|  |
| --- |
| Implementation Generic Blue icon |
| Application Design, Implementation & Protocol Document  Part One: Protocol Design & Proposed Implementation Part Two: Software Architecture & Implementation Part Three: Protocol Document |
| |  |  |  | | --- | --- | --- | | Deirdre Lee | 3/12/24 | Distributed Computing | |

Contents

[Part 1: Protocol Design 3](#_Toc162448965)

[1. Objectives and Overview 3](#_Toc162448966)

[2. Design Philosophy 3](#_Toc162448967)

[3. Protocol for Client-Server Communication 3](#_Toc162448968)

[4. Pseudo-code for Functional Requirements 3](#_Toc162448969)

[4.1. Client Side 3](#_Toc162448974)

[4.2. Server Side 3](#_Toc162448975)

[5. Inter-process Communication 4](#_Toc162448980)

[5.1. Sequence of Inter-process Communication 4](#_Toc162448982)

[6. Conclusion 4](#_Toc162448983)

[Part 2: Application Implementation 4](#_Toc162448984)

[1. Objectives and Overview 4](#_Toc162448985)

[2. Source Files 4](#_Toc162448986)

[2.1. Client.java 4](#_Toc162448987)

[2.2. ClientHelper.java 4](#_Toc162448988)

[2.3. Server.java 5](#_Toc162448989)

[2.4. ServerThread.java 5](#_Toc162448990)

[2.5. MyStreamSocket.java 5](#_Toc162448991)

[3. Tiers 5](#_Toc162448992)

[3.1. Application Tier 5](#_Toc162448993)

[3.2. Presentation Tier 5](#_Toc162448994)

[3.3. Session Tier 5](#_Toc162448995)

[4. UML Diagrams 6](#_Toc162448996)

[5. Sample Input & Output Data 7](#_Toc162448997)

[Running Instructions 9](#_Toc162448998)

[5.1. IDE Used 9](#_Toc162448999)

[5.2. Username/Password 9](#_Toc162449000)

[5.3. Dependencies 9](#_Toc162449001)

[5.4. Steps to Run 9](#_Toc162449002)

[6. Conclusion 11](#_Toc162449003)

[7. Link to Project Repository 11](#_Toc162449004)

[Part 3: Protocol Document 12](#_Toc162449005)

[1. Introduction 12](#_Toc162449006)

[2. Protocol Objective 12](#_Toc162449007)

[3. Protocol Overview 12](#_Toc162449010)

[4. Protocol Specification 12](#_Toc162449013)

[4.1. Service Location 12](#_Toc162449014)

[4.2. Sequence of Inter-process Communication 12](#_Toc162449015)

[4.3. Representation and Interpretation of Data Exchanged 12](#_Toc162449016)

[4.4. Description of Message Formats 13](#_Toc162449024)

[4.5. Error Handling 16](#_Toc162449025)

[4.6. Service Session Management 16](#_Toc162449026)

[5. Implementation of Functions 16](#_Toc162449027)

[5.1. Log On 16](#_Toc162449033)

[5.2. Upload Request 16](#_Toc162449037)

[5.3. Download Request 17](#_Toc162449038)

[5.4. Download All Requests 17](#_Toc162449039)

[5.5. Log Off 17](#_Toc162449040)

[5.6. Quit 17](#_Toc162449041)

[6. Sequence Diagram 18](#_Toc162449042)

[7. Security Considerations 19](#_Toc162449043)

[8. Author 19](#_Toc162449044)

[9. References 19](#_Toc162449045)

# Part 1: Protocol Design

# Objectives and Overview

The protocol aims to ensure secure, concurrent, reliable and error-free communication. It provides guidelines for implementation, ensuring the application meets the criteria for simplicity, security, and concurrency.

# Design Philosophy

This protocol facilitates communication between clients and a server prioritising simplicity, security and concurrency. It enables users to log in, upload messages to the server, download single or all messages from the server, log out, and quit the program. Security measures are implemented to ensure secure communication and authentication, Concurrency is achieved through multithreading, allowing the server to handle multiple client connections simultaneously.

