|  |
| --- |
| MTU Kerry |
| Design and Implementation of a Secure, Concurrent, Client-Server Short Message Protocol (SMP) |
| Module: Distributed Computing  Name: Luke Foley  T-Number: T00224345 |

# Summary

This project developed a secure, concurrent client-server application using a Short Message Protocol (SMP) using the Java Socket API. The system allows for user login, message upload, message download, and logoff. A text-based protocol was designed for this functionality. The application was built with Java, consisting of a three-layer architecture and multi-threading for concurrency, and SSL for secure communication.

# Table of Contents

[Summary 1](#_Toc195280645)

[Table of Contents 1](#_Toc195280646)

[Introduction 2](#_Toc195280647)

[Protocol Design 3](#_Toc195280648)

[Overview and Objective 3](#_Toc195280649)

[Message Format 3](#_Toc195280650)

[Message Definitions 3](#_Toc195280651)

[Login 3](#_Toc195280652)

[Upload Message 3](#_Toc195280653)

[Download Message 4](#_Toc195280654)

[Download All Messages 4](#_Toc195280655)

[Log Off 4](#_Toc195280656)

[Description of Implementation of Functional Requirements 5](#_Toc195280657)

[Login 5](#_Toc195280658)

[Upload 5](#_Toc195280659)

[Download 5](#_Toc195280660)

[Download All 5](#_Toc195280661)

[Log Off 6](#_Toc195280662)

[Sequence Diagram of Client-Server Communication 6](#_Toc195280663)

[Protocol Recap 7](#_Toc195280664)

[Application Design 7](#_Toc195280665)

[Design Overview 7](#_Toc195280666)

[UML Class Diagram 8](#_Toc195280667)

[Overall View 8](#_Toc195280668)

[Client-Side UML 9](#_Toc195280669)

[Server-Side UML 10](#_Toc195280670)

[Description of IPC 11](#_Toc195280671)

[Layer Sequence Diagrams 12](#_Toc195280672)

[Presentation Layer 12](#_Toc195280673)

[Application Layer 12](#_Toc195280674)

[Session Layer 13](#_Toc195280675)

[Application Implementation 14](#_Toc195280676)

[Starting Point Classes Used 14](#_Toc195280677)

[Login 14](#_Toc195280678)

[Upload 16](#_Toc195280679)

[Download 18](#_Toc195280680)

[Download All 19](#_Toc195280681)

[Log Off 21](#_Toc195280682)

[SSL Implementation 23](#_Toc195280683)

[Server-Side 23](#_Toc195280684)

[Client-Side 24](#_Toc195280685)

[Wireshark Capture of SSL 25](#_Toc195280686)

[Running Application 26](#_Toc195280687)

[Screenshots of Running Application 26](#_Toc195280688)

[Instructions for Running the Application 28](#_Toc195280689)

[Conclusion 31](#_Toc195280690)

# Introduction

This document outlines the design and implementation of a secure, concurrent client-server Short Message Protocol (SMP) application using the Java Socket API. The goal of this project was to create a system that allows multiple clients to securely interact with a server for basic messaging operations. These operations are user login, upload messages, download messages, and log off.

The project involved defining a text-based communication protocol. A layered application architecture was designed, consisting of three layers: a Presentation Layer, a Session Layer, and an Application Layer.

This document provides the details for protocol specifications, the application's design with UML and sequence diagrams, implementation details for each functional requirement, implementation of SSL, and instructions for running the application.

# Protocol Design

## Overview and Objective

The Short Message Protocol (SMP) is designed to facilitate communication between the client and server by allowing users to:

1. Login.
2. Upload a short message to the server.
3. Download a specific message from the server.
4. Download all messages stored on the server.
5. Log out

## Message Format

All messages are in the format of:

***<request\_code/response\_code>:<parameter>,<parameter>,<parameter>…***

Request/Response codes are separated from parameters with a colon “:”. The request and response codes are integer values, and parameters are strings separated by commas “,”.

