A picture containing text

Description automatically generated

Distributed Computing

Joseph Diggins – T00172862

Simple Message Protocol

# Summary

The client-server project detailed in this document encompasses a comprehensive exploration of modern communication paradigms, ranging from traditional socket-based communication to secure SSL protocols and the utilisation of WebSockets for real-time interactions. The project addresses the diverse needs of both graphical and command-line interfaces, providing a versatile solution for users across various platforms. At its core, the project establishes a robust communication protocol between the server and multiple clients, ensuring seamless data exchange while prioritising security and reliability. Through meticulous implementation, the project exemplifies the integration of advanced technologies to create a scalable and efficient client-server architecture.

The project begins with a thorough Protocol Design, delineating the communication protocol for client-server interactions. This section defines message formats for key functionalities such as Log On, Upload Message, Download Message(s), and Log Off. Additionally, the integration of SSL protocols ensures secure communication channels, safeguarding sensitive data transmission between clients and the server.

Subsequently, the RFC Documentation section formalises specifications following the RFC standard, providing detailed insights into the objectives, sequence diagrams, and message formats for each functionality. This section serves as a comprehensive reference guide, offering a clear understanding of the communication protocols and system requirements.

Moving on to Software Architecture & Implementation, the project showcases the execution of requested functionalities, encompassing both graphical and command-line interfaces. The integration of WebSockets enables real-time communication, facilitating dynamic interactions between clients and the server. The implementation of a GUI (Graphical User Interface) enhances user experience, while the CLI (Command-Line Interface) caters to users preferring a text-based interaction paradigm.

Table of Contents

[Summary 2](#_Toc162299108)

[1. Introduction 4](#_Toc162299109)

[2. Protocol Design 5](#_Toc162299110)

[2.1. Protocol for Client-Server Communication 5](#_Toc162299111)

[2.2. Format of Each Message Type 5](#_Toc162299112)

[2.2.1. Log On 5](#_Toc162299113)

[2.2.2. Upload Message 5](#_Toc162299114)

[2.2.3. Download Message(s) 5](#_Toc162299115)

[2.2.4. Log Off 6](#_Toc162299116)

[2.2.5. Server Response 6](#_Toc162299117)

[2.3. Pseudo-code for Functional Requirements 6](#_Toc162299118)

[2.3.1. Log On 6](#_Toc162299119)

[2.3.2. Upload Message 6](#_Toc162299120)

[2.3.3. Download Message(s) 6](#_Toc162299121)

[2.3.4. Log Off 7](#_Toc162299122)

[2.4. Inter-process Communication 7](#_Toc162299123)

[3. RFC Documentation 7](#_Toc162299124)

[3.1. Introduction 7](#_Toc162299125)

[3.2. Sequence Diagram 8](#_Toc162299126)

[3.3. Message Formats 8](#_Toc162299127)

[3.3.1. Log On 8](#_Toc162299128)

[3.3.2. Upload Message 9](#_Toc162299129)

[3.3.3. Download Message(s) 9](#_Toc162299130)

[3.3.4. Log Off 10](#_Toc162299131)

[3.4. UML Diagrams 11](#_Toc162299132)

[3.4.3. Implementation of Functions Required 13](#_Toc162299133)

[4. Software Architecture & Implementation 14](#_Toc162299134)

[4.1. Overview & Objectives 14](#_Toc162299135)

[4.2. Application Layer 14](#_Toc162299136)

[4.3. Presentation Layer 15](#_Toc162299137)

[4.4. Service Layer 15](#_Toc162299138)

[4.5. Execute Requested Functionalities (Command Line) 16](#_Toc162299139)

[4.5.1. Log On 16](#_Toc162299140)

[4.5.2. Upload Message 17](#_Toc162299141)

[4.5.3. Download Message(s) 17](#_Toc162299142)

[4.5.4. Log Off 18](#_Toc162299143)

[4.6. Execute Requested Functionalities (GUI) 19](#_Toc162299144)

[4.6.1. Connect to Server 19](#_Toc162299145)

[4.6.2. Log On 19](#_Toc162299146)

[4.6.3. Upload Message 20](#_Toc162299147)

[4.6.4. Download Message(s) 20](#_Toc162299148)

[4.6.5. Log Off 21](#_Toc162299149)