# Protocol for Client-Server Communication

* A secure connection will be established using Secure Sockets Layer/Transport Layer Security (SSL/TLS) protocol.
* The client communicates with the server using TCP/IP as outlined in the requirements.
* The server manages concurrent communication through multithreading.
* Messages are exchanged between the client and server in plain text format, with a predefined structure, and appropriate error messages are provided for incorrect input.
* SSL/TLS encrypts and decrypts communication between the client and the server.

# 

# 

# **Pseudo-code for Functional Requirements**



## **Client Side**

* Connect to the server
* Establish an initial handshake to establish a secure connection
* Send requests to the server
* Receive responses from the server
* Close the connection when done

## **Server Side**

* Listen for incoming client connections
* Accept client connections and establish communication
* Receive requests from the client(s)
* Process requests and generate responses
* Send responses to the client(s)
* Close the connection with the client(s) when done



# **Conclusion**

In conclusion, the protocol design section provides an overview of the objectives, design philosophy, and communication protocols for ensuring secure, concurrent, and reliable communication between clients and the server. It outlines the functional requirements with pseudocode examples and describes the sequence of inter-process communication, highlighting the use of SSL/TLS for secure connections.

# **Part 2: Application Implementation**

# Objectives and Overview

The application serves as a platform for secure and efficient client-server communication. Key functionalities include establishing secure connections, transmitting messages, managing user authentication, error handling, and supporting concurrent communication.

# Source Files

### Client.java

This file contains the main class for the client application. It handles user input, establishes connection with the server using SSL/TLS, sends requests, and receives responses. The class also implements error handling and logging functionalities to catch and report any exceptions or issues during communication process.

### ClientHelper.java

This class helps the client in handling operations such as login, sending and receiving messages, logging out, and quitting the application. It also manages the clients socket connection to the server, ensuring proper initialisation and error handling is implemented throughout the communication process.

### Server.java

This file contains the main class for the server application. It listens for incoming client connections, establishes SSL/TLS connections, and creates threads to handle multiple client communication.

### ServerThread.java

This class represents a thread on the server side responsible for handling communication with a single client. It implements the Runnable interface and overrides the run() method to execute the threads tasks. The class contains methods for processing client requests and generating responses.

### MyStreamSocket.java

This class provides a wrapper for Java socket functionality, enabling the handling of input and output streams in the communication process. It provides methods such as **sendMessage()** and **receiveMessage()** for sending and receiving messages, and **close()** for closing the socket.

# Tiers

## Application Tier

The application tier is responsible for executing the business logic and essential application functions. (TechTarget Contributor, 2021)

In this application, the classes that make up the application tier are the Server, MyStreamSocket and ClientHelper classes.

The Server class manages the creation of SSL/TLS sockets, handles incoming client connections and manages communication with clients via threads.

The MyStreamSocket class also facilitates communication between the client and server by managing input and output streams over the socket.

The ClientHelper class helps the client, providing methods for necessary operations, and contains the logic for establishing connections with the server.

## Presentation Tier

The presentation tier provides the interface for the application, allowing users to communicate with the system. (TechTarget Contributor, 2021) It establishes communication with the other tiers through user input and forwards requests for processing.

In this application, the class that fulfils this role is the Client class, which manages user interaction and initiates communication with the server. It provides a user interface through console input and output.

## Session Tier

The session tier, or data tier, typically manages reading and writing to a database or storage system. In this application, although there is no database involved, the ServerThread class fulfils a similar role. While its primary responsibility is managing communication with clients, it also handles individual client sessions, including the storage of the message and all related read and write operations.

# UML Diagrams

The following UML diagrams show the client and server packages as generated in IntelliJ.