## Message Definitions

### Login

**Description**: The user logs into the server with a username and password.

**Code**: 100

**Message Parameters:**

* username (text)
* password (text)

**Request Message:** 100:<username>,<password>

**Response Messages:**

* 101: Welcome <username>
* 201: Invalid username or password

### Upload Message

**Description**: The user uploads a message to the server.

**Code**: 200

**Message Parameters:**

* Message (text)

**Request Message:** 200:<message>

**Response Messages:**

* 102: Upload Successful. Message ID: <id>
* 204: Invalid message format: <request>. Must be: 200:<message>
* 205: Unable to upload message: <error message>

### Download Message

**Description**: The user downloads a specified message from the server with an ID.

**Code**: 300

**Message Parameters:**

* Message ID (text)

**Request Message:** 300:<message\_id>

**Response Messages:**

* 103: <message>
* 202: Unable to find message with ID: <message\_id>
* 204: Invalid message format: <request>. Must be: 300:<message\_id>
* 205: Unable to download message: <error message>

### Download All Messages

**Description**: The user downloads all messages from the server.

**Code**: 4

**Request Message:** 400:

**Response Messages:**

* 104: <message>|<message>|<message>…
* 204: Invalid message format: <request>. Must be: 400:
* 205: Unable to download messages: <error message>

### Log Off

**Description**: The user logs off from the server.

**Code**: 5

**Request Message:** 500:

**Response Messages:**

* 105: Log Off Successful. Goodbye <username>.
* 204: Invalid message format: <request>. Must be: 500:

## Description of Implementation of Functional Requirements

### Login

* The client takes in the username and password input from the login GUI.
* The GUI calls the client helper method login(), which sends the request to the server via stream socket in the correct message format. (e.g. 100:peter,123)
* The server thread receives the request, splits it by ":", and parses the request code (100).
* The username and password are passed to the correct handler method, handleLogin()
* If the username and password are not empty, the server responds with code 101 and a welcome message and assigns this session username variable. Otherwise, it responds with code 201, with a failure message
* The login GUI receives the server response. If successful (101), it opens the main app GUI. If failed, it displays an error message.

### Upload

* The client takes in the message input from the GUI.
* The GUI calls the client helper method uploadMessage(), sending the request to the server in the correct message format. (e.g. 200:Hello!)
* The server thread receives the request, splits it by ":", and parses the request code (200).
* The message is passed to the correct handler method handleUpload().
* The server generates a message ID and writes the message to the messages file.
* The server responds with code 102, and a success message.
* The client GUI receives the response. If successful, it displays the response. If failed, it displays an error message.

### Download

* The client takes in the message ID input from the GUI.
* The GUI calls the client helper method downloadMessage(), which sends the request to the server in the correct message format. (e.g. 300:5)
* The server thread receives the request, splits it by ":", and parses the request code (300).
* The message is passed to the correct handler method handleDownload().
* The server reads the messages text file and searches for the message with the given ID.
* If found, the server responds with code 103, and the message data.
* If not found, it responds with code 202, and a “Not Found” message.
* The client GUI receives the response. If successful, it displays the requested the message data. If failed, it displays an error message.

### Download All

* The client triggers the download all messages action from the GUI.
* The GUI calls the client helper method downloadAllMessages(), which sends a request to the server in the correct message format. (400:)
* The server thread receives the request, splits it by ":", and parses the request code (400).
* The request is passed to the correct handler method handleDownloadAll().
* The method reads all of the messages from the messages text file and joins them with a pipe symbol “|”.
* The server responds with code 104, and the messages data (<message1>|<message2>|…).
* The client GUI receives the response, splits the message string by "|", and displays the messages. If failed, it displays an error message.

### Log Off

* The client triggers the log off action from the GUI.
* The GUI calls the client helper function logoff() which sends the request a request to the server in the correct message format. (500:)
* The server thread receives the request, splits it by ":", and parses the request code (500)
* The request is passed to the correct handler method handleLogOff().
* The server clears the session user variable, and the server responds with code 105, and a confirmation message.
* The client GUI receives the response, displays a success message, closes the socket, and disposes of the window.

## Sequence Diagram of Client-Server Communication

Below is a sequence diagram of a typical session where the user logs in, uploads a message, downloads messages, and logs off, including possible error messages.

A screenshot of a computer screen

AI-generated content may be incorrect.

