A blue and white logo

Description automatically generated

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 3](#_Toc185712926)

[Server and Client Information 4](#_Toc185712927)

[Code screenshots and explanation 5](#_Toc185712928)

[UDP transport 5](#_Toc185712929)

[TCP transport 7](#_Toc185712930)

[Parsing dns query 9](#_Toc185712931)

[Main 10](#_Toc185712932)

[Start DNS server 11](#_Toc185712933)

[Process queries 12](#_Toc185712934)

[Resolve query 13](#_Toc185712935)

[Authoritative server cache 16](#_Toc185712936)

[Cache 19](#_Toc185712937)

[Testing the server 21](#_Toc185712938)

# 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  
A screenshot of a computer

Description automatically generated

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  
A screen shot of a computer program

Description automatically generated

**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.

A screen shot of a computer code

Description automatically generated

**\_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.

A screen shot of a computer code

Description automatically generated**\_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.A screen shot of a computer program

Description automatically generated

**Accept Connections (\_accept\_connections)**: Manages new client connections and assigns each to a handler.

A screen shot of a computer program

Description automatically generated

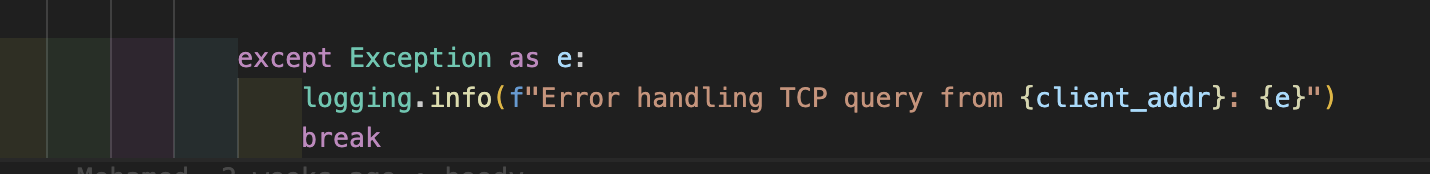
Once a client connects:

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.

A screen shot of a computer code

Description automatically generated

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



A black screen with text

Description automatically generated

Parsing dns query  
A screenshot of a computer program

Description automatically generated

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   
A computer screen shot of a program

Description automatically generated

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 serverA screen shot of a computer

Description automatically generated 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  
A computer screen with text and images

Description automatically generated

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

A screen shot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

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

A screen shot of a computer program

Description automatically generated

**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.

A computer screen shot of text

Description automatically generated

**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>").

A computer screen shot of a computer code

Description automatically generated

**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

A screenshot of a computer program

Description automatically generated

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:

A screen shot of a computer

Description automatically generated

Client side terminal (querying for networking.net):

A computer screen shot of a computer

Description automatically generated

A black screen with white text

Description automatically generatedServer 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:

A black background with white text

Description automatically generated

Response stored in Name cache:  
A computer screen with white text

Description automatically generated

Networking.net record in Authoritative.py file:  
A computer screen shot of a black screen

Description automatically generated