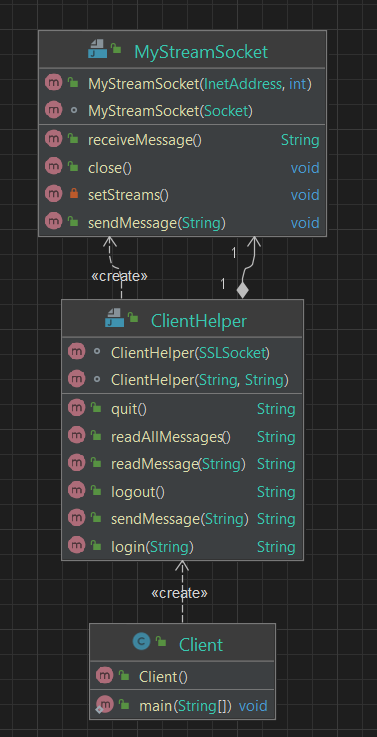


Fig 1 Client Package

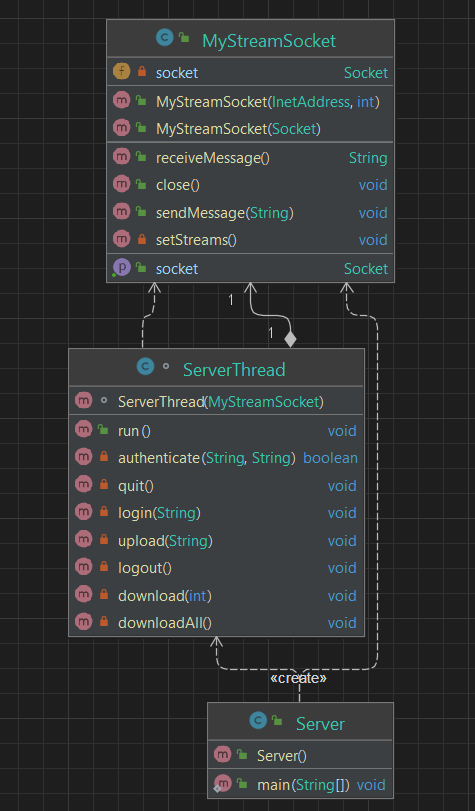


Fig 2 Server package

# Sample Input & Output Data

The following screenshots show interaction following valid requests to server

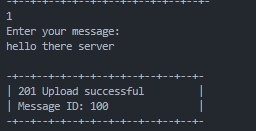


Fig 3 1. Upload message (valid)

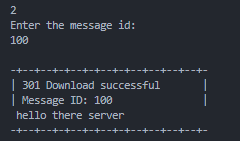


Fig 4 2. Read message using id (valid)

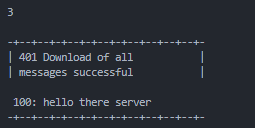


Fig 5 3. Read all messages (valid)

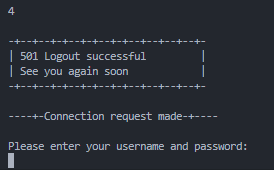


Fig 6 4. Logout

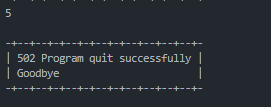


Fig 7 5. Quit program

Fig 8 shows the server output provided during the previous interaction.

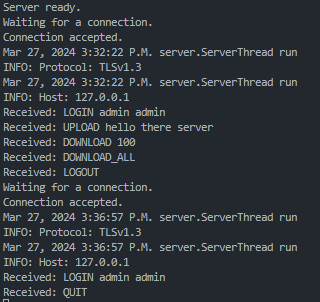


Fig 8 Server side

# Running Instructions

## IDE Used

The IDE used when building, testing and running the application is VSCode. I have also run it using IntelliJ with no apparent issues.

## Username/Password

The login credentials for the application is lowercase ‘admin’ for both the username and for the password. The very basic authentication entails checking there are two words provided, as shown in Fig 9, and that they match the hardcoded values, as shown in Fig 10.