## Protocol Recap

|  |  |  |
| --- | --- | --- |
| Message | Client Request | Server Responses |
| Login | 100 | 101/201 |
| Upload | 200 | 102/204/205 |
| Download | 300 | 103/202/204/205 |
| Download All | 400 | 104/204/205 |
| Log Off | 500 | 105/204 |

# Application Design

## Design Overview

The SMP app consists of three layers. These are presentation, session, and application. A full UML class diagram view of these layers and classes can be seen in the following section.

**Presentation:**

The presentation layer has the sole responsibility of collecting user input and displaying server responses. The layer consists of GUI forms built using IntelliJ’s Swing UI Designer Plugin.

Classes:

* LoginForm.java (client login screen)
* MainForm.java (main client interface for uploading, and downloading messages)
* ServerForm.java (server log view)

**Session**:

The session layer is responsible for managing communication sessions between the client and the server. The MyStreamSocket class is a wrapper class that uses a TCP stream to send and receive text messages via sockets. The class also adds security with SSL which will be explained in the SSL implementation section. Both the client-side and server-side make use of this class.

Classes:

* MyStreamSocket.java

**Application**:

The application layer contains the core application and business logic. This layer is for processing user requests and managing server-side processes.

Classes:

* ClientHelper.java (helper method to encapsulate client-side protocol logic)
* ServerThread.java (manages individual client connections, sending responses back to the client)
* Server.java (initialises the server, accepts client connections, spawns threads for concurrency handling)

## UML Class Diagram

### A screenshot of a computer AI-generated content may be incorrect.Overall View

### Client-Side UML

A screenshot of a computer

AI-generated content may be incorrect.

### Server-Side UML

A screenshot of a computer

AI-generated content may be incorrect.

## Description of IPC

The following is a description of the process communication flow in the SMP app:

1. **Server Initialisation**

* The Server’s main() method is executed to start the server.
* The server loads its keystore, which contains its SSL certificate and private key.
* The server creates an SSLServerSocket to listen for incoming client connections.

1. **Server Listening:**

* The server enters a forever loop, waiting for client connection requests.

1. **Client Initialisation**

* When the client application starts (LoginForm), a ClientHelper object is created.
* The ClientHelper creates a MyStreamSocket to connect to the server.
* The MyStreamSocket loads the client's truststore, which contains certificates of trusted servers. An SSLSocket connection is then established to the server.

1. **Client Connection Request:**

* The client's SSLSocket initiates the SSL/TLS handshake with the server's SSLServerSocket.

1. **Server Connection Acceptance:**

* The server's SSLServerSocket accepts the client's connection request and creates an SSLSocket for communication with that client.
* The server creates a MyStreamSocket, which wraps the SSLSocket, to communicate with the client.
* The server spawns a new ServerThread to handle communication with this client concurrently.

1. **Client Request:**

* The user interacts with the client's GUI (logging in, uploading messages, etc.).
* The client's GUI calls a method in ClientHelper (login(), uploadMessage(), etc.) to initiate the action.
* ClientHelper formats the request according to the protocol.
* ClientHelper sends the request string to the server using MyStreamSocket.sendMessage().
* MyStreamSocket transmits the request string over the secure SSL connection.

1. **Server Request Processing:**

* The server's MyStreamSocket receives the request string.
* The ServerThread receives the request string from MyStreamSocket.
* The ServerThread parses the request string and calls the appropriate handler method (handleLogin(), handleUpload(), etc.).
* The handler performs the server-side logic (validate credentials, upload message, download message, etc.).

1. **Server Response:**

* The handler method in the ServerThread creates a formatted response string.
* The ServerThread sends the response string to the client using MyStreamSocket.sendMessage().
* The server's MyStreamSocket transmits the response string over the SSL.

1. **Client Response Handling:**

* The client's MyStreamSocket receives the response string.
* The ClientHelper receives the response string from MyStreamSocket.
* The ClientHelper returns the response to the client GUI.

1. **Client Output:**

* The client GUI receives the response string from the ClientHelper.
* The client's GUI displays the response to the user.

## Layer Sequence Diagrams

### Presentation Layer

A diagram of a software process

AI-generated content may be incorrect.

### Application Layer

A diagram of a server thread

AI-generated content may be incorrect.

### Session Layer

A diagram of a process

AI-generated content may be incorrect.

