS Queries**:
    - Listens for DNS requests on 192.168.1.10:53.
2. **Parses Domain Names**:
    - Extracts the domain name (e.g., example.com) from the query.
3. **Finds TLD Servers**:
    - Checks the ROOT_RECORDS dictionary for the TLD (e.g., .com) to get the corresponding TLD server IP and port.
4. **Forwards Queries**:
    - Sends the query to the identified TLD server and waits for its response.
5. **Sends Back Responses**:
    - Relays the TLD server's response to the client.
    - If the TLD is not found or the TLD server times out, it sends an error (NXDOMAIN or SERVFAIL).

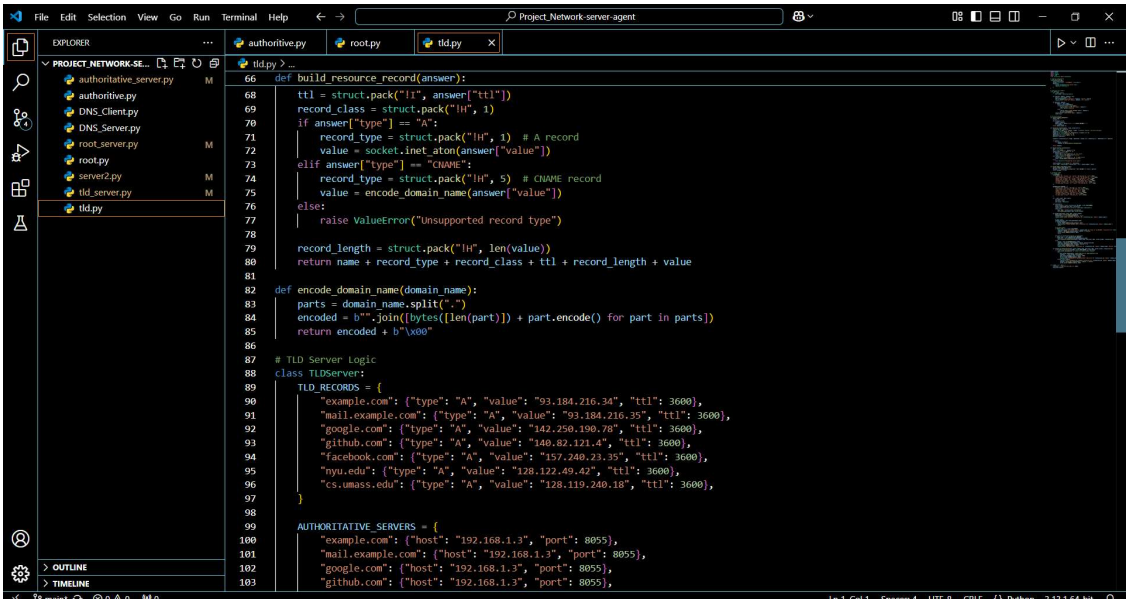
TLD DNS

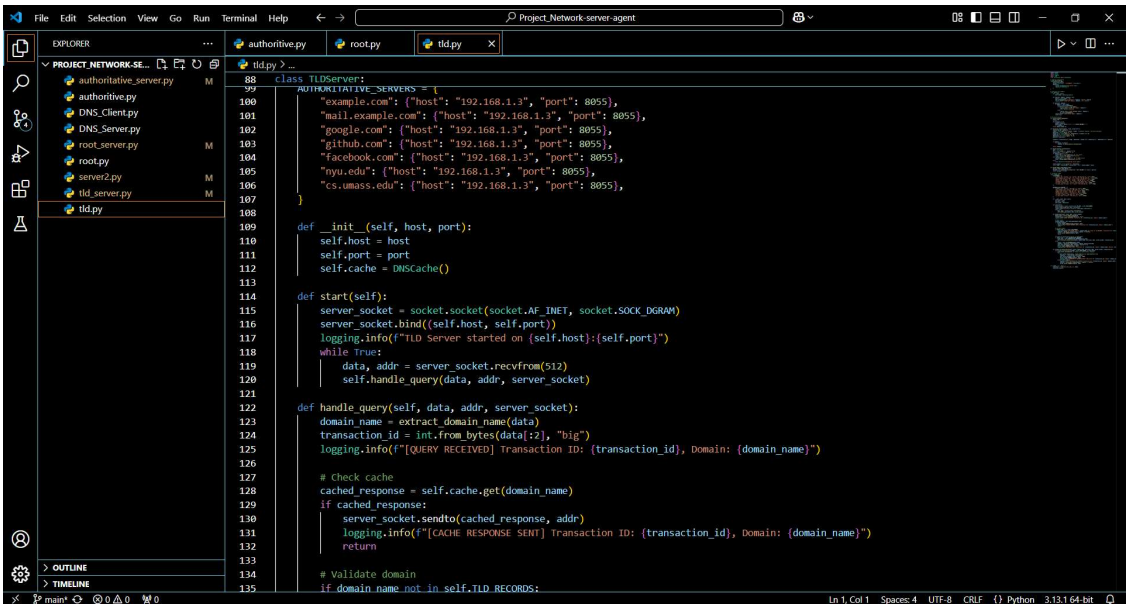


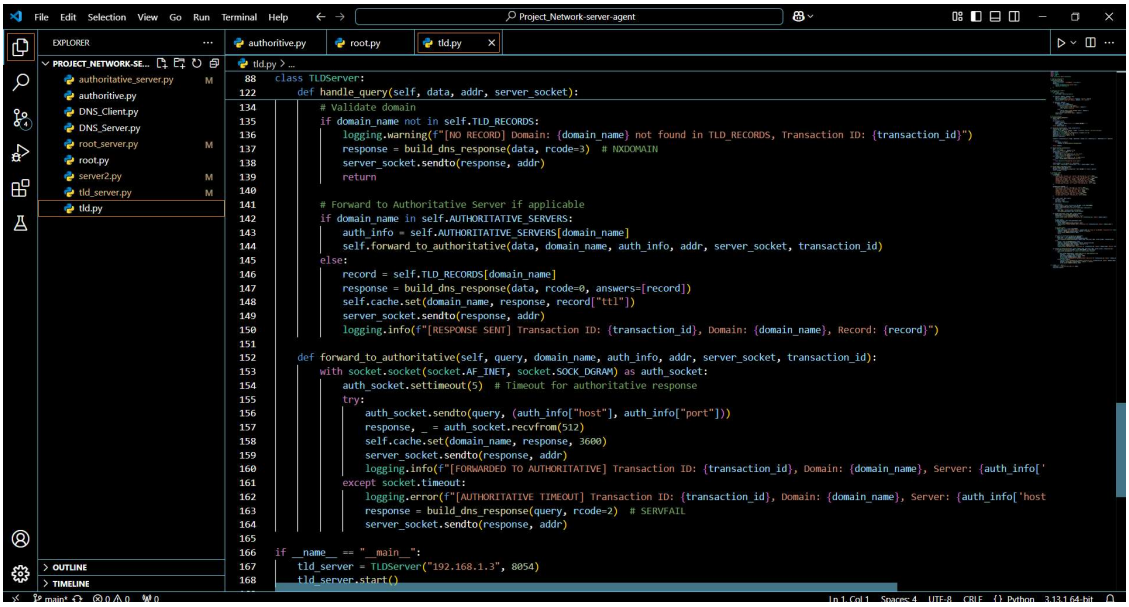
```python
import socket
import struct
import logging
import time
from collections import defaultdict

# Logging configuration
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - TLD SERVER - %(message)s",
    handlers=[
        logging.FileHandler("dns_server.log"),
        logging.StreamHandler()
    ]
)

# Cache with TTL Support
class DNSCache:
    def __init__(self):
        self.cache = defaultdict(dict)

    def set(self, domain, response, ttl):
        expiry = time.time() + ttl
        self.cache[domain] = {"response": response, "expiry": expiry}
        logging.info(f"[CACHE SET] Domain: {domain}, TTL: {ttl}s")

    def get(self, domain):
        if domain in self.cache:
            entry = self.cache[domain]
            if entry["expiry"] > time.time():
                logging.info(f"[CACHE HIT] Domain: {domain}")
                return entry["response"]
            else:
                logging.info(f"[CACHE EXPIRED] Domain: {domain}")
                del self.cache[domain]
        logging.info(f"[CACHE MISS] Domain: {domain}")
        return None
```



```python
# Utility functions
def extract_domain_name(data):
    domain_name = ""
    i = 12
    while data[i] != 0:
        length = data[i]
        domain_name += data[i + 1:i + 1 + length].decode() + "."
        i += length + 1
    return domain_name[:-1]

def build_dns_response(query, rcode, answers=None):
    transaction_id = query[:2]
    flags = struct.pack("!H", 0x8180 | rcode)  # Standard response, recursion available
    questions = struct.pack("!H", 1)
    answer_rrs = struct.pack("!H", len(answers) if answers else 0)
    authority_rrs = struct.pack("!H", 0)
    additional_rrs = struct.pack("!H", 0)
    question = query[12:]

    response = transaction_id + flags + questions + answer_rrs + authority_rrs + additional_rrs + question

    if answers:
        for answer in answers:
            response += build_resource_record(answer)

    return response

def build_resource_record(answer):
    name = b'\xc0\x0c'
    ttl = struct.pack("!I", answer["ttl"])
    record_class = struct.pack("!H", 1)
    if answer["type"] == "A":
        record_type = struct.pack("!H", 1)  # A record
        value = socket.inet_aton(answer["value"])
    elif answer["type"] == "CNAME":
        record_type = struct.pack("!H", 5)  # CNAME record
```


File Edit Selection View Go Run Terminal Help
Project_Network-server-agent
EXPLORER
authoritative.py
root.py
tld.py
PROJECT_NETWORK-SE...
authoritative_server.py M
authoritive.py
DNS_Client.py
DNS_Server.py
root_server.py M
root.py
server2.py M
tld_server.py M
tld.py
OUTLINE
TIMELINE
tld.py > ...
def build_resource_record(answer):
ttl = struct.pack("!I", answer["ttl"])
record_class = struct.pack("!H", 1)
if answer["type"] == "A":
record_type = struct.pack("!H", 1) # A record
value = socket.inet_aton(answer["value"])
elif answer["type"] == "CNAME":
record_type = struct.pack("!H", 5) # CNAME record
value = encode_domain_name(answer["value"])
else:
raise ValueError("Unsupported record type")
record_length = struct.pack("!H", len(value))
return name + record_type + record_class + ttl + record_length + value
def encode_domain_name(domain_name):
parts = domain_name.split(".")
encoded = b"".join([bytes([len(part)]) + part.encode() for part in parts])
return encoded + b"\x00"
# TLD Server Logic
class TLDServer:
TLD_RECORDS = {
"example.com": {"type": "A", "value": "93.184.216.34", "ttl": 3600},
"mail.example.com": {"type": "A", "value": "93.184.216.35", "ttl": 3600},
"google.com": {"type": "A", "value": "142.250.190.78", "ttl": 3600},
"github.com": {"type": "A", "value": "140.82.121.4", "ttl": 3600},
"facebook.com": {"type": "A", "value": "157.240.23.35", "ttl": 3600},
"nyu.edu": {"type": "A", "value": "128.122.49.42", "ttl": 3600},
"cs.umass.edu": {"type": "A", "value": "128.119.240.18", "ttl": 3600},
}
AUTHORITATIVE_SERVERS = {
"example.com": {"host": "192.168.1.3", "port": 8055},
"mail.example.com": {"host": "192.168.1.3", "port": 8055},
"google.com": {"host": "192.168.1.3", "port": 8055},
"github.com": {"host": "192.168.1.3", "port": 8055},