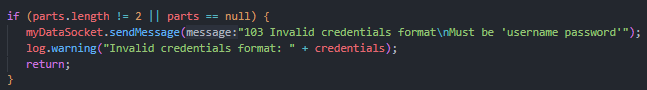


Fig 9 Check two words provided



Fig 10 Check credentials match hardcoded details

## Dependencies

No additional dependencies are needed for the application.

## Steps to Run

**Step 1:** Compile the java files in the client and server package:

**javac client/\*.java**

**javac server/\*.java**

**Step 2:** Run the server:

**java server/Server**



Fig 11 Server

**Step 3:** Run the client:

**java client/Client**

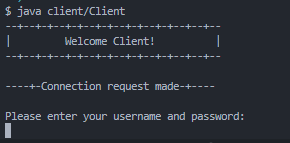
****

Fig 12 Client

**Step 4:** Enter username and password ‘admin admin’

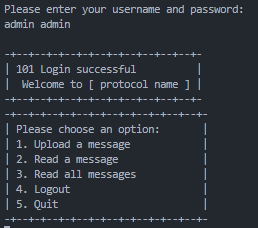


Fig 13 Client

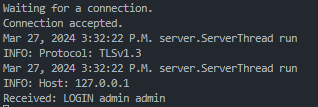


Fig 14 Server

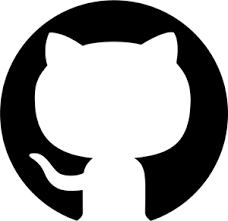
**Step 5:** Interact with client using onscreen instructions. See output examples in [part 5](#_Sample_Input_&)

**Step 6:** Once finished, terminate the server using ‘Ctrl + C’ and the client using the Quit option or ‘Ctrl + C’.

# Conclusion

Part two of the document, Application Implementation, outlines the client-server design, functionality and implementation details. A breakdown of the Java sources files involved is provided, along with an explanation of the application architecture. UML diagrams of the application are included. Sample input and output data are provided in the form of screenshots to illustrate the interaction between the server and the client. Finally, running instructions are provided to demonstrate the set up and running of the application, including necessary steps for execution.

# Link to Project Repository

 <https://github.com/EmoSense/distributed_computing>

C:\Users\Deirdre\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\D63FA85E.tmp <https://deetralee.atlassian.net/jira/software/projects/DC/boards/3>

# Part 3: Protocol Document

# 1. Introduction

The protocol is intended to provide secure and efficient interactions between clients and the server.

This document describes the SMProtocol a protocol designed for facilitating communication between clients and a server concurrently and securely.

# 2. Protocol Objective

The objectives of the SMProtocol protocol are to enable users to log on to the server, upload and download messages, log off securely, and quit the application.

The protocol ensures concurrency, confidentiality, and reliability of data exchanged between clients and servers.

(Information Sciences Institute, University of Southern California, 1981)



# Protocol Overview

SMProtocol is a simple client-server protocol based on a request-response model. It supports secure communication using SSL/TLS encryption and concurrency using multithreading over TCP/IP.



# 4. Protocol Specification

## 4.1. Service Location

Clients connect to the server's IP address (127.0.0.1) and port number (7).

Communication is secured using SSL/TLS encryption.

## 4.2. Sequence of Inter-process Communication

In the system, communication between the client and the server processes is facilitated through a secure connection established using SSL/TLS.

* The client initiates a secure connection request to the server.
* The server listens for incoming client connections over SSL/TLS.
* Upon successful connection establishment, the client and server perform an SSL/TLS handshake to authenticate and establish a secure communication channel.
* Once the handshake is completed, the server responds to the client’s connection request.
* The client acknowledges the server's response over the secure connection.
* Communication continues bidirectionally over the established SSL/TLS connection until session termination. Session termination can occur due to termination by either party.

## 4.3. Representation and Interpretation of Data Exchanged

Requests and responses follow a predefined format agreed upon by both client and server.

Plain text data format is used for encoding requests, responses, and other data over the secure connection.