[4.7. Concurrent Access to the Server 22](#_Toc162299150)

[4.8. SSL 24](#_Toc162299151)

[4.8.1. Overview 24](#_Toc162299152)

[4.8.2. Generating Keys and Certs 25](#_Toc162299153)

[4.9. Handling Errors 27](#_Toc162299154)

[4.9.1. User Input 27](#_Toc162299155)

[4.9.2. Server 27](#_Toc162299156)

[4.9.3. Network 27](#_Toc162299157)

[5. Conclusion 28](#_Toc162299158)

[6. References 29](#_Toc162299159)

# Introduction

The client-server project presented in this document encompasses the design, implementation, and documentation of a robust communication system. Beginning with Protocol Design, the project outlines the communication protocol between the client and server, specifying the format for each message type and providing pseudo-code for functional requirements. The RFC Documentation section formalises specifications following the RFC standard, ensuring clarity and consistency in communication protocols. Finally, the Software Architecture & Implementation section details the execution of requested functionalities, demonstrating the implementation of key features such as Log On, Upload Message, Download Messages, Download A Message, and Log Off.

# Protocol Design

## Protocol for Client-Server Communication

The protocol will be based on a request-response model, where the client sends requests to the server and the server responds accordingly.

Communication will be initiated by the client for actions like logging on, uploading messages, downloading all messages, and logging off.

The server will validate requests, perform necessary actions, and respond with appropriate status codes or data.

## Format of Each Message Type

### Log On

**Format:** logon <username> <password>

**Description:** The client uses this message type to authenticate itself with the server. It includes the username and password of the client.

**Example:** logon joseph\_diggins mypassword

### Upload Message

**Format:** uploadmsg <message>

**Description:** The client uses this message type to upload a message to the server. The message could be any data or content that the client wants to store on the server.

**Example:** uploadmsg Hello

### Download Message(s)

**Format:** downloadmsgs <OPTIONAL message\_id>

**Description:** The client uses this message type to request downloading all messages stored on the server. The server will respond with all available messages. If a message\_id is passed the server will respond with a message with the corresponding message\_id.

**Example:** downloadmsgs, downloadmsgs 3

### Log Off

**Format:** logoff

**Description:** The client uses this message type to signal to the server that it wants to log off and terminate the session.

**Example:** logoff

### Server Response

**Format:** <status\_code> <message>

**Description:** The server uses this message type to signal to the client the outcome of a request.

**Status codes:**

* ‘200: OK’: Indicates that the request was successful and the server was able to fulfil it.
* ‘400: Bad Request’: Indicates that the client's request was malformed or had other errors, and the server could not process it.
* ‘401: Unauthorized’: Indicates that the client does not have access to the server.
* ‘404: Not Found’: Indicates that the requested resource could not be found on the server.

**Example:** 200 OK "Logon successful"

## Pseudo-code for Functional Requirements

### Log On

client sends: logon <username> <password>

server receives and validates credentials

if valid:

server responds: 200 OK "Login successful"

else:

server responds: 400 Bad Request "This message requires 2 parameters <username><password>"

### Upload Message

client sends: uploadmsg <message>

server receives and stores the message

server responds: 200 OK "Message uploaded successfully"

### Download Message(s)

client sends: downloadmsgs or downloadmsgs<message\_id>

server retrieves all messages or message matching message\_id

server responds with a message or list of messages OR 404 Not Found if no messages are found.

### Log Off

client sends: logoff

server logs the user off

server responds: 200 OK "Logoff successful"

## Inter-process Communication

To carry out inter-process communication using the described protocol over TCP (Transmission Control Protocol), both the client and server need to establish a TCP connection. The client initiates the connection by creating a TCP socket and specifying the server's IP address and port number. Upon receiving the connection request, the server creates a socket and accepts the connection. Once the connection is established, the client can start sending messages by encoding them according to the defined message formats and transmitting them through the TCP connection. The server, upon receiving a message, decodes it to determine the type of request and processes it accordingly. It then formulates a response message using the defined server response format and sends it back to the client through the same TCP connection. Throughout this communication process, TCP ensures reliable, ordered, and error-checked delivery of messages between the client and server, handling issues such as packet loss, reordering, and congestion control. Upon completing the communication session, either the client or the server can terminate the TCP connection, signalling the end of the inter-process communication session.

