Samia Esha, Sotiris Emmanouil, Djiedjom Gbonkou

Professor: Dr. Mahbubur Rahman

CSCI 348

Project Title: Secure Client – to - Server Payment System - A Zelle Clone

Abstract:

This paper presents an in-depth exploration of a secure client-to-server payment system inspired by Zelle, including the structure, implementation, and evaluation. The system facilitates secure money transfers between users through a central server, ensuring data confidentiality and integrity using TCP connection and SSL/TLS encryption. We will explore the project's components, tools used, line-by-line code explanation, and experimental evaluations conducted under various scenarios.

1. Introduction:

This project focuses on creating a secure client-to-server payment system similar to Zelle, with the primary aim of enabling users to safely send and receive money through a central server, fortified by SSL/TLS encryption for data protection.

2. Tools Used:

The project utilizes the Java programming language. Additionally, it leverages the Java Secure Socket Extension (JSSE) for implementing SSL/TLS encryption and secure socket communication. The Java Keytool is employed for managing SSL certificates and keystores, which play a vital role in establishing secure connections.

This project was created using several IDE's such as the Eclipse IDE and Visual Studio Code.

3. Implementation Details:

3.1 Client-Side Implementation:

The client-side code of the Zelle Clone project is responsible for establishing a secure connection with the server, sending and receiving transaction-related information, and enabling users to interact with the system. The code is designed to facilitate seamless money transfers and ensure the security of sensitive data using the Java Secure Socket Layer (SSL) framework.

Imports and Initializations

The client-side code begins with a series of import statements that bring in essential classes and packages required for secure communication, input/output handling, and data manipulation.

These include classes related to SSL, security, and I/O operations. Following the imports, the Client class is initialized with private variables that will be used for establishing connections and data transmission.

Constructor for Sending Money (Client):

1. **Key and Trust Store Initialization**: The constructor starts by initializing the client's keystore and truststore. The keystore contains the client's private key and certificate, while the truststore holds certificates of trusted entities. These keystores are crucial for secure SSL communication.
2. **Key and Trust Manager Factories**: KeyManagerFactory and TrustManagerFactory are responsible for managing keys and certificates. The KeyManagerFactory is initialized with the client's keystore, while the TrustManagerFactory is initialized with the truststore. These factories play a pivotal role in setting up the SSL context.
3. **SSL Context Creation:** An SSLContext is created to establish a secure communication channel. It's initialized with the KeyManager and TrustManager from the respective factories. The context ensures encrypted data transmission between the client and the server.
4. **SSL Socket Initialization**: Using the SSLContext, an SSL socket factory is obtained. This factory is then used to create an SSL socket that connects to the server using the provided IP address and port.
5. **Input and Output Streams**: Input and output streams are initialized to handle data communication. The input stream reads user input, while the out stream writes data to the socket's output stream.
6. **Transaction Object Creation**: A Transaction object is instantiated with the user's message and transaction amount. This object will hold the details of the money transfer.
7. **Data Transmission**: The user's username, balance, transaction message, amount, and sender's username are sent to the server via the output stream. This information is crucial for processing the money transfer.

Input Loop

1. **Continuous User Interaction:** Following the data transmission, the code enters an input loop. Within this loop, the user can continuously interact with the system, inputting various commands and details related to money transfers.

Constructor for Receiving Money (Client):

1. **Similar Initialization**: This constructor follows a similar initialization process as the sending constructor. However, instead of sending transaction details, it only sends the user's username and balance to the server. This is intended for clients who are expecting to receive money.

Main Function

1. **User Input Collection**: The main function starts by collecting user information, including username, balance, and the intention to send or receive money. These details are essential for constructing the appropriate type of client.
2. **Sending Client Construction**: If the user intends to send money, a sending client is constructed. This involves providing the IP address, port, user details, message, amount, balance, and sender's username to the Client constructor.
3. **Receiving Client Construction:** If the user intends to receive money, a receiving client is constructed. Similar to the sending client, the receiving client provides necessary information to the Client constructor.

The client-side code forms the bridge between users and the server, enabling secure communication and transaction execution. It establishes an SSL connection, sends transaction details, and allows users to interact with the system. This comprehensive explanation provides insights into the intricate workings of the client-side code and its pivotal role in the Zelle Clone project.