## Description of Message Formats

The message formats are in plain text. Each message includes a " Message type" field to indicate the action requested, along with any other necessary parameters. The server processes these messages according to the protocol and responds to the client accordingly. Responses are also in string format. Once the connection with the server has been initialised by the client, the client can then send messages to the server. When the user selects an option, the corresponding method from the ClientHelper class is called. Inside this class, the message is formatted according to the protocol - the message is appended to the appropriate message type. Messages are then exchanged using the sendMessage() and receiveMessage() methods in MyStreamSocket. An instance of ServerThread is created for each client. When a message is received, the ServerThread processes it based on its message type. Afterwards, it sends back a response message according to the protocol. The client received the response via MyStreamSocket and displays the message to the user.

* + 1. **Login Request Message Format**

**Description:** The user can login to the server with a username and password.

The login request message format includes the following elements:

**Parameters:**

Message type: LOGIN

Username: [username]

Password: [password]

**Example:**

+---------------------+

| **LOGIN user password** |

+---------------------+

**Response Message OK:**

Code: 101 - Success

Text: “Login successful! Welcome to SMProtocol”

**+-----------------------------------+**

**| 101 Login successful! |**

**| Welcome to SMProtocol |**

**+-----------------------------------+**

**Response Message Error:**

Code: 102 - Unauthenticated

Text: “Login unsuccessful. Please try again. Check logs for details”

**+-----------------------------+**

**| 102 Login unsuccessful |**

**| Please try again |**

**| Check logs for details |**

**+-----------------------------+**

**Response Message Error:**

Code: 103 – Invalid format

Text: “Invalid credentials format. Must be ‘username password’”

**+-----------------------------------+**

**| 103 Invalid credentials format |**

**| Please try again |**

**| Check logs for details |**

**+-----------------------------------+**

* + 1. **Upload Request Message Format**

**Description**: The user can upload a message to the server once authenticated.

The upload request message format includes the following elements:

**Parameters:**

Message type: UPLOAD

Message: [ message content ]

**Example:**

**+--------------------------+**

**| UPLOAD This is a message |**

**+--------------------------+**

**Response Message OK:**

Code: 201 - Success

Text: “Upload successful. Message ID: [id]”

**+-----------------------+**

**| 201 Upload successful |**

**| Message Id: [id] |**

**+-----------------------+**

**Response Message Error:**

Code: 202 – Empty message

Text: “Upload unsuccessful. Attempted to upload an empty message.”

**+-----------------------------+**

**| 202 Upload unsuccessful |**

**| Attempted to upload |**

**| null message |**

**+-----------------------------+**

* + 1. **Download Request Message Format**

**Description**: The user can download a specific message from the server once authenticated.

The download request message format includes the following elements:

**Parameters:**

Message type: Download

Message ID: [ ID of message to download ]

**+-----------------+**

**| DOWNLOAD 12345 |**

**+-----------------+**

**Response Message OK:**

Code: 301 - Success

Text: “Download successful. Message ID: [id] [message]”

**+-----------------------------+**

**| 301 Download successful |**

**| Message ID: [id] |**

**[message]**

**+-----------------------------+**

**Response Message Error:**

Code: 302 – No message with that id

Text: “Download unsuccessful. Message id not found”

**+-----------------------------+**

**| 302 Download unsuccessful |**

**| Message id not found |**

**+-----------------------------+**

* + 1. **Download All Request Message Format**

**Description**: The user can download all available message from the server once authenticated.

The download all request message format includes the following elements:

**Parameters:**

Message type: Download\_all

**+-----------------+**

**| DOWNLOAD\_ALL |**

**+-----------------+**

**Response Message OK:**

Code: 401 - Success

Text: “Download of all messages successful. [messages]”

**+----------------------------+**

**| 401 Download of all |**

**| messages successful |**

**[messages]**

**+----------------------------+**

**Response Message Error:**

Code: 402 – HashMap empty

Text: “No messages available”

