Multiplayer Guessing Game Platform using Java Sockets

Amr Abuzayyad (0214597), Saleh Alghobari (0214044), Marlo Rizkalla (2211499)

1. Introduction

This project involves the design and implementation of a TCP-based multiplayer guessing game using Java's socket programming API. The game platform allows a player (the challenger) to initiate a challenge by selecting a secret name from a category, while other players join to guess the name by asking yes/no questions. The game proceeds in structured turns and supports concurrent games, synchronized interactions, and dynamic client-server communication.

This report describes the system architecture, communication flow, synchronization mechanisms, and design choices made during implementation.

2. System Components

The application is structured in a classic client-server model:

- Server: Manages connections, game rooms, players, and challenge state. It handles message routing, player turn control, and game conclusion logic.
- Client: Interacts with the user via the terminal, sends inputs to the server, and responds to prompts received from the server.

Key Components:
- Server.java: The entry point for the server application.
- ClientHandler.java: Manages each client's communication and room interaction, creates a thread for each client and interacts with its respective client.
- Room.java: Maintains game state such as players, genre, and guesses.
- RoomManager.java: A singleton managing all rooms.
- Client.java: Client-side application for game interaction.
- Player.java and Challenger.java: Represent game participants after they take a role.

3. Game Flow

Startup Phase:
1. The player enters a nickname.
2. Chooses to host or join a game.

Hosting a Challenge:
- Select category and provide secret word.
- Choose limited or unlimited attempts.
- Challenge becomes open and visible.

Joining a Challenge:
- Select from active rooms.
- When two players are in, game begins.

- Players alternate asking yes/no questions.
- Challenger responds with Yes/No/I don't know.
- After the answer, the player can choose to guess.
- The game ends with a correct guess or if a player's attempts run out (in limited mode).

4. Methodology

The way this designed the system is as follows:

1- The server catches any connection and creates a socket to connect with the client that is trying to connect.

2- A client handler is created from the received socket and a thread is ran to manage that client (ClientHandler.java implements runnable, which means it has a run() function that specify what this threads does)

3- An initialization sequence begins in the run() method of the thread where it interacts with the client to get the name and the future info it needs, using the TCP socket used to construct the class and buffered reader and writers.

4- The client is presented with a choice to make it choose between if it wants to host a room (in which case it becomes a challenger) or join an existing room(becomes a player)
4.1- if the client chooses to host a room, the client handler creates a challenger class and passes itself as an attribute, and start the initial sequence to get the necessary info to create a room (category, secret, number of attempts) and then creates one and the rest of the code is within the ChallengerLogic function.
4.2- if the client chooses to join a room, it is presented with a list of existing rooms to choose from, if the player chooses to join a valid room, a player class is created and the client handler passes itself as an attribute , and it is also added to the room class the challenger created, when two players join the room the game starts.

5- The game starts by default with player one (the player who joined the room first) and then it is turn based from there, there are two main mechanisms that synchronizes and manages turns:
5.1- turn management on the server side: each of the three client handles in the room have an "isTurn" flag, and all the client handlers have access to each other's flags, the game starts with player one's flag enabled and the rest disabled, and each time a player is done with their logic they disable their own flag and enable the next player's flag in the following order: p1 -> ch -> p1 -> ch -> p2 -> ch -> p2 -> ch-> repeat.
5.2- turn management on the client side: the clients start with an initialization sequence (they get their name and role etc.) then enter the main game loop, where

they are stuck in a reading state until they receive a certain flag from the server that triggers their turn, when their turn is triggered, they continue the game logic until their turn ends then they go back to the reading loop.

