CSE 351 Computer Networks

Fall 2024/2025

Group 11

Phase 2

Submitted to:

Dr. Ayman Bahaa

Eng. Noha Wahdan

Submitted by:

Mohamed Mostafa Mamdouh 21P0244

Gaser Zaghloul Hassan 21P0052

Ali Tarek 21P0123

Abdelrahman Sherif 21P0098

Contents

Phase 2: Basic Client-Server Setup

1. **Objective**: Implement the foundational elements of the system, including basic client-server communication using Python and sockets.

2. **Deliverables**:

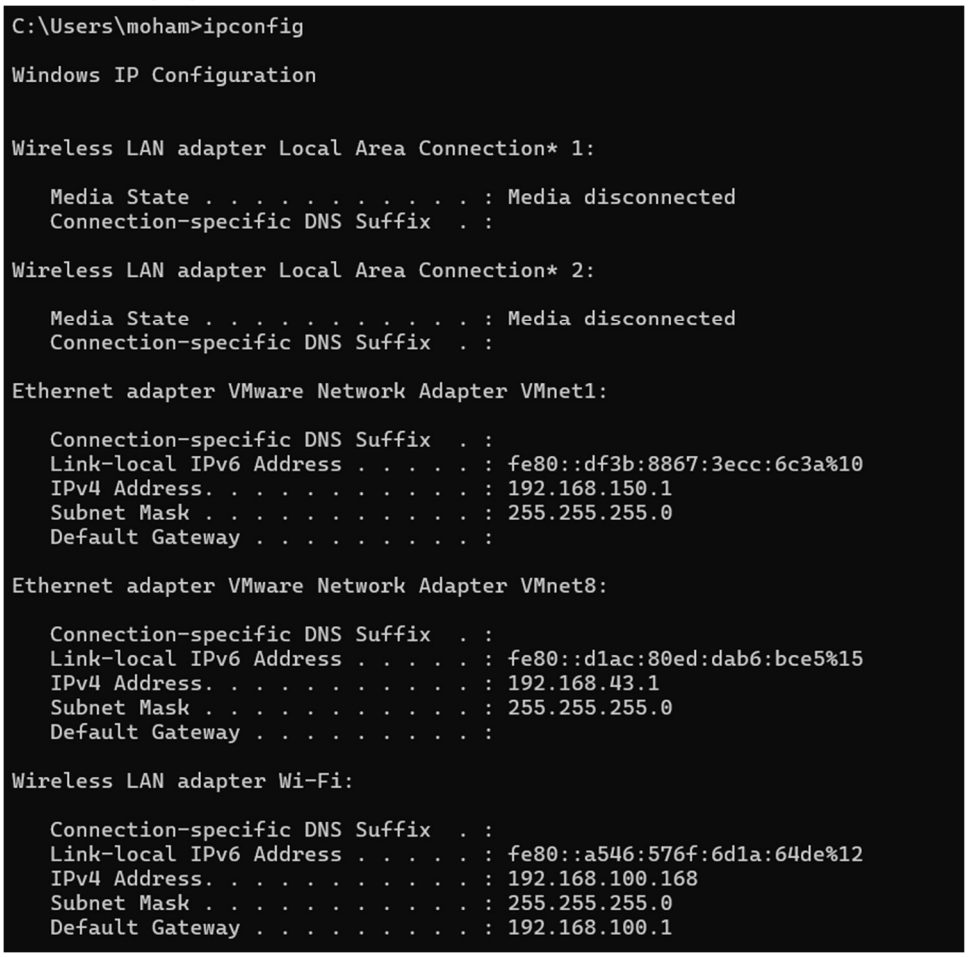
- Basic server application.
- Establish a connection for basic RFC features.

Steps

- Implement a basic server application capable of handling multiple client connections.
- Establish a TCP connection for user authentication (if required).
- Implement a simple command-line interface showing the server status.

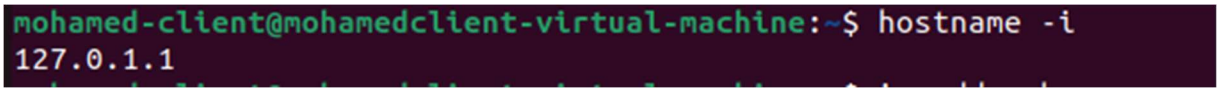
Server and Client Information

Server is deployed on a windows device at IP address 192.168.100.168



Server listens on port 1053 for DNS requests. When using nslookup from client side, the flag -port=1053 should be used.