**+-----------------------------+**

**| 402 No messages available |**

**+-----------------------------+**

* + 1. **Logout Message Format**

**Description**: The user can log off from the server.

The logout request message format includes the following elements:

**Parameters:**

Message type: Logout

**+-----------+**

**| LOGOUT |**

**+-----------+**

**Response Message OK:**

Code: 501 - Success

Text: “Logout successful, see you again soon”

**+-----------------------------+**

**| 501 Logout successful |**

**| See you again soon |**

**+-----------------------------+**

* + 1. **Quit Message Format**

**Description**: The user can quit the program.

The quit request message format includes the following elements:

**Parameters:**

Message type: Quit

**+-----------+**

**| QUIT |**

**+-----------+**

**Response Message OK:**

Code: 502 -Success

Text: “Logout successful, see you again soon”

**+----------------------------------+**

**| 502 Program quit successfully |**

**| Goodbye |**

**+----------------------------------+**

## Error Handling

Error codes and messages above are defined for common error scenarios.

Both client and server handle exceptions and errors appropriately over the secure connection and log all error messages appropriately.

## Service Session Management

Upon connection establishment, a secure service session is initiated. Session management includes maintaining the session state and the stored messages securely over the SSL/TLS connection. Details about the session are logged upon initialisation as depicted in Fig 15.

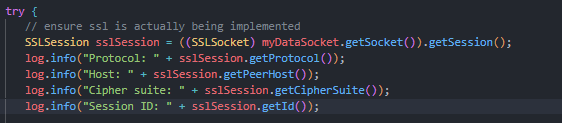


Fig 15 ServerThread – log SSL connection details

The session validity is then checked before proceeding, as shown in Fig 16. The handling of all message functions occurs within the context of a valid SSL session.

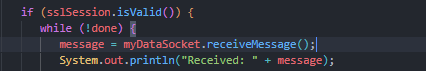


Fig 16 Session validity check

The output of a successful connection is shown in Fig 17.



Fig 17 Output on a successful connection (server)

If the session is invalid a warning is logged and the connection is closed, as shown in Fig 18. The code snippet in Fig 19 shows the catching of SSLHandshakeExceptions and actions taken on invalid sessions.

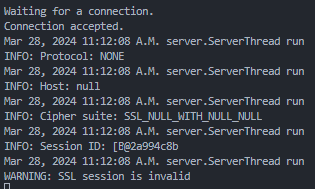


Fig 18 Output on an unsuccessful connection (server)

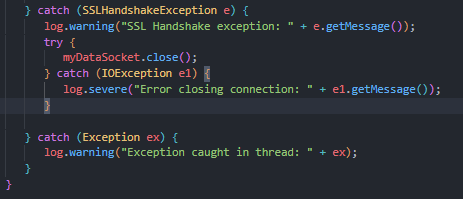


Fig 19 Exception handling on server

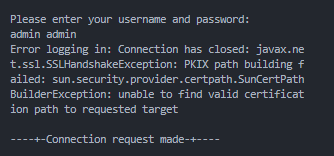


Fig 20 Client side details

Session clean-up is performed upon session termination. This is handled by both the quit() and logout() methods in ServerThread, shown in Fig 21.

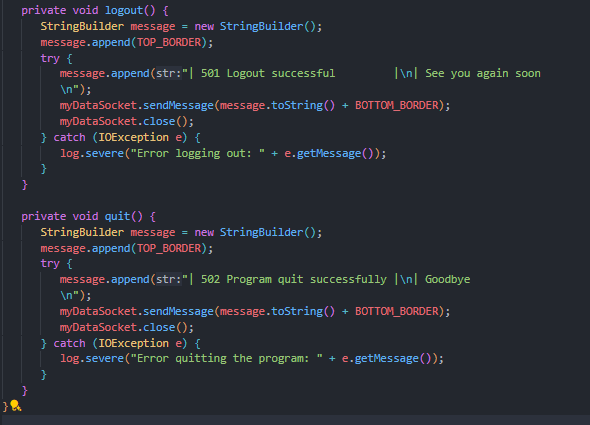


Fig 21 Logout and Quit method

An example of the output on the server and client of these functions can be seen in Fig 22. The initial session with ID [B@980e41c is closed successfully on logout by the user. The new session with ID [B@5bfa350e is also closed successfully when the user quits the program.