# RFC Documentation

## Introduction

The Client-Server Short Message Protocol (SMP), is designed to facilitate efficient and reliable communication between client and server applications. The protocol follows a request-response model, where clients initiate actions by sending requests to the server, and the server responds accordingly. The primary objectives encompass ensuring secure data transmission, robust user authentication, and streamlined message management. The logon process authenticates users before granting access to server resources, requiring clients to provide valid credentials. Upon successful authentication, clients can proceed to upload, download, or manage messages stored on the server. The upload functionality allows clients to send messages to the server for storage, while the download feature enables clients to retrieve stored messages based on specific criteria. Finally, the logoff process ensures a graceful termination of client-server connections, releasing associated resources upon client request.SMP does not address encryption or authentication mechanisms directly. Implementers are encouraged to incorporate additional security measures to protect sensitive data and prevent unauthorized access.

## Sequence Diagram

A diagram of a program

Description automatically generated

*Figure 1 – Sequence Diagram of the full interaction.*

## Message Formats

### Log On

Description: Allows the user to log into the server with a username and password.

Message Parameters:

* username (text): The username of the client.
* password (text): The password associated with the username.

Response Messages:

* Code: 200

Text: Logon successful.

* Code: 400

Text: Logon unsuccessful. This message requires two parameters <username><password>.

* Code:401

Text: Unauthorized Access. Please enter a valid username and password.

### Upload Message

Description: Enables the client to upload a message to the server.

Message Parameters:

message (text): The content or data to be uploaded.

Response Message:

* Code: 200

Text: Message uploaded successfully.

* Code: 400

Text: No Message Entered. Please enter a message.

### Download Message(s)

Description: Requests the server to download all messages, or a specific message, stored on it.

Message Parameters:

OPTIONAL = Message\_id (int): Retrieves the message matching the message\_id

Response Messages:

* Code: 200

Text: List of messages.

* Code: 400

Text: This function takes a maximum of one parameter.

* Code: 404

Text: No messages found (with message\_id: <message\_id>).

### Log Off

Description: Indicates the client's intention to terminate the session.

No additional parameters.

Response Message:

* Code: 200

Text: Logoff successful.

## UML Diagrams

#### Server Side

A screenshot of a computer

Description automatically generated

*Figure 2 – Server-side UML Diagram.*

#### Client Side

A screenshot of a computer

Description automatically generated

*Figure 3 – Client side UML diagram.*

### Implementation of Functions Required

The server-side implementation comprises several key functionalities to manage client interactions effectively. Firstly, the "logon" functionality verifies user credentials upon receiving a logon request. The server checks the provided username and password against predefined constants. If the credentials match, access is granted by setting the isLoggedIn flag to true, allowing subsequent interactions. In case of invalid credentials or an incorrect command format, the server promptly notifies the client with an appropriate error message, ensuring secure authentication.

Another functionality is the "uploadmsg" feature, enabling clients to upload messages to the server. Upon receiving an upload request, the server validates whether a message is provided. If a message exists, it is added to the server's message list, and a confirmation message is sent back to the client, indicating successful upload. However, if no message is provided, the server notifies the client of the requirement, preventing incomplete submissions.

Additionally, the server facilitates the "downloadmsgs" functionality, allowing clients to retrieve messages from the server. This feature handles requests with or without a specified message ID parameter. If no message ID is provided, the server responds with a list of available messages along with their count. On the other hand, if a valid message ID is provided, the server retrieves and sends the requested message to the client. In case of an invalid message ID, the server informs the client about the absence of the requested message.

Finally, the "logoff" functionality enables clients to gracefully terminate their session with the server. Upon receiving a logoff request, the server acknowledges the request with a confirmation message, indicating successful logoff. Subsequently, the server closes the clientSocket, effectively terminating the connection. This ensures a smooth end to the client-server interaction, maintaining system integrity and resource efficiency.