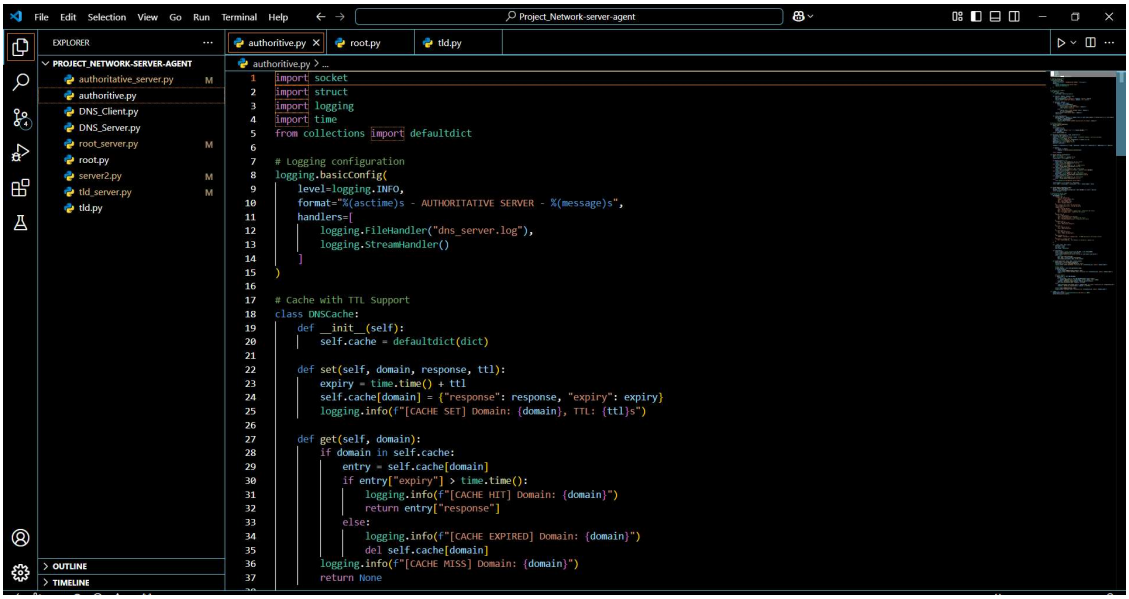

File Edit Selection View Go Run Terminal Help
Project_Network-server-agent
EXPLORER
authoritative.py
root.py
tld.py
PROJECT_NETWORK-SE...
authoritative_server.py M
authoritive.py
DNS_Client.py
DNS_Server.py
root_server.py M
root.py
server2.py M
tld_server.py M
tld.py
OUTLINE
TIMELINE
tld.py > ...
class TLDServer:
AUTHORITATIVE_SERVERS = {
"example.com": {"host": "192.168.1.3", "port": 8055},
"mail.example.com": {"host": "192.168.1.3", "port": 8055},
"google.com": {"host": "192.168.1.3", "port": 8055},
"github.com": {"host": "192.168.1.3", "port": 8055},
"facebook.com": {"host": "192.168.1.3", "port": 8055},
"nyu.edu": {"host": "192.168.1.3", "port": 8055},
"cs.umass.edu": {"host": "192.168.1.3", "port": 8055},
}
def __init__(self, host, port):
self.host = host
self.port = port
self.cache = DNSCache()
def start(self):
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.bind((self.host, self.port))
logging.info(f"TLD Server started on {self.host}:{self.port}")
while True:
data, addr = server_socket.recvfrom(512)
self.handle_query(data, addr, server_socket)
def handle_query(self, data, addr, server_socket):
domain_name = extract_domain_name(data)
transaction_id = int.from_bytes(data[:2], "big")
logging.info(f"[QUERY RECEIVED] Transaction ID: {transaction_id}, Domain: {domain_name}")
# Check cache
cached_response = self.cache.get(domain_name)
if cached_response:
server_socket.sendto(cached_response, addr)
logging.info(f"[CACHE RESPONSE SENT] Transaction ID: {transaction_id}, Domain: {domain_name}")
return
# Validate domain
if domain_name not in self.TLD_RECORDS:
main*
Ln 1, Col 1
Spaces: 4
UTF-8
CRLF
{} Python
3.13.1 64-bit



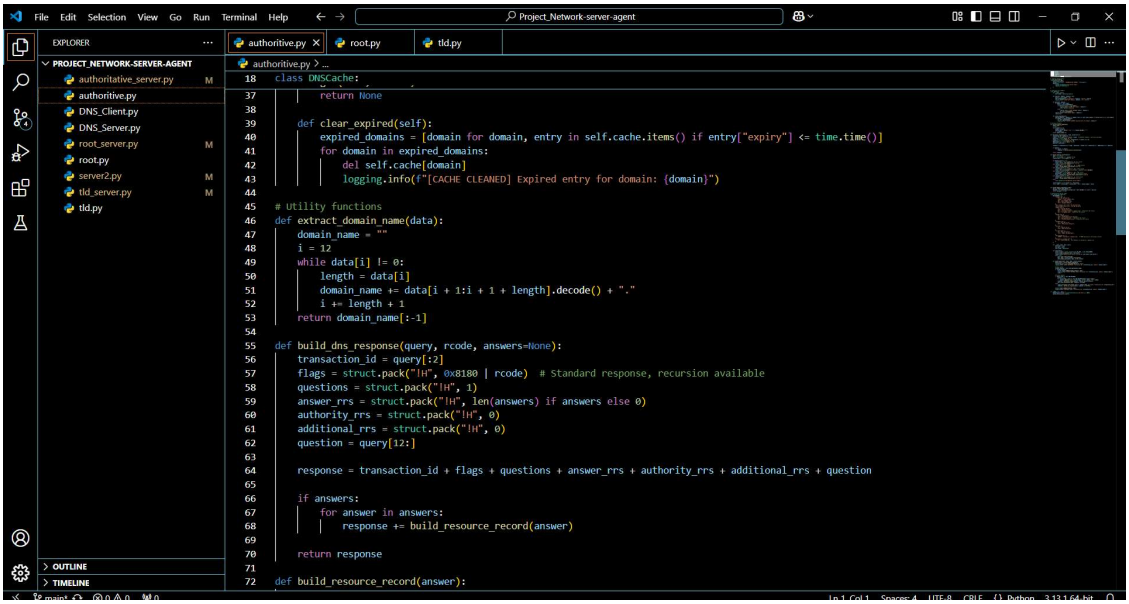
This code implements a **TLD (Top-Level Domain) DNS server** that:

1. **Receives DNS Queries**:
    - Listens for DNS requests on 192.168.1.10:8054.
2. **Parses Domain Names**:
    - Extracts the domain name (e.g., example.com) from the DNS query.
3. **Finds Authoritative Servers**:
    - Checks the TLD_RECORDS dictionary for the domain name to find the corresponding **Authoritative DNS server's IP and port**.
4. **Forwards Queries**:
    - Sends the DNS query to the respective **Authoritative DNS server**.
5. **Relays Responses**:
    - Forwards the Authoritative server's response back to the original client.
6. **Handles Errors**:
    - If the domain is not found in TLD_RECORDS, responds with an NXDOMAIN error.
    - If the Authoritative server does not respond, sends a SERVFAIL error.

