# Introduction to Computer Networking
# Mail Server Project Report

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

December 16, 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 Concurrency Management: From a Global Lock to a Per-User Model

The first version of our server used a simple but problematic synchronization approach for the `MailStorageManager`. All methods manipulating files (like `saveEmail` or `getMessages`) were declared as `static synchronized`.

## 4.1 The Global Lock Problem

A lock on a static method acts as a global lock for the entire class. This meant that only one storage operation could occur at any given time across the whole server, regardless of the user involved. For example, if an IMAP session for `user1` was reading their messages, the delivery of a new email via SMTP for `user2` was blocked, waiting for the lock to be released. This approach, while safe, created a major bottleneck and severely limited the server's performance and scalability.

## 4.2 New Architecture: Fine-Grained Locking with `ReadWriteLock`

To solve this issue, we completely refactored the concurrency management by adopting a much more performant per-user locking model.

- **Creation of `MailboxLockManager`:** This new central class manages a `ConcurrentHashMap` that maps each username to a `java.util.concurrent.locks.ReadWriteLock` object. This type of lock is ideal for our use case:
  - It allows an unlimited number of readers (**read locks**) to access a mailbox simultaneously, as they do not modify it.
  - It ensures that only one writer (**write lock**) can obtain a lock, and it blocks all other readers and writers while held.
- **Overhaul of `MailStorageManager`:** The class is no longer static. An instance is now created for a specific user (e.g., `new MailStorageManager("user1")`). Each method now acquires the appropriate lock (read or write) via the `MailboxLockManager` before its execution and systematically releases it in a `finally` block, thereby ensuring the absence of deadlocks.

## 4.3 Protocol Implications and Trade-offs

This new architecture has profound implications for the behavior of the protocols.

- **Benefits for IMAP and SMTP:** The IMAP protocol, designed for concurrent access, benefits immensely from this change. Multiple clients for the same user can now read the mailbox simultaneously. Furthermore, email delivery via SMTP is never blocked by IMAP or POP3 reading activities anymore.
- **Trade-off with the POP3 Standard:** The RFC 1939 for POP3 requires an **exclusive session-wide lock** to prevent any concurrent access. Our new model, which locks on a per-operation basis, no longer strictly adheres to this requirement. In return, it prevents a POP3 client from blocking the delivery of new emails, which

is more desirable behavior in a modern server. We therefore made a pragmatic design choice that favors service availability over literal compliance with an older POP3 specification.

# 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`