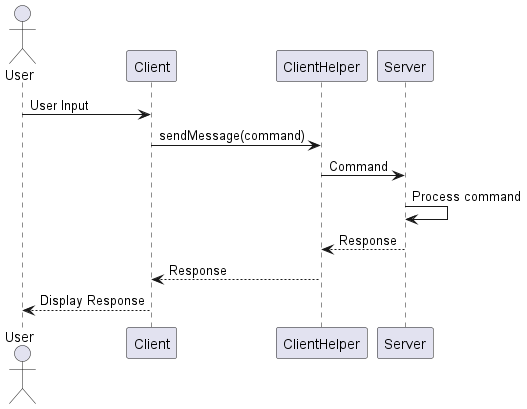
# Software Architecture & Implementation

## Overview & Objectives

The implementation of the client-server project seeks to materialise the design objectives through the development of robust and efficient components responsible for secure communication, user authentication, and message management. This implementation encompasses three distinct layers: the application layer, the presentation layer, and the service layer. The application layer orchestrates the flow of communication between clients and the server, managing user input, processing requests, and generating responses. Meanwhile, the presentation layer handles user interaction, providing an interface through which users can interact with the system. This layer manages user input and output, presenting prompts, receiving commands, and displaying responses. The service layer encapsulates core business logic and services, including user authentication, message management, and network communication. Here, user credentials are validated, messages are stored and retrieved, and secure communication channels between clients and the server are facilitated. By implementing these layers effectively, the client-server project achieves its overarching objectives of ensuring secure and efficient communication, robust user authentication, and streamlined message management, thereby enhancing overall system reliability and user experience.

## Application Layer

In the SMP, the Application Layer serves as the top-level component responsible for orchestrating the overall functionality of the system. This layer oversees the interaction between the user-facing Presentation Layer and the underlying Service Layer, ensuring smooth communication and coordination between different parts of the application. It manages high-level tasks such as handling user requests, coordinating data flow, and enforcing business logic rules. Additionally, the Application Layer may include components for managing application-wide configurations, error handling, and overall system control, providing a cohesive framework for the entire application to operate within.



*Figure 4 – Sequence diagram for Application Layer.*

## Presentation Layer

The Presentation Layer, positioned just below the Application Layer, focuses on presenting information to users and gathering user inputs. The ClientGUI class embodies the Presentation Layer, providing a graphical user interface for users to interact with the application. This layer encapsulates elements such as user interface components and event-handling mechanisms. It ensures a user-friendly and intuitive interaction experience by presenting information in a visually appealing format and facilitating user interactions through graphical controls and widgets.

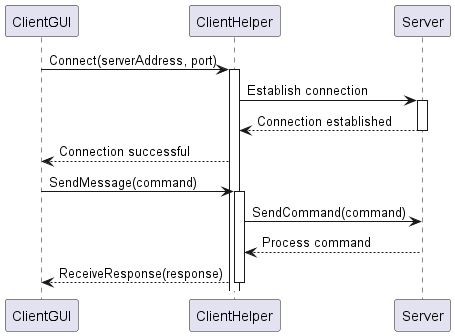
A diagram of a server

Description automatically generated

*Figure 5 – Sequence diagram for Presentation Layer.*

## Service Layer

Beneath the Presentation Layer lies the Service Layer, which handles the processing and execution of business logic and application-specific tasks. The ClientHelper class embodies the Service Layer, encapsulating functionality related to establishing connections with the server, sending commands, and receiving responses. This layer abstracts away the complexities of network communication and business logic implementation, providing a clean and modular interface for the Presentation Layer to interact with. By separating concerns and encapsulating functionality within the Service Layer, your application achieves a higher level of modularity, reusability, and maintainability.



*Figure 6 – Sequence diagram for Service Layer.*

## Execute Requested Functionalities (Command Line)

### Log On

The command-line client facilitates server authentication. It prompts users for the server's address and port, connects to the server, and allows users to input commands. The primary command is "logon," requiring a username and password. Upon sending this information to the server, it responds with status codes indicating success or failure. The client then prompts for further commands, maintaining an interactive loop for continued interaction. This setup provides a straightforward means for users to authenticate with the server securely.

A screen shot of a computer

Description automatically generated

*Figure 7 – Command Line showing functionality of logon function.*

### Upload Message

The updated command-line interface ensures secure interaction with the server. Users are first prompted to log in using the "logon" command, providing a username and password. Upon successful authentication, indicated by a "200 OK" response, users can proceed to use the "uploadmsg" command to upload messages. Subsequent uploads prompt users to provide a message; otherwise, a "400 No Message Entered" response is returned. Each successful upload is acknowledged with a "200 OK" message, confirming the successful transmission of the message to the server. This streamlined process maintains security and efficiency, facilitating seamless communication between users and the server.