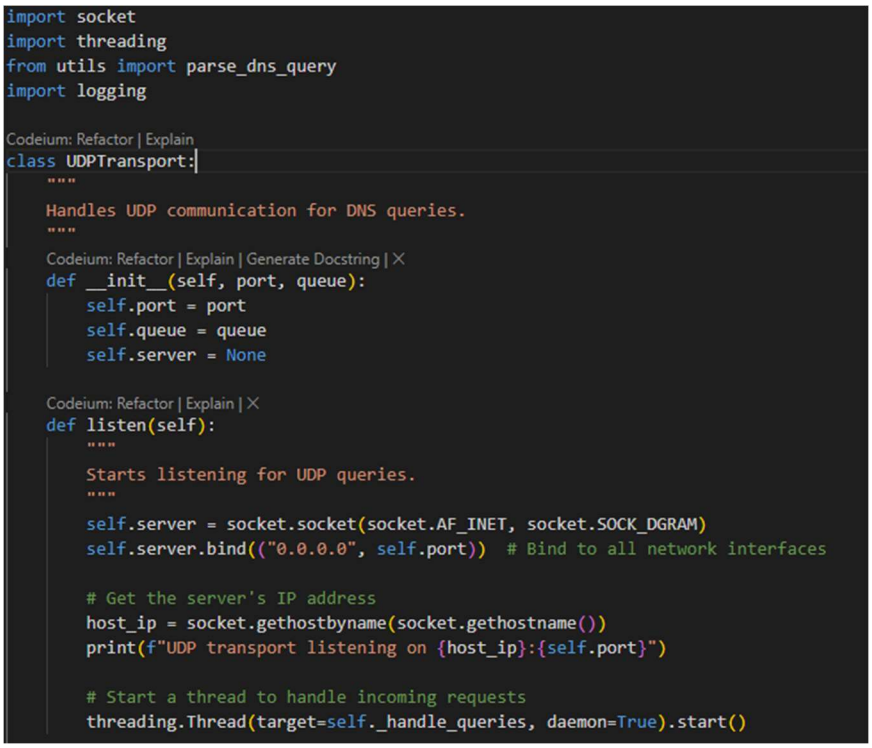
Client is a linux virtual machine with IP address 127.0.1.1



Threading was used to be able to handle multiple connections/requests at the same time.

Code screenshots and explanation

UDP transport



```python
import socket
import threading
from utils import parse_dns_query
import logging

Codeium: Refactor | Explain
class UDPTransport:
    """
    Handles UDP communication for DNS queries.
    """
    Codeium: Refactor | Explain | Generate Docstring | X
    def __init__(self, port, queue):
        self.port = port
        self.queue = queue
        self.server = None

    Codeium: Refactor | Explain | X
    def listen(self):
        """
        Starts listening for UDP queries.
        """
        self.server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.server.bind(("0.0.0.0", self.port))  # Bind to all network interfaces

        # Get the server's IP address
        host_ip = socket.gethostbyname(socket.gethostname())
        print(f"UDP transport listening on {host_ip}:{self.port}")

        # Start a thread to handle incoming requests
        threading.Thread(target=self._handle_queries, daemon=True).start()
```

def __init__(self, port, queue):

This is the class constructor, initializing the UDP transport service.

- **port**: The port on which the UDP server will listen.
- **queue**: A queue to pass parsed DNS query data for further processing.
- **self.server**: Initially set to None, it will later hold the UDP socket.

def listen(self)

Starts the UDP server to listen for incoming DNS queries.

1. Creates a UDP socket (socket.AF_INET for IPv4, socket.SOCK_DGRAM for UDP).
2. Binds the socket to all network interfaces (0.0.0.0) and the specified port.
3. Retrieves the server's IP address using socket.gethostbyname(socket.gethostname()).
4. Logs that the server is listening on the specified IP and port.
5. Spawns a daemon thread to handle incoming requests using _handle_queries.

```python
def _handle_queries(self):
    """
    Handles incoming DNS queries from the UDP socket.
    """
    while True:
        try:
            data, client_addr = self.server.recvfrom(512)  # 512 bytes is max for DNS over UDP
            threading.Thread(target=self._handle_udp_query, args=(data, client_addr), daemon=True).start()
        except Exception as e:
            logging.info(f"Error reading from UDP socket: {e}")
```

_handle_queries(self)

Continuously listens for incoming DNS queries on the UDP socket.

1. Enters an infinite loop (while True) to receive data from the socket using recvfrom.
2. When data is received, spawns a new daemon thread to process the query with _handle_udp_query.
3. If an error occurs while reading from the socket, logs the exception.

```python
def _handle_udp_query(self, query_data, client_addr):
    logging.info(f"query data is {query_data}")
    try:
        query_raw = query_data
        transaction_id, domain_name, qtype, qclass = parse_dns_query(query_raw)
        logging.info(f"Parsed DNS query for domain: {domain_name} from {client_addr} with transaction ID: {transaction_id}")

        self.queue.put({
            "domain_name": domain_name,
            "transaction_id": transaction_id,
            "qtype": qtype,
            "qclass": qclass,
            "respond": lambda response: self.server.sendto(response, client_addr),
            "raw_query": query_raw  # Ensure this is the raw query bytes
        })

    except Exception as e:
        logging.info(f"Error unpacking DNS query from {client_addr}: {e}")
```

_handle_udp_query(self, query_data, client_addr)

Processes individual DNS queries.

1. Logs the raw query data.
2. Attempts to parse the query using a helper function (parse_dns_query) that extracts:
    - transaction_id: Identifies the query for matching responses.
    - domain_name: The domain being queried.
    - qtype: The type of query (e.g., A, NS, MX).
    - qclass: The class of the query (commonly IN for internet).
3. Logs the parsed query details.
4. Places a dictionary with the parsed query information into the queue, including:
    - A respond lambda function to send a response back to the client.
    - The raw_query for reference or retransmission.
5. Logs any error encountered during query parsing.

close(self)

Gracefully shuts down the UDP server.

1. Checks if the server is initialized.
2. Closes the UDP socket.
3. Logs a message indicating that the transport has been closed.

TCP transport

The tcp_transport.py file handles DNS queries over TCP. It creates a server that listens for incoming connections, processes client queries, and sends responses.

Initialize (__init__) : Sets up the server with a port and a queue for query processing.