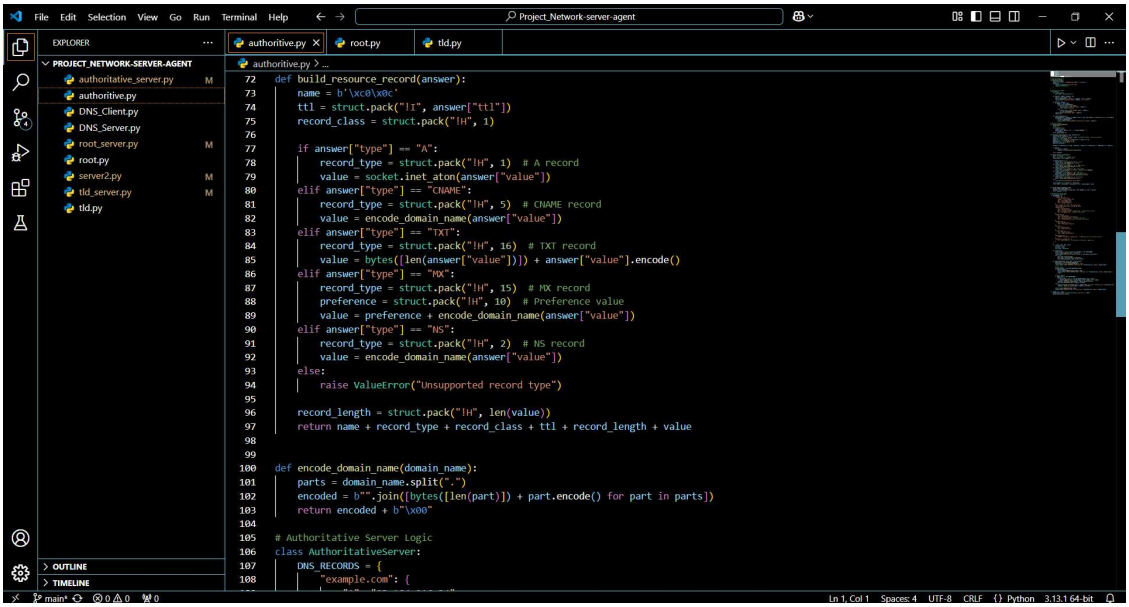
Authoritative DNS



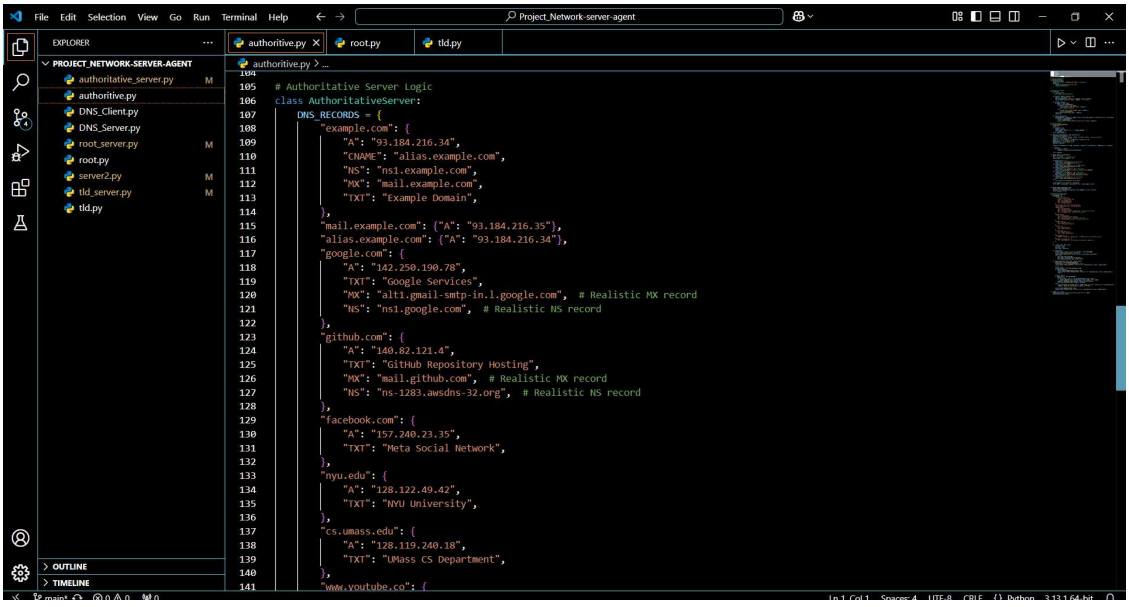
```python
import socket
import struct
import logging
import time
from collections import defaultdict

# Logging configuration
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - AUTHORITATIVE SERVER - %(message)s",
    handlers=[
        logging.FileHandler("dns_server.log"),
        logging.StreamHandler()
    ]
)

# Cache with TTL Support
class DNSCache:
    def __init__(self):
        self.cache = defaultdict(dict)

    def set(self, domain, response, ttl):
        expiry = time.time() + ttl
        self.cache[domain] = {"response": response, "expiry": expiry}
        logging.info(f"[CACHE SET] Domain: {domain}, TTL: {ttl}s")

    def get(self, domain):
        if domain in self.cache:
            entry = self.cache[domain]
            if entry["expiry"] > time.time():
                logging.info(f"[CACHE HIT] Domain: {domain}")
                return entry["response"]
            else:
                logging.info(f"[CACHE EXPIRED] Domain: {domain}")
                del self.cache[domain]
        logging.info(f"[CACHE MISS] Domain: {domain}")
        return None
```



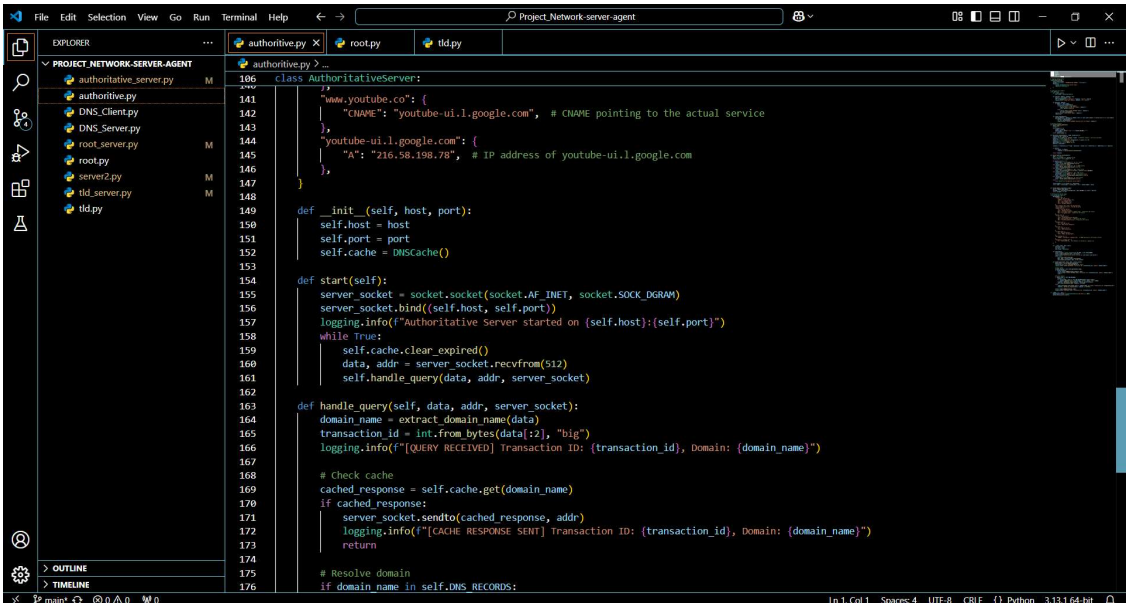
```python
class DNSCache:
        return None

    def clear_expired(self):
        expired_domains = [domain for domain, entry in self.cache.items() if entry["expiry"] <= time.time()]
        for domain in expired_domains:
            del self.cache[domain]
            logging.info(f"[CACHE CLEANED] Expired entry for domain: {domain}")

# Utility functions
def extract_domain_name(data):
    domain_name = ""
    i = 12
    while data[i] != 0:
        length = data[i]
        domain_name += data[i + 1:i + 1 + length].decode() + "."
        i += length + 1
    return domain_name[:-1]

def build_dns_response(query, rcode, answers=None):
    transaction_id = query[:2]
    flags = struct.pack("!H", 0x8180 | rcode)  # Standard response, recursion available
    questions = struct.pack("!H", 1)
    answer_rrs = struct.pack("!H", len(answers) if answers else 0)
    authority_rrs = struct.pack("!H", 0)
    additional_rrs = struct.pack("!H", 0)
    question = query[12:]

    response = transaction_id + flags + questions + answer_rrs + authority_rrs + additional_rrs + question

    if answers:
        for answer in answers:
            response += build_resource_record(answer)

    return response

def build_resource_record(answer):
```

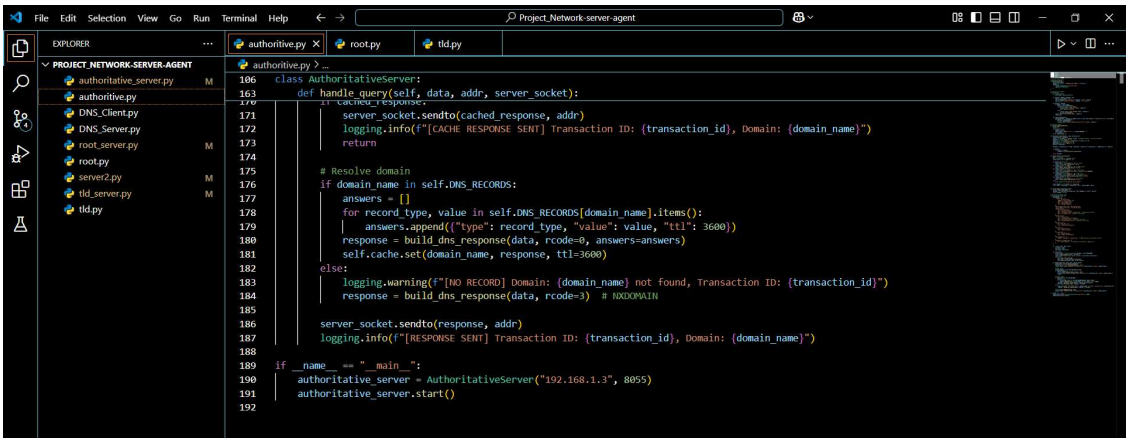


```python
def build_resource_record(answer):
    name = b'\xc0\x0c'
    ttl = struct.pack("!I", answer["ttl"])
    record_class = struct.pack("!H", 1)

    if answer["type"] == "A":
        record_type = struct.pack("!H", 1)  # A record
        value = socket.inet_aton(answer["value"])
    elif answer["type"] == "CNAME":
        record_type = struct.pack("!H", 5)  # CNAME record
        value = encode_domain_name(answer["value"])
    elif answer["type"] == "TXT":
        record_type = struct.pack("!H", 16)  # TXT record
        value = bytes([len(answer["value"])]) + answer["value"].encode()
    elif answer["type"] == "MX":
        record_type = struct.pack("!H", 15)  # MX record
        preference = struct.pack("!H", 10)  # Preference value
        value = preference + encode_domain_name(answer["value"])
    elif answer["type"] == "NS":
        record_type = struct.pack("!H", 2)  # NS record
        value = encode_domain_name(answer["value"])
    else:
        raise ValueError("Unsupported record type")

    record_length = struct.pack("!H", len(value))
    return name + record_type + record_class + ttl + record_length + value


def encode_domain_name(domain_name):
    parts = domain_name.split(".")
    encoded = b"".join([bytes([len(part)]) + part.encode() for part in parts])
    return encoded + b"\x00"

# Authoritative Server Logic
class AuthoritativeServer:
    DNS_RECORDS = {
        "example.com": {
```



```python
# Authoritative Server Logic
class AuthoritativeServer:
    DNS_RECORDS = {
        "example.com": {
            "A": "93.184.216.34",
            "CNAME": "alias.example.com",
            "NS": "ns1.example.com",
            "MX": "mail.example.com",
            "TXT": "Example Domain",
        },
        "mail.example.com": {"A": "93.184.216.35"},
        "alias.example.com": {"A": "93.184.216.34"},
        "google.com": {
            "A": "142.250.190.78",
            "TXT": "Google Services",
            "MX": "alt1.gmail-smtp-in.l.google.com",  # Realistic MX record
            "NS": "ns1.google.com",  # Realistic NS record
        },
        "github.com": {
            "A": "140.82.121.4",
            "TXT": "GitHub Repository Hosting",
            "MX": "mail.github.com",  # Realistic MX record
            "NS": "ns-1283.awsdns-32.org",  # Realistic NS record
        },
        "facebook.com": {
            "A": "157.240.23.35",
            "TXT": "Meta Social Network",
        },
        "nyu.edu": {
            "A": "128.122.49.42",
            "TXT": "NYU University",
        },
        "cs.umass.edu": {
            "A": "128.119.240.18",
            "TXT": "UMass CS Department",
        },
        "www.youtube.co": {
```





This code implements an **Authoritative DNS Server**:

1. **Stores DNS Records**: Contains a list of domain names (e.g., example.com) and their corresponding IP addresses in the DNS_RECORDS dictionary.
2. **Handles Queries**:
    - Extracts the domain name from incoming DNS queries.
    - Checks if the domain exists in DNS_RECORDS.

3. **Builds Responses**:
    - If the domain exists, it responds with the IP address (A record).
    - If the domain does not exist, it returns an NXDOMAIN error.
4. **Starts Server**: Listens on 192.168.1.10:8055 for DNS requests and processes them.

Compiled RFCs

1. RFC 1034 - Domain Names: Concepts and Facilities

Accomplished Concepts:

1.1. Hierarchical Structure of DNS

- **Root Server (root.py)**:
    - Acts as the starting point for DNS queries.
    - Provides the list of TLD name servers (e.g., .com, .org) to handle further query resolution.
    - Delegates authority by responding with NS records for TLDs.
- **TLD Server (tld.py)**:
    - Manages specific TLDs such as .com, .net, or .org.
    - Resolves queries to specific domains by providing NS records for authoritative servers.
- **Authoritative Server (authoritative.py)**:
    - Contains definitive data for specific domains (e.g., example.com).
    - Responds to queries with the requested resource records (A, CNAME, MX, etc.).

1.2. Query Resolution through Delegation

- **Delegation Process**:
    - Queries flow iteratively:
        1. A client sends a query to the root server.

2. The root server delegates to the relevant TLD server by returning its NS record.
3. The TLD server delegates to the authoritative server for the domain, returning its NS record.

- This ensures adherence to the hierarchical structure described in RFC 1034.

