COMP90015 Assignment2 Report

Student ID: 1394392

Name: Renfei Yu (Klaus)

Problem Context of assignment

The assignment aims to build a distributed Tic-Tac-Toe gaming system with 3*3 grid in Java, which can play game and real-time chatting between players. The project can be implemented by both Java Sockets or RMI. The game involves a GUI on the client side, whereas the server will be a central entity without a GUI.

Basic System

- Client which includes features of:

1. Username input from a command-line argument.
2. A GUI to visually represent the game board.
3. Real-time game state reflection.
4. Ability to chat with opponents with a limitation of the message history size.
5. Option to exit the game anytime.
6. Displays game outcome.
7. Option to either rematch or exit after a game.

- Server which includes features of:

1. Handling multiple clients.
2. Managing active players and matching them to play.
3. Randomly assigning the first turn and symbols to players.
4. Keeping track of the game state.
5. Checking for a winning condition and communicating the result.
6. Handling players who quit midway.
7. Design Considerations:
8. Concurrency and access to shared resources.
9. Networked communication and establishing an exchange protocol.
10. Choice of either Java sockets or RMI for communication.

Components of the system

In this assignment, I choose to use client-server model same as the assignment1. One server will serve multiple clients and I will use RMI to implement the program. There are several classes that will be implemented in the project.

1. ServerInterface

The server Interface will be implemented as an interface that declares the remote functionalities of server. The ServerInterface describes methods that can be remotely called on the game server by RMI clients. These are the actions that a client can request of the server in the context of a Tic-Tac-Toe game.

Methods:

1. **move(int gameId, String username, int i, int j)**:
    - This method requests a move to be made on behalf of a player.
    - Parameters:
        - **gameId**: The unique identifier for a game.
        - **username**: The name of the player making the move.
        - **i** and **j**: The board coordinates where the player wants to place their symbol.
2. **initPlayer(String username, ClientInterface client)**:
    - Initializes a player with the provided username and their corresponding client interface.

- Parameters:
    - **username**: The name of the player.
    - **client**: An interface to the client, allowing the server to make callbacks.

3. **quitGame(int gameId, String username)**:
    - Requests that a player quits a specific game.
    - Parameters:
        - **gameId**: The unique identifier for the game that the player wants to quit.
        - **username**: The name of the player who wants to quit.
4. **handleMessageSending(int gameId, String username, String text)**:
    - Sends a chat or message from a player in a specific game.
    - Parameters:
        - **gameId**: The unique identifier for the game where the message is being sent.
        - **username**: The name of the player sending the message.
        - **text**: The content of the message.
5. **quitWaiting(String username)**:
    - Requests that a waiting player (probably in a queue for a game) quits waiting.
    - Parameters:
        - **username**: The name of the player who wants to quit waiting.

6. **getGameBoardByid(int gameId)**:

    - Retrieves the current state of the game board for a specific game.
    - Parameters:
        - **gameId**: The unique identifier for the game.
    - Returns:
        - A 2D char array representing the game board.

2. GameServer

The server of the Tic Tac Toe game, it will be used to handle client connections and sessions allocation. It is the server-side code for a Tic-Tac-Toe game implemented using Java's RMI (Remote Method Invocation) system. The game is set up to handle multiple games concurrently with different pairs of players and chat functionalities.

Variables:

- **players:** A list of all players.
- **queuingPlayers:** A list of players who are waiting for a match.
- **currentGames**: A hashmap storing the ongoing games with game IDs as keys.

random: A random generator used in various places in the code.

Methods

Player Management:

- **getAllPlayers()**: Returns all registered players.
- **getAllQueuingPlayers()**: Returns all players currently in the queue.
- **getRandomPlayer()**: Returns a random player from the queuing players.
- **assignRandomSymbols(Player tempPlayer, Player enemy)**: Randomly assigns 'X' or 'O' to the given players.
- **assignRandomStartTurn(Player tempPlayer, Player enemy):** Randomly decides which player gets the first move.

Game Flow and Logic:

- **initPlayer(String username, ClientInterface client):** Initializes a player and either waits for an opponent or starts a game if an opponent is found in the queue.
- **reRank():** Reranks players based on their points.
- **quitGame(int gameId, String username):** Handles a player quitting the game.
- **move(int gameId, String username, int i, int j)**: Processes a player's move on the board.
- **handleMessageSending(int gameId, String username, String text)**: Handles the chat messages sent by players.

Information Retrieval:

- **getGameByID(int id)**: Returns the Game object for a given game ID.
- **getGameBoardByid(int gameId):** Returns the game board of a given game ID.

Server Initialization:

- **main(String[] args)**: The main entry point that initializes the RMI registry and binds the server object, making it available for remote calls.

3. Game

The class that allocated by the GameServer class and the class represents a two-player game with mechanisms to track the current state, switch turns, and check for game-end conditions.