Start Server (listen) : Creates a TCP socket, binds it to a port, and begins listening for connections.

tcp_transport.py

app > tcp_transport.py > TCPTransport > _accept_connections

Mohamed, 8 hours ago | 1 author (Mohamed)

```python
import socket
import threading
from utils import parse_dns_query
import logging


Mohamed, 8 hours ago | 1 author (Mohamed)
class TCPTransport:
    """
    Handles TCP communication for DNS queries.
    """
    def __init__(self, port, queue):
        self.port = port
        self.queue = queue
        self.server = None

    def listen(self):
        """
        Starts listening for TCP queries.
        """
        self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server.bind(("0.0.0.0", self.port))  # Bind to all network interfaces
        self.server.listen(5)  # Listen for up to 5 connections
        host_ip = socket.gethostbyname(socket.gethostname())
        logging.info(f"TCP transport listening on port {host_ip}:{self.port}")

        # Start a thread to accept incoming connections
        threading.Thread(target=self._accept_connections, daemon=True).start()
```

Accept Connections (_accept_connections): Manages new client connections and assigns each to a handler.

```python
def _accept_connections(self):
    """
    Accepts incoming TCP connections.
    """      Mohamed, 8 hours ago • phase 2
    while True:
        try:
            conn, addr = self.server.accept()
            threading.Thread(target=self._handle_connection, args=(conn, addr), daemon=True).start()
        except Exception as e:
            logging.info(f"Error accepting TCP connection: {e}")
```

Once a client connects:

1. It reads the incoming DNS query (the first 2 bytes tell the query's size).
2. Parses the query to extract useful details like the domain name and query type.
3. Logs the details for debugging or tracking.
4. Stores the query in the queue for further processing.
5. Prepares a way to send responses back to the client.

```python
def _handle_connection(self, conn, client_addr):
    """
    Handles individual TCP connections.
    """
    with conn:
        while True:
            try:
                # Read the length of the incoming DNS query
                length_bytes = conn.recv(2)
                if not length_bytes:
                    break

                query_length = int.from_bytes(length_bytes, "big")
                query_data = conn.recv(query_length)

                # Parse the query using the same utility as UDP
                transaction_id, domain_name, qtype, qclass = parse_dns_query(query_data)
                logging.info(f"Parsed DNS query for domain: {domain_name} from {client_addr} with transaction ID: {transaction_id}")

                # Add query data to the queue
                self.queue.put({
                    "domain_name": domain_name,
                    "transaction_id": transaction_id,
                    "qtype": qtype,
                    "qclass": qclass,
                    "respond": lambda response: conn.sendall(len(response).to_bytes(2, "big") + response),
                    "raw_query": query_data
                })
```

If something goes wrong (e.g., the client sends bad data), it logs the error and stops processing.

```python
            except Exception as e:
                logging.info(f"Error handling TCP query from {client_addr}: {e}")
                break
```

```python
    def close(self):
        """
        Closes the TCP socket.
        """
        if self.server:
            self.server.close()
            logging.info("TCP transport closed")
```

Parsing dns query

```python
def parse_dns_query(query):
    """
    Parses a DNS query to extract the transaction ID, domain name, query type (QTYPE), and query class (QCLASS).
    """
    # Ensure the query is long enough to contain a header
    if len(query) < 12:
        raise ValueError("Invalid DNS query: Too short")

    # Transaction ID is the first 2 bytes
    transaction_id = struct.unpack("!H", query[:2])[0]

    # Parse the domain name, which starts after the first 12 bytes (header)
    domain_parts = []
    idx = 12
    try:
        while query[idx] != 0:  # A label is terminated by a 0 byte
            length = query[idx]
            idx += 1
            if idx + length > len(query):
                raise ValueError("Invalid DNS query: Domain name length exceeds query size")
            domain_parts.append(query[idx:idx + length].decode())
            idx += length
    except IndexError:
        raise ValueError("Invalid DNS query: Domain name parsing failed")

    domain_name = ".".join(domain_parts)

    # Skip the next byte before reading QTYPE and QCLASS
    idx += 1

    # Now that the domain is fully parsed, the next 4 bytes should be QTYPE and QCLASS
    # Ensure there's enough data for QTYPE and QCLASS (2 bytes each)
    if len(query) < idx + 4:
        raise ValueError("Invalid DNS query: Missing QTYPE or QCLASS")

    # Unpack QTYPE and QCLASS (each are 2 bytes long, so we use "!HH")
    qtype = struct.unpack("!H", query[idx:idx + 2])[0]
    qclass = struct.unpack("!H", query[idx + 2:idx + 4])[0]

    # Debugging output to check what values are being parsed
    # logging.debug(f"Transaction ID: {transaction_id}, Domain: {domain_name}, QTYPE: {qtype}, QCLASS: {qclass}")

    # If QCLASS is not valid, raise an error
    if qclass != 1:  # Only support IN class (1)
        logging.error(f"Invalid query: Unsupported query class {qclass}")
        raise ValueError(f"Unsupported query class: {qclass}")

    return transaction_id, domain_name, qtype, qclass
```

This function parses a raw DNS query to extract the **transaction ID**, **domain name**, **query type (QTYPE)**, and **query class (QCLASS)**. It validates the query format, decodes the domain name from its labels, and ensures the query class is supported (only Internet class, 1). It raises errors for malformed or unsupported queries.