A computer screen shot of a black screen

Description automatically generated

*Figure 8 – Command Line showing functionality of uploadmsg function.*

### Download Message(s)

This function allows users to retrieve stored messages from the server. Upon issuing the command, if messages are available, the server responds with a "200 OK" status code followed by a list of messages. Users can select a message by index for download, receiving the content with another "200 OK" response. If no messages are found, the server returns a "404 No messages found" message. If attempting to download a non-existent message, the server responds with a "404 Message not found" error.

A screen shot of a computer

Description automatically generated

*Figure 9 – Command Line showing functionality of downlaodmsgs function.*

A screenshot of a computer screen

Description automatically generated

*Figure 10 – Command Line showing error hangling of downlaodmsgs function.*

### Log Off

Upon establishing a connection to the server, users are prompted to input commands. If users attempt to execute the "logoff" command without first logging in, the server responds with a "401 Unauthorized" message, indicating the need for authentication. After successfully logging in using the "logon" command with appropriate credentials, the server confirms with a "200 OK Logon successful" response. Subsequently, executing the "logoff" command results in a "200 OK Logoff successful" response, signifying a successful termination of the session.

A screenshot of a computer

Description automatically generated

*Figure 11 – Command Line showing functionality of logoff function.*

## Execute Requested Functionalities (GUI)

### Connect to Server

The GUI facilitates the process of connecting to the server by providing input fields for the server address and port. Users input the required information and initiate the connection by clicking the "Connect" button. Upon successful connection establishment, the GUI initialises communication channels with the server and displays a confirmation message in the message area. Additionally, instructions for further actions are provided to guide users on the available commands and their usage.

A screenshot of a computer

Description automatically generated

*Figure 12 – GUI showing successful connection to server.*

### Log On

To log onto the server, users input their username and password in the designated command field. From the dropdown menu, users select the "logon" command and click "Send" to initiate the logon process. The GUI combines the selected command with the provided parameters and sends the request to the server. Subsequently, the server processes the logon request, verifying the credentials and responding accordingly. Users receive feedback in the message area, indicating whether the logon attempt was successful or unsuccessful.

A screenshot of a computer

Description automatically generated

*Figure 13 – GUI showing functionality of logon function.*

### Upload Message

Users can upload messages to the server by entering their message in the command field. From the dropdown menu, users select the "uploadmsg" command and click "Send" to transmit the message to the server. The GUI constructs the appropriate command string and forwards it to the server for processing. Following the upload request, the server handles the incoming message, confirming successful upload or reporting any errors encountered during the process. Users receive feedback in the message area, reflecting the outcome of the upload attempt.

A screenshot of a computer

Description automatically generated

*Figure 14 – GUI showing functionality of uploadmsg function.*

### Download Message(s)

To download messages from the server, users have the option to specify a message ID in the command field, although this parameter is optional. Users select the "downloadmsgs" command from the dropdown menu and click "Send" to initiate the download process. The GUI constructs the command string, including the optional message ID, and sends it to the server for processing. Upon receiving the request, the server retrieves the requested message(s) and sends them back to the client. The downloaded messages are displayed in the message area for users to view.

A screenshot of a computer

Description automatically generated

*Figure 15 – GUI showing functionality of downlaodmsgs function.*

### Log Off

Users can log off from the server by selecting the "logoff" command from the dropdown menu and clicking "Send." The GUI sends the logoff request to the server, indicating the client's intention to disconnect. Upon receiving the logoff request, the server terminates the connection with the client. Notably, the GUI does not automatically close upon logoff, allowing users to perform additional actions or reconnect to the server if needed. Feedback regarding the logoff process is displayed in the message area, ensuring users are informed about the status of their logoff request.