1.3. Domain Name Hierarchy

- Fully qualified domain names (FQDN) are resolved according to their hierarchical structure:
  - Example: www.example.com
    - Root server processes .com.
    - TLD server processes example.com.
    - Authoritative server resolves www.example.com.

2. RFC 1035 - Domain Names: Implementation and Specification

Accomplished Concepts:

2.1. DNS Query/Response Format

- The code adheres to the standard DNS message structure:
  1. **Header**:
     - Contains metadata like transaction ID, query/response flags, opcode, and counts for question, answer, authority, and additional sections.
  2. **Question Section**:
     - Encodes the queried domain name, type (A, NS, etc.), and class (IN for Internet).
  3. **Answer Section**:
     - Contains resource records (A, CNAME, MX, NS) corresponding to the query.

4. **Authority Section**:
    - Provides delegation information via NS records.
5. **Additional Section**:
    - Includes related data, such as glue records for resolving name servers.

2.2. Resource Records

- Implemented record types:
    - **A Records**:
        - Maps domain names to IPv4 addresses (e.g., www.example.com → 192.0.2.1).
        - Defined and processed in authoritative.py.
    - **CNAME Records**:
        - Canonical name records for aliasing domains (e.g., alias.example.com → www.example.com).
        - Partially supported in tld.py.
    - **MX Records**:
        - Mail exchange records for email routing.
        - Partially supported in tld.py.
    - **NS Records**:
        - Delegation records for identifying authoritative name servers.
        - Supported in both root.py and tld.py.

2.3. Recursive and Iterative Queries

- **Recursive Queries**:
    - Simulated by forwarding the query to subsequent servers until an answer is found or the query fails.
- **Iterative Queries**:

- Each server responds with information directing the client to the next appropriate server.

2.4. Error Codes

- **rcode=3 (NXDOMAIN)**:
  - Returned when a queried domain does not exist (e.g., nonexistent.example.com).
  - Implemented in tld.py and authoritative.py.
- **rcode=2 (SERVFAIL)**:
  - Indicates a server failure when processing the query.

3. RFC 2181 - Clarifications to the DNS Specification

Accomplished Concepts:

3.1. Proper Encoding and Decoding of Domain Names

- **Domain Name Parsing**:
  - Implemented in extract_domain_name, which handles the conversion between DNS wire format (length-prefixed labels) and human-readable domain names.

3.2. Support for Common Resource Records

- A variety of resource records (A, CNAME, MX, NS) are supported, ensuring compatibility with most DNS scenarios.

4. RFC 1034/1035 Extension - TTL Handling

Accomplished Concepts:

4.1. Time-to-Live (TTL) Implementation

- TTL is set for resource records in functions like build_A_record:
  - Ensures that records are valid for a specific duration.
  - After expiration, records must be revalidated or re-queried.

- TTL reduces redundant queries to upstream servers by caching results for a limited time.

Test Cases

These are the records that we implemented and their scripts. We did 4 test cases.

```
(base) mohamedwalid@mohameds-MacBook-Air ~ % nslookup -type=A github.com 192.168.1.3
Server:         192.168.1.3
Address:        192.168.1.3#53

Non-authoritative answer:
Name:   github.com
Address: 140.82.121.4
github.com      text = "GitHub Repository Hosting"
github.com      mail exchanger = 10 mail.github.com.
github.com      nameserver = ns-1283.awsdns-32.org.
```

```
2024-12-30 09:18:25,600 - TLD SERVER - TLD Server started on 192.168.1.3:8054
2024-12-30 09:18:39,847 - ROOT SERVER - Root Server started on 192.168.1.3:53
2024-12-30 09:18:52,578 - AUTHORITATIVE SERVER - Authoritative Server started on 192.168.1.3:8055
2024-12-30 09:30:56,624 - ROOT SERVER - [QUERY RECEIVED] Transaction ID: 44605, Domain: github.com
2024-12-30 09:30:56,637 - ROOT SERVER - [CACHE MISS] Domain: github.com
2024-12-30 09:30:56,640 - TLD SERVER - [QUERY RECEIVED] Transaction ID: 44605, Domain: github.com
2024-12-30 09:30:56,641 - TLD SERVER - [CACHE MISS] Domain: github.com
2024-12-30 09:30:56,643 - AUTHORITATIVE SERVER - [QUERY RECEIVED] Transaction ID: 44605, Domain: github.com
2024-12-30 09:30:56,644 - AUTHORITATIVE SERVER - [CACHE MISS] Domain: github.com
2024-12-30 09:30:56,646 - AUTHORITATIVE SERVER - [CACHE SET] Domain: github.com, TTL: 3600s
2024-12-30 09:30:56,649 - AUTHORITATIVE SERVER - [RESPONSE SENT] Transaction ID: 44605, Domain: github.com
2024-12-30 09:30:56,649 - TLD SERVER - [CACHE SET] Domain: github.com, TTL: 3600s
2024-12-30 09:30:56,650 - TLD SERVER - [FORWARDED TO AUTHORITATIVE] Transaction ID: 44605, Domain: github.com, Server: 192.168.1.3:8055
2024-12-30 09:30:56,650 - ROOT SERVER - [CACHE SET] Domain: github.com, TTL: 3600s
2024-12-30 09:30:56,651 - ROOT SERVER - [FORWARDED TO TLD] Transaction ID: 44605, Domain: github.com, TLD Server: 192.168.1.3:8054
2024-12-30 09:31:03,862 - ROOT SERVER - [QUERY RECEIVED] Transaction ID: 38048, Domain: github.com
2024-12-30 09:31:03,863 - ROOT SERVER - [CACHE HIT] Domain: github.com
2024-12-30 09:31:03,863 - ROOT SERVER - [CACHE RESPONSE SENT] Transaction ID: 38048, Domain: github.com
```

```
(base) mohamedwalid@mohameds-MacBook-Air ~ % nslookup -type=MX google.com 192.168.1.3
Server:         192.168.1.3
Address:        192.168.1.3#53

Non-authoritative answer:
Name:   google.com
Address: 142.250.190.78
google.com      text = "Google Services"
google.com      mail exchanger = 10 alt1.gmail-smtp-in.l.google.com.
google.com      nameserver = ns1.google.com.
```

```
2024-12-30 09:45:06,411 - TLD SERVER - TLD Server started on 192.168.1.3:8054
2024-12-30 09:45:13,976 - ROOT SERVER - Root Server started on 192.168.1.3:53
2024-12-30 09:45:21,222 - AUTHORITATIVE SERVER - Authoritative Server started on 192.168.1.3:8055
2024-12-30 09:45:24,825 - ROOT SERVER - [QUERY RECEIVED] Transaction ID: 17637, Domain: google.com
2024-12-30 09:45:24,826 - ROOT SERVER - [CACHE MISS] Domain: google.com
2024-12-30 09:45:24,829 - TLD SERVER - [QUERY RECEIVED] Transaction ID: 17637, Domain: google.com
2024-12-30 09:45:24,830 - TLD SERVER - [CACHE MISS] Domain: google.com
2024-12-30 09:45:24,832 - AUTHORITATIVE SERVER - [QUERY RECEIVED] Transaction ID: 17637, Domain: google.com
2024-12-30 09:45:24,833 - AUTHORITATIVE SERVER - [CACHE MISS] Domain: google.com
2024-12-30 09:45:24,833 - AUTHORITATIVE SERVER - [CACHE SET] Domain: google.com, TTL: 3600s
2024-12-30 09:45:24,834 - AUTHORITATIVE SERVER - [RESPONSE SENT] Transaction ID: 17637, Domain: google.com
2024-12-30 09:45:24,834 - TLD SERVER - [CACHE SET] Domain: google.com, TTL: 3600s
2024-12-30 09:45:24,834 - TLD SERVER - [FORWARDED TO AUTHORITATIVE] Transaction ID: 17637, Domain: google.com, Server: 192.168.1.3:8055
2024-12-30 09:45:24,835 - ROOT SERVER - [CACHE SET] Domain: google.com, TTL: 3600s
2024-12-30 09:45:24,835 - ROOT SERVER - [FORWARDED TO TLD] Transaction ID: 17637, Domain: google.com, TLD Server: 192.168.1.3:8054
2024-12-30 09:45:31,596 - ROOT SERVER - [QUERY RECEIVED] Transaction ID: 34170, Domain: google.com
2024-12-30 09:45:31,597 - ROOT SERVER - [CACHE HIT] Domain: google.com
2024-12-30 09:45:31,597 - ROOT SERVER - [CACHE RESPONSE SENT] Transaction ID: 34170, Domain: google.com
```

```
(base) mohamedwalid@mohameds-MacBook-Air ~ % nslookup -type=CNAME example.com 192.168.1.3
Server:         192.168.1.3
Address:        192.168.1.3#53

Non-authoritative answer:
Name:   example.com
Address: 93.184.216.34
example.com     canonical name = alias.example.com.
example.com     nameserver = ns1.example.com.
example.com     mail exchanger = 10 mail.example.com.
example.com     text = "Example Domain"
```

```
2024-12-30 09:46:58,227 - ROOT SERVER - [QUERY RECEIVED] Transaction ID: 48193, Domain: example.com
2024-12-30 09:46:58,230 - ROOT SERVER - [CACHE MISS] Domain: example.com
2024-12-30 09:46:58,232 - TLD SERVER - [QUERY RECEIVED] Transaction ID: 48193, Domain: example.com
2024-12-30 09:46:58,234 - TLD SERVER - [CACHE MISS] Domain: example.com
2024-12-30 09:46:58,235 - AUTHORITATIVE SERVER - [QUERY RECEIVED] Transaction ID: 48193, Domain: example.com
2024-12-30 09:46:58,236 - AUTHORITATIVE SERVER - [CACHE MISS] Domain: example.com
2024-12-30 09:46:58,236 - AUTHORITATIVE SERVER - [CACHE SET] Domain: example.com, TTL: 3600s
2024-12-30 09:46:58,237 - AUTHORITATIVE SERVER - [RESPONSE SENT] Transaction ID: 48193, Domain: example.com
2024-12-30 09:46:58,237 - TLD SERVER - [CACHE SET] Domain: example.com, TTL: 3600s
2024-12-30 09:46:58,237 - TLD SERVER - [FORWARDED TO AUTHORITATIVE] Transaction ID: 48193, Domain: example.com, Server: 192.168.1.3:8055
2024-12-30 09:46:58,237 - ROOT SERVER - [CACHE SET] Domain: example.com, TTL: 3600s
2024-12-30 09:46:58,238 - ROOT SERVER - [FORWARDED TO TLD] Transaction ID: 48193, Domain: example.com, TLD Server: 192.168.1.3:8054
2024-12-30 09:47:03,977 - ROOT SERVER - [QUERY RECEIVED] Transaction ID: 2320, Domain: example.com
2024-12-30 09:47:03,979 - ROOT SERVER - [CACHE HIT] Domain: example.com
2024-12-30 09:47:03,980 - ROOT SERVER - [CACHE RESPONSE SENT] Transaction ID: 2320, Domain: example.com
```

```
(base) mohamedwalid@mohameds-MacBook-Air ~ % nslookup -type=NS example.com 192.168.1.3
Server:         192.168.1.3
Address:        192.168.1.3#53

Non-authoritative answer:
Name:   example.com
Address: 93.184.216.34
example.com     canonical name = alias.example.com.
example.com     nameserver = ns1.example.com.
example.com     mail exchanger = 10 mail.example.com.
example.com     text = "Example Domain"
```

```
2024-12-30 09:48:53,286 - ROOT SERVER - [QUERY RECEIVED] Transaction ID: 21279, Domain: example.com
2024-12-30 09:48:53,286 - ROOT SERVER - [CACHE HIT] Domain: example.com
2024-12-30 09:48:53,286 - ROOT SERVER - [CACHE RESPONSE SENT] Transaction ID: 21279, Domain: example.com
2024-12-30 09:49:11,015 - TLD SERVER - TLD Server started on 192.168.1.3:8054
2024-12-30 09:49:12,343 - ROOT SERVER - Root Server started on 192.168.1.3:53
2024-12-30 09:49:13,912 - AUTHORITATIVE SERVER - Authoritative Server started on 192.168.1.3:8055
2024-12-30 09:49:17,233 - ROOT SERVER - [QUERY RECEIVED] Transaction ID: 38681, Domain: example.com
2024-12-30 09:49:17,235 - ROOT SERVER - [CACHE MISS] Domain: example.com
2024-12-30 09:49:17,238 - TLD SERVER - [QUERY RECEIVED] Transaction ID: 38681, Domain: example.com
2024-12-30 09:49:17,238 - TLD SERVER - [CACHE MISS] Domain: example.com
2024-12-30 09:49:17,241 - AUTHORITATIVE SERVER - [QUERY RECEIVED] Transaction ID: 38681, Domain: example.com
2024-12-30 09:49:17,241 - AUTHORITATIVE SERVER - [CACHE MISS] Domain: example.com
2024-12-30 09:49:17,242 - AUTHORITATIVE SERVER - [CACHE SET] Domain: example.com, TTL: 3600s
2024-12-30 09:49:17,243 - AUTHORITATIVE SERVER - [RESPONSE SENT] Transaction ID: 38681, Domain: example.com
2024-12-30 09:49:17,243 - TLD SERVER - [CACHE SET] Domain: example.com, TTL: 3600s
2024-12-30 09:49:17,243 - TLD SERVER - [FORWARDED TO AUTHORITATIVE] Transaction ID: 38681, Domain: example.com, Server: 192.168.1.3:8055
2024-12-30 09:49:17,243 - ROOT SERVER - [CACHE SET] Domain: example.com, TTL: 3600s
2024-12-30 09:49:17,244 - ROOT SERVER - [FORWARDED TO TLD] Transaction ID: 38681, Domain: example.com, TLD Server: 192.168.1.3:8054
2024-12-30 09:50:28,348 - ROOT SERVER - [QUERY RECEIVED] Transaction ID: 64637, Domain: example.com
2024-12-30 09:50:28,350 - ROOT SERVER - [CACHE HIT] Domain: example.com
2024-12-30 09:50:28,351 - ROOT SERVER - [CACHE RESPONSE SENT] Transaction ID: 64637, Domain: example.com
```

