Introduction to Computer Networking
Mail Server Project Report

Heuba Franck Duval, Firstname Lastname
Students ID: S227629, XXXXX

December 3, 2025

1 Introduction

The objective of this assignment was to design and implement a fully functional Mail Server in Java, capable of operating within a distributed network environment simulated via Docker. Our server acts as both a Mail Transfer Agent (MTA) and a Mail Delivery Agent (MDA). It implements three core application-layer protocols: SMTP (for sending and relaying emails), POP3 (for retrieving emails), and IMAP (for synchronizing mailboxes). A key feature of our implementation is the development of a custom raw UDP DNS client to resolve Mail Exchange (MX) records without relying on high-level libraries, strictly adhering to RFC specifications.

2 Software Architecture

We adopted a modular architecture centered around a main server class that delegates protocol handling to specific worker threads.

- **MailServer (Entry Point):** This class initializes the server configuration and the thread pool. It listens on ports 25 (SMTP), 110 (POP3), and 143 (IMAP). Upon accepting a connection, it wraps the client socket in a generic `MailClient` task and submits it to the thread pool.
- **MailClient & Protocols:** The `MailClient` class determines which protocol to instantiate based on the port connected. We implemented three abstract classes extending `MailProtocol`:
  - `SMTPProtocol`: Implements the SMTP state machine. It handles local delivery by writing to the file system and performs remote relaying by acting as a client to other mail servers.
  - `POP3Protocol`: Handles authentication and basic retrieval commands (`USER`, `PASS`, `LIST`, `RETR`, `DELE`).
  - `IMAPProtocol`: A more complex handler supporting folder management and persistent UIDs (`SELECT`, `UID FETCH`, `EXPUNGE`). It parses complex arguments (e.g., `BODY.PEEK`) using Regex.

- **MailDNSClient (Custom DNS):** A standalone utility that constructs raw DNS query packets (headers, questions) byte-by-byte and parses the binary response to extract MX and A records. It handles DNS pointer compression (RFC 1035).
- **MailStorageManager:** A static helper class acting as the data access layer. It synchronizes access to the file system (storage directory) and manages metadata (UIDs, flags like `\Seen`) to ensure data consistency across threads.

Interaction Flow: When a request arrives, `MailServer` spawns a thread. The `SMTPProtocol` uses `MailDNSClient` to resolve destinations. If delivery is local, or if a client uses POP3/IMAP, the protocols interact with `MailStorageManager` to read/write '.eml' files and update the '.metadata' files.

3 Mail forwarding and resolution

In the scenario where `dcd@gembloux.uliege.be` sends an email to `vj@info.uliege.be` (with empty caches), the following steps occur:

1. **SMTP Submission:** The Thunderbird client connects to our mail server at `gembloux.uliege.be` on port 25. The `SMTPProtocol` receives the `RCPT TO:<vj@info.uliege.be>` command.
2. **Domain Analysis:** The server identifies that the recipient domain (`info.uliege.be`) is different from its own (`gembloux.uliege.be`). It initiates the relaying process.
3. **DNS Query Construction:** The `MailDNSClient` reads `/etc/resolv.conf` to find the local nameserver (e.g., 10.0.2.2). It constructs a raw query packet with `QTYPE=MX` (15) for `info.uliege.be`.
4. **DNS Resolution:**
   - The packet is sent over UDP port 53.
   - Since caches are empty, the local DNS (Gembloux) may query the root/parent DNS (Uliege).
   - The DNS responds with the MX record: `mail.info.uliege.be` (Preference 10).
   - Our client parses the binary response, handling pointer compression to read the hostname.
   - A second lookup is performed (Type A) to resolve `mail.info.uliege.be` to its IP address (10.0.3.7).
5. **SMTP Relay:** Our server opens a TCP socket to 10.0.3.7 on port 25. It acts as a client, sending `HELO`, `MAIL FROM`, `RCPT TO`, and `DATA`.
6. **Delivery:** The remote server (`info`) accepts the message and stores it locally via its own `MailStorageManager`.

4 Multi-thread coordination

To ensure robustness and scalability, we avoided creating a new thread for every single connection, which could lead to resource exhaustion (DoS). Instead, we utilized a **Thread Pool** pattern.

- We used `java.util.concurrent.ExecutorService` with a fixed pool size (configurable via command line arguments, typically 10 threads).
- When a socket is accepted, the job is submitted to the pool. If all threads are busy, the request waits in a queue.

Data Synchronization: Since multiple threads (IMAP and SMTP) might access the same mailbox simultaneously, we synchronized critical sections in `MailStorageManager`. Methods modifying the file system or the metadata files (e.g., `saveEmail`, `updateFlag`, `getNextUID`) are marked with the `synchronized` keyword. This guarantees that file writes are atomic and prevents race conditions where two emails could receive the same UID.

5 Limits

While functional, our implementation has certain limitations:

- **Security:** Authentication is performed in plain text. We do not support START-TLS or SSL, making the server vulnerable to packet sniffing.
- **Robustness of Parsing:** The IMAP parser uses Regular Expressions (`java.util.regex`). While sufficient for the project scope, this is fragile against malformed or extremely complex nested commands compared to a tokenizer/lexer approach.
- **DNS Size Limit:** Our UDP DNS client uses a fixed buffer of 512 bytes. If a response exceeds this (setting the Truncation flag), we do not implement the TCP fallback required by RFC 1035, meaning large DNS responses would be lost.

6 Possible Improvements

- **Encryption:** Implementing TLS context to support secure SMTPS and IMAPS connections.
- **Database Integration:** Replacing the file-based storage with a relational database (e.g., SQLite or PostgreSQL) would significantly improve performance for large mailboxes and simplify concurrency management.
- **Spam Filtering:** Adding a simple content filter in the SMTP `DATA` phase to reject messages containing specific keywords.

7 Conclusion

This project was a challenging but rewarding deep dive into application-layer protocols. The most difficult part was implementing the *MailDNSClient*. Manually constructing binary packets and handling DNS compression pointers required a precise understanding of bits and bytes, which we learned to manage effectively using Java's streams and bitwise operators. We heavily relied on online resources to understand the packet structure, notably an article on "DNS Request and Response in Java"[cite: 1].

We also learned how to manage state in stateful protocols like SMTP and IMAP versus the stateless nature of HTTP. Implementing the synchronization logic for the shared file storage taught us the practical importance of thread safety in concurrent systems.

References

[1] Aditya. *DNS Request and Response in Java*. GitConnected, 2023. Available at: `https://levelup.gitconnected.com/dns-request-and-response-in-java-acbd51ad3467`