A screenshot of a computer

Description automatically generated

*Figure 16 – GUI showing functionality of logoff function.*

## Concurrent Access to the Server

A screenshot of a computer program

Description automatically generated

*Figure 17 – Concurrency shown in the Server command line output.*

In the server command line output provided, we can observe a clear demonstration of concurrency between the GUI Client and the CLI Client. Upon starting the server, it reports that it has successfully started and is actively listening on port 5665, ready to receive incoming connections. Shortly after, the server announces the arrival of two clients, each connecting from their respective addresses. This initial setup indicates the readiness of the server to handle multiple client connections concurrently, laying the groundwork for concurrent interactions.

As the client interactions unfold, we witness both the GUI Client and the CLI Client engaging with the server in parallel. Both clients initiate logon requests almost simultaneously, signaling their intent to authenticate with the server using the provided username and password. The server processes these login attempts concurrently, granting access to both clients without delay and responding promptly with an "Access Granted" message, showcasing its ability to handle multiple authentication requests concurrently.

Following successful authentication, both clients proceed to interact with the server by uploading messages. The GUI Client and the CLI Client upload their respective messages, indicating their source, to the server. The server adeptly processes them without contention, responding promptly with confirmation messages for each upload.

Subsequently, as both clients request to download messages from the server, we witness another instance of concurrency in action. The server, equipped to handle concurrent download requests, promptly responds to each client's request by providing the requested messages. What's notable here is that the server facilitates cross-client interaction seamlessly. Both the GUI Client and the CLI Client can download messages sent by the other, showcasing bidirectional communication. This bidirectional functionality enhances the collaborative aspect of the system, allowing clients to share and access messages regardless of their client type. Even when clients specify individual message IDs for download, indicating specific messages they wish to retrieve, the server efficiently retrieves and delivers the requested messages concurrently.

A screenshot of a computer

Description automatically generated

*Figure 18 – CLI client downloading message sent by GUI client.*

A screenshot of a computer

Description automatically generated

*Figure 19 – GUI client downloading all messages and one sent by CLI client.*

Finally, as the client sessions draw to a close, both the GUI Client and the CLI Client initiate logoff requests almost simultaneously. The server, capable of handling concurrent logoff requests, promptly acknowledges each request with a "Logoff successful" response, effectively terminating the client sessions in parallel.

## SSL

### Overview

SSL (Secure Sockets Layer) is utilised to establish a secure communication channel between a server and a client. On the server side, an SSL context is created with the "TLS" protocol, configuring it to use a Java KeyStore (JKS) containing the server's certificate and private key. The KeyManagerFactory is initialised with the key store, which facilitates the management of keys for the server's side of the SSL connection. Additionally, a TrustManagerFactory is initialised with the same key store, allowing the server to verify the authenticity of clients connecting to it. This SSL context is then used to create an SSLServerSocketFactory and subsequently an SSLServerSocket bound to a specific port. On the client side, a similar SSL context is established with trust in the server's certificate. The client loads a trust store containing the server's certificate, allowing it to validate the server's identity. The client-side SSL context is then used to create an SSLSocketFactory, which generates SSLSocket instances for connecting to the server securely. This setup ensures that both the server and client can authenticate each other's identities and encrypt data exchanged over the network, thereby maintaining the confidentiality and integrity of the communication.

A screen shot of a computer screen

Description automatically generated

*Figure 20 – Debugging of SSL from client side (i).*

A computer screen with text on it

Description automatically generated

*Figure 21 – Debugging of SSL from client side (ii).*

### Generating Keys and Certs

To set up SSL, a new directory named 'ssl' was created. Within this directory, a public/private key pair was generated and stored in a keystore file named 'server.jks' using the command 'keytool -genkeypair -v -keystore server.jks -alias JKS'. Necessary details were provided during this process, and a password for the keystore was chosen. After generating the public/private key pair and storing it, a recommendation was made to migrate to the PKCS12 (Personal Information Exchange Syntax Standard) format for improved compatibility and security. This recommendation was followed, and the necessary steps were taken to migrate the keystore to the PKCS12 format.

A screenshot of a computer program

Description automatically generated

*Figure 22 – Creating keystore for server..*

To establish trust on the client side, the server's certificate was extracted by running 'keytool -exportcert -alias JKS -file my\_cert.crt -keystore server.jks -storepass password'. This saved the certificate as 'my\_cert.crt'.