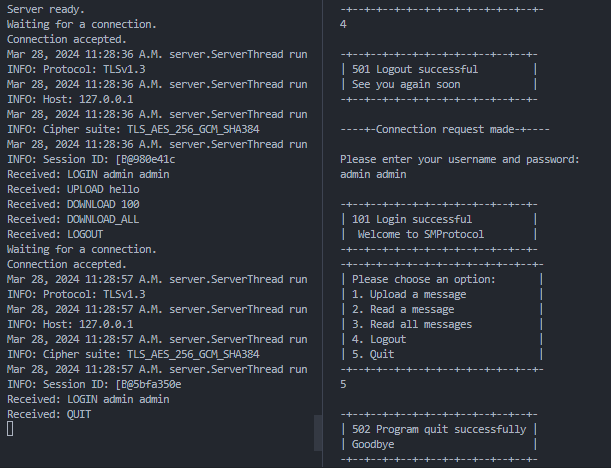


Fig 22 Logout and quit

# Implementation of Functions

The Server implements functions for handling login, upload, download, download all messages, logout, and quit requests from clients. These functions are managed by the ServerThread. In addition to the listed functionality, the server is also responsible for handling concurrent connections using multithreading, ensuring multiple clients can be dealt with without blocking or delay.



## Log On

**login(String credentials)**

* If login successful return success message (101)
* If login !successful return error message (102)
* If credential format incorrect return error message (103)
* Log any exceptions encountered



## Upload Request

**upload(String message)**

* Attempt to store message and it’s ID to pre-initialised HashMap
* If stored successfully return success message (201)
* If message empty return error message (202)
* Log any exceptions encountered

## Download Request

**download(int id)**

* Get value from HashMap using id
* If successful return success message (301)
* If !successful return error (302)
* Log any exceptions encountered

## Download All Requests

**downloadAll ()**

* + Get all messages from HashMap
  + If successful return success message (401)
  + If ! successful return error (402)
  + Log any exceptions encountered

## Log Off

**logout()**

* Perform logout operations
* If successful return success message (501)
* Log any exceptions encountered

## Quit

**quit()**

* Perform quit operations
* If successful return success message (502)
* Log any exceptions encountered