Main

```python
def main():
    logging.info("Starting the DNS Server Agent...")

    cache, authoritative_server, tld_server, root_server, udp_transport, tcp_transport, udp_thread = start_dns_server()

    if cache is None or authoritative_server is None:
        logging.error("Failed to start DNS server.")
    else:
        try:
            # Choose interface
            choice = input("Enter '1' for terminal interface or '2' for GUI: ").strip()
            if choice == '1':
                start_terminal_interface(cache, authoritative_server, tld_server, root_server)
            elif choice == '2':
                start_gui(cache, authoritative_server, tld_server, root_server)
            else:
                print("Invalid choice. Exiting.")
        except KeyboardInterrupt:
            print("\nShutting down the DNS Server. Goodbye!")
            logging.info("Shutting down DNS server...")
            # Close resources
            udp_transport.close()
            tcp_transport.close()
            udp_thread.join()

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

This function serves as the main entry point for the DNS Server Agent. Here's what it does:

1. **Logs Start-Up**: Logs a message indicating the DNS server is starting.
2. **Initializes Server Components**: Calls start_dns_server() to set up essential components like the cache, authoritative server, TLD server, root server, UDP and TCP transports, and a UDP thread.
3. **Checks Initialization Success**: Verifies if the critical components (e.g., cache and authoritative server) were successfully started.
4. **Graceful Shutdown**: Catches KeyboardInterrupt (e.g., Ctrl+C), logs a shutdown message, and ensures all resources (UDP transport, TCP transport, and UDP thread) are properly closed.

This function orchestrates the setup, user interaction, and shutdown of the DNS Server Agent.

Note that the functions start_terminal_interface were created for local development only before deploying the server and testing on the client device.

Start DNS server

```python
def start_dns_server():
    """
    Starts the DNS server that listens for queries over UDP and TCP.
    """
    # Initialize components
    authoritative_cache = Cache1(redis_host="localhost", redis_port=6380)  # Authoritative server cache
    resolver_cache = Cache2(redis_host="localhost", redis_port=6379)  # Resolver cache
    authoritative_server = AuthoritativeServer(authoritative_cache)  # Handle authoritative queries    root_server = RootServer()  # Initialize RootServer
    root_server = RootServer()  # Initialize RootServer
    tld_server = TLDServer()    # Initialize TLDServer

    query_queue = Queue()

    # Start UDP transport
    udp_transport = UDPTransport(DNS_SERVER_UDP_PORT, query_queue)
    udp_transport.listen()

    # Start TCP transport
    tcp_transport = TCPTransport(DNS_SERVER_TCP_PORT, query_queue)
    tcp_transport.listen()

    logging.info(f"DNS server is running on {DNS_SERVER_IP}:{DNS_SERVER_UDP_PORT} for UDP...")
    logging.info(f"DNS server is running on {DNS_SERVER_IP}:{DNS_SERVER_TCP_PORT} for TCP...")

    udp_thread = threading.Thread(target=process_queries, args=(query_queue, resolver_cache, root_server, tld_server, authoritative_server))
    udp_thread.start()

    return resolver_cache, authoritative_server, tld_server, root_server, udp_transport, tcp_transport, udp_thread
```

The start_dns_server function initializes and starts a DNS server that listens for queries over both UDP and TCP. Here's what it does:

1. **Initialize Components**:
    - Sets up two cache systems: authoritative_cache (for authoritative server) and resolver_cache (for resolver).
    - Initializes the authoritative_server, root_server, and tld_server components, which handle different types of DNS queries.
2. **Create a Query Queue**:
    - Creates a Queue to manage incoming DNS queries.
3. **Start UDP Transport**:
    - Initializes and starts the UDPTransport to listen for DNS queries over UDP on a specified port.
4. **Start TCP Transport**:
    - Initializes and starts the TCPTransport to listen for DNS queries over TCP on a specified port.
5. **Logging**:
    - Logs the DNS server's status, indicating it's running and listening on both UDP and TCP ports.
6. **Start Query Processing**:
    - Starts a new thread to process incoming queries from the query_queue, using the process_queries function.
7. **Returns Components**:
    - Returns the components necessary for handling DNS queries, including the caches, servers, and transport layers, along with the query processing thread.

Process queries

```python
def process_queries(queue, cache, root_server, tld_server, authoritative_server):
    while True:
        query_data = queue.get()
        if query_data:
            query_raw = query_data.get('raw_query')
            if query_raw:
                try:
                    transaction_id, domain_name, qtype, qclass = parse_dns_query(query_raw)

                    # Handle IPv6 queries
                    if qtype == 28:
                        logging.info(f"IPv6 query received for {domain_name}, sending Not Implemented")
                        response = build_error_response(query_raw, rcode=4)
                        query_data['respond'](response)
                        continue

                    # Normal processing for other queries
                    response = resolve_query(
                        query_raw,
                        cache,
                        root_server,
                        tld_server,
                        authoritative_server,
                        recursive=True,
                        is_tcp=False
                    )
                    if response:
                        query_data['respond'](response)

                except Exception as e:
                    logging.error(f"Error processing query: {e}")
```

The process_queries function processes DNS queries from a queue and handles them accordingly.

1. **Continuous Loop**:
    - Continuously retrieves query data from the queue and processes it.
2. **Extract Query Data**:
    - Extracts the raw query from query_data using get('raw_query').
3. **Parse the Query**:
    - Uses parse_dns_query to parse the raw query, extracting the transaction ID, domain name, query type (QTYPE), and query class (QCLASS).
4. **Handle IPv6 Queries**:
    - If the query is of type 28 (IPv6), it logs an error message indicating IPv6 is not implemented and sends a "Not Implemented" response (RCode 4).
5. **Normal Query Processing**:
    - For other queries, it resolves the query using the resolve_query function, which utilizes the provided caches and server components to resolve the query.
    - The response is sent back to the client using the respond function from the query data.

6. **Error Handling**:
    - If an error occurs during query processing (e.g., parsing or resolving the query), it logs the error.

Resolve query

