KU LEUVEN

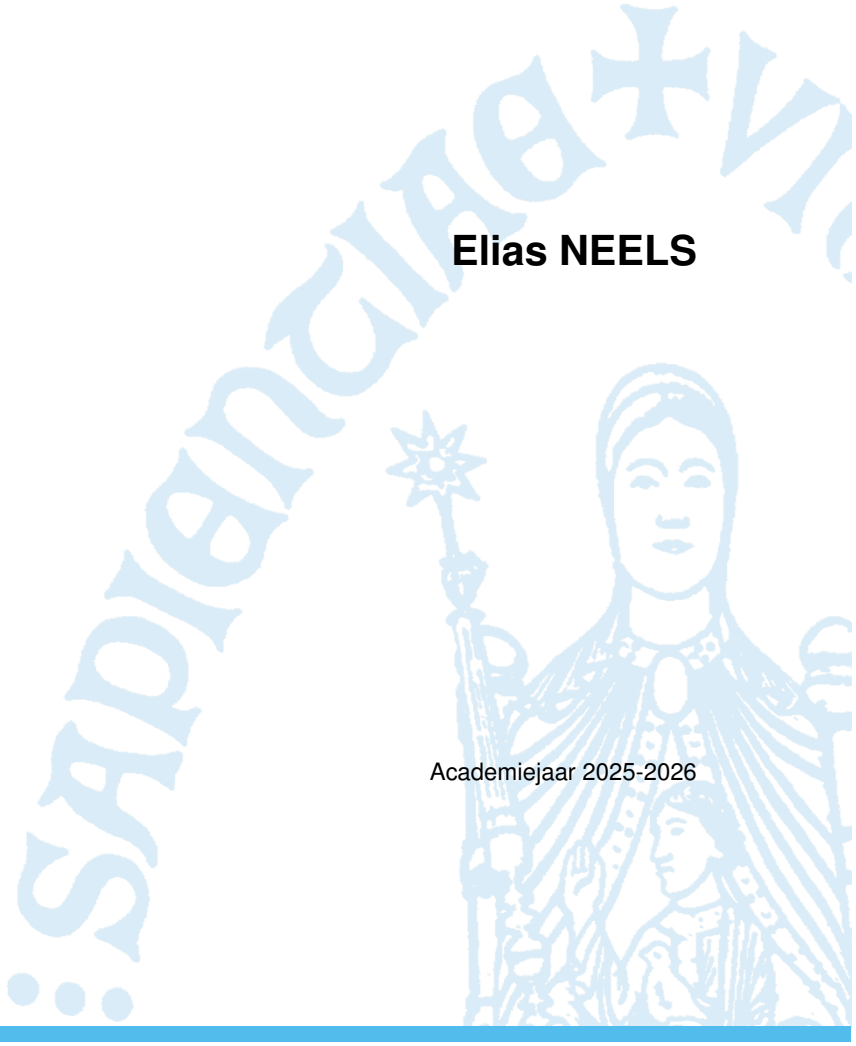
FACULTEIT INDUSTRIËLE
INGENIEURSWETENSCHAPPEN

TECHNOLOGIECAMPUS GENT

Labo Report

Distributed Systems

Elias NEELS



Verantwoordelijke: V. Naessens

Academiejaar 2025-2026

Introduction

This report summarizes the implementation details and architectural decisions made for the distributed systems lab.

1 Lab 1: Sockets

The goal of this lab was to establish foundational knowledge of network communication using Java Sockets. We progressed from a simple text-based protocol to a multi-threaded chat application.

1.1 Knock Knock Protocol

The initial exercise involved implementing a ”Knock Knock”joke server.

- **State Management:** The `KnockKnockProtocol` class acts as the core logic engine. It uses integer constants (WAITING, SENTKNOCKKNOCK, SENTCLUE, ANOTHER) to track the conversation state. This ensures that if a client sends ”Who’s there?” at the wrong time, the server rejects it with a prompt to try again.
- **Concurrency:** The `KnockKnockServer` was initially single-threaded, blocking other users while one joke was in progress. This was upgraded to `KKMultiServer`, which accepts connections in a `while(true)` loop and spawns a new `KKMultiServerThread` for each client.

1.2 Group Chat Application

The second phase was a full-featured group chat system.

1.2.1 Server Architecture

The `Server` class acts as a central relay.

- **Thread Safety:** To handle concurrent users safely, the server uses `synchronized` methods for critical operations like `addUser()` and `removeUser()`. This prevents race conditions where two threads might try to modify the user list simultaneously.
- **Client Threads:** Each connection is wrapped in a `ClientThread`. This thread is responsible for the initial handshake (reading the username) and the main message loop. When it receives a message, it calls `server.broadcast()`, which iterates through all other active threads to relay the text.

1.2.2 Client Architecture

The client features a GUI to display messages.

- **Asynchronous Listening:** The client starts a separate thread to listen for incoming data from the server. This prevents the GUI from freezing while waiting for network input.
- **Protocol parsing:** The client distinguishes between chat messages and control commands (e.g., `USERLIST`) to update the sidebar or chat window accordingly.

2 Lab 2: RMI

The objective of Lab 2 was to refactor the chat application to use Java RMI. This abstracts the low-level details of sockets and allows to invoke methods on remote objects as if they were local.