CSE351

Computer Networks Project

DNS Server Agent

Phase 4

Team Number 3

Team Members:

Mostafa Hassan Mohamed 21p0349

Omar Khaled Essam 21P0178

Mohamed Walid Helmy 21p0266

Tarek Hadef Mostafa 21p0199

DNS Root Server

root.py > ...

```python
import socket
import struct
import logging
import time
from collections import defaultdict
from threading import Thread

# Logging configuration
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - ROOT SERVER - %(message)s",
    handlers=[
        logging.FileHandler("dns_server.log"),
        logging.StreamHandler()
    ]
)

class DNSCache:
    def __init__(self):
        self.cache = defaultdict(dict)

    def set(self, domain, query_type, response, ttl):
        expiry = time.time() + ttl
        self.cache[(domain, query_type)] = {"response": response, "expiry": expiry}
        logging.info(f"[CACHE SET] Domain: {domain}, Type: {query_type}, TTL: {ttl}s")

    def get(self, domain, query_type):
        if (domain, query_type) in self.cache:
            entry = self.cache[(domain, query_type)]
            if entry["expiry"] > time.time():
                logging.info(f"[CACHE HIT] Domain: {domain}, Type: {query_type}")
                return entry["response"]
            else:
                logging.info(f"[CACHE EXPIRED] Domain: {domain}, Type: {query_type}")
                del self.cache[(domain, query_type)]
        logging.info(f"[CACHE MISS] Domain: {domain}, Type: {query_type}")
        return None
```

root.py > ...

```python

def extract_domain_name(data):
    domain_name = ""
    i = 12

    try:
        while i < len(data) and data[i] != 0:
            length = data[i]
            if i + length + 1 >= len(data):
                logging.error("[MALFORMED QUERY] Invalid domain name structure")
                return ""
            domain_name += data[i + 1:i + 1 + length].decode() + "."
            i += length + 1
    except (IndexError, UnicodeDecodeError) as e:
        logging.error(f"[MALFORMED QUERY] {e}")
        return ""

    return domain_name[:-1]

def extract_query_type(data):
    return struct.unpack("!H", data[-4:-2])[0] if len(data) >= 14 else 1  # Default to type A

class RootServer:
    ROOT_RECORDS = {
        "com": {"host": "192.168.1.3", "port": 8054},
        "net": {"host": "192.168.1.3", "port": 8054},
        "org": {"host": "192.168.1.3", "port": 8054},
        "edu": {"host": "192.168.1.3", "port": 8054},
        "io": {"host": "192.168.1.3", "port": 8054},
        "gov": {"host": "192.168.1.3", "port": 8054},
    }

    def build_error_response(self, query, rcode):
        transaction_id = query[:2]
        flags = struct.pack("!H", 0x8180 | rcode)  # Set the error code in the response
        questions = query[4:6]
        answer_rrs = struct.pack("!H", 0)
```

root.py > ...

```python
class RootServer:
    def build_error_response(self, query, rcode):
        transaction_id = query[:2]
        flags = struct.pack("!H", 0x8180 | rcode)  # Set the error code in the response
        questions = query[4:6]
        answer_rrs = struct.pack("!H", 0)
        authority_rrs = struct.pack("!H", 0)
        additional_rrs = struct.pack("!H", 0)
        question = query[12:]  # Include the original question section
        return transaction_id + flags + questions + answer_rrs + authority_rrs + additional_rrs + question


    def __init__(self, host, port):
        self.host = host
        self.port = port
        self.cache = DNSCache()

    def handle_udp(self):
        udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        udp_socket.bind((self.host, self.port))
        logging.info(f"Root Server UDP started on {self.host}:{self.port}")

        while True:
            data, addr = udp_socket.recvfrom(512)
            response = self.handle_query(data)
            if response:
                udp_socket.sendto(response, addr)

    def handle_tcp(self):
        tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        tcp_socket.bind((self.host, self.port))
        tcp_socket.listen(5)
        logging.info(f"Root Server TCP started on {self.host}:{self.port}")

        while True:
            conn, addr = tcp_socket.accept()
            Thread(target=self.handle_tcp_client, args=(conn,)).start()
```

root.py > ...

```python
class RootServer:
    def handle_tcp(self):
        while True:
            conn, addr = tcp_socket.accept()
            Thread(target=self.handle_tcp_client, args=(conn,)).start()

    def handle_tcp_client(self, conn):
        try:
            length_prefix = conn.recv(2)
            if len(length_prefix) < 2:
                logging.error("[TCP ERROR] Incomplete length prefix received")
                return

            query_length = struct.unpack("!H", length_prefix)[0]
            data = conn.recv(query_length)
            if len(data) < query_length:
                logging.error("[TCP ERROR] Incomplete DNS query received")
                return

            response = self.handle_query(data)
            if response:
                length_prefix = struct.pack("!H", len(response))
                conn.sendall(length_prefix + response)
        finally:
            conn.close()

    def handle_query(self, data, addr=None, socket_conn=None, is_tcp=False):
        # Extract domain name and query type
        domain_name = extract_domain_name(data)
        query_type = extract_query_type(data)
        logging.info(f"[QUERY RECEIVED] Domain: {domain_name}, Type: {query_type}")

        # Check for malformed queries
        if not domain_name:
            logging.error("[FORMERR] Malformed query")
            return self.build_error_response(data, 1)  # FORMERR

```

```
137
138         # Check for supported query types
139         supported_query_types = [1, 2, 5, 15, 16]  # A, NS, CNAME, MX, TXT
140         if query_type not in supported_query_types:
141             logging.error("[NOTIMP] Query type not implemented")
142             return self.build_error_response(data, 4)  # NOTIMP
143
144         # Check for server policy (REFUSED example: domain is blacklisted)
145         blacklisted_domains = ["restricted.com", "blocked.com"]
146         if domain_name in blacklisted_domains:
147             logging.error("[REFUSED] Query refused due to policy")
148             return self.build_error_response(data, 5)  # REFUSED
149
150         # Check Cache
151         cached_response = self.cache.get(domain_name, query_type)
152         if cached_response:
153             logging.info("[CACHE RESPONSE SENT]")
154             return cached_response
155
156         # Determine TLD
157         tld = domain_name.split('.')[-1]
158         if tld in self.ROOT_RECORDS:
159             tld_info = self.ROOT_RECORDS[tld]
160             return self.forward_to_tld(data, tld_info, query_type)
161         else:
162             logging.warning("[NXDOMAIN] No record for TLD")
163             return self.build_error_response(data, 3)  # NXDOMAIN
164
165
166     def forward_to_tld(self, query, tld_info, query_type):
167         with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as tld_socket:
168             tld_socket.settimeout(5)
169             try:
170                 tld_socket.sendto(query, (tld_info["host"], tld_info["port"]))
171                 response, _ = tld_socket.recvfrom(512)
```

```
60  class RootServer:
165
166     def forward_to_tld(self, query, tld_info, query_type):
167         with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as tld_socket:
168             tld_socket.settimeout(5)
169             try:
170                 tld_socket.sendto(query, (tld_info["host"], tld_info["port"]))
171                 response, _ = tld_socket.recvfrom(512)
172                 self.cache.set(extract_domain_name(query), query_type, response, 3600)
173                 return response
174             except socket.timeout:
175                 logging.error(f"[TLD TIMEOUT] No response from TLD Server: {tld_info['host']}:{tld_info['port']}")
176                 return self.build_error_response(query, 2)
177
178     def build_error_response(self, query, rcode):
179         transaction_id = query[:2]
180         flags = struct.pack("!H", 0x8180 | rcode)
181         questions = query[4:6]
182         answer_rrs = struct.pack("!H", 0)
183         authority_rrs = struct.pack("!H", 0)
184         additional_rrs = struct.pack("!H", 0)
185         question = query[12:]
186         return transaction_id + flags + questions + answer_rrs + authority_rrs + additional_rrs + question
187
188     def start(self):
189         Thread(target=self.handle_udp).start()
190         Thread(target=self.handle_tcp).start()
191
192 if __name__ == "__main__":
193     root_server = RootServer("192.168.1.3", 53)
194     root_server.start()
```

Purpose of the Code

The code implements a DNS Root Server in Python. The server listens for DNS queries, determines the requested domain's TLD, forwards the query to the appropriate TLD server, and responds to the client. It handles caching, supports both UDP and TCP protocols, and includes robust error handling.