A black screen with white text

Description automatically generated

*Figure 23 – Creating server certificate.*

This certificate was then moved to the 'client' folder, and a truststore was created on the client side by importing the certificate into it using the command 'keytool -importcert -alias herong\_home -file my\_cert.crt -keystore public.jks -storepass password'. This command created a truststore named 'public.jks' and imported the server's certificate into it, allowing the client to trust the server's identity during SSL communication.

A computer screen shot of a computer program

Description automatically generated

*Figure 24 – Importing servers certificate to the truststore.*

## Handling Errors

### User Input

In the user input handling aspect of the code, there's a robust strategy for dealing with potential errors. This includes catching NumberFormatException and IOException when parsing the port number input from the user in the Server class. By implementing this error handling, the program can gracefully handle situations where the user provides invalid input, such as non-numeric characters for the port number or unexpected I/O errors during input retrieval.

Moreover, both the ClientGUI and Client classes implement command validation mechanisms to guide users in providing correct inputs. By prompting the user for commands and offering guidance on the required format, these components reduce the likelihood of errors stemming from incorrect user input. This helps users understand how to interact with the application and also minimises the occurrence of input-related issues, enhancing the overall user experience and reducing potential frustration.

### Server

The Server class implements a robust error handling mechanism to address potential exceptions that may arise during SSL/TLS setup and client interaction. Specifically, the code employs a combination of try-catch blocks to catch various types of exceptions that could occur during server operation.

Firstly, the code surrounds the SSL/TLS setup process within a try block. Within this block, exceptions such as NoSuchAlgorithmException, KeyStoreException, CertificateException, UnrecoverableKeyException, and KeyManagementException are caught. These exceptions may occur during the initialisation of SSLContext, KeyStore, KeyManagerFactory, or TrustManagerFactory instances.

Additionally, the code includes a try block to catch IOException that may occur during client interaction, particularly during socket creation, accepting client connections, or handling client input/output streams. By catching IOException, the server can handle potential I/O errors, ensuring uninterrupted server operation.

When an exception is caught, the code typically logs relevant error information using e.printStackTrace() or a similar logging mechanism. This allows the server to record details about the encountered error, facilitating debugging and troubleshooting. Depending on the severity of the error, the server may choose to continue functioning if possible or terminate execution to prevent further issues or data corruption.

### Network

The Client and ClientHelper classes implement error handling mechanisms to manage potential issues encountered during client-server communication. Specifically, these components address errors related to both SSL connection establishment and message sending/receiving operations.

For SSL connection establishment, the code handles common exceptions such as IOException, NoSuchAlgorithmException, KeyManagementException, KeyStoreException, and CertificateException. Again, these exceptions may occur during SSL context initialisation, truststore loading, or socket creation. By catching these exceptions within the connect() method of the ClientHelper class, the code ensures that errors during SSL connection setup are managed, preventing application crashes and providing informative feedback to users.

Furthermore, the error handling mechanisms in the ClientHelper class also address potential errors encountered during message transmission and reception. Specifically, the code handles IOException instances that may arise during message sending or receiving operations. By capturing these exceptions in the sendMessage() and receiveMessage() methods, the code maintains robustness in client-server communication, allowing the application to adapt to network interruptions or other communication issues without disrupting the user experience.

# Conclusion

In conclusion, the SMP project exemplifies the integration of advanced communication technologies and secure protocols. The project's comprehensive approach to protocol design ensures clarity and consistency in communication, while the utilisation of SSL and WebSockets enhances security and enables real-time interactions. With robust support for both GUI and CLI interfaces, the project provides flexibility and convenience in accessing server functionalities.

Through meticulous implementation and documentation, the project demonstrates the feasibility of creating scalable and efficient client-server architectures. As a versatile solution for various communication needs, the project stands ready to serve its intended purpose effectively, facilitating seamless data exchange between multiple clients and the server.

# References

Hartman, J. (2023) *Java Swing Tutorial: How to create a GUI application in Java*, *Guru99*. Available at: https://www.guru99.com/java-swing-gui.html (Accessed: 12 March 2024).

*How can I use multi-threading to allow multiple clients on to a server?* (1961) *Stack Overflow*. Available at: https://stackoverflow.com/questions/33584373/how-can-i-use-multi-threading-to-allow-multiple-clients-on-to-a-server (Accessed: 15 March 2024).