3.2 Server-Side Implementation

The server-side code of the Zelle Clone project plays a pivotal role in enabling secure communication, managing user balances, and facilitating money transfers between clients. Let's delve into the code step by step to understand its functionality and significance.

Imports and Initializations

The code starts with import statements that bring in essential packages and classes required for SSL communication and I/O operations. The Server class is then initialized with private instance variables, such as the server socket, client socket, and flags to manage the server's running state.

Constructor and Initialization

1.**Keystore and Truststore Initialization**: The keystore contains the server's private key and corresponding certificate, while the truststore holds certificates from trusted parties. In this code, the server's keystore and truststore are loaded from files using FileInputStream, and their respective passwords are set.

2.Key and Trust Manager Factories: The KeyManagerFactory and TrustManagerFactory classes are crucial for managing keys and certificates. The KeyManagerFactory is initialized with the server's keystore, enabling the server to authenticate itself to clients. The

TrustManagerFactory is initialized with the truststore, allowing the server to verify client certificates.

3. SSL Context Creation: The SSLContext is responsible for establishing secure connections using SSL/TLS protocols. It is initialized with KeyManagers (obtained from the KeyManagerFactory) and TrustManagers (from the TrustManagerFactory). The SSL context forms the foundation for secure communication between the server and clients.

4.SSL Server Socket Initialization: With the SSLContext ready, an SSLServerSocketFactory is obtained. This factory creates an SSL server socket, which is a secure version of the standard server socket. The server socket is bound to a specific port and awaits incoming client connections.

Connection Listening Loop

1.Accepting Client Connections: The server enters a continuous loop to listen for incoming client connections. When a client attempts to connect, the server accepts the connection and retrieves a corresponding SSLSocket instance.

2**.Client Multiple Thread Handling:** To ensure concurrent communication with multiple clients, a new thread is spawned for each client connection. This thread is dedicated to managing communication with the specific client.

Handling Client Communication

1.Data Input Stream: Within the client thread, a DataInputStream is initialized. This stream reads data from the client's input stream. In this code, it first reads the user's username and then their balance.

2.Balance Storage: The username and balance information obtained from the client are stored in a HashMap named balances. This map serves as a repository for user balances, associating each username with its corresponding balance.

3.Transaction Processing: The client thread proceeds to read additional information from the client, such as the reason for the transaction, the amount, and the receiver's username. With this information, the code performs a transaction by deducting the specified amount from the sender's balance and adding it to the receiver's balance in the balances map.

4.**Communication with Clients:** After processing the transaction, the client thread communicates back to the client, indicating the successful completion of the transaction. It

provides updates on both the sender's and receiver's balances, confirming that the money transfer was executed accurately.

Main Function

1.Server Initialization: The main function initiates the server by creating an instance of the Server class. This instance is configured to listen on a specified port, allowing clients to connect and engage in secure and authenticated transactions.

The server demonstrates how the server establishes secure connections using SSL, manages user balances, processes transactions, and ensures robust communication between clients. By employing SSL/TLS protocols, handling multiple threads, and maintaining balance records, the server-side code is a fundamental component of the Zelle Clone project's functionality.

3.3 Transaction Class Explanation:

The Transaction class in the Zelle Clone project encapsulates the concept of a financial transaction between users. This class is responsible for representing transaction details, such as the transaction message and the transaction amount. Let's delve into the code step by step to understand its functionality and role in the project.

Class Declaration and Attributes

The Transaction class is declared with the attributes necessary to define a transaction:

private double money: This attribute represents the amount of money involved in the transaction.

private String message: This attribute stores a message associated with the transaction.

Constructor: The class contains a parameterized constructor that initializes the Transaction object with the provided message and amount. The constructor sets the message and money attributes to the values passed as arguments.

Getters:

public double getAmount(): This method is a getter for the money attribute. It allows external code to retrieve the amount of money involved in the transaction.

public String getMessage(): This method is a getter for the message attribute. It enables external code to retrieve the transaction message.

Example Usage

When a client initiates a transaction by sending money to another user, the client-side code creates a Transaction object using the provided message and amount. This Transaction object is then serialized and sent over the SSL/TLS-secured connection to the server, which processes the transaction details and updates the users' balances accordingly.

4.SSL/TLS Handshake Process

The SSL/TLS handshake is a fundamental aspect of secure communication in the Zelle Clone project. It plays a pivotal role in establishing a secure channel for transmitting sensitive transaction data between the client and server. Let's break down the SSL/TLS handshake process as implemented in the provided code.

Client Side Handshake

In the client-side code (Client.java), the handshake process involves loading necessary certificates, initializing SSL context, creating an SSL socket, and exchanging relevant transaction data.

1.Loading Keystore and Truststore: The client loads its keystore and truststore using Java's KeyStore class. The keystore contains the client's private key and certificate, while the truststore holds certificates of trusted entities, such as the server's certificate

2.Creating KeyManagerFactory and TrustManagerFactory: The client initializes a KeyManagerFactory with its keystore. The KeyManagerFactory manages the client's credentials for authentication.The client also initializes a TrustManagerFactory with its truststore. The TrustManagerFactory manages certificates for verifying the authenticity of the server.

3.Initializing SSL Context: An SSLContext is created, and it's initialized with the KeyManagerFactory and TrustManagerFactory. This SSLContext encapsulates the configuration needed for SSL/TLS communication.

4.Creating SSL Socket: Using the initialized SSLContext, the client creates an SSLSocket to establish a secure connection to the server. The SSLSocket will be responsible for encrypting and decrypting data.

5.Sending Data to Server: The client sends user-related data (username, balance, transaction details, amount, and sender information) to the server using the SSLSocket's DataOutputStream. This data is written to the output stream and will be encrypted using SSL/TLS.

Server Side Handshake:

1.Accepting Client Connection: The server initializes an SSLServerSocket that listens for incoming connections from clients. When a client connects, the SSL handshake process begins.

2.Loading Keystore and Truststore: Similar to the client, the server loads its keystore and truststore using the KeyStore class.

3.Creating KeyManagerFactory and TrustManagerFactory: The server initializes a KeyManagerFactory with its keystore, which manages the server's credentials. It also initializes a TrustManagerFactory with its truststore, which manages certificates for client authentication.

4.Initializing SSL Context: An SSLContext is created on the server side and is initialized with the KeyManagerFactory and TrustManagerFactory. This context defines the security parameters for the SSL/TLS communication.

5.Client Authentication and Data Reception: When a client connects, the server reads the user's information (username and balance) sent by the client using the DataInputStream. The server stores this information in a HashMap for reference.

6.Handling Data Exchange: The server then reads the encrypted transaction data (reason, amount, receiver) sent by the client. It processes the transaction, updates the user balances, and communicates the results back to the client using the SSLSocket's DataOutputStream.

5. How to run the project :

Server Side (Server.java):

To run the server, these steps need to be taken:

Compile the Server.java file: javac Server.java

Run the server: java Server.java

The server will start listening for incoming connections on the specified port.

You will see a message Server started. Waiting for Client...

Client Side (Client.java):

To run the client, these steps need to be taken:

Compile the Client.java file: javac Client.java

Run the client: java Client.java

(Please ensure that you enter the correct IP address of the machine where the server is running. The IP address entered by the client should match the IP address of the server machine for successful communication.)

Receiver Side:

The receiver will join first. On the client side, it will ask to put a username and password. The password is '1223'. Then it will ask about your balance and a question if you want to send money? (YES/NO). As we are on the receiver side, the user will say "NO". Now on the server side, it will show a message "username1 has connected" with balance '_' , indicating a successful connection to the server has been made.

Sender Side:

After running client.java from the sender side,it will ask the same questions and now user will say "YES" . Then it will ask to "enter the username of the receiver" and "the amount you want to send" and the transaction details. When it successfully runs, there will be a message "Connected" on the client side and "Money will be sent" , indicating a successful connection to the server has been made.

On the server side, now it shows "username2 is connected with balance __" And, a message will be shown "Amount being sent from user: username2 to user: username1 is ____" . It will also show the current and updated balance of the users with the message "Transaction successful" .

Contribution:

Each member contributed 33.33% for this project.

References:

https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/doors/9.6.1?topic=tls-ssl-keystores-certificates

https://www.ibm.com/docs/en/was/9.0.5?topic=communications-secure-using-ssl