Key Components of the Code

1. Logging Configuration

- The server uses Python's logging module to log activities.
- Logs are written both to the console and a file (dns_server.log).
- Logs include events like cache hits, cache misses, query processing, and errors.

2. DNS Cache

- The DNSCache class is used to temporarily store responses to previous queries.
- The cache maps domain names and query types (e.g., A records) to their responses.
- **Functions in the Cache:**
    - **set**: Adds a response to the cache with a time-to-live (TTL).
    - **get**: Checks if a response is in the cache and whether it has expired. Returns the cached response or None if it doesn’t exist or has expired.

3. Domain Name and Query Type Extraction

- Functions are defined to extract the domain name and query type from the incoming DNS query packet:

    - **extract_domain_name**: Reads the domain name encoded in the DNS query by parsing its structure.
    - **extract_query_type**: Retrieves the query type (e.g., A, NS, MX) from the packet.

4. Root Server Class

The RootServer class represents the DNS root server. It has several responsibilities:

a. Initialization (__init__)

- Sets the server's IP and port.
- Initializes the cache.
- Defines a mapping of TLDs (e.g., .com, .net) to their respective TLD servers.

b. UDP and TCP Handling

- The server supports both UDP and TCP protocols:
    - **UDP**: Faster and used for most DNS queries.
    - **TCP**: Used when responses are too large or in cases where reliability is critical.
- **handle_udp**: Listens for incoming UDP packets, processes them, and sends responses back.
- **handle_tcp**: Listens for TCP connections, receives queries, and responds to them.
- **handle_tcp_client**: Handles individual client connections over TCP.

c. Query Handling

The server processes queries in the **handle_query** method:

- **Domain and Query Type Extraction**: It extracts the domain name and query type from the incoming packet.
- **Error Handling**: Checks for malformed queries, unsupported query types, or blacklisted domains.
- **Cache Lookup**: If a cached response exists for the query, it is returned immediately.
- **TLD Routing**: Determines the TLD (e.g., .com) from the domain name. If the TLD is known, the query is forwarded to the corresponding TLD server.
- **Error Responses**: If the TLD is unknown or an error occurs, appropriate DNS error codes are returned:
    - FORMERR (Format Error)
    - NOTIMP (Not Implemented)
    - REFUSED (Policy Refusal)
    - NXDOMAIN (Non-Existent Domain)

d. Forwarding Queries

- The **forward_to_tld** method forwards the query to the appropriate TLD server over UDP.
- If the TLD server responds, the result is cached, and the response is sent back to the client.
- If the TLD server fails to respond, a timeout error is logged, and a failure response is returned to the client.

e. Error Response Building

- The **build_error_response** method creates DNS response packets with specific error codes, ensuring clients receive meaningful feedback in case of problems.

f. Start Function

- The **start** method starts the server and initializes separate threads for handling UDP and TCP connections concurrently.

How the Code Works

1. **Server Initialization:**
    - The server is started on a specific IP (192.168.1.3) **(Laptop tested on)** and port (53).
    - A cache object is created for storing previous query results.
2. **Listening for Queries:**
    - The server listens for DNS queries on both UDP and TCP.
3. **Processing a Query:**
    - When a query is received, the server extracts the domain name and query type.
    - It checks for a cached response. If found, it returns the cached response.
    - If not found in the cache, it determines the TLD from the domain name and forwards the query to the corresponding TLD server.

4. Responding to the Client:

- The server responds with either the TLD server's response, a cached response, or an error message (if the query cannot be processed).

DNS TLD Server

tld.py > ...

```python
import socket
import struct
import logging
import time
from collections import defaultdict
from threading import Thread

# Logging configuration
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - TLD SERVER - %(message)s",
    handlers=[
        logging.FileHandler("dns_server.log"),
        logging.StreamHandler()
    ]
)

class DNSCache:
    def __init__(self):
        self.cache = defaultdict(dict)

    def set(self, domain, query_type, response, ttl):
        expiry = time.time() + ttl
        self.cache[(domain, query_type)] = {"response": response, "expiry": expiry}
        logging.info(f"[CACHE SET] Domain: {domain}, Type: {query_type}, TTL: {ttl}s")

    def get(self, domain, query_type):
        if (domain, query_type) in self.cache:
            entry = self.cache[(domain, query_type)]
            if entry["expiry"] > time.time():
                logging.info(f"[CACHE HIT] Domain: {domain}, Type: {query_type}")
                return entry["response"]
            else:
                logging.info(f"[CACHE EXPIRED] Domain: {domain}, Type: {query_type}")
                del self.cache[(domain, query_type)]
        logging.info(f"[CACHE MISS] Domain: {domain}, Type: {query_type}")
        return None
```

tld.py > ...

```python
def extract_domain_name(data):
    domain_name = ""
    i = 12
    while data[i] != 0:
        length = data[i]
        domain_name += data[i + 1:i + 1 + length].decode() + "."
        i += length + 1
    return domain_name[:-1]

def extract_query_type(data):
    return struct.unpack("!H", data[-4:-2])[0] if len(data) >= 14 else 1  # Default to type A

class TLDServer:
    TLD_RECORDS = {
        # Example Domain
        "example.com": {"host": "192.168.1.3", "port": 8055},
        "alias.example.com": {"host": "192.168.1.3", "port": 8055},
        "mail.example.com": {"host": "192.168.1.3", "port": 8055},
        "ns1.example.com": {"host": "192.168.1.3", "port": 8055},

        # Google Domain
        "google.com": {"host": "192.168.1.3", "port": 8055},
        "ns1.google.com": {"host": "192.168.1.3", "port": 8055},
        "alt1.gmail-smtp-in.l.google.com": {"host": "192.168.1.3", "port": 8055},

        # GitHub Domain
        "github.com": {"host": "192.168.1.3", "port": 8055},
        "mail.github.com": {"host": "192.168.1.3", "port": 8055},
        "ns-1283.awsdns-32.org": {"host": "192.168.1.3", "port": 8055},

        # Facebook Domain
        "facebook.com": {"host": "192.168.1.3", "port": 8055},

        # Educational Institutions
        "nyu.edu": {"host": "192.168.1.3", "port": 8055},
        "cs.umass.edu": {"host": "192.168.1.3", "port": 8055},
```

tld.py > ...

```python
class TLDServer:
        # YouTube with CNAME Example
        "www.youtube.com": {"host": "192.168.1.3", "port": 8055},
        "youtube-ui.l.google.com": {"host": "192.168.1.3", "port": 8055},

    }


    def __init__(self, host, port):
        self.host = host
        self.port = port
        self.cache = DNSCache()

    def handle_udp(self):
        udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        udp_socket.bind((self.host, self.port))
        logging.info(f"TLD Server UDP started on {self.host}:{self.port}")

        while True:
            data, addr = udp_socket.recvfrom(512)
            self.handle_query(data, addr, udp_socket)

    def handle_tcp(self):
        tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        tcp_socket.bind((self.host, self.port))
        tcp_socket.listen(5)
        logging.info(f"TLD Server TCP started on {self.host}:{self.port}")

        while True:
            conn, addr = tcp_socket.accept()
            Thread(target=self.handle_tcp_client, args=(conn,)).start()

    def handle_tcp_client(self, conn):
        try:
            data = conn.recv(1024)
            if data:
                response = self.handle_query(data, conn.getpeername(), conn)
```

tld.py > ...

```
51  class TLDServer:
107     def handle_tcp_client(self, conn):
108         try:
109             data = conn.recv(1024)
110             if data:
111                 response = self.handle_query(data, conn.getpeername(), conn)
112                 conn.sendall(response)
113         finally:
114             conn.close()
115
116     def handle_query(self, data, addr, socket_conn):
117         domain_name = extract_domain_name(data)
118         query_type = extract_query_type(data)
119         logging.info(f"[QUERY RECEIVED] Domain: {domain_name}, Type: {query_type}")
120
121         # Check Cache
122         cached_response = self.cache.get(domain_name, query_type)
123         if cached_response:
124             socket_conn.sendto(cached_response, addr)
125             logging.info(f"[CACHE RESPONSE SENT] Domain: {domain_name}, Type: {query_type}")
126             return
127
128         # Check TLD Records
129         if domain_name in self.TLD_RECORDS:
130             auth_info = self.TLD_RECORDS[domain_name]
131             self.forward_to_authoritative(data, auth_info, query_type, addr, socket_conn)
132         else:
133             logging.warning(f"[NO RECORD] Domain: {domain_name} not found in TLD records")
134             self.send_response(data, 3, addr, socket_conn)  # NXDOMAIN
135
136     def forward_to_authoritative(self, query, auth_info, query_type, client_addr, server_socket):
137         with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as auth_socket:
138             auth_socket.settimeout(5)
139             try:
140                 logging.info(f"[FORWARDING TO AUTHORITATIVE] Server: {auth_info['host']}:{auth_info['port']}")
141                 auth_socket.sendto(query, (auth_info["host"], auth_info["port"]))
142                 response, _ = auth_socket.recvfrom(512)
```

tld.py > ...

```
51  class TLDServer:
136     def forward_to_authoritative(self, query, auth_info, query_type, client_addr, server_socket):
142                 response, _ = auth_socket.recvfrom(512)
143                 self.cache.set(extract_domain_name(query), query_type, response, 3600)
144                 server_socket.sendto(response, client_addr)
145             except socket.timeout:
146                 logging.error(f"[AUTHORITATIVE TIMEOUT] No response from Authoritative Server")
147                 self.send_response(query, 2, client_addr, server_socket)  # SERVFAIL
148
149     def send_response(self, query, rcode, client_addr, server_socket):
150         response = query[:2] + struct.pack("!H", 0x8180 | rcode) + query[4:]
151         server_socket.sendto(response, client_addr)
152
153     def start(self):
154         Thread(target=self.handle_udp).start()
155         Thread(target=self.handle_tcp).start()
156
157 if __name__ == "__main__":
158     tld_server = TLDServer("192.168.1.3", 8054)
159     tld_server.start()
```

Purpose of the Code

This code implements a TLD (Top-Level Domain) DNS server. The TLD server receives DNS queries, checks if the requested domain exists in its authoritative records, and either responds with the cached or authoritative information or forwards the query to the appropriate authoritative server. The server supports both UDP and TCP protocols.

Key Components of the Code

1. Logging Configuration

- The server uses the logging module to log events, such as received queries, cache hits/misses, forwarded queries, and errors.
- Logs are written to both the console and a file named dns_server.log.

2. DNS Cache

The **DNSCache** class handles the caching mechanism:

- **set**: Stores a response in the cache along with its time-to-live (TTL).
- **get**: Retrieves a cached response for a domain and query type. If the entry has expired, it is removed from the cache, and a cache miss is logged.

This caching mechanism improves performance by reducing the number of repetitive requests sent to authoritative servers.

3. Domain Name and Query Type Extraction

- **extract_domain_name**:
    - Extracts the domain name from a DNS query packet by decoding its structure.
    - The domain name is encoded as a series of labels (length-prefixed strings) in the query. This function iteratively parses them to reconstruct the full domain name.
- **extract_query_type**:
    - Extracts the query type (e.g., A, NS, MX) from the packet. By default, it assumes type A if the data length is insufficient.