Variables:

- **currentPlayer:** This holds the current player object, indicating whose turn it is.
- **p1 and p2**: These represent the two players in the game.
- **playerDictionary:** A HashMap that maps player usernames to the respective player objects.
- **gameId:** An identifier for the game.
- **currentGameState:** Represents the current state of the game, including the board status, chat messages, and other game-specific details.

Methods

- **constrctor(Player p1, Player p2, int gameId)**: Initializes a new game with two players and a game ID. It sets the starting player, initializes the game state, and maps the players to their usernames in the dictionary.
- **getCurrentPlayer():** Returns the current player.
- **getPlayerByName(String username):** Retrieves a player based on a provided username.
- **getThePlayerNotCurrent():** Returns the player who is not currently taking their turn.
- **getplayerDictionary():** Returns the dictionary mapping usernames to player objects.
- **setCurrentPlayer(Player temp):** Sets the current player.
- **getGameId():** Retrieves the game ID.
- **getGameState():** Returns the current game state.
- **setGameState(int x_coordinate, int y_coordinate):** Modifies the game board at the given coordinates based on the current player's symbol. Returns a boolean indicating the success or failure of the operation.
- **getP1() and getP2():** Return the respective player objects.
- **switchPlayer():** Toggles the current player between p1 and p2.
- **checkWinning():** Checks if the current player has a winning combination on the board, either via row, column, or diagonal. If there's a win, it updates the winning situation in the current game state.

- **checkDraw():** Checks if the game has ended in a draw. If it's a draw, it updates the game state accordingly.
- **setChatMessage(String content):** Allows adding chat messages to the current game state.

4. GameState

The class represents the current state of a Tic-Tac-Toe game. It is allocated by the Game and bound with specific Game object, this class is designed for storing all information in the Game including game states, chat information, winning situation and player information.

Variables:

- **gameBoard**: A 2D character array that represents the Tic-Tac-Toe board. Each entry in this array can be either 'X', 'O', or '\0' (indicating an empty cell).
- **chatHistory:** An ArrayList storing the chat messages related to the game.
- **winningSituation**: A string variable that describes the state of the game, whether it's still ongoing ("gaming"), won by a player, or ended in a draw.

Methods

- **constructor():** Initializes the game board as a 3x3 empty grid and sets up the chat history ArrayList.

Board Management:

- **modifyBoard(int row, int col, char value)**: Tries to place a character (value, which can be 'X' or 'O') on the game board at the specified row and column. It will only modify the board if the targeted cell is empty ('\0'). If the cell is modified, it returns true; otherwise, it returns false.
- **getGameBoard()**: Returns the current game board.

Chat History Management:

- **addChat(String content)**: Adds a new chat message to the chat history. It ensures that the chat history does not exceed 10 messages. If there are already 10 messages, it removes the oldest message before adding a new one.
- **getChatHistory()**: Returns the chat history of the game.

Game Status Management:

- **setWinningSituation(String situation)**: Updates the winning situation of the game.
- **getWinningSituation()**: Returns the current winning situation.

5. Player

The player class represents an individual player participating in the Tic-Tac-Toe game which contains player's name, connected client that will be communicated to this player, current points of the player, the ranking of player and current using symbols which is randomly chosen by server. Each Player object also has a corresponding MatchStatus object which indicating the Player is currently waiting for game or playing game.

Variables:

- **playerName:** The name of the player.
- **symbol:** A character ('X' or 'O') that represents the player's symbol in the Tic-Tac-Toe game.
- **connectedClient:** An interface of the client that is connected to this player. This interface allows the server to make remote method calls on the player's client.
- **point:** The number of points the player has accumulated. Points increase with wins and decrease with losses.
- **ranking:** The player's rank.
- **matchingStatus:** An instance of the MatchStatus class, representing the current status of the player in the game.

Methods

- **Constructor** of Player(): Initializes a player with a given name and associated client interface.

Setters and Getters:

- Methods to get and set the playerName, symbol, connectedClient, point, and ranking attributes.
- There are specific methods to get and set the player's status (geStatus() and setStatus()).

Game Outcome Handling:

- **winAction():** Increases the player's points by 5 and announces to the client that they've won.
- **loseAction(String winnerName, String winnerSymbol):** Decreases the player's points by 5 and announces to the client that they've lost, providing information about the winner.
- **enemyQuitAction():** Increases the player's points by 5 and announces to the client that their opponent has quit the game, declaring them the winner.
- **drawAction():** Increases the player's points by 2 and announces to the client that the game has ended in a draw.

Utility Methods:

- **playerFormat():** Returns a string that provides a concise representation of the player's rank, name, and symbol.
- **simpleAnnouncement(String announce):** Makes a simple announcement to the connected client using RMI.