# Application Implementation

## Starting Point Classes Used

* MyStreamSocket.java
* EchoServerThread.java
* EchoServer3.java
* EchoClientHelper2

## Login

**Presentation Layer – LoginForm.java:**

A white background with text

AI-generated content may be incorrect.

A login form is displayed, and the user is prompted to enter username and password. On login button click, a new client helper is instantiated with the defined hostname and port number. The helper.login() method is then called, passing in the username and password. If response is successful, the main GUI is launched, passing in the helper instance to ensure it remains in the same session

**Application Layer – ClientHelper.java**

A screen shot of a computer code

AI-generated content may be incorrect.

When the ClientHelper is instantiated, a new stream socket is created, which requests a connection to the server. When the helper login() method is called, the message string is built and tagged with the correct request code. In this instance it is 100 for login. The message is then sent via mySocket.sendMessage(), and the response is returned via mySocket.receiveMessage().

A white background with black text

AI-generated content may be incorrect.

A simple MessageCodes class with definitions for each code is used throughout the system.

**Session Layer – MyStreamSocket.java**

**A screenshot of a computer code

AI-generated content may be incorrect.**

The login request then passes through the stream socket class and gets written to the output stream of the TCP socket. The server’s response is then read from the input stream of the TCP socket and returned to the client.

**Application Layer – ServerThread.java**

**A computer code with colorful text

AI-generated content may be incorrect.**

The run method within the server thread (which was created by the server on connection request) receives the login request string.

A white background with text

AI-generated content may be incorrect.

The request string is split by “:” and “,” to extract the request code and parameters. A switch statement reads the request and passes it to the correct handler, in this case the login handler.

A computer screen shot of text

AI-generated content may be incorrect.

The login handler validates the credentials provided (will always be accepted for simplicity), sets the session currentUser variable, and sends a login success message back to the client (101).

## Upload

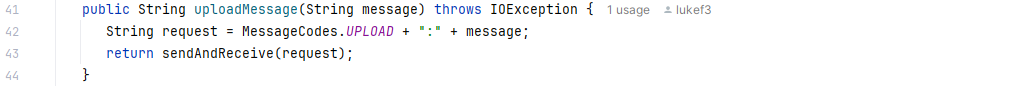
**Presentation Layer – MainForm.java:**

A white background with text

AI-generated content may be incorrect.

From the main form, the client sends a message to the server with the helper function uploadMessage(). The field is validated to ensure that the message does not contain the “|” pipe symbol which is used to divide each message in the download all function.

**Application Layer – ClientHelper.java**

****

As with the login function, the ClientHelper instance receives the message from the UI, attaches the correct message code, in this case 200, and sends it to the server.

**Session Layer – MyStreamSocket.java**

The message then passes through the stream socket with the send and receive methods.

**Application Layer – ServerThread.java**

A screenshot of a computer code

AI-generated content may be incorrect.

The session server thread then receives the request, parses the request code, and passes the request data to the upload handler. The upload handler then writes the message to the messages text file in the format “id-[*current time]-[username] : message”*. If the message is successfully written to the file, the server responds with an upload success message (code 102). If unsuccessful, the server responds with an error message (code 205).

## Download

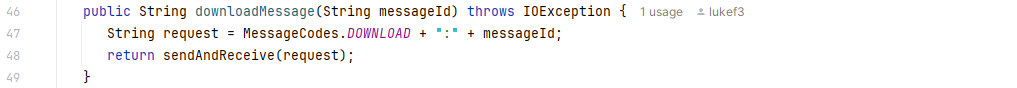
**Presentation Layer – MainForm.java**

A screenshot of a computer code

AI-generated content may be incorrect.

From the main form, the client enters a message ID to download. On download button click, the ID field is validated, and the helper method downloadMessage() is called, passing in the ID. On receiving a response, the response is displayed in the text area.

**Application Layer – ClientHelper.java**

****

The ClientHelper instance receives the message ID from the UI, attaches the correct message code (300), and sends the message to the server through the stream socket.

**Session Layer – MyStreamSocket.java**

The message then passes through the stream socket with the send and receive methods.

**Application Layer – ServerThread.java**

A screenshot of a computer code

AI-generated content may be incorrect.