```python
def resolve_query(query, cache: Cache2, root_server: RootServer, tld_server: TLDServer, authoritative_server: AuthoritativeServer, recursive, is_tcp=False):
    """
    Resolves a DNS query by checking the cache and querying the Root, TLD, and Authoritative servers in sequence.
    The recursive flag indicates whether to resolve the query recursively.
    """
    # Validate the query format
    try:
        transaction_id, domain_name, qtype, qclass = validate_query(query)
    except ValueError as e:
        logging.error(f"Invalid query: {e}")
        return build_error_response(query, rcode=1)  # Format error (RCODE 1)

    cache_key = (domain_name, qtype, qclass)
    # Check the cache for the response
    cached_response = cache.get(cache_key, transaction_id)
    if cached_response:
        logging.info(f"Resolver cache hit for domain: {domain_name}")
        human_readable = parse_dns_response(cached_response)
        print(human_readable)
        return cached_response
        # return human_readable

    logging.info(f"Cache miss for domain: {domain_name}. Querying root server.")

    if recursive:
        # Query Root Server and follow the chain for recursive resolution
        logging.info(f"recursive query")
        root_response = root_server.handle_root_query(query)
        logging.info(f"root response is {root_response}")

        if not root_response:
            logging.error(f"Root server could not resolve domain: {domain_name}")
            return build_error_response(query, rcode=3)  # NXDOMAIN

        # Query TLD Server
        tld_server_ip = extract_referred_ip(root_response)
        logging.debug(f"Referred TLD server IP: {tld_server_ip}")

        tld_response = tld_server.handle_tld_query(root_response)
        if not tld_response:
            logging.error(f"TLD server could not resolve domain: {domain_name}")
            return build_error_response(query, rcode=3)  # NXDOMAIN

        # Query Authoritative Server
        logging.info(f"your tld response is {tld_response}")
        authoritative_server_ip = extract_referred_ip(tld_response)
        logging.debug(f"Referred authoritative server IP: {authoritative_server_ip}")
        authoritative_response = authoritative_server.handle_name_query(tld_response)
        if not authoritative_response:
            logging.error(f"Authoritative server could not resolve domain: {domain_name}")
            return build_error_response(query, rcode=3)  # NXDOMAIN

        # Cache the successful response
        logging.info(f"Caching response for domain: {domain_name}")
        cache.store(authoritative_response)
    else:
        logging.info(f"iterative query")
```

```
        # Iterative query: Simply send back the referral or best possible response
        root_response = root_server.handle_root_query(query)
        logging.info(f"root response is {root_response}")
        if not root_response:
            logging.error(f"Root server could not resolve domain: {domain_name}")
            return build_error_response(query, rcode=3)  # NXDOMAIN

        # Return the referral to TLD server
        tld_server_ip = extract_referred_ip(root_response)
        tld_response = tld_server.handle_tld_query(query)
        return tld_response

    # Check if the response needs to be sent over TCP (due to TC flag)
    if len(authoritative_response) > 512:
        logging.info("Response size exceeds 512 bytes, setting TC flag and returning over TCP.")
        if not is_tcp:  # If this is a UDP query and response is truncated, we should use TCP
            # Set the TC flag in the DNS header to indicate truncation
            authoritative_response = set_tc_flag(authoritative_response)
            return authoritative_response
        else:
            human_readable = parse_dns_response(authoritative_response)
            return human_readable

    # Return the response as a regular UDP response
    human_readable = parse_dns_response(authoritative_response)
    print(human_readable)
    return authoritative_response
    # return human_readable