6. MatchStatus

The class which binding with a Player object, it shows the matching status which could be waiting for game or playing game

Variables:

- status: the status that could be playing or waiting
- username: the corresponding username of the player

Methods

- **Setter() and getter()** of status
- **Getter()** of username

7. Client

The Client class is responsible for handling the client-side logic and UI of a presumably multiplayer game. It communicates with a remote server to exchange game states, chat messages, and other information, using Java's RMI (Remote Method Invocation) technology. The Client class that will implements ClientInterface and the instance variables contains the Client GUI and server that it will be connected to.

Variables:

- **myTurn**: Indicates whether it's the client's turn.

- **gui**: Handles the client-side graphical user interface.
- **username**: Stores the username of the client.
- **ServerInterface server**: Used for communication with the server.
- **playingGameId**: Stores the ID of the game session that the client is playing.

Important Methods

- **Constructor(String username)**: Initializes the client, the UI, and registers the client on the server.
- **boardClick(int I, int j):** receive the board click from the GUI and send a movement to the server
- **getChatMessage():** return the messages that the other client sends.
- **setmyTurn():** the function to set myTurn to true or false
- **startOrWait(String decision)**: Handles starting the game or waiting based on the server's response.
- **ServerInterface getGameServer()**: Returns the server interface for communication.
- **quitWaitingQueue()**: quit the waiting queue, exit from the client and the corresponding GUI.
- **quiGame()**: Attempts to quit the game, communicating the decision to the server.
- **sendMessage(String text)**: Sends a chat message to the server.

- **getChatHistory(ArrayList<String> chat)**: Updates the client’s chat UI with the chat history.
- **getChatMessage()**: return the messages that the other client sends.
- **myTurn()**: the function to indicate whether it’s currently this client turn to move
- **getGameId()**: the function to return the game id of the current client playing with
- **setGameId(int id)**: the function to set the game id
- **getGameBoard():** returns the current playing game board
- **getAvailableMoves(char[][] board):** this function will return the List of points indicates what coordinate can be choose
- **announceTurn(String string):** send the current turn message from server to the GUI and make changes on GUI
- **announceWinning(String string):** send the current game winning message from server to the GUI and make changes on GUI
- **askForReRegisterPlayer():** ask the server to register this player again using the same username
- **updateBoard(GameState currGameState):** update the game board in GUI using the input argument.

Main Method

- **main(String[] args)**:
  - Expects command-line arguments for username, server IP, and server port.
  - Looks up the remote server object from the RMI registry and initializes the client.
  - Handles possible exceptions (like if the server isn't ready).

8. ClientGUI

The Graphic User Interface that will serve for the client. It defines the graphical user interface (GUI) for a Tic-Tac-Toe game. The file uses Java's Swing library for the GUI components.

Variables:

- **frame:** Represents the main window of the game.
- **timerLabel:** Displays the remaining time, as in "Time Left: XX".
- **chatInput:** A text field where the user types in chat messages.
- **statusLabel:** Indicates the current game status.
- **moveTimer:** A timer that counts down for a player's move.
- **time:** The default countdown value, set to 20 seconds.
- **chatArea:** A text area where chat messages are displayed.

- **timeLeft:** A counter that represents the remaining time.
- **boardButtons:** A 3x3 matrix of buttons, representing the Tic-Tac-Toe grid.
- **username:** The username of the player.
- **correspondClient:** A Client object that represents the current user.

Methods

Window Control and Components Management:

- **constructor(String username, Client client)**: Initializes the GUI components, sets the properties of various components, adds listeners to handle button clicks and other events, and makes the main frame visible.
- **getStatusLabel()**: Returns the status label.
- **getFrame()**: Returns the main frame.
- **startTime()**: Starts the timer with the default countdown value.
- **resetTime()**: Stops the timer and resets it.
- **modifyWinInformation(String string)**: Modifies the displayed winning information.
- **updateBoard(char[][] board)**: Updates the displayed game board.
- **refresh()**: Resets the board, chat area, and timer to their initial states.
- **geButtons()**: Returns the board buttons.
- **getUsername():** Returns the username.
- **getClient()**: Returns the correspondClient object.

- **modifTurnyInformation(String string):** Modifies turn-based information on the GUI.
- **setChatHistory(String finalChat)**: Updates the chat area with the provided chat history.
- **banButton()**: Disables all the board buttons.
- **restartButton()**: Re-enables all the board buttons.

Game Flow Logic:

- **handleOutCome(String message):** Handles the outcome of the game (win, lose, or draw) and prompts the user if they want to play again.
- **promptForReplay(String message)**: Displays a dialog asking if the user wants to play again.
- **tryToPlayAgain():** Resets the game to play again.
- **quitGame()**: Checks the game's state and asks the user if they want to quit. If they confirm, it quits the game and closes the application.
- **userWantsToQuit()**: Displays a dialog asking if the user wants to quit the game.
- **quitCurrentGame()**: Calls the quiGame() method of the client to end the game.
- **closeClientApplication()**: Exits the application.
- **handleError(RemoteException e)**: Handles exceptions, prints stack traces, and exits with an error code.
- **exitGame()**: Simply exits the game.