The session server thread receives the request, parses the request code, and passes the request data to the download handler. The download handler then reads each line of the messages file, finds the message with the matching ID, and returns the message with a success code 103. The function also handles messages not found, and IO exceptions, with appropriate responses.

## Download All

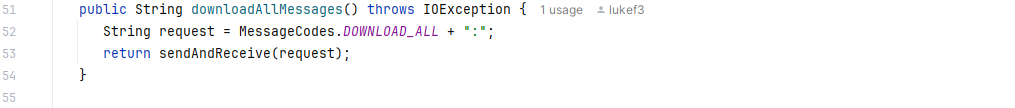
**Presentation Layer – MainForm.java**

**A screenshot of a computer code

AI-generated content may be incorrect.**

From the main form, the client clicks the download button, and the helper method downloadAllMessages() is called. On receiving a successful response, the messages are split by the pipe “|” delimiter and displayed individually in the text area. If unsuccessful, an error message is displayed.

**Application Layer – ClientHelper.java**

****

The ClientHelper instance sends the correct message code (400) to the server via the stream socket and returns the response to the client UI.

**Session Layer – MyStreamSocket.java**

The message then passes through the stream socket with the send and receive methods.

**Application Layer – ServerThread.java**

**A computer code with many text

AI-generated content may be incorrect.**

The session server thread receives the request, parses the request code, and passes the request to the download all messages handler. The handler reads all messages in the messages file, joining each message by a pipe “|” delimiter. The server responds with the delimited messages and the appropriate message code (104). I/O exceptions are also caught and transmitted to the client with the appropriate error code.

## Log Off

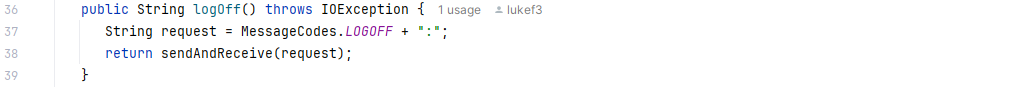
**Presentation Layer – MainForm.java**

**A computer screen shot of text

AI-generated content may be incorrect.**

From the main form, the client clicks the log off button, and the helper method logoff() is called. On receiving a successful response, the UI displays a confirmation message, calls the helper method done() which closes the socket, and disposes the window. On receiving an unsuccessful response, an error is displayed.

**Application Layer – ClientHelper.java**

****

****

The ClientHelper instance sends the correct message code (500) to the server via the stream socket and returns the response to the client UI.

**Session Layer – MyStreamSocket.java**

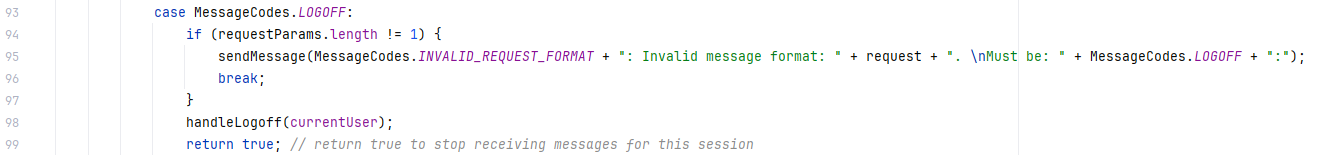
The message then passes through the stream socket with the send and receive methods.

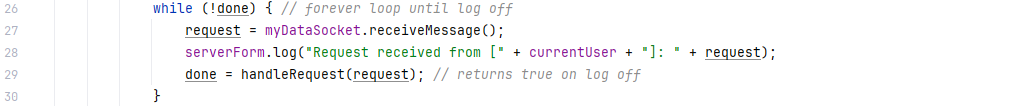
**Application Layer – ServerThread.java**

**A group of colorful text

AI-generated content may be incorrect.**

The session server thread receives the request, parses the request code, and passes the request to the log off handler. The handler sets the session currentUser variable to blank, and sends a log off success message to the client with the appropriate code (105).

****

****

Once the log off handler has executed successfully, the log off case returns true, which stops this thread forever loop that receives messages.

## SSL Implementation

The server keystore and client trust store were generated via command line with **java keytool.**

### Server-Side

A screenshot of a computer

AI-generated content may be incorrect.