2.1 Architectural Design

Unlike the socket lab where we parsed raw strings, RMI requires defining interfaces that extend `java.rmi.Remote`.

- **ServerInterface:** This interface defines the actions a client can perform. It includes `registerClient(ClientInterface client, String username)` to join the chat and `sendMessage(String username, String message)` to post messages.
- **ClientInterface (Callback):** This interface defines how the server pushes data back to the client. It includes `receiveMessage(String msg)` and `updateUserList(Set<String> users)`.

2.2 Server Implementation

The `ServerImpl` class extends `UnicastRemoteObject` to export itself as a remote service.

- **Client Storage:** The list of threads was replaced with a `Map<String, ClientInterface> clients`. This maps usernames to their specific remote callback objects.
- **Broadcasting via Callbacks:** When `sendMessage` is called by a client, the server does not just write to a stream. Instead, it iterates through the values in the `clients` map and calls the `receiveMessage()` method on each client's remote reference.
- **Registry Setup:** The `Main` class locates the RMI registry on port 1099 and binds the `ServerImpl` object under the name "ChatServer", making it discoverable.

2.3 Client Implementation

The client also extends `UnicastRemoteObject` because it must act as a server for callbacks.

- **Connection Logic:** On startup, the client looks up "ChatServer" in the registry. It then creates an instance of itself and passes 'this' to the server's `registerClient` method.
- **GUI Integration:** When the server calls `receiveMessage`, the client uses `SwingUtilities.invokeLater()` to safely update the GUI from the RMI thread, ensuring thread safety.

3 Lab 3: JCA/JCE

In Lab 3, I secured communication using the Java Cryptography Architecture (JCA). The goal was to implement a secure channel that ensures Confidentiality, Integrity, and Non-repudiation.

3.1 Cryptographic Primitives

Three core helper classes were implemented to handle different security aspects:

- **Symmetric Encryption (AES):** The `SymmetricEncryption` class wraps AES (128-bit) with ECB mode and PKCS5Padding. It is used to encrypt the large data payload (message + signature) because it is computationally efficient.
- **Asymmetric Encryption (RSA):** The `AsymmetricEncryption` class wraps RSA (2048-bit). It is used specifically to encrypt the AES session key (K), allowing to securely transmit the key to the receiver.
- **Digital Signatures:** The `DigitalSignature` class uses the SHA256withRSA algorithm. It allows the sender to sign the hash of the message, proving that the data has not been altered and confirming the sender's identity.

3.2 Secure Communication Workflow

The `Main.java` file simulates a secure transmission between a Sender and a Receiver using the following steps:

3.2.1 1. Sender

1. **Signing:** The sender hashes the message (SHA-256) and signs it using her private key (retrieved from 'storeA.jks').
2. **Payload Encryption:** Then combines the message, the signature, and her alias into a byte array. Next a random AES session key (K) is generated and this entire payload is encrypted with it.
3. **Key Encapsulation:** Finally, The session key (K) gets encrypted using the receiver's public key (retrieved from the receiver's certificate in the sender's keystore).

3.2.2 2. Receiver

1. **Key Decryption:** Uses the private key (from 'storeB.jks') to decrypt the encrypted session key, recovering the original AES key (K).
2. **Payload Decryption:** Using K, the main payload gets decrypted to extract the message and the signature.
3. **Verification:** The signature gets verified using the sender's public key. If 'verify()' returns true, it is known the message is authentic and integrity is preserved.

Lab 5 & 6: Data Anonymization)

Objective: The goal was to anonymize a ”Fraud Detection”dataset containing 100,000 records of citizens (suspects and non-suspects) so it could be shared with a data analyst for machine learning, ensuring no individual could be re-identified with certainty.

Methodology using ARX Tool:

- **Attribute Configuration:** Direct identifiers (*name, surname, phone_number*) were removed. The attribute *fraud* was marked as sensitive. Quasi-identifiers (*age, sex, zip, car, company_type*) were generalized using hierarchies (e.g., grouping ages into 5/10/20-year intervals, masking zip codes, and grouping car models by brand).
- **Privacy Models:** We applied k**-Anonymity** to prevent re-identification and l**-Diversity)** to prevent attribute disclosure.

Results and Experiments:

1. **Initial Configuration (**$k = 5, l = 2$**):** We achieved an optimal transformation with minimal data loss. The dataset retained high utility, preserving exact zip codes and company types, while generalizing car models to brands and ages to intervals.
2. **Increasing Privacy (**$k = 10$**):** We increased the group size to $k = 10$. This halved the re-identification risk (from 20% to 10%) without suppressing any records (0% data loss). The trade-off was a slight reduction in precision for the *Age* attribute (higher generalization level), but attributes like *Car* and *Zip* remained the same as in $k = 5$.
3. **Utility Trade-off:** Further increasing privacy requirements led to a sharp increase in suppressed records (rising to 7.5%). We concluded that this level of data loss would be bad for the usability of the dataset.

Conclusion

This series of labs provided a comprehensive overview of distributed systems security. We moved from building raw communication channels with Sockets to abstracting them with RMI. We then secured these channels using a robust hybrid encryption protocol involving AES and RSA. Finally, we addressed data privacy at rest through k-anonymity, ensuring a good understanding of cybersecurity.