Network Systems Final Project Technical Report: Secure Multi-Threaded CLI Chat Application

Jonathan Setiawan, Kawata Ryota, Soncillan Nika Kristin Esclamado

Ritsumeikan University

January 12, 2026

1. Introduction

This application was developed for the purpose of the Network Systems Class final project which involves the development of a command-line based chat application using Java. The project is held with the purpose of showcasing how we can build a real-time communication system over a network for 2 different users using different devices. In the domain of network systems in particular, being able to conduct data exchange between multiple users simultaneously would require us to understand how transport protocols, socket programming, and concurrency work. Moreover, the challenge or the problem of this project is to be able to create a robust and comprehensive system that can handle communication without fault, handle multiple different users sessions at the same time, and packaging the entire thing in a way that is also safe and functional. To address these challenges, this project aims to build a system using Transmission Control Protocol (TCP) which would guarantee ordered and error-free delivery of data streams to make sure that users can have a reliable chat messaging experience.

The scope of the solution starts from a Client-Server architecture that is created using Java's standard networking libraries. According to the requirements discussed in the Network Systems Class week 9 presentation held by our instructor, the project demands a server and a client application. The server application is setup on a main computer and it will not only serve to wait for but also process user's requests. On the other hand, the client application is used by different users to connect to the server and communicate with each other. Additionally, we must fulfill the requirements of being able to handle the exchange of text messages, files, and direct messages between users. It is also recommended that we implement a list of channels on a server and users must be able to send a message to a channel for all currently connected users in that channel to see. In terms of technicality, we must implement and use TCP sockets, handle parsing for line-based commands, server concurrency using threads, broadcast against direct message routing, error handling, and protocol states. Notably, it is also suggested that we implement a text-based protocol and preferably a real standard protocol such as the IRC protocol that is used in this project.

In addition to the prescribed requirements for the scope of our solution, we decided to implement a multi-threaded server that is capable of handling multiple client connections at the same time without blocking anyone. This would mean that there is the need to implement a custom, text-based application layer protocol to handle user sessions, authentications, and message routing.

However, as a challenge, we decided to take it further and improve the server's features by adding a turn-based multiplayer battleship game to demonstrate our understanding of the client-server architecture. This additional challenge would require the server to go beyond simple message relaying and have it also be able to handle the states of numerous game sessions, establish game rules, and validate player moves. By extending the scope of the project to this extent, we believe that we can also learn more about session handling and relaying states across the server.

2. Related Work

Current architecture of real-time communication systems necessitates low-latency data exchange and robust security. As of now, the industry recognizes two primary models for full-duplex communication, the traditional socket-based model and the newer WebSocket protocol [3]. Although WebSockets are the standard for browser-based applications, low-level TCP Sockets still remain the global standard for CLI and system level tools because they provide a direct stream without the overhead of the HTTP handshaking [4]. We decided to use TCP Sockets as it aligned with the principles of the Internet Relay Chat (IRC) protocol [1], which is established to be the standard for line-based command parsing in multi-user environments.

In terms of establishing security in a messaging app, it is no longer an optional thing and is now a mandatory layer typically implemented via Transport Layer Security (TLS) especially in most modern messaging applications such as slack, whatsapp, discord, and snapchat. These apps use TLS version 1.2 or higher to prevent man-in-the-middle attacks during confidential data exchange. According to RFC 5246 which is the official technical documentation for TLS protocol version 1.2, TLS provides a secure tunnel for TCP traffic and ensures confidentiality through symmetric encryption using Message Authentication Codes (MACs) [2]. Furthermore, Secure Hash Standard (SHS), specifically the SHA-256 algorithm is widely known to be the industry standard for verifying the integrity of data transfers such as message or file exchange [5]. SHA-256 generates a 256-bit digital code that applications can detect even single bit corruptions during transit.

Lastly, to accommodate for the battleship game feature, Server-Authoritative logic is the global standard for secure multiplayer interactions such as World of Warcraft (WoW), Counter-Strike 2 , and League of Legends (LoL) [4]. In Counter-Strike 2 for example, the client acts as a passive terminal that only relays the user's inputs like movement or shooting while the server manages the secret state of the game environment. In professional game engineering, the server becomes the moderator holding the definitive state of the environment to prevent client-side exploitation or any forms of cheating. This design pattern is identical to the implementation of the battleship protocol, where coordinate validation and hit detection are handled purely on the server side.

3. Method

3.1 Analysis

To fulfill the project requirements, we want to be able to implement consistent message delivery, the ability to support multiple users on different laptops under the same internet connection, and secure data exchange using the IRC protocol. We decided that a client server architecture was the best choice for managing protocol states and synchronizing game data. Moreover, we decided on using TCP sockets as the transport layer so we can make sure that the line-based commands are ordered, error-checked, and are accurately parsing our IRC protocol syntax. Furthermore, we determined that a full two way communication flow through a dedicated receiver thread on the chat client. With this, we can ensure that users receive real time server updates for private messages or battleship actions without getting their own command input interrupted.

Due to the fact that we must be able to handle multiple users at the same time for server concurrency, there is a need to transition from a thread per client model to a fixed thread pool architecture using Java's ExecutorService. Through this, we also prevent server side crashes from excessive thread creation by essentially limiting the resource usage while maintaining fast response time from from all active ClientHandler instances. Each of these ClientHandler instances are also responsible for parsing and executing specific commands like NICK, JOIN, or MSG. Additionally, the battleship protocol requires a server authoritative logic flow to make sure that the game runs properly. In short, the server acts as the moderator to hold the private board states within the game session class and only broadcasting results, such as "HIT," or "MISS," to the relevant player. This kind of architecture not only prevents client-side cheating by hiding the opponent's coordinates but it also simplifies the client's display to an easy to understand interface.

3.2 Design (UML Diagrams)

3.2.1 Class Diagram

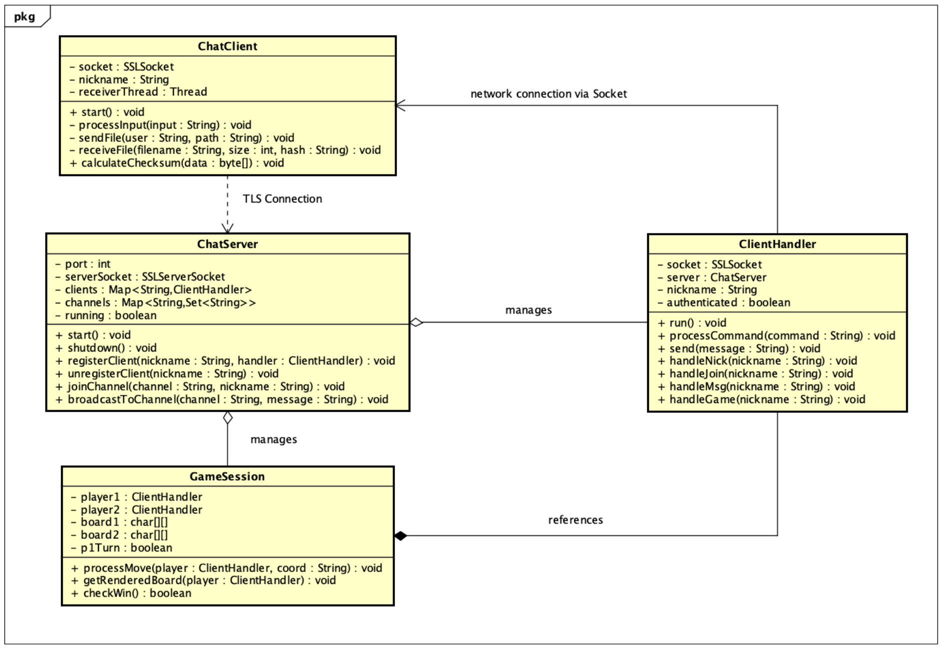
pkg
ChatClient
– socket : SSLSocket
– nickname : String
– receiverThread : Thread
+ start() : void
– processInput(input : String) : void
– sendFile(user : String, path : String) : void
– receiveFile(filename : String, size : int, hash : String) : void
+ calculateChecksum(data : byte[]) : void
network connection via Socket
TLS Connection
ChatServer
– port : int
– serverSocket : SSLServerSocket
– clients : Map<String,ClientHandler>
– channels : Map<String,Set<String>>
– running : boolean
+ start() : void
+ shutdown() : void
+ registerClient(nickname : String, handler : ClientHandler) : void
+ unregisterClient(nickname : String) : void
+ joinChannel(channel : String, nickname : String) : void
+ broadcastToChannel(channel : String, message : String) : void
manages
ClientHandler
– socket : SSLSocket
– server : ChatServer
– nickname : String
– authenticated : boolean
+ run() : void
+ processCommand(command : String) : void
+ send(message : String) : void
+ handleNick(nickname : String) : void
+ handleJoin(nickname : String) : void
+ handleMsg(nickname : String) : void
+ handleGame(nickname : String) : void
manages
GameSession
– player1 : ClientHandler
– player2 : ClientHandler
– board1 : char[][]
– board2 : char[][]
– p1Turn : boolean
+ processMove(player : ClientHandler, coord : String) : void
+ getRenderedBoard(player : ClientHandler) : void
+ checkWin() : boolean
references

Diagram 1

In Diagram 1, the chat server acts as the center of the entire system which manages a SSLServerSocket to satisfy security requirements and utilizing a map of nicknames to ClientHandler objects for direct message routing. To handle server concurrency, the server uses an ExecutorService thread pool to manage multiple ClientHandler instances which are the server side representatives for each connected user. Each of these ClientHandler instances contain dedicated logic for parsing the line-based commands that the users can type in the command line through method such as handeNick, handleJoin, handleMsg, and handleGame.

As for the ChatClient, it manages the end-user interface and maintains a TLS connection to the server for communication. It implements a receiverThread to allow for two way communication and includes a calculateChecksum method to make sure that the file exchanges between users are secure. Finally, the GameSession class is an encapsulation for the battleship game logic to hold private board data and turn states while acting as a server-authoritative moderator to manage the interaction between two players.

3.2.2 Sequence Diagram (IRC protocol / Client Server Management)

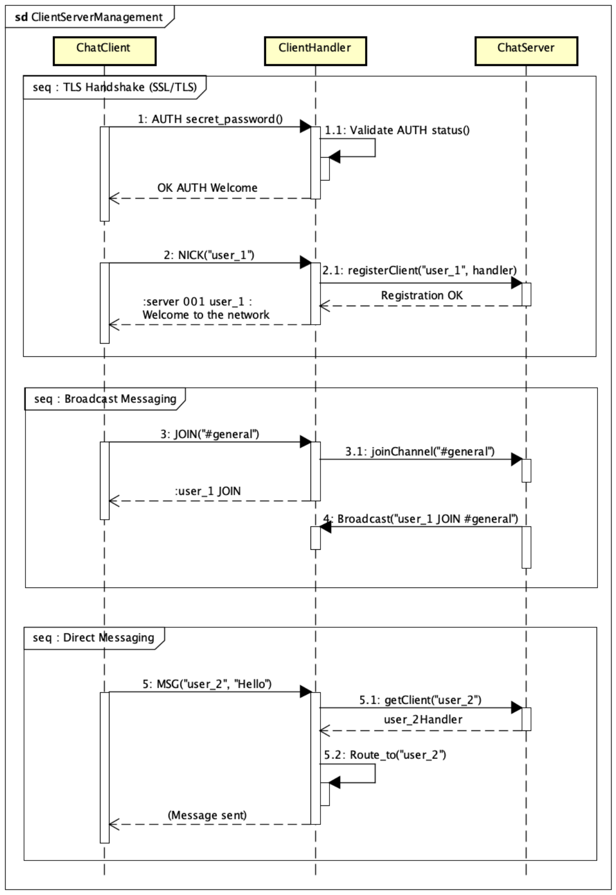
sd ClientServerManagement
ChatClient
ClientHandler
ChatServer
seq : TLS Handshake (SSL/TLS)
1: AUTH secret_password()
1.1: Validate AUTH status()
OK AUTH Welcome
2: NICK("user_1")
2.1: registerClient("user_1", handler)
Registration OK
:server 001 user_1 : Welcome to the network
seq : Broadcast Messaging
3: JOIN("#general")
3.1: joinChannel("#general")
:user_1 JOIN
4: Broadcast("user_1 JOIN #general")
seq : Direct Messaging
5: MSG("user_2", "Hello")
5.1: getClient("user_2")
user_2Handler
5.2: Route_to("user_2")
(Message sent)

Diagram 2

The first section of the sequence is the TLS handshake and authentication. This section is used to establish a secure session between the ChatClient and the server using TLS encryption over TCP sockets. During this section, the client must successfully complete an AUTH sequence with a secret password before they are permitted to set a nickname via the NICK command. The ClientHandler validates these protocol states and only after a successful registration with the ChatServer does it return a welcome message.

For the second section which is broadcast messaging, it illustrates the system's ability to manage group communications within virtual channels. When a user types a JOIN command for a specific channel, for example #general, the ClientHandler queries the server's joinChannel. The ChatServer then processes this request and performs a broadcast to all other connected ClientHandler instances or users to that channel.

The final section is direct messaging, this fulfills the direct message routing logic used for one to one private communication. In this scenario, when user_1 sends a MSG command targeted at user_2, the ClientHandler requests the specific recipient's handler from the ChatServer using a nickname lookup. Once received, the message is routed directly to the target's output stream through the Route_to method so that instead of having that message be broadcasted to the entire network, it's only sent specifically to user_2. Eventually, the message will be received by user_2 after the routing is completed and a notification will be sent to user_1 that the message has been sent and user_2 will be able to see the message.

3.2.3 Sequence Diagram (Game Session Management)

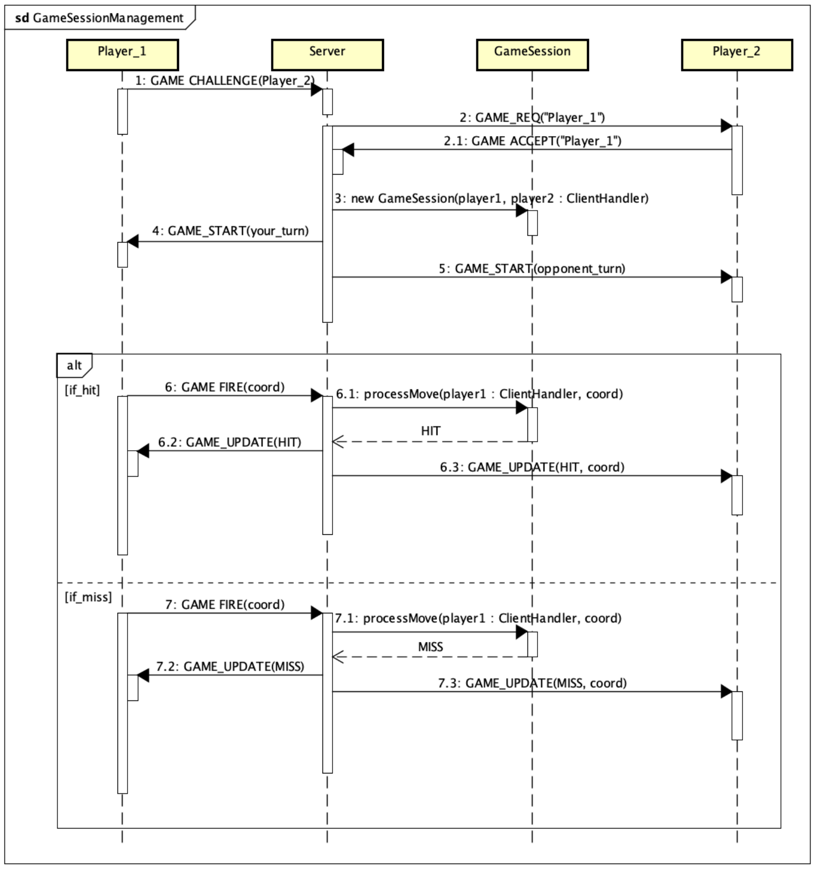
sd GameSessionManagement
Player_1
Server
GameSession
Player_2
1: GAME CHALLENGE(Player_2)
2: GAME_REQ("Player_1")
2.1: GAME ACCEPT("Player_1")
3: new GameSession(player1, player2 : ClientHandler)
4: GAME_START(your_turn)
5: GAME_START(opponent_turn)
alt
[if_hit]
6: GAME FIRE(coord)
6.1: processMove(player1 : ClientHandler, coord)
HIT
6.2: GAME_UPDATE(HIT)
6.3: GAME_UPDATE(HIT, coord)
[if_miss]
7: GAME FIRE(coord)
7.1: processMove(player1 : ClientHandler, coord)
MISS
7.2: GAME_UPDATE(MISS)
7.3: GAME_UPDATE(MISS, coord)

Diagram 3

The game begins with a connection phase where a player issues a game challenge to another user through the server. The server then acts as a relay for that request and upon receiving a game accept from the other player, it will initialize a new GameSession containing both players' ClientHandler references. After it is created, the server will notify both of the users that the session has started and displays the rules and interface for the battleship game.

The next phase is the gameplay phase which requires a moderator to maintain the game session and prevent any unfairness. When player1 issues a GAME FIRE command with specific coordinates, the request is routed to the server which then prompts the processMove method within the GameSession class. Moreover, the server also hides all the board data from the opponents and only the results of the move are validated by the internal game state. This makes sure that the hidden information of the battleship board from the opponent's board state remains visible only to their side. Lastly, there are two possible outcomes from when a player fires, it will either result in a HIT or a MISS depending if they successfully hit a ship or not. If the server determines that a move is a HIT, then it will broadcast GAME_UPDATE(HIT) to the attacker while also informing the defender that the specific coordinate was hit. However, if it was a MISS, then it will trigger a GAME_UPDATE(MISS) broadcast to both parties. This makes it a turn based game with proper state and property management.

3.3 Implementation Details

The implementation of the ChatServer required concurrency and secure communication. To do that, we had to use FixedThreadPool via Java's ExecutorService. This allows the server to manage multiple ClientHandler instances effectively to prevent the server from being overwhelmed by too many requests. As for the security, the system uses SSL/TLS encrypted sockets for all the network traffic to make sure that private messages and important game data are protected against packet sniffing which can be seen when using wireshark. The communication itself follows a strict set of rules defined by a line-based text protocol where the processCommand method in the ClientHandler acts as a central parser for IRC-style commands like /nick, /join, and /msg.

In terms of how the battleship game logic and file exchange are handled, the GameSession class implements a server-authoritative moderator model where the server maintains the only copy of the 10x10 grids and validates all coordinates via the processMove method before broadcasting the results back to the players during the game. This design makes it so that the protocol state is maintained by forcing the the players to take turn orders and prevents the clients from viewing hidden board data. Furthermore, the file transfer functionality includes a SHA-256 checksum verification process. The calculateChecksum method generates a unique hash for each file which the receiving client validates to ensure that the data was not corrupted during the TCP socket transfer.

4. Discussion

4.1 Implemented Solution

The implemented system is a strong, secure, multi threaded chat environment that combines IRC style communication with an interactive battleship game module. At its core, the solution utilizes a client server architecture built using Java’s SSLSocket and SSLServerSocket classes, to ensure that all data in transit is encrypted. The server utilises a FixedThreadPool via ExecutorService to handle concurrency which separates the task of message processing from threat management preventing resource exhaustion. The communication protocol is strictly line based, allowing for easy terms of commands such as NICK, JOIN and MSG. For the file transfers, the implementation includes a SHA-256 integrity check, providing a layer of security against data corruption or tampering during the TCP stream transfer.

4.2 Strong Points

This project's strong points are mainly divided into three sections. Enhanced security, scalability and resource management, and asynchronous communication. The architecture of our chat application in enhanced security by moving beyond simple socket communication. By approving a TLS handshake before any data exchange, we ensure that every packet, whether it’s a private message or a command, is encrypted against eavesdropping. This is further strengthened by the initial AUTH sequence, which prevents unauthorised access to the server. By combining SHA-256 checksums for file transfers, the system provides end to end integrity allowing the recipient to mathematically verify that file has not been tampered with or corrupted during transit. In terms of scalability and resource management the transition from a “threat per client," model to a FixedThreadPool via Java’s ExecutorService represents a significant technical advantage. Traditional multi-threaded servers are vulnerable to resource exhaustion if too many users connect at the same time. Our implementation reduces the risk by capping the number of active threads ensuring that server’s CPU and memory usage remain predictable. This design allows the system to maintain high responsiveness and stability even as the number of concurrent connections grows, preventing the server crashes with excessive thread creation.

Finally the implementation of asynchronous communication significantly improves the user experience. By utilising a dedicated receiverThread on the client side, the application separates the user’s input stream from the network’s output stream. This means that user can continuously type their commands or messaging without being interrupted by incoming data. This means that users can continuously type their commands or messages without being interrupted by incoming data. This full dual communication flow ensures the real time updates are displayed instantly while the user maintains full control over the CLI creating an interactive environment that mimics high level chat protocols.

4.3 Weak Points

The weak point is reliance on a centralised failure point. If the ChatServer goes down, all the active communications will be lost because there are no distributed failover mechanisms. Furthermore, the lack of persistence means that all user sessions, nicknames, and channel data are stored in volatile memory. If the server restarts, all current states are wiped. Lastly the fixed thread pool introduces a hard limit on capacity, once all threads are occupied, new users may experience latency or connection delays until a slot becomes available.

5. Conclusion

This project is the application of network systems principles through the development of a secure multi threaded CLI Chat Application. By using Java's SSL/TLS capabilities and FixedThreadPool, we managed to create a platform that prioritises both data privacy and operational stability. The implementation of an IRC-style messaging protocol allowed us to explore complex state management, direct message routing and the challenge of real time synchronisation.

The project met all primary objectives, particularly the requirement for consistent message delivery and secure data exchange for multiple users operating under a shared network environment. A key takeaway from this development process was the transition from a naive "thread-per-client," architecture to a sophisticated fixed thread pool model.

This shift provided deep insights into how resource-constrained environments such as a personal laptop can be optimised to handle concurrent connections without succumbing to memory exhaustion or CPU thrashing. Furthermore, the combination of a server authoritative battleship game logic proved that a well designed network protocol can support not only basic communication but also complex, state dependent applications while preventing client side manipulation.

The use of SHA-256 checksums for file transfers and the enforcement of an AUTH sequence before protocol interaction underscored the importance of defense in depth in network security. These layers ensure that the system remains strong against both packet sniffing and unauthorised access. However, as noted in the discussion, the current iteration remains a volatile system. While highly effective for real time interaction, the reliance on a single, non persistent server instance identifies clear paths for future optimisation.

The system could be further developed into a service level by introducing a database layer (such as PostgreSQL or Redis) to provide chat logs and user profiles, ensuring that data is not lost during server restarts. Additionally, implementing a dynamic thread pool or an asynchronous I/O model (like Java NIO) could allow the server to scale to thousands of users with even less overhead.

6. References

[1] J. Oikarinen and D. Reed, "Internet Relay Chat Protocol," RFC 1459, May 1993.

[2] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, Aug. 2008.

[3] I. Fette and A. Melnikov, "The WebSocket Protocol," RFC 6455, Dec. 2011.

[4] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 8th ed. Pearson, 2021.

[5] Q. Dang, "Secure Hash Standard (SHS)," Federal Information Processing Standards (FIPS) Pub 180-4, July 2015.