The keystore path, keystore password and cert password are first defined. When the server is run, the server loads its keystore (server.jks) with the keystore password. A KeyManagerFactory is then initialised with the server’s keystore and the private key password. An SSL context is then created with the TLSv1.2 protocol. The context is initialised with the key manager. An SSLServerSocketFactory is then created from the SSL context, allowing SSLServerSockets to be created.

An SSLServerSocket is created using the factory, listening on the specified port. The server starts an infinite loop to continuously listen for incoming connections. When a connection request is received, the connection request is accepted, and an SSLSocket is created, establishing a secure communication channel with the client. After accepting the connection, a MyStreamSocket instance is created, which wraps the SSLSocket and provides methods for sending and receiving messages. A new ServerThread is then created, passing in the MyStreamSocket to allow use of the send and receive methods.

### Client-Side

A screenshot of a computer screen

AI-generated content may be incorrect.

The MyStreamSocket class was refactored to use SSLSockets instead of regular sockets, and the constructor was altered to encapsulate loading the client’s trust store and SSL setup. The constructor first instantiates a keystore to load the trust store file with the trust store password. A TrustManagerFactory is then instantiated to manage the trusted certs. An SSLContext is then created with the same cryptographic protocol as the server, TLSv1.2. The SSLContext is then initialised with the trust managers from the TrustManagerFactory. An SSLSocketFactory is created from the SSLContext so that SSLSockets can be created. An SSLSocket is then created from the factory, and the I/O streams are set. The rest of the MyStreamSocket class remains unchanged.

### Wireshark Capture of SSL

A screenshot of a computer

AI-generated content may be incorrect.

The above is a Wireshark capture of the SSL handshake in action, on port 12345.

1. TCP Connection gets made.
2. **Client Hello**: The client kicks off the handshake process.
3. **Server Hello, Certificate, Key Exchange, Server Hello Done**: The server responds with its details and certificate.
4. **Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message**: The client processes the server info, agrees on encryption, and sends its part, now encrypted.
5. **Server Change Cipher Spec, Encrypted Handshake Message**: The server confirms the encryption switch.
6. **Application Data**: Secure communication is established, and the encrypted application messages start flowing.

# Running Application

## Screenshots of Running Application

The following is a collection of screenshots in which 2 users concurrently connect to the server. Both users upload a message, download the other user’s message, download all messages, and finally log off. Session specific outputs can be seen from the user’s text areas. Full server logs can be seen from the server log window.

A screenshot of a login box

AI-generated content may be incorrect.A screenshot of a login box

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

## Instructions for Running the Application

**Build Details:  
IDE:** IntelliJ IDEA 2024.3.4.1 (Ultimate Edition)  
**Plugins:** SwingUI Designer (not needed to run the application)  
**Java Version:** OpenJDK 23

1. From Command Line or PowerShell, ensure that you are in the correct directory: “\SMP\_APP”  
   A screenshot of a computer

   AI-generated content may be incorrect.
2. Compile all java files in the directory with “javac \*.java”  
   A screenshot of a computer

   AI-generated content may be incorrect.
3. Run the java file “APPTEST.java” with “java .\APPTEST.java”:  
     
     
   Running this java file will automatically launch the server, two client login windows, and a server log window:  
   A screenshot of a login screen

   AI-generated content may be incorrect.A screenshot of a login box

   AI-generated content may be incorrect.A screenshot of a computer

   AI-generated content may be incorrect.

From here the application can be tested and actions for the two clients can be performed concurrently.

# Conclusion

In conclusion, this project successfully resulted in the secure and concurrent client-server application built around the SMP protocol that was designed for this application. The process of building this application was a valuable learning experience, helping me become very familiar with Java Sockets, and multi-threading, and solidifying my understanding of communication protocols and SSL security. Utilising the initial starting code and sticking to a layered architecture made it a manageable and enjoyable process.

Project Goals Achieved:

* **Concurrency:** The server effectively manages multiple clients simultaneously using threads.
* **Security:** Implemented SSL through SSLSockets, keystores, and truststores.
* **Functionality:** All necessary SMP operations (login, upload, download, log off) are fully implemented and functional via the client-side Swing GUI.

While the application has much room for improvement, the application meets the assignment requirements and demonstrates a working solution of secure, concurrent, client-server communication.