Codeium: Refactor | Explain | ×
def set_tc_flag(response):
    """
    Sets the TC flag in the DNS header to indicate that the response is truncated.
    This is used when sending the response over TCP.
    """
    # Unpack the DNS header
    header = response[:12]  # First 12 bytes are the DNS header
    transaction_id = struct.unpack("!H", header[:2])[0]
    flags = struct.unpack("!H", header[2:4])[0]
    # Set the TC flag (bit 1) to 1
    flags |= 0x0200  # 0x0200 corresponds to the TC bit (bit 1 of the flags byte)
    # Repack the DNS header with the updated flags
    header = struct.pack("!HHHHHH", transaction_id, flags, 1, 0, 0, 0)
    # Return the response with the updated header
    return header + response[12:]
```

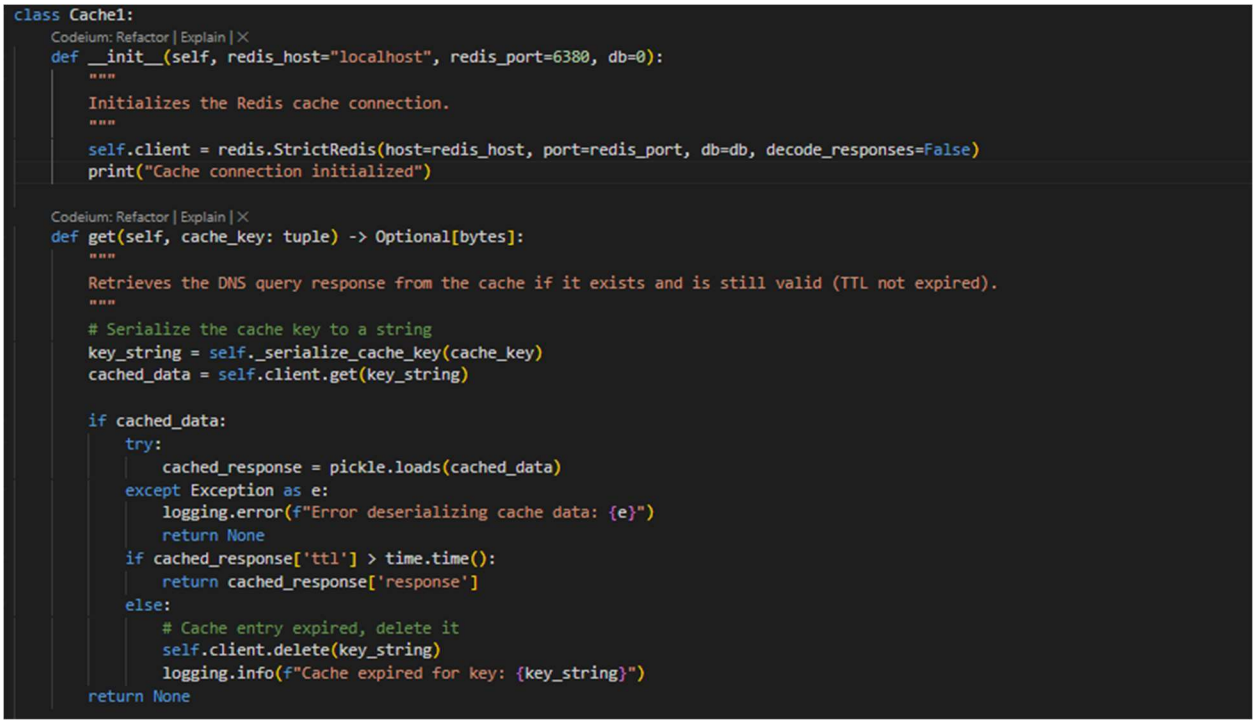
The resolve_query function processes a DNS query by attempting to resolve the domain name through multiple stages, including cache checking, root server querying, TLD server querying, and authoritative server querying. Here's a brief breakdown:

1. **Query Validation**:
    - Validates the incoming query's format. If invalid, it returns an error response (RCODE 1).
2. **Cache Lookup**:
    - Checks if the response is already in the cache for the given domain, query type, and query class.
    - If found, it logs a cache hit, logs a human-readable version of the response, and returns the cached response.

3. **Recursive Query Resolution**:
    - If the recursive flag is set:
        - Queries the root server for the domain.
        - If the root server cannot resolve the domain, it returns an NXDOMAIN error.
        - If the root server responds, queries the TLD server, followed by the authoritative server.
        - Each server's response is validated and processed. If any server cannot resolve the domain, an NXDOMAIN error is returned.
        - The final authoritative response is cached for future use.
4. **Iterative Query Resolution**:
    - If recursive is not set, the function returns the referral to the TLD server from the root server.
5. **TCP Handling**:
    - If the response exceeds 512 bytes (standard UDP DNS size), it sets the TC (truncation) flag in the DNS header.
    - If the query was over UDP and the response is too large, the function sets the TC flag and returns the response over TCP.
6. **Response Formatting**:
    - After resolving the query, the response is either returned as a DNS response (for UDP or TCP) or a human-readable version of the response is printed.

In summary, this function either resolves the query recursively or iteratively, caches the result, and ensures that responses larger than 512 bytes are handled appropriately for TCP.

Authoritative server cache



```
class Cache1:
    Codeium: Refactor | Explain | ×
    def __init__(self, redis_host="localhost", redis_port=6380, db=0):
        """
        Initializes the Redis cache connection.
        """
        self.client = redis.StrictRedis(host=redis_host, port=redis_port, db=db, decode_responses=False)
        print("Cache connection initialized")

    Codeium: Refactor | Explain | ×
    def get(self, cache_key: tuple) -> Optional[bytes]:
        """
        Retrieves the DNS query response from the cache if it exists and is still valid (TTL not expired).
        """
        # Serialize the cache key to a string
        key_string = self._serialize_cache_key(cache_key)
        cached_data = self.client.get(key_string)