# Sequence Diagram

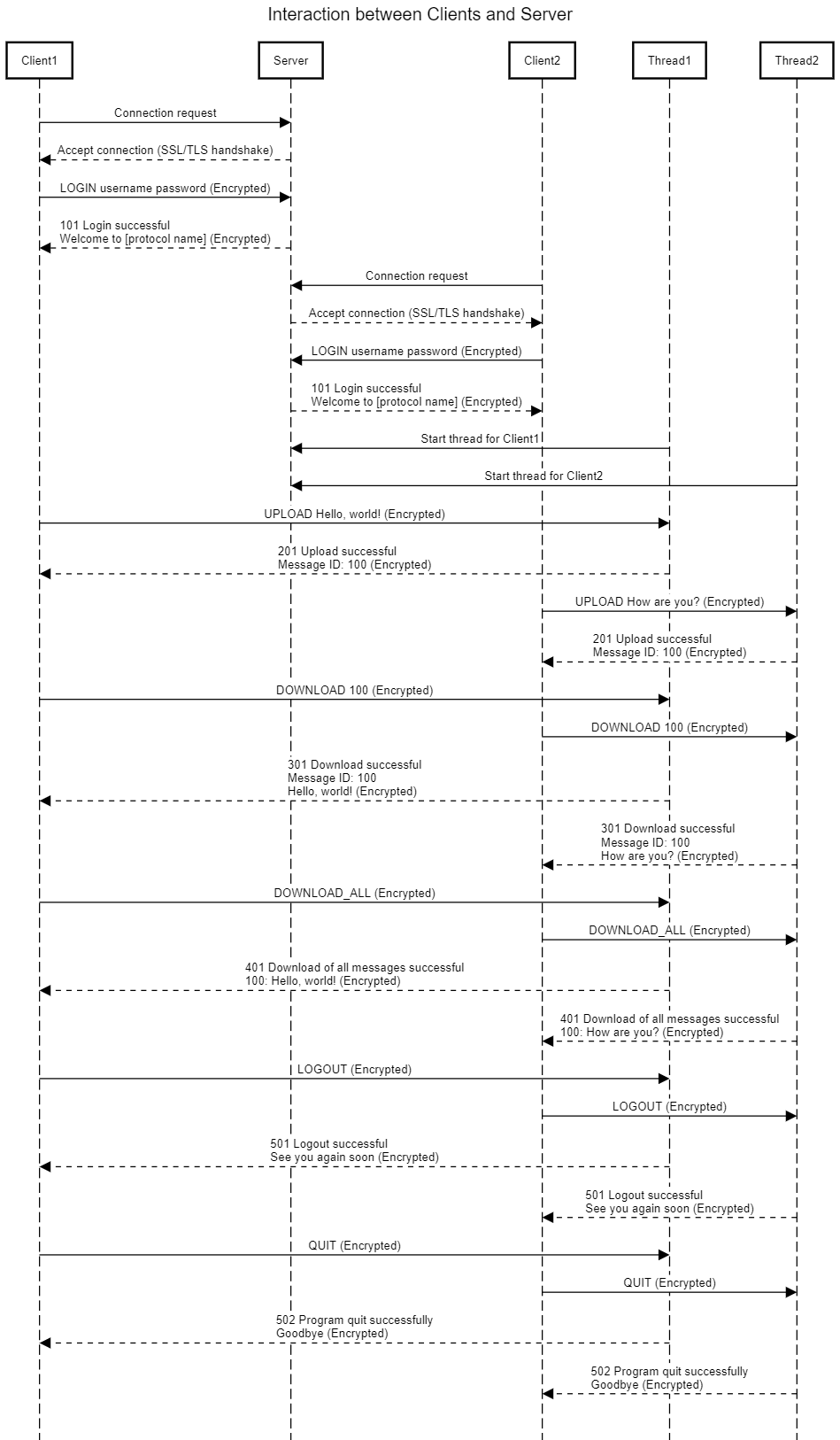


Fig 23 Created using sequencediagram.org

# Security Considerations

Communication between the client and server is encrypted using SSL/TLS, providing confidentiality and integrity for communication between server and client by encrypting the data being exchanged. SSL/TLS also enables mutual authentication via the handshake process. During this process, both the client and the server verify each other’s identity, and a secure communication channel is established.



# Author

Deirdre Lee

Munster Technological University

Dept. of Computer Science

Distributed Computing

Kerry

# References

Information Sciences Institute, University of Southern California, 1981. *TRANSMISSION CONTROL PROTOCOL DARPA INTERNET PROGRAM PROTOCOL SPECIFICATION.* [Online]   
Available at: https://datatracker.ietf.org/doc/html/rfc793  
[Accessed 4 March 2024].

Sengul, C. & Kirby, A., 2023. *Message Queuing Telemetry Transport (MQTT) and Transport Layer Security (TLS) Profile of Authentication and Authorization for Constrained Environments (ACE) Framework.* [Online]   
Available at: https://datatracker.ietf.org/doc/html/rfc9431  
[Accessed 4 March 2024].

TechTarget Contributor, 2021. *3-tier application architecture.* [Online]   
Available at: https://www.techtarget.com/searchsoftwarequality/definition/3-tier-application  
[Accessed 25 March 2024].