9. ClientInterface

This is an interface that will be implemented by the Client that defines the remote methods of client.

Methods

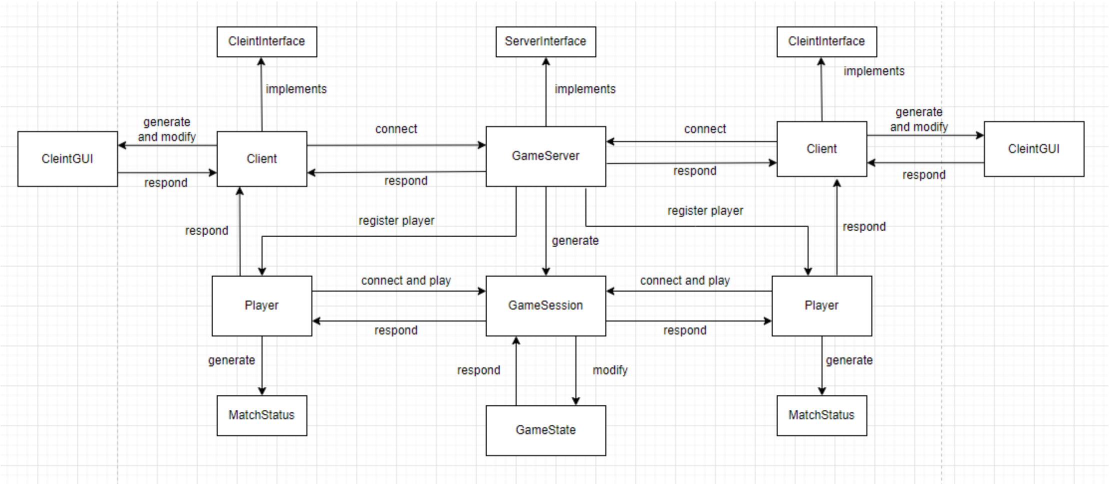
1. **updateBoard(GameState currGameState)**:
    - Updates the client's view of the game board to reflect the current game state.
    - Parameters:
        - **currGameState**: An object representing the current state of the game.
2. **setmyTurn(boolean temp)**:
    - Sets whether it is currently this client's turn to move.
    - Parameters:
        - **temp**: A boolean indicating if it's the client's turn or not.
3. **myTurn()**:
    - Checks if it's currently this client's turn to move.
    - Returns:
        - A boolean indicating if it's the client's turn or not.
4. **announceTurn(String string)**:
    - Informs the client whose turn it is.
    - Parameters:

        - **string**: A message indicating whose turn it is.

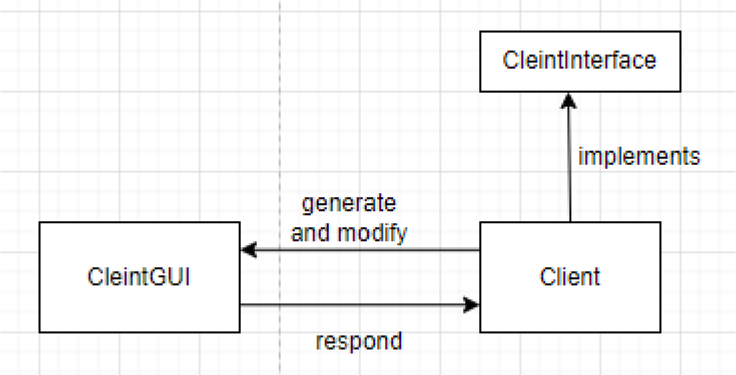
5. **announceWinning(String string)**:
    - Informs the client about the outcome of the game (e.g., win, lose, draw).
    - Parameters:
        - **string**: A message indicating the outcome.
6. **getChatHistory(ArrayList<String> chat)**:
    - Updates the client's chat history.
    - Parameters:
        - **chat**: An array list containing chat messages.
7. **setGameId(int tempGameId)**:
    - Assigns an ID to the game this client is currently part of.
    - Parameters:
        - **tempGameId**: The game's unique identifier.
8. **startOrWait(String decision)**:
    - Informs the client whether they should start the game or wait (probably in a queue).
    - Parameters:
        - **decision**: A string indicating the client's status (either start the game or wait).