        if cached_data:
            try:
                cached_response = pickle.loads(cached_data)
            except Exception as e:
                logging.error(f"Error deserializing cache data: {e}")
                return None
            if cached_response['ttl'] > time.time():
                return cached_response['response']
            else:
                # Cache entry expired, delete it
                self.client.delete(key_string)
                logging.info(f"Cache expired for key: {key_string}")
        return None
```

1. __init__(self, redis_host="localhost", redis_port=6379, db=0)

- **Purpose:** Initializes a connection to a Redis database.
- **Parameters:**
    - redis_host: The hostname of the Redis server (default: "localhost").
    - redis_port: The port of the Redis server (default: 6379).
    - db: The Redis database number to use (default: 0).
- **Steps:**
    1. Sets up a connection to Redis using the provided host, port, and database.
    2. The decode_responses=False ensures that binary data is preserved.
    3. Prints a message indicating the cache connection has been initialized.

2. get(self, cache_key: tuple, transaction_id: int) -> Optional[bytes]

- **Purpose:** Retrieves a DNS query response from the cache if it exists and is still valid (TTL not expired).
- **Steps:**
    1. Serializes the provided cache_key (tuple format) into a unique string key using the _serialize_cache_key method.
    2. Attempts to retrieve the cached data from Redis using the serialized key.
    3. If data is found:
        - Attempts to deserialize it using pickle.
        - Checks if the entry's TTL (expiration time) has passed:
            - **Valid:** Returns the cached DNS response.

- **Expired:** Deletes the entry from the cache and logs the expiration.

4. If no valid data is found or if an error occurs, returns None.

```python
def _serialize_cache_key(self, cache_key: tuple) -> str:
    """
    Serializes a tuple cache key into a string format suitable for Redis.

    Parameters:
        cache_key (tuple): The cache key as (domain_name, qtype, qclass).

    Returns:
        str: A serialized string suitable for Redis.
    """
    return f"dns:{hashlib.sha256(':'.join(map(str, cache_key)).encode()).hexdigest()}"
```

3. _serialize_cache_key(self, cache_key: tuple) -> str

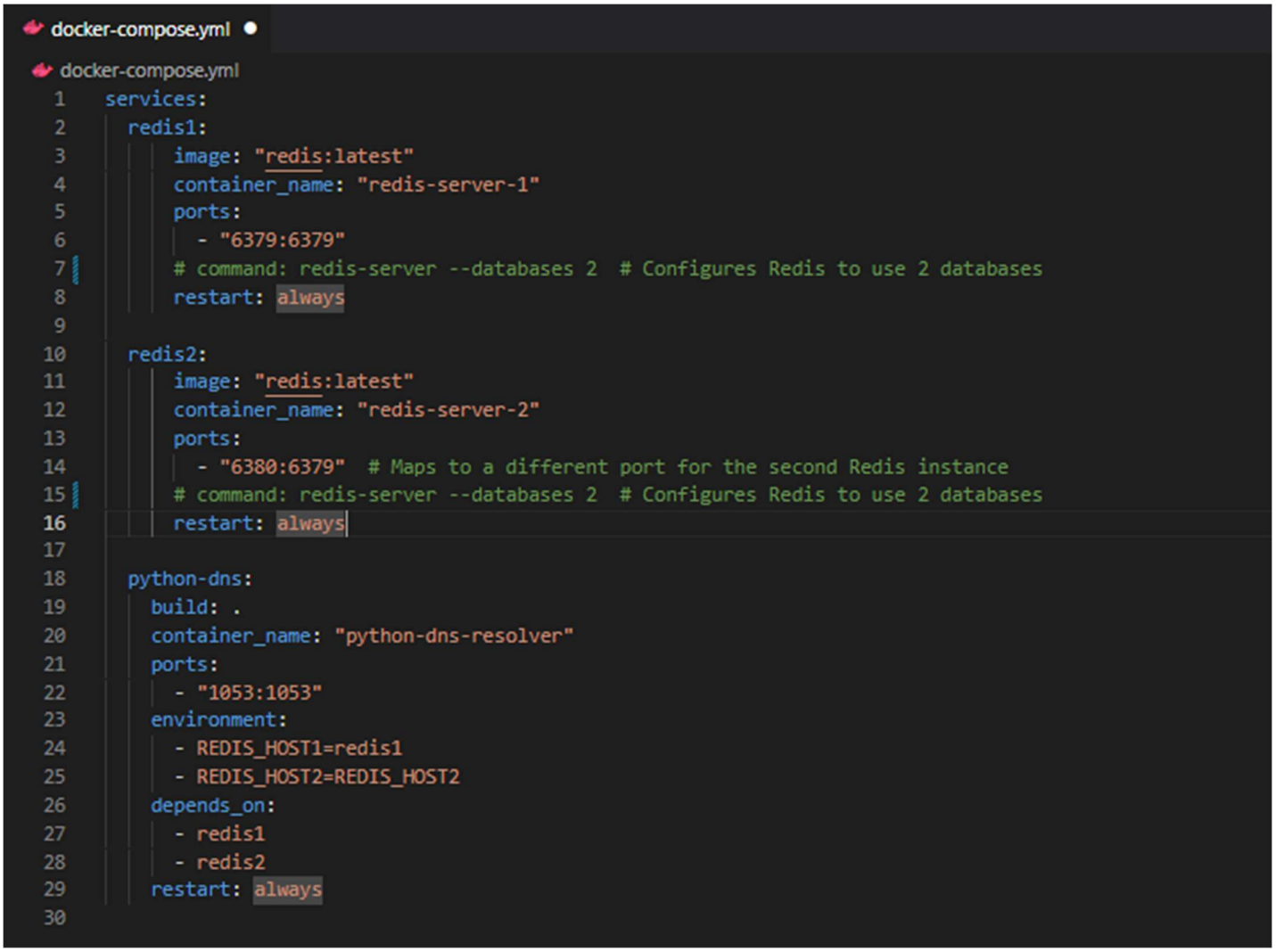
- **Purpose:** Converts a tuple-based cache key (e.g., domain name, query type, query class) into a unique string format.
- **Steps:**
    1. Joins the elements of the cache_key tuple into a single string separated by colons.
    2. Generates a SHA-256 hash of this string to ensure uniqueness and a consistent format.
    3. Prefixes the hash with "dns:" to namespace it for Redis storage.
- **Returns:** The resulting string (e.g., "dns:<hashed_key>").