4. TLD Records

The **TLD_RECORDS** dictionary contains the TLD server's authoritative records. It maps specific domain names (e.g., example.com, google.com) to their authoritative server's host and port. This mapping is used to forward queries to the correct server.

Example:

- example.com: Mapped to an authoritative server at IP 192.168.1.3 and port 8055.

5. TLD Server Class

The **TLDServer** class represents the TLD DNS server.

a. Initialization (__init__)

- Initializes the server's IP and port, DNS cache, and logging.
- Loads the TLD records that define the domains the server is responsible for.

b. Protocol Handling

The server supports both **UDP** and **TCP** protocols:

- ### UDP Handling:
  - A socket is created for UDP communication and bound to the specified IP and port.
  - The server listens for incoming DNS queries on the UDP socket.
  - Queries are processed by the handle_query method.
- ### TCP Handling:
  - A socket is created for TCP communication, bound to the specified IP and port, and set to listen for connections.
  - When a client establishes a connection, the server creates a new thread to handle it using the handle_tcp_client method.
  - The server processes the query, sends the response back to the client, and closes the connection.

c. Query Handling

The **handle_query** method processes DNS queries:

1. **Domain Name and Query Type Extraction:**
    - The domain name and query type are extracted from the incoming DNS packet.
2. **Cache Lookup:**
    - The server first checks its cache for a matching response.
    - If a cached response exists, it is sent back to the client immediately.
3. **TLD Records Lookup:**
    - If the domain exists in the TLD server's records, the query is forwarded to the authoritative server using the **forward_to_authoritative** method.
    - If the domain does not exist in the TLD records, an NXDOMAIN (Non-Existent Domain) response is sent to the client.

d. Forwarding to Authoritative Server

The **forward_to_authoritative** method handles forwarding queries to authoritative servers:

- A UDP socket is created to communicate with the authoritative server.
- The query is forwarded to the authoritative server's host and port (retrieved from the TLD records).
- If the authoritative server responds, the response is cached and forwarded to the client.

- If the authoritative server does not respond within a timeout period, a SERVFAIL (Server Failure) response is sent to the client.

e. Sending Error Responses

The **send_response** method constructs and sends DNS error responses:

- The response includes the query's transaction ID and an error code (rcode) indicating the type of error:
    - NXDOMAIN (3): Non-Existent Domain.
    - SERVFAIL (2): Server Failure.

f. Starting the Server

The **start** method starts the server:

- Separate threads are created for handling UDP and TCP connections concurrently, ensuring the server can process multiple queries simultaneously.

How the Code Works

1. Server Initialization:

- The server is started on IP 192.168.1.3 and port 8054.
- TLD records and the cache are initialized.

2. Listening for Queries:

- The server listens for incoming DNS queries over both UDP and TCP.

3. Processing a Query:

- When a query is received, the server extracts the domain name and query type.
- It checks the cache for a matching response.
- If the response is not in the cache, the server checks if the domain exists in its TLD records.

4. Handling the Query:

- If the domain exists in the TLD records:
  - The query is forwarded to the authoritative server.
  - The authoritative server's response is cached and forwarded to the client.
- If the domain does not exist:
  - An NXDOMAIN response is sent to the client.

5. Responding to the Client:

- The server responds with either:
  - A cached response.
  - A response from the authoritative server.
  - An error response (e.g., NXDOMAIN, SERVFAIL).

DNS Authoritative Server

authoritive.py > ...

```python
import socket
import struct
import logging
import time
from collections import defaultdict
from threading import Thread

# Logging configuration
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - AUTHORITATIVE SERVER - %(message)s",
    handlers=[
        logging.FileHandler("dns_server.log"),
        logging.StreamHandler()
    ]
)

class DNSCache:
    def __init__(self):
        self.cache = defaultdict(dict)

    def set(self, domain, query_type, response, ttl):
        expiry = time.time() + ttl
        self.cache[(domain, query_type)] = {"response": response, "expiry": expiry}
        logging.info(f"[CACHE SET] Domain: {domain}, Type: {query_type}, TTL: {ttl}s")

    def get(self, domain, query_type):
        if (domain, query_type) in self.cache:
            entry = self.cache[(domain, query_type)]
            if entry["expiry"] > time.time():
                logging.info(f"[CACHE HIT] Domain: {domain}, Type: {query_type}")
                return entry["response"]
            else:
                logging.info(f"[CACHE EXPIRED] Domain: {domain}, Type: {query_type}")
                del self.cache[(domain, query_type)]
        logging.info(f"[CACHE MISS] Domain: {domain}, Type: {query_type}")
        return None
```

authoritive.py > ...

```python
class DNSCache:
        return None

def extract_domain_name(data):
    domain_name = ""
    i = 12
    while data[i] != 0:
        length = data[i]
        domain_name += data[i + 1:i + 1 + length].decode() + "."
        i += length + 1
    return domain_name[:-1]

def extract_query_type(data):
    return struct.unpack("!H", data[-4:-2])[0] if len(data) >= 14 else 1  # Default to type A

def encode_domain_name(domain_name):
    parts = domain_name.split(".")
    encoded = b"".join([bytes([len(part)]) + part.encode() for part in parts])
    return encoded + b"\x00"

class AuthoritativeServer:
    DNS_RECORDS = {
        # Example Domain Records
        "example.com": {
            "A": "93.184.216.34",
            "CNAME": "alias.example.com",
            "TXT": "Example Domain",
            "MX": "mail.example.com",
            "NS": "ns1.example.com",
        },
        "mail.example.com": {"A": "93.184.216.35"},  # Mail server record
        "alias.example.com": {"A": "93.184.216.34"},  # Alias points to main server

        # Google Domain Records
        "google.com": {
            "A": "142.250.190.78",
            "TXT": "Google Services",
            "MX": "alt1.gmail-smtp-in.l.google.com",
```

```python
class AuthoritativeServer:
        # Google Domain Records
        "google.com": {
            "A": "142.250.190.78",
            "TXT": "Google Services",
            "MX": "alt1.gmail-smtp-in.l.google.com",
            "NS": "ns1.google.com",
        },
        "alt1.gmail-smtp-in.l.google.com": {"A": "172.217.10.27"},  # MX server

        # GitHub Domain Records
        "github.com": {
            "A": "140.82.121.4",
            "TXT": "GitHub Repository Hosting",
            "MX": "mail.github.com",
            "NS": "ns-1283.awsdns-32.org",
        },
        "mail.github.com": {"A": "192.30.252.1"},  # Mail server for GitHub

        # Facebook Domain Records
        "facebook.com": {
            "A": "157.240.23.35",
            "TXT": "Meta Social Network",
        },

        # Educational Institutions
        "nyu.edu": {
            "A": "128.122.49.42",
            "TXT": "NYU University",
        },
        "cs.umass.edu": {
            "A": "128.119.240.18",
            "TXT": "UMass CS Department",
        },

        # YouTube with CNAME Example
        "www.youtube.com": {
```

authoritive.py > ...

```python
class AuthoritativeServer:
        # YouTube with CNAME Example
        "www.youtube.com": {
            "CNAME": "youtube-ui.l.google.com",
        },
        "youtube-ui.l.google.com": {
            "A": "216.58.198.78",
        },
    }



    def __init__(self, host, port):
        self.host = host
        self.port = port
        self.cache = DNSCache()

    def handle_udp(self):
        udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        udp_socket.bind((self.host, self.port))
        logging.info(f"Authoritative Server UDP started on {self.host}:{self.port}")

        while True:
            data, addr = udp_socket.recvfrom(512)
            self.handle_query(data, addr, udp_socket)

    def handle_tcp(self):
        tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        tcp_socket.bind((self.host, self.port))
        tcp_socket.listen(5)
        logging.info(f"Authoritative Server TCP started on {self.host}:{self.port}")

        while True:
            conn, addr = tcp_socket.accept()
            Thread(target=self.handle_tcp_client, args=(conn,)).start()

    def handle_tcp_client(self, conn):
        try:
```

authoritive.py > ...

```python
class AuthoritativeServer:
    def handle_tcp_client(self, conn):
        try:
            data = conn.recv(1024)
            if data:
                response = self.handle_query(data, conn.getpeername(), conn)
                conn.sendall(response)
        finally:
            conn.close()

    def handle_query(self, data, addr, socket_conn):
        domain_name = extract_domain_name(data)
        query_type = extract_query_type(data)
        logging.info(f"[QUERY RECEIVED] Domain: {domain_name}, Type: {query_type}")

        # Check Cache
        cached_response = self.cache.get(domain_name, query_type)
        if cached_response:
            socket_conn.sendto(cached_response, addr)
            logging.info(f"[CACHE RESPONSE SENT] Domain: {domain_name}, Type: {query_type}")
            return

        # Check DNS Records
        if domain_name in self.DNS_RECORDS:
            if query_type == 1:  # A Record
                answer = self.DNS_RECORDS[domain_name].get("A", None)
                if answer:
                    response = self.build_response(data, 0, [{"type": "A", "value": answer, "ttl": 3600}])
                    self.cache.set(domain_name, query_type, response, 3600)
                    socket_conn.sendto(response, addr)
                    logging.info(f"[RESPONSE SENT] A Record for {domain_name}")
                    return

            elif query_type == 5:  # CNAME Record
                cname_record = self.DNS_RECORDS[domain_name].get("CNAME", None)
                if cname_record:
                    response = self.build_response(data, 0, [{"type": "CNAME", "value": cname_record, "ttl": 3600}])
```

```python
class AuthoritativeServer:
    def handle_query(self, data, addr, socket_conn):
            elif query_type == 5:  # CNAME Record
                cname_record = self.DNS_RECORDS[domain_name].get("CNAME", None)
                if cname_record:
                    response = self.build_response(data, 0, [{"type": "CNAME", "value": cname_record, "ttl": 3600}])
                    self.cache.set(domain_name, query_type, response, 3600)
                    socket_conn.sendto(response, addr)
                    logging.info(f"[RESPONSE SENT] CNAME Record for {domain_name}")
                    return

            elif query_type == 15:  # MX Record
                mx_record = self.DNS_RECORDS[domain_name].get("MX", None)
                if mx_record:
                    response = self.build_response(data, 0, [{"type": "MX", "value": mx_record, "ttl": 3600}])
                    self.cache.set(domain_name, query_type, response, 3600)
                    socket_conn.sendto(response, addr)
                    logging.info(f"[RESPONSE SENT] MX Record for {domain_name}")
                    return

            elif query_type == 2:  # NS Record
                ns_record = self.DNS_RECORDS[domain_name].get("NS", None)
                if ns_record:
                    response = self.build_response(data, 0, [{"type": "NS", "value": ns_record, "ttl": 3600}])
                    self.cache.set(domain_name, query_type, response, 3600)
                    socket_conn.sendto(response, addr)
                    logging.info(f"[RESPONSE SENT] NS Record for {domain_name}")
                    return

            elif query_type == 16:  # TXT Record
                txt_record = self.DNS_RECORDS[domain_name].get("TXT", None)
                if txt_record:
                    response = self.build_response(data, 0, [{"type": "TXT", "value": txt_record, "ttl": 3600}])
                    self.cache.set(domain_name, query_type, response, 3600)
                    socket_conn.sendto(response, addr)
                    logging.info(f"[RESPONSE SENT] TXT Record for {domain_name}")
                    return
```