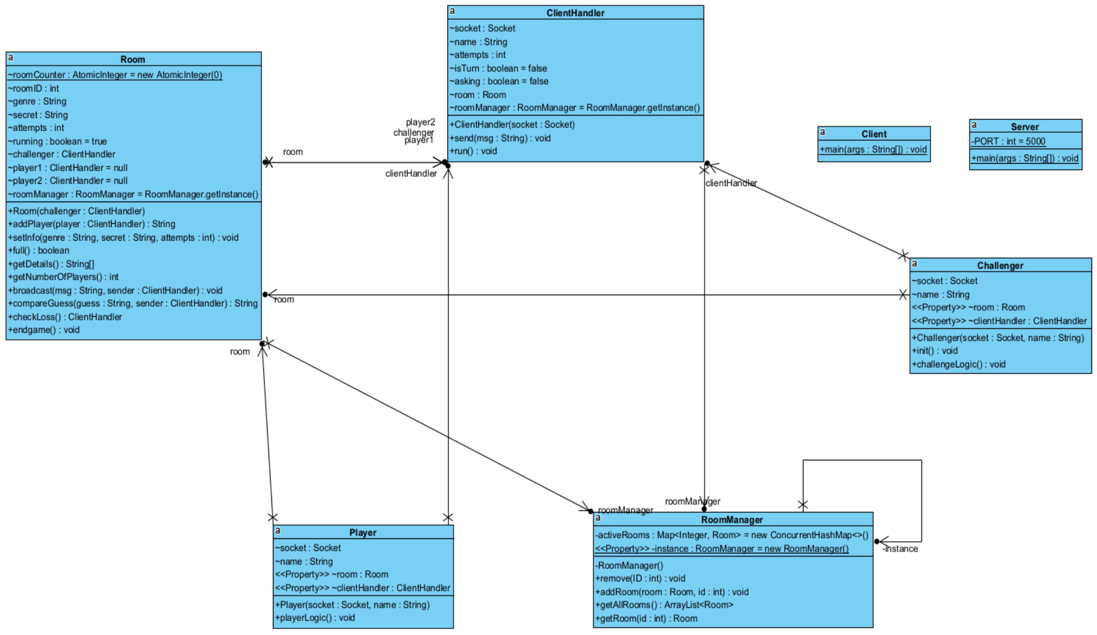
6- The game continues on until a player exhausts all their attempts or until a player wins. When either of these conditions occur the main loop of the challenger and the players exits using a flag then the run() functions in the client handles ends closing the thread and respective sockets, and the end flag is sent to the client to prompt the exit of the main loop.

7- The class room managers is a single instance that contains all the current room so they can be displayed for the clients to choose a room to enter, the room class has a lot of helper functions that help with the game design like broadcast(). It is also worth noting that most classes have access to most other classes so that communication is made easier, however a lot functions inside the room class are synchronized to ensure another layer of safety (beyond the flags) so that clients in the same room do not create a race condition.

5. Sample Client Interaction

```
Welcome to the guessing game!
Enter your name: Alice
Do you want to:
1. Host a room
2. Join a room
> 1
Choose a genre:
1. Movies ...
> 1
Enter a word for the players to guess:
> Inception
Do you want to specify attempts? (y/n)
> y
Enter number of attempts:
> 5
Please wait until all the players join...
```

6. UML Class Diagram


a
ClientHandler
~socket : Socket
~name : String
~attempts : int
~isTurn : boolean = false
~asking : boolean = false
~room : Room
~roomManager : RoomManager = RoomManager.getInstance()
+ClientHandler(socket : Socket)
+send(msg : String) : void
+run() : void
a
Room
~roomCounter : AtomicInteger = new AtomicInteger(0)
~roomID : int
~genre : String
~secret : String
~attempts : int
~running : boolean = true
~challenger : ClientHandler
~player1 : ClientHandler = null
~player2 : ClientHandler = null
~roomManager : RoomManager = RoomManager.getInstance()
+Room(challenger : ClientHandler)
+addPlayer(player : ClientHandler) : String
+setInfo(genre : String, secret : String, attempts : int) : void
+full() : boolean
+getDetails() : String[]
+getNumberOfPlayers() : int
+broadcast(msg : String, sender : ClientHandler) : void
+compareGuess(guess : String, sender : ClientHandler) : String
+checkLoss() : ClientHandler
+endgame() : void
player2
challenger
player1
room
clientHandler
a
Client
+main(args : String[]) : void
a
Server
-PORT : int = 5000
+main(args : String[]) : void
clientHandler
a
Challenger
~socket : Socket
~name : String
<<Property>> ~room : Room
<<Property>> ~clientHandler : ClientHandler
+Challenger(socket : Socket, name : String)
+init() : void
+challengeLogic() : void
room
room
roomManager
roomManager
a
RoomManager
-activeRooms : Map<Integer, Room> = new ConcurrentHashMap<>()
<<Property>> -instance : RoomManager = new RoomManager()
-RoomManager()
+remove(ID : int) : void
+addRoom(room : Room, id : int) : void
+getAllRooms() : ArrayList<Room>
+getRoom(id : int) : Room
-instance
a
Player
~socket : Socket
~name : String
<<Property>> ~room : Room
<<Property>> ~clientHandler : ClientHandler
+Player(socket : Socket, name : String)
+playerLogic() : void

7. Conclusion

The project successfully demonstrