COMP90015: Distributed Systems
Assignment 2 Report

Student Name: Yiran Wang
Student ID: 1366272

1. Problem Context

The objective of this project is to follow the principles of distributed computing to develop a real-time Tic-Tac-Toe gaming ecosystem. This system comprises two primary elements: the game server and the game client. Both components must employ either Remote Method Invocation (RMI) or Java sockets to facilitate real-time networked communication and gameplay between players.

The functional requirements for this system include:

- Player matching
- Turn-based game play
- Real-time chat in the game
- Quit game option
- Option to play a new game after the current one finished
- Timeout mechanism for each move
- Player ranking system
- Both Client-side and server-side fault tolerance in the event of a system crash

A non-functional requirement of this system is the design and integration of a client-side user-friendly interface. This interface should not only provide a seamless gameplay experience but also allow for real-time chat functionalities, thereby enabling users to interact with each other.

2. System Components

2.1 Overview

This project employs RMI techniques to manage networked interactions. When the server launches, it generates a remote object and binds it to the registry. This enables clients to fetch the remote object and invoke its methods to interact with the server. On the other hand, each client, upon launching its application, also creates a remote object, which is sent to the server, enabling the server to invoke its methods. A graphical representation offering a high-level overview of the interactions between the server and clients is provided in Figure 2.1.

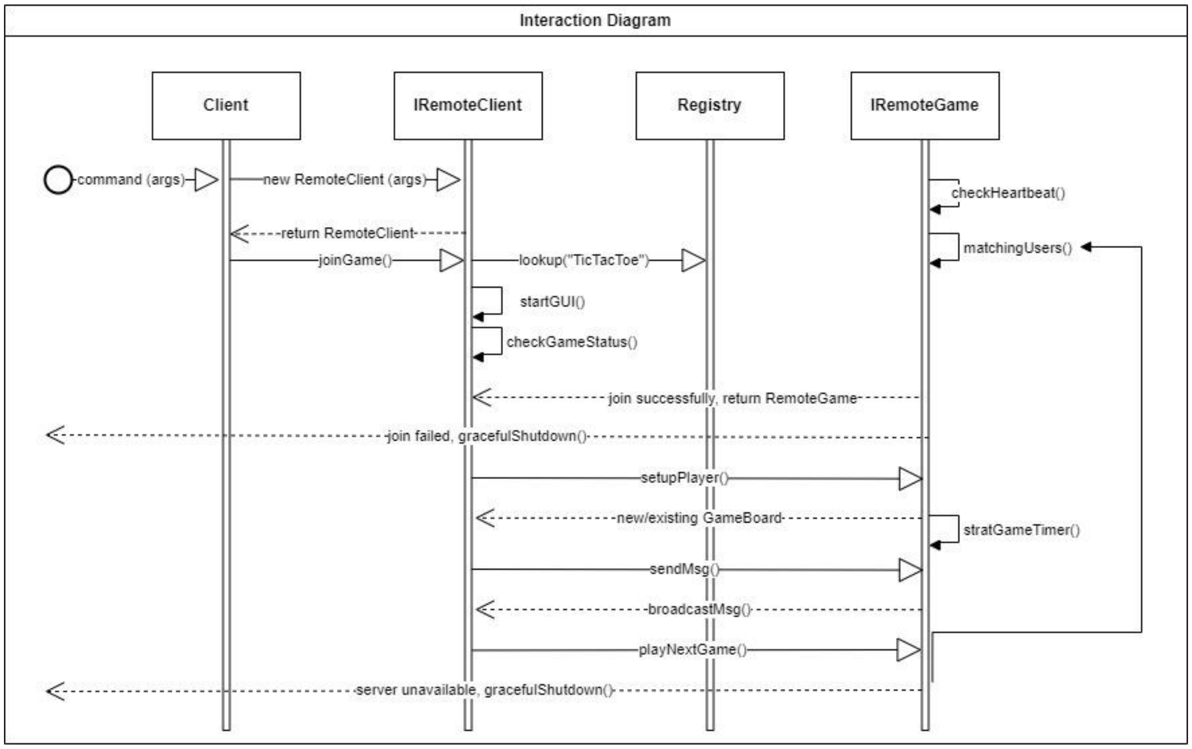


Figure 2.1 Interaction diagram

2.2 Server

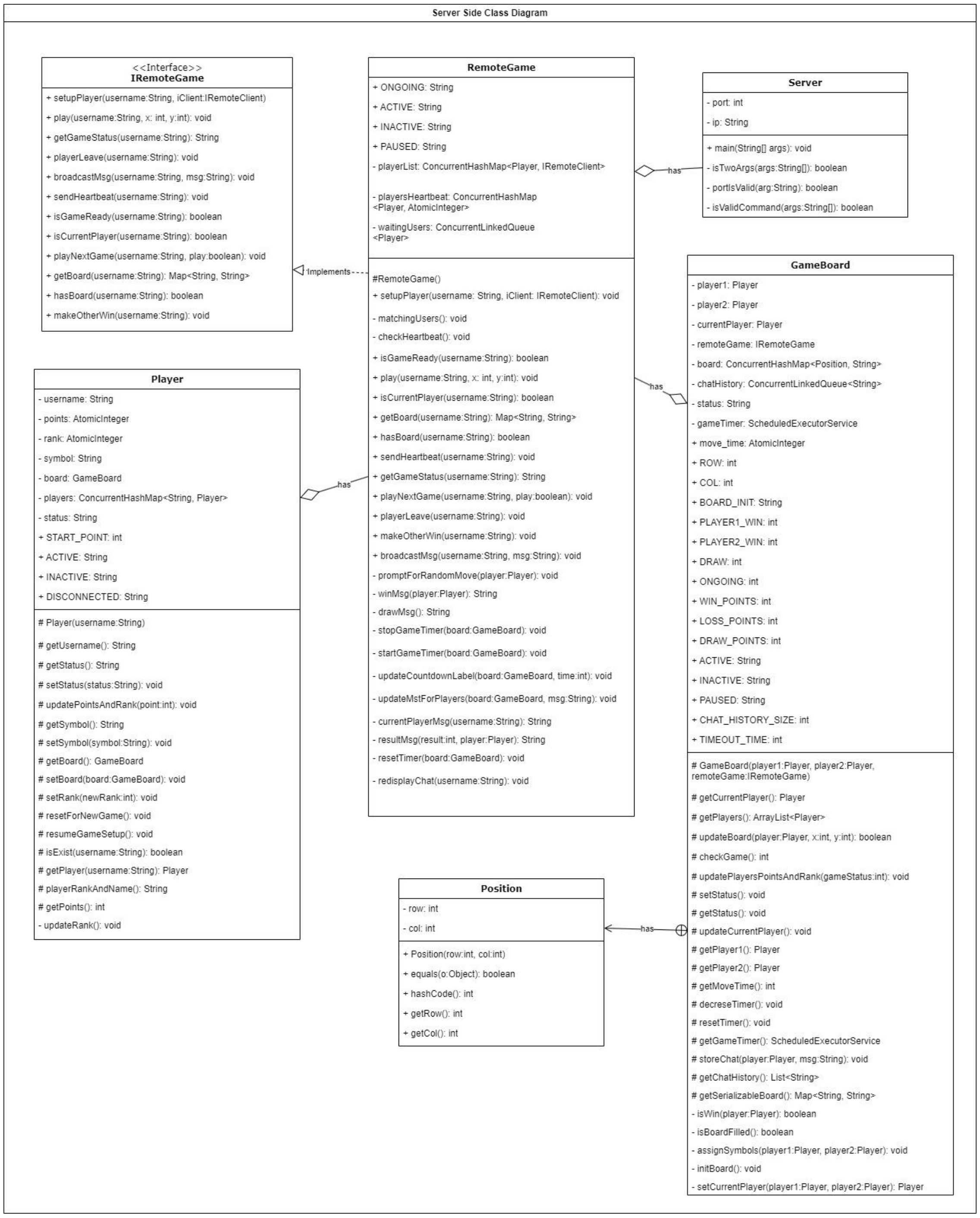
As can be seen in Figure 2.2, the server has three main parts: *Player*, *GameBoard* and *RemoteGame* which implements *IRemoteGame*. Each of them is responsible for different functionalities.

2.2.1 Player

The *Player* class serves as a repository for various attributes and states of each player, including *symbol*, *points*, *rank*, and game *board*. It maintains a static *ConcurrentHashMap* to track all player instances. This same *ConcurrentHashMap* is also used for updating each player's *rank*. When a game concludes, the player's *points* and *rank* are updated. Upon initiating a new game, each player is allocated a new *symbol* and game *board*. The player *status* is set to *ACTIVE* by default but switches to *DISCONNECTED* if the client failed to send a heartbeat in a 3-second window. Additionally, when a player quits a game, the status is immediately set to *INACTIVE*.

2.2.2 GameBoard

The GameBoard class serves as a game manager for overseeing real-time board status, player chats, and game state control. The class maintains a *currentPlayer* attribute, which is dynamically updated after each move. Additionally, the moment a game is created, a dedicated *ScheduledExecutorService,* referred to as *gameTimer,* is activated. This timer enforces a 20-second time limit for each move. The class also features a game *status* attribute that can transition among ACTIVE, INACTIVE, and PAUSED states. This game *status* is contingent on whether a game has concluded, and on the statuses of the players. To ensure thread-safe operations, the game *board* and *chatHistory* are stored in a *ConcurrentHashMap* and *ConcurrentLinkedQueue*, respectively.



Server Side Class Diagram
<<Interface>>
IRemoteGame
+ setupPlayer(username:String, iClient:IRemoteClient)
+ play(username:String, x: int, y:int): void
+ getGameStatus(username:String): String
+ playerLeave(username:String): void
+ broadcastMsg(username:String, msg:String): void
+ sendHeartbeat(username:String): void
+ isGameReady(username:String): boolean
+ isCurrentPlayer(username:String): boolean
+ playNextGame(username:String, play:boolean): void
+ getBoard(username:String): Map<String, String>
+ hasBoard(username:String): boolean
+ makeOtherWin(username:String): void
implements
RemoteGame
+ ONGOING: String
+ ACTIVE: String
+ INACTIVE: String
+ PAUSED: String
- playerList: ConcurrentHashMap<Player, IRemoteClient>
- playersHeartbeat: ConcurrentHashMap <Player, AtomicInteger>
- waitingUsers: ConcurrentLinkedQueue <Player>
#RemoteGame()
+ setupPlayer(username: String, iClient: IRemoteClient): void
- matchingUsers(): void
- checkHeartbeat(): void
+ isGameReady(username:String): boolean
+ play(username:String, x: int, y:int): void
+ isCurrentPlayer(username:String): boolean
+ getBoard(username:String): Map<String, String>
+ hasBoard(username:String): boolean
+ sendHeartbeat(username:String): void
+ getGameStatus(username:String): String
+ playNextGame(username:String, play:boolean): void
+ playerLeave(username:String): void
+ makeOtherWin(username:String): void
+ broadcastMsg(username:String, msg:String): void
- promptForRandomMove(player:Player): void
- winMsg(player:Player): String
- drawMsg(): String
- stopGameTimer(board:GameBoard): void
- startGameTimer(board:GameBoard): void
- updateCountdownLabel(board:GameBoard, time:int): void
- updateMstForPlayers(board:GameBoard, msg:String): void
- currentPlayerMsg(username:String): String
- resultMsg(result:int, player:Player): String
- resetTimer(board:GameBoard): void
- redisplayChat(username:String): void
has
Server
- port: int
- ip: String
+ main(String[] args): void
- isTwoArgs(args:String[]): boolean
- portIsValid(arg:String): boolean
- isValidCommand(args:String[]): boolean
Player
- username: String
- points: AtomicInteger
- rank: AtomicInteger
- symbol: String
- board: GameBoard
- players: ConcurrentHashMap<String, Player>
- status: String
+ START_POINT: int
+ ACTIVE: String
+ INACTIVE: String
+ DISCONNECTED: String
# Player(username:String)
# getUsername(): String
# getStatus(): String
# setStatus(status:String): void
# updatePointsAndRank(point:int): void
# getSymbol(): String
# setSymbol(symbol:String): void
# getBoard(): GameBoard
# setBoard(board:GameBoard): void
# setRank(newRank:int): void
# resetForNewGame(): void
# resumeGameSetup(): void
# isExist(username:String): boolean
# getPlayer(username:String): Player
# playerRankAndName(): String
# getPoints(): int
- updateRank(): void
has
has
GameBoard
- player1: Player
- player2: Player
- currentPlayer: Player
- remoteGame: IRemoteGame
- board: ConcurrentHashMap<Position, String>
- chatHistory: ConcurrentLinkedQueue<String>
- status: String
- gameTimer: ScheduledExecutorService
+ move_time: AtomicInteger
+ ROW: int
+ COL: int
+ BOARD_INIT: String
+ PLAYER1_WIN: int
+ PLAYER2_WIN: int
+ DRAW: int
+ ONGOING: int
+ WIN_POINTS: int
+ LOSS_POINTS: int
+ DRAW_POINTS: int
+ ACTIVE: String
+ INACTIVE: String
+ PAUSED: String
+ CHAT_HISTORY_SIZE: int
+ TIMEOUT_TIME: int
# GameBoard(player1:Player, player2:Player, remoteGame:IRemoteGame)
# getCurrentPlayer(): Player
# getPlayers(): ArrayList<Player>
# updateBoard(player:Player, x:int, y:int): boolean
# checkGame(): int
# updatePlayersPointsAndRank(gameStatus:int): void
# setStatus(): void
# getStatus(): void
# updateCurrentPlayer(): void
# getPlayer1(): Player
# getPlayer2(): Player
# getMoveTime(): int
# decreseTimer(): void
# resetTimer(): void
# getGameTimer(): ScheduledExecutorService
# storeChat(player:Player, msg:String): void
# getChatHistory(): List<String>
# getSerializableBoard(): Map<String, String>
- isWin(player:Player): boolean
- isBoardFilled(): boolean
- assignSymbols(player1:Player, player2:Player): void
- initBoard(): void
- setCurrentPlayer(player1:Player, player2:Player): Player
has
Position
- row: int
- col: int
+ Position(row:int, col:int)
+ equals(o:Object): boolean
+ hashCode(): int
+ getRow(): int
+ getCol(): int

Figure 2.2 Server-side UML diagram

2.2.3 Remote Game

The *RemoteGame* implements the *IRemoteGame* interface along with its remote methods. It is responsible for managing the client connections and interactions, while also overseeing all players' status. When the server starts, two threads – *matchingUsers* and *checkHeartbeat*– run continuously. The *matchingUsers* thread scans the waiting queue to check whether there are sufficient players to be matched up for a game. On the other hand, *checkHeartbeat* thread monitors client disconnects. and updates both game and player statuses accordingly.

2.3. Client

The *RemoteClient* class, implementing the *IRemoteClient* interface, serves as the client-side hub for this system. Upon starting the application, a RemoteClient object is created, and a graphical user interface (ClientGUI) is started. The class is responsible for configuring the client-side environment, including joining a game, making moves, and rendering real-time updates to the GUI. To ensure the server is informed of its connection status,the class utilises a *ScheduledExecutorService* to dispatch periodic heartbeats. Additionally, the client starts a separate thread, *checkGameStatus*, upon game launch. This thread is tasked with monitoring the game's *status*. Should the game conclude or the other player exit, the user is offered the choice to enter a new game via a pop-up window. If the other player becomes disconnected, the user is alerted through the chat box, and the game board is subsequently frozen. The class also incorporates exception-handling and gracefulShutdown mechanisms to address potential network failures or server crashes.

2.4 Message Exchange Protocol

Two primary categories of messages are exchanged within the system: game board updates and chat messages. To reduce the bandwidth, the *updateBoard* method transmits only the incremental changes in game board to each player after every move. These incremental changes are stored in a *ConcurrentHashMap<Position, String>* on the server-side to reflect the current state of the board. Should a player crash and subsequently rejoin the game, the server serialises the game board into a *Map<String, String>* and transmits it to the rejoining player. In terms of chat messages, they are stored in a *ConcurrentLinkedQueue* as simple strings. Each incoming message is added to the queue and then broadcasted by the server to both players engaged in the same game.

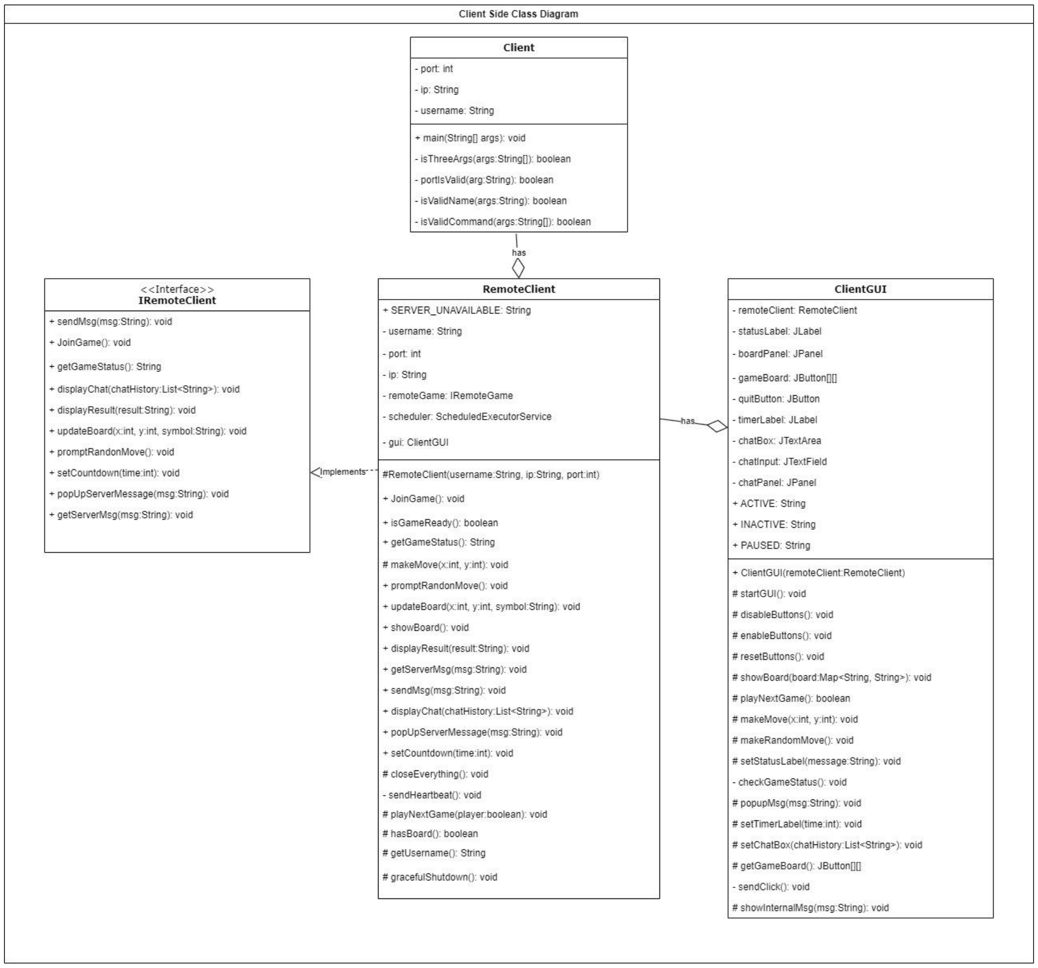

Client Side Class Diagram
Client
- port: int
- ip: String
- username: String
+ main(String[] args): void
- isThreeArgs(args:String[]): boolean
- portIsValid(arg:String): boolean
- isValidName(args:String): boolean
- isValidCommand(args:String[]): boolean
has
<<Interface>>
IRemoteClient
+ sendMsg(msg:String): void
+ JoinGame(): void
+ getGameStatus(): String
+ displayChat(chatHistory:List<String>): void
+ displayResult(result:String): void
+ updateBoard(x:int, y:int, symbol:String): void
+ promptRandonMove(): void
+ setCountdown(time:int): void
+ popUpServerMessage(msg:String): void
+ getServerMsg(msg:String): void
Implements
RemoteClient
+ SERVER_UNAVAILABLE: String
- username: String
- port: int
- ip: String
- remoteGame: IRemoteGame
- scheduler: ScheduledExecutorService
- gui: ClientGUI
#RemoteClient(username:String, ip:String, port:int)
+ JoinGame(): void
+ isGameReady(): boolean
+ getGameStatus(): String
# makeMove(x:int, y:int): void
+ promptRandonMove(): void
+ updateBoard(x:int, y:int, symbol:String): void
+ showBoard(): void
+ displayResult(result:String): void
+ getServerMsg(msg:String): void
+ sendMsg(msg:String): void
+ displayChat(chatHistory:List<String>): void
+ popUpServerMessage(msg:String): void
+ setCountdown(time:int): void
# closeEverything(): void
- sendHeartbeat(): void
# playNextGame(player:boolean): void
# hasBoard(): boolean
# getUsername(): String
# gracefulShutdown(): void
has
ClientGUI
- remoteClient: RemoteClient
- statusLabel: JLabel
- boardPanel: JPanel
- gameBoard: JButton[][]
- quitButton: JButton
- timerLabel: JLabel
- chatBox: JTextArea
- chatInput: JTextField
- chatPanel: JPanel
+ ACTIVE: String
+ INACTIVE: String
+ PAUSED: String
+ ClientGUI(remoteClient:RemoteClient)
# startGUI(): void
# disableButtons(): void
# enableButtons(): void
# resetButtons(): void
# showBoard(board:Map<String, String>): void
# playNextGame(): boolean
# makeMove(x:int, y:int): void
# makeRandomMove(): void
# setStatusLabel(message:String): void
- checkGameStatus(): void
# popupMsg(msg:String): void
# setTimerLabel(time:int): void
# setChatBox(chatHistory:List<String>): void
# getGameBoard(): JButton[][]
- sendClick(): void
# showInternalMsg(msg:String): void

Figure 2.3 Client-side UML diagram

3. Critical Analysis

3.1 Strength

3.1.1 Centralised Server Architecture

This system adopts the centralised server architecture featuring a singular server and multiple clients. This design choice simplifies state management and eases both deployment and maintenance. With all data and logic housed in a single control point, it becomes straightforward to implement features like matchmaking, player reconnection, and real-time updates while ensuring data consistency.

3.1.2 Use of TCP Protocol

The system uses TCP, facilitated through Java RMI, to ensure reliable, ordered, and error-checked delivery of messages between the server and clients. TCP's inherent features like flow control and congestion management contribute to a more stable user experience. As Tic-Tac-Toe is a turn-based game where every move is critical, the guaranteed delivery and data integrity offered by TCP is critical to the success of the system.

3.1.3 Error Handling

Both client and server components incorporate comprehensive exception handling mechanisms. This is evident in the way the system handles AccessException, AlreadyBoundException, ConnectException, NumberFormatException, NoSuchObjectException and InterruptedException. Additionally, the system further elevates its fault tolerance through its effective utilisation of Java's RemoteException. This exception serves as an alert to the opposite end of the communication channel in the event of a crash. When a client crashes, the server encounters a RemoteException during the next remote method invocation, thereby receiving a notification that the player has disconnected. If the server crashes, the client will receive a RemoteException, triggering the gracefulShutdown method to enact a graceful exit within a 5-second window.

3.2 Weakness

3.2.1 Unencrypted communications

The system's lack of encrypted communications between the server and clients constitutes a notable vulnerability. This absence of encryption opens the door to various security threats, including eavesdropping and data manipulation. For instance, players might share sensitive information through in-game chat. Without secure channels, such data remains susceptible to unauthorised interception and access.

3.2.2 Single Server

While the centralised server architecture presents merits in terms of control and simplicity, it imposes significant scalability limitations. As the user base expands, the single-server setup could become overwhelmed, thus leading to slower response times or even potential service outages.

3.2.3 Vulnerable to System Abuse

The system's current design inadequately addresses both edge cases and the potential for malicious exploitation. For example, a player could exploit the system to falsely secure a win by repeatedly disconnecting to force the other player to quit. Such vulnerabilities expose the system to misuse and compromise the enjoyment of the gameplay experience.

4. Conclusion

In conclusion, the project successfully delivers a distributed Tic-Tac-Toe game that meets all stipulated functional and non-functional criteria outlined in the problem context. The system has a blend of merits and drawbacks. Its strengths lie in a centralised server architecture that ensures control and data consistency, the utilisation of TCP for reliable message transmission, and meticulous error handling strategies. However, to elevate its scalability, robustness, and security, addressing its inherent limitations remains crucial. These include mitigating the single-server bottleneck, fortifying the system against potential abuse, and implementing encrypted communications for enhanced security.