authoritive.py > ...

```
56  class AuthoritativeServer:
147     def handle_query(self, data, addr, socket_conn):
206             # Unsupported query type or domain not found
207             logging.warning(f"[NXDOMAIN] Domain: {domain_name} not found in records")
208             response = self.build_response(data, 3)  # NXDOMAIN
209             socket_conn.sendto(response, addr)
210
211
212
213     def build_response(self, query, rcode, answers=None):
214         transaction_id = query[:2]
215         flags = struct.pack("!H", 0x8180 | rcode)  # Standard response
216         questions = struct.pack("!H", 1)
217         answer_rrs = struct.pack("!H", len(answers) if answers else 0)
218         authority_rrs = struct.pack("!H", 0)
219         additional_rrs = struct.pack("!H", 0)
220         question = query[12:]
221
222         response = transaction_id + flags + questions + answer_rrs + authority_rrs + additional_rrs + question
223
224         if answers:
225             for answer in answers:
226                 response += self.build_resource_record(answer)
227
228         return response
229
230     def build_resource_record(self, answer):
231         name = b'\xc0\x0c'  # Pointer to the domain name
232         ttl = struct.pack("!I", answer["ttl"])  # Time-to-Live value
233         record_class = struct.pack("!H", 1)  # IN (Internet)
234
235         if answer["type"] == "A":
236             record_type = struct.pack("!H", 1)  # A record
237             value = socket.inet_aton(answer["value"])  # IP address
238         elif answer["type"] == "CNAME":
239             record_type = struct.pack("!H", 5)  # CNAME record
240             value = encode_domain_name(answer["value"])  # Canonical name
```

authoritive.py > ...

```
56  class AuthoritativeServer:
230     def build_resource_record(self, answer):
236             record_type = struct.pack("!H", 1)  # A record
237             value = socket.inet_aton(answer["value"])  # IP address
238         elif answer["type"] == "CNAME":
239             record_type = struct.pack("!H", 5)  # CNAME record
240             value = encode_domain_name(answer["value"])  # Canonical name
241         elif answer["type"] == "MX":
242             record_type = struct.pack("!H", 15)  # MX record
243             preference = struct.pack("!H", 10)  # Priority value (can be adjusted as needed)
244             value = preference + encode_domain_name(answer["value"])  # Priority + Mail exchanger
245         elif answer["type"] == "NS":
246             record_type = struct.pack("!H", 2)  # NS record
247             value = encode_domain_name(answer["value"])  # Name server
248         elif answer["type"] == "TXT":
249             record_type = struct.pack("!H", 16)  # TXT record
250             value = bytes([len(answer["value"])]) + answer["value"].encode()  # TXT data
251         else:
252             raise ValueError(f"Unsupported record type: {answer['type']}")
253
254         record_length = struct.pack("!H", len(value))  # Length of the record value
255         return name + record_type + record_class + ttl + record_length + value
256
257
258     def start(self):
259         Thread(target=self.handle_udp).start()
260         Thread(target=self.handle_tcp).start()
261
262 if __name__ == "__main__":
263     authoritative_server = AuthoritativeServer("192.168.1.3", 8055)
264     authoritative_server.start()
```

1. Purpose of the Code

This code implements an **Authoritative DNS Server**. Its primary responsibility is to respond to DNS queries for specific domains by returning authoritative answers (like A, CNAME, MX, NS, or TXT records). It also includes a **caching mechanism** to speed up responses for frequently requested domains.

The server supports both **UDP** and **TCP** protocols, which are standard for DNS communication.

2. Key Components

a. Logging

The code uses Python's logging module to log important events (like cache hits, cache misses, query processing, etc.) to both the console and a file (dns_server.log).

b. DNSCache Class

The DNSCache class is responsible for caching DNS responses to avoid repeatedly querying for the same domain.

- **Methods**:
    - set(domain, query_type, response, ttl): Stores a response in the cache with a TTL (time-to-live).
    - get(domain, query_type): Retrieves a cached response if it is still valid (not expired). Otherwise, it returns None.

c. Helper Functions

1. **extract_domain_name(data)**
   Extracts the domain name from the incoming DNS query packet.

2. **extract_query_type(data)**
   Extracts the DNS query type (e.g., A, CNAME, etc.) from the query packet.

   - Default type is set to 1 (A record) if the packet doesn't specify it.

3. **encode_domain_name(domain_name)**
   Encodes a domain name (e.g., example.com) into the DNS wire format, where each label is prefixed by its length.
   For example:

   - example.com → \x07example\x03com\x00

d. AuthoritativeServer Class

This is the core of the DNS server. It contains the DNS logic, record storage, and query-handling methods.

- **Attributes**:

  - DNS_RECORDS: A dictionary of DNS records for specific domains.
    Each domain can have multiple record types (A, CNAME, TXT, etc.).

  - cache: An instance of the DNSCache class.

- **Key Methods**:

1. handle_udp()

Handles DNS queries over UDP (connectionless). It listens for incoming queries, processes them, and sends a response back to the client.

2. handle_tcp()

Handles DNS queries over TCP (connection-oriented). It accepts incoming connections, processes queries, and sends responses back.

3. handle_tcp_client(conn)

Processes individual client connections for TCP. It reads data from the client, processes the DNS query, and sends a response.

4. handle_query(data, addr, socket_conn)

This is the main method that processes DNS queries.

- Extracts the domain name and query type from the query.
- **Checks the cache** for a response. If found, it sends the cached response.
- If not cached, it looks up the domain in DNS_RECORDS and builds the appropriate response (e.g., A, CNAME, etc.).
- Sends a response or returns an NXDOMAIN error if the domain or query type is unsupported.

5. build_response(query, rcode, answers=None)

Constructs a DNS response packet based on the query, the response code (rcode), and the answers.

- Includes sections like the transaction ID, flags, questions, answers, etc.

6. build_resource_record(answer)

Builds an individual resource record (e.g., A, CNAME, MX, etc.) for the DNS response packet.

e. DNS Records Storage

The DNS_RECORDS dictionary acts as the database of DNS records. Here's an example structure for example.com:

```
"example.com": {
   "A": "93.184.216.34",
   "CNAME": "alias.example.com",
   "TXT": "Example Domain",
   "MX": "mail.example.com",
   "NS": "ns1.example.com",
},
```

- A: Maps the domain to an IP address (IPv4).
- CNAME: Canonical Name, an alias for another domain.
- TXT: Text records, often used for additional metadata.
- MX: Mail Exchange records, used for email servers.
- NS: Name Server records, identifying authoritative name servers.

3. Main Flow

When the server starts, it listens for DNS queries over both UDP and TCP. Here's how it works step by step:

1. **UDP/TCP Listener Threads**
    - handle_udp() starts a thread for handling UDP queries.
    - handle_tcp() starts a thread for handling TCP queries.

2. **Receiving and Processing Queries**
    - A query arrives (e.g., "What is the A record for example.com?").
    - The server extracts the **domain name** and **query type** from the query packet.
3. **Cache Check**
    - The server checks if the response is already cached.
    - If cached and valid, it sends the cached response.
4. **DNS Records Lookup**
    - If not cached, it looks up the DNS_RECORDS dictionary.
    - Based on the query type (A, CNAME, etc.), it builds the response.
5. **Building and Sending the Response**
    - A DNS response is constructed using build_response() and sent back to the client.
    - If the domain or query type is not found, it sends an NXDOMAIN error.
6. **Cache Update**
    - The response is added to the cache for future queries.

4. Supported Record Types

The server supports the following record types:

1. **A (Type 1)**: IPv4 address of the domain.

2. **CNAME (Type 5)**: Alias to another domain.
3. **MX (Type 15)**: Mail server for the domain.
4. **NS (Type 2)**: Name server for the domain.
5. **TXT (Type 16)**: Metadata or descriptive text.

RFC 1035:

UDP and TCP Support:

Code Implementation:

1. handle_udp processes UDP-based DNS queries.
2. handle_tcp processes TCP-based DNS queries, including handling length prefixes for TCP responses.

Test Cases

```
PS C:\Users\boody> nslookup -type=A facebook.com 192.168.1.3
Server:  UnKnown
Address:  192.168.1.3

Non-authoritative answer:
Name:    facebook.com
Address:  157.240.23.35
```

```
PS C:\Users\boody> nslookup -type=NS google.com 192.168.1.3
Server:  UnKnown
Address:  192.168.1.3

Non-authoritative answer:
google.com      nameserver = ns1.google.com
```

```
PS C:\Users\boody> nslookup -type=CNAME example.com 192.168.1.3
Server:  UnKnown
Address:  192.168.1.3

Non-authoritative answer:
example.com     canonical name = alias.example.com
```

```
PS C:\Users\boody> nslookup -type=MX google.com 192.168.1.3
Server:  UnKnown
Address:  192.168.1.3

Non-authoritative answer:
google.com      MX preference = 10, mail exchanger = alt1.gmail-smtp-in.l.google.com
```

```
PS C:\Users\boody> nslookup -type=TXT google.com 192.168.1.3
Server:  UnKnown
Address:  192.168.1.3

Non-authoritative answer:
google.com      text =

        "Google Services"
```

```
PS C:\Users\boody> nslookup -type=AAAA google.com 192.168.1.3
Server:  UnKnown
Address:  192.168.1.3

*** UnKnown can't find google.com: Not implemented
```

```
PS C:\Users\boody> nslookup -vc -type=TXT google.com 192.168.1.3
Server:  UnKnown
Address:  192.168.1.3

Non-authoritative answer:
google.com      text =

        "Google Services"
```

```
PS C:\Users\boody> nslookup  amazon.com 192.168.1.3
Server:  UnKnown
Address:  192.168.1.3

*** UnKnown can't find amazon.com: Non-existent domain
```

```
(base) mohamedwalid@mohameds-MacBook-Air ~ % nslookup -type=A example.com 192.168.1.3
Server:         192.168.1.3
Address:        192.168.1.3#53

** server can't find example.com: SERVFAIL
```

Installation and configuration steps:

Step 1: Install Python

1. Check if Python is Installed:
    - Open a terminal and type: python --version or python3 --version
    - If Python is not installed, download and install it from the official Python website.

Step 2: Required Libraries

The provided code uses standard Python libraries (`socket`, `threading`, `time`, `logging`), which are built into Python. You don't need to install additional third-party libraries.

Step 3: Setting Up Visual Studio Code

1. Install VS Code:
    - If you don't have Visual Studio Code installed, download it from the official VS Code website.
2. Install the Python Extension:
    - Open VS Code.
    - Go to the Extensions Marketplace (Ctrl+Shift+X or Cmd+Shift+X on Mac).
    - Search for "Python" and install the official extension by Microsoft.

Step 4: Runing the program

1. Run root server in terminal using “python root.py”.
2. Run tld server in terminal using “python tld.py”.
3. Run authoritative server in terminal using “python authoritative.py”.
4. Open command prompt and run the server using the keyword nslookup.