```python
def store(self, response: bytes):
    """
    Stores the DNS query response in the cache with a specified TTL.

    Parameters:
        response (bytes): The authoritative DNS response to store.
        ttl (int): Time-to-live in seconds for the cached entry.
    """
    ttl = 3600
    try:
        # Ensure TTL is an integer
        if not isinstance(ttl, int):
            raise ValueError(f"TTL must be an integer, got {type(ttl)}: {ttl}")

        # Extract the question section to build the cache key
        qname, qtype, qclass, _ = parse_question_section(response, 12)
        cache_key = (qname, qtype, qclass)

        # Serialize the cache key to a string
        key_string = self._serialize_cache_key(cache_key)

        # Create the cache entry
        cache_entry = {
            'response': response,  # Store the entire authoritative response
            'ttl': time.time() + ttl  # Set TTL based on current time
        }

        # Store in Redis
        self.client.setex(key_string, ttl, pickle.dumps(cache_entry))
        logging.info(f"Stored response in cache for key: {key_string}")
    except Exception as e:
        logging.error(f"Error storing response in cache: {e}")
```

4. store(self, response: bytes)

- **Purpose:** Saves a DNS query response to a cache (likely a Redis instance) with a specified time-to-live (TTL).
- **Steps:**
    1. Verifies that the TTL is an integer.
    2. Extracts the DNS question section from the response to create a cache key.
    3. Serializes the cache key into a string format suitable for storing in Redis.
    4. Creates a cache entry that includes the DNS response and an expiration timestamp (current time + TTL).
    5. Stores the serialized cache entry in Redis with the generated key and TTL.
- **Error Handling:** Logs errors if any occur during the process.

Cache



For easier connectivity and deployment, server was packaged in a docker-compose container.

1. **redis1 (authoritative server cache)**:
    - The container is named redis-server-1.
    - Exposes port 6379 on the host to the container's 6379.
    - Restarts automatically if the container stops.
2. **redis2 (resolver cache)**:
    - Another Redis instance, but running on a different port (6380 on the host to 6379 inside the container).
    - Restarts automatically if the container stops.
3. **python-dns**:
    - A custom-built Docker image (from the current directory) for a Python-based DNS resolver.
    - The container is named python-dns-resolver.
    - Exposes port 1053 for the DNS service.

- Defines environment variables to connect to the Redis instances (REDIS_HOST1=redis1 and REDIS_HOST2=redis2).
- Specifies that it depends on redis1 and redis2 to ensure those services are started first.
- Restarts automatically if the container stops.

This configuration sets up two Redis instances and a DNS resolver that interacts with both Redis services for caching.

Testing the server

Server initiation:

```
[Running] python -u "c:\Users\moham\Documents\uni\semesters\fall 2025\networks\Networks-project\app\main.py"
INFO:root:Starting the DNS Server Agent...
Cache connection initialized
Cache connection initialized
UDP transport listening on 192.168.100.168:1053
INFO:root:TCP transport listening on port 192.168.100.168:1053
INFO:root:DNS server is running on 0.0.0.0:1053 for UDP...
Enter '1' for terminal interface or '2' for GUI:
```

Client side terminal (querying for networking.net):

```
mohamed-client@mohamedclient-virtual-machine:~$ nslookup -port=1053 -type=A  networking.net 192.168.100.168
Server:         192.168.100.168
Address:        192.168.100.168#1053

Non-authoritative answer:
Name:   networking.net
Address: 93.184.216.37
Name:   networking.net
Address: 93.184.216.38
```

Server side terminal:

Note that excessive logging was used for debugging purposes. This will all be cleaned up by phase 4 submission.

Response stored in Resolver cache:

```
127.0.0.1:6379> keys *
1) "dns:cb60c948c52e3f812124df0b1a0ac96018aeb9c086cd7f298be8612e4e47e196"
127.0.0.1:6379>
```

Response stored in Name cache:

```
PS C:\Users\moham\Documents\uni\semesters\fall 2025\networks\Networks-project> docker exec -it redis-server-2 /bin/bash
root@847e88485f88:/data# redis-cli
127.0.0.1:6379> keys *
1) "dns:b14c9e8dd23e71c03ecaaae3c5aa4ed58bdd58b29b1109380c95c0d5d9464b86"
2) "dns:cb60c948c52e3f812124df0b1a0ac96018aeb9c086cd7f298be8612e4e47e196"
3) "dns:7927b7f2a4faada0e932f80f3a914e91a85e339158cd4062bbebdde3fc835a8a"
4) "dns:f652beeaa71cda9beb46e7b048e9d2dfd3e155975945ede9a27c4869ef29bdef"
127.0.0.1:6379>
```

Networking.net record in Authoritative.py file:

```
"networking.net": {
    "A": ["93.184.216.37", "93.184.216.38"],
    "NS": ["ns1.networking.net.", "ns2.networking.net."],
    "MX": ["10 mail.networking.net.", "20 backup.mail.networking.net."],
    "SOA": ["ns1.networking.net. admin.networking.net. 2023120304 7200 3600 1209600 86400"],
    "PTR": ["networking.net.", "reverse.networking.net."]
},
```