Overall design and Interaction Diagram

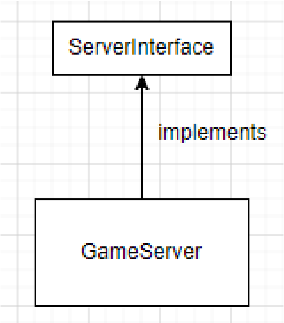
Overall Design Diagram:



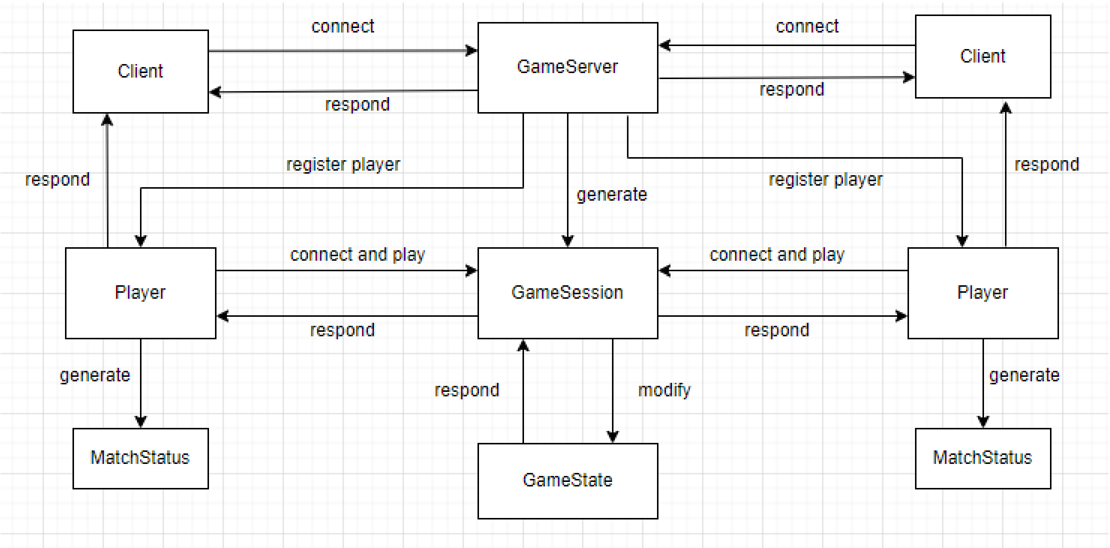
Client part:



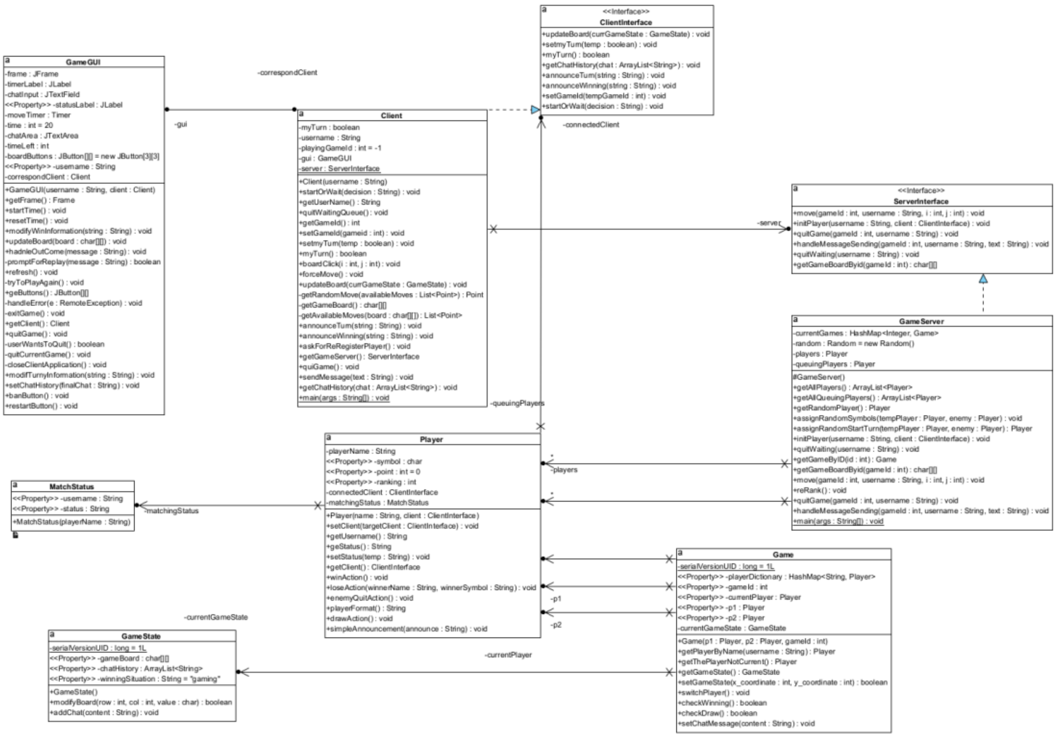
Server part：



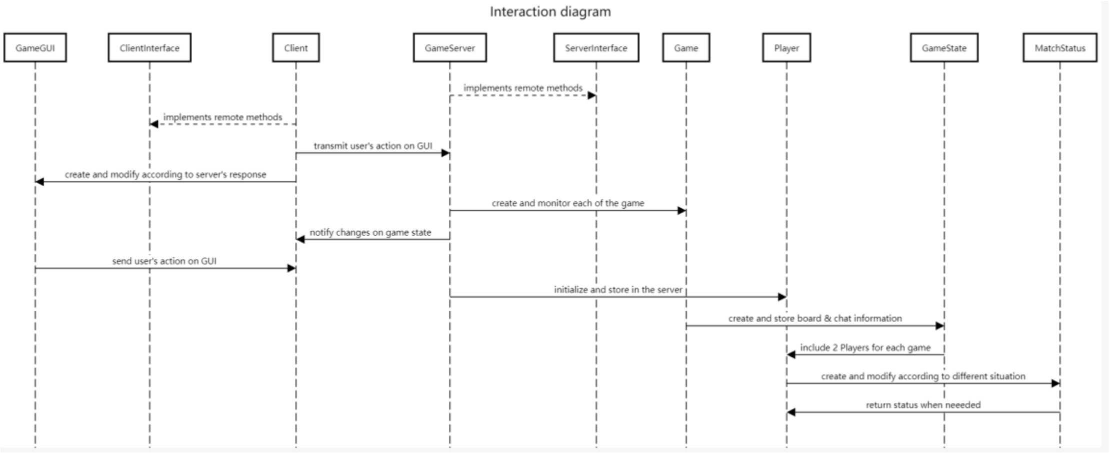
Player and Game Session Part:



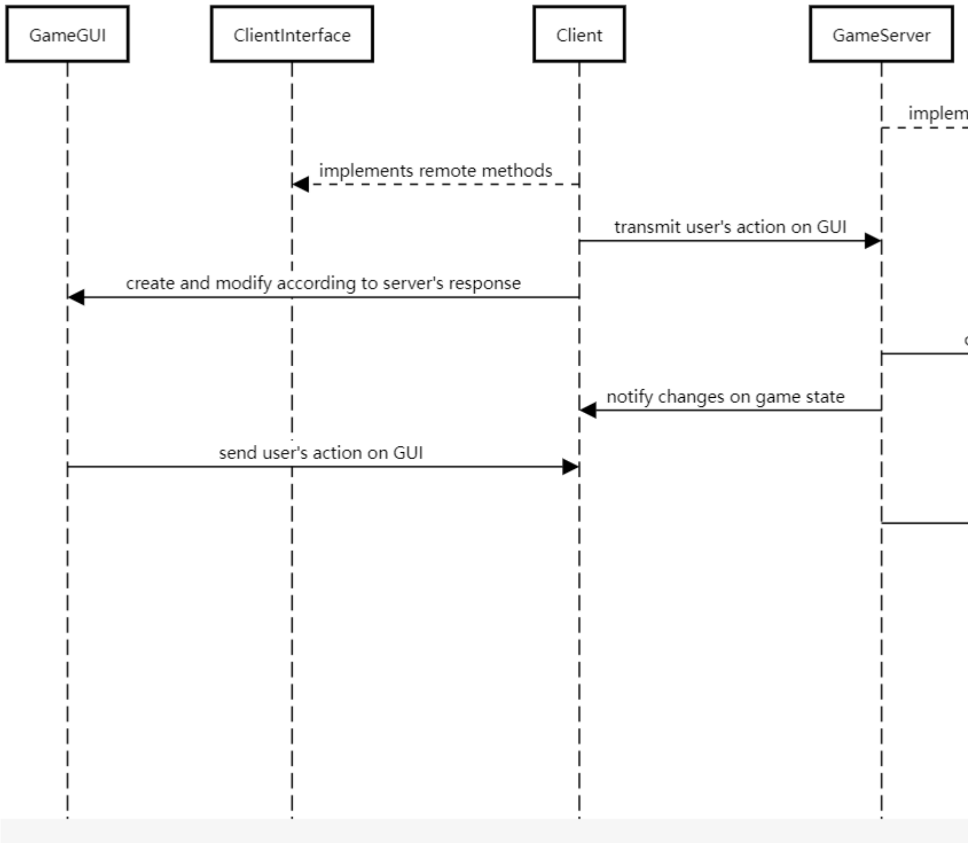
Overall Design by UML diagram:


<<Interface>>
ClientInterface
+updateBoard(currGameState : GameState) : void
+setmyTurn(temp : boolean) : void
+myTurn() : boolean
+getChatHistory(chat : ArrayList<String>) : void
+announceTurn(string : String) : void
+announceWinning(string : String) : void
+setGameId(tempGameId : int) : void
+startOrWait(decision : String) : void
GameGUI
-frame : JFrame
-timerLabel : JLabel
-chatInput : JTextField
<<Property>> -statusLabel : JLabel
-moveTimer : Timer
-time : int = 20
-chatArea : JTextArea
-timeLeft : int
-boardButtons : JButton[][] = new JButton[3][3]
<<Property>> -username : String
-correspondClient : Client
+GameGUI(username : String, client : Client)
+getFrame() : Frame
+startTime() : void
+resetTime() : void
+modifyWinInformation(string : String) : void
+updateBoard(board : char[][]) : void
+hadnleOutCome(message : String) : void
-promptForReplay(message : String) : boolean
+refresh() : void
-tryToPlayAgain() : void
+geButtons() : JButton[][]
-handleError(e : RemoteException) : void
-exitGame() : void
+getClient() : Client
+quitGame() : void
-userWantsToQuit() : boolean
-quitCurrentGame() : void
-closeClientApplication() : void
+modifTurnyInformation(string : String) : void
+setChatHistory(finalChat : String) : void
+banButton() : void
+restartButton() : void
-correspondClient
-gui
Client
-myTurn : boolean
-username : String
-playingGameId : int = -1
-gui : GameGUI
-server : ServerInterface
+Client(username : String)
+startOrWait(decision : String) : void
+getUserName() : String
+quitWaitingQueue() : void
+getGameId() : int
+setGameId(gameid : int) : void
+setmyTurn(temp : boolean) : void
+myTurn() : boolean
+boardClick(i : int, j : int) : void
+forceMove() : void
+updateBoard(currGameState : GameState) : void
-getRandomMove(availableMoves : List<Point>) : Point
-getGameBoard() : char[][]
-getAvailableMoves(board : char[][]) : List<Point>
+announceTurn(string : String) : void
+announceWinning(string : String) : void
+askForReRegisterPlayer() : void
+getGameServer() : ServerInterface
+quiGame() : void
+sendMessage(text : String) : void
+getChatHistory(chat : ArrayList<String>) : void
+main(args : String[]) : void
-connectedClient
-server
<<Interface>>
ServerInterface
+move(gameId : int, username : String, i : int, j : int) : void
+initPlayer(username : String, client : ClientInterface) : void
+quitGame(gameId : int, username : String) : void
+handleMessageSending(gameId : int, username : String, text : String) : void
+quitWaiting(username : String) : void
+getGameBoardByid(gameId : int) : char[][]
GameServer
-currentGames : HashMap<Integer, Game>
-random : Random = new Random()
-players : Player
-queuingPlayers : Player
#GameServer()
+getAllPlayers() : ArrayList<Player>
+getAllQueuingPlayers() : ArrayList<Player>
+getRandomPlayer() : Player
+assignRandomSymbols(tempPlayer : Player, enemy : Player) : void
+assignRandomStartTurn(tempPlayer : Player, enemy : Player) : Player
+initPlayer(username : String, client : ClientInterface) : void
+quitWaiting(username : String) : void
+getGameByID(id : int) : Game
+getGameBoardByid(gameId : int) : char[][]
+move(gameId : int, username : String, i : int, j : int) : void
+reRank() : void
+quitGame(gameId : int, username : String) : void
+handleMessageSending(gameId : int, username : String, text : String) : void
+main(args : String[]) : void
-queuingPlayers
-players
Player
-playerName : String
<<Property>> -symbol : char
<<Property>> -point : int = 0
<<Property>> -ranking : int
-connectedClient : ClientInterface
-matchingStatus : MatchStatus
+Player(name : String, client : ClientInterface)
+setClient(targetClient : ClientInterface) : void
+getUsername() : String
+geStatus() : String
+setStatus(temp : String) : void
+getClient() : ClientInterface
+winAction() : void
+loseAction(winnerName : String, winnerSymbol : String) : void
+enemyQuitAction() : void
+playerFormat() : String
+drawAction() : void
+simpleAnnouncement(announce : String) : void
MatchStatus
<<Property>> -username : String
<<Property>> -status : String
+MatchStatus(playerName : String)
-matchingStatus
-p1
-p2
-currentGameState
-currentPlayer
GameState
-serialVersionUID : long = 1L
<<Property>> -gameBoard : char[][]
<<Property>> -chatHistory : ArrayList<String>
<<Property>> -winningSituation : String = "gaming"
+GameState()
+modifyBoard(row : int, col : int, value : char) : boolean
+addChat(content : String) : void
Game
-serialVersionUID : long = 1L
<<Property>> -playerDictionary : HashMap<String, Player>
<<Property>> -gameId : int
<<Property>> -currentPlayer : Player
<<Property>> -p1 : Player
<<Property>> -p2 : Player
-currentGameState : GameState
+Game(p1 : Player, p2 : Player, gameId : int)
+getPlayerByName(username : String) : Player
+getThePlayerNotCurrent() : Player
+getGameState() : GameState
+setGameState(x_coordinate : int, y_coordinate : int) : boolean
+switchPlayer() : void
+checkWinning() : boolean
+checkDraw() : boolean
+setChatMessage(content : String) : void

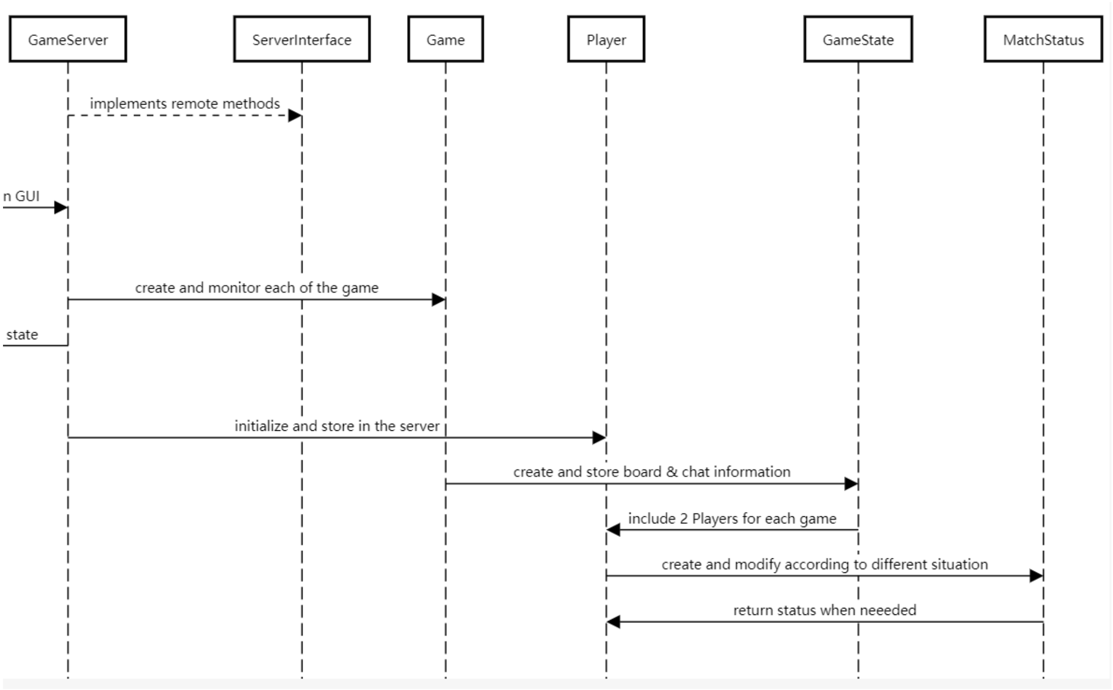
Overall Interaction diagram:


Interaction diagram
GameGUI
ClientInterface
Client
GameServer
ServerInterface
Game
Player
GameState
MatchStatus
implements remote methods
implements remote methods
transmit user's action on GUI
create and modify according to server's response
create and monitor each of the game
notify changes on game state
send user's action on GUI
initialize and store in the server
create and store board & chat information
include 2 Players for each game
create and modify according to different situation
return status when neeeded

Client to Server:


GameGUI
ClientInterface
Client
GameServer
implem
implements remote methods
transmit user's action on GUI
create and modify according to server's response
notify changes on game state
send user's action on GUI

Server side:


GameServer
ServerInterface
Game
Player
GameState
MatchStatus
implements remote methods
n GUI
create and monitor each of the game
state
initialize and store in the server
create and store board & chat information
include 2 Players for each game
create and modify according to different situation
return status when neeeded

Critical analysis of the work done

Strength

1. The project used Distributed Design: The use of Java RMI or sockets indicates a distributed system approach, which is flexible.

2. The GUI are implemented completely according to the assignment specification and enhance the interactive aspect of the game

3. Structured and Object-Oriented Design: The code are modular and object-oriented, which makes it easier to manage, modify and scale.

4. Error Handling: The client and server sides both account for potential errors, ensuring a smooth user experience.

Weakness:

1. Code Redundancy: Some of the methods and action, like error handling are repetitive. Using design patterns or further abstraction might make the code more elegant.

2. Considering the complexity of the assignment, fault tolerance was not implemented in this project. The fault tolerance I envisioned should be that the server continues to send connection requests to the client for 30 seconds after the client crashes. If the client accepts, the game will restart. if the client does not respond, the game ends with a draw and exits, but when actually implementing this function, I found that it is not realistic to make the server continue to send connection requests to the disconnected client, because the clients are connected through the same port, the client After reconnecting to the client, it will be regarded as a new user. I have no good way to solve this problem without changing the original code framework, so I did not implement this function. When a player quit, the server will let the opponent win the game immediately. Also, if one of the client quit, and another client tries to connect with the same username, the server will let the new client inherit the previous player's ranking and points.

Conclusion

In summary, this project implemented complete distributed systems, GUI design, and real-time interaction. It could be an even more robust and user-friendly application if the fault tolerance has been implemented and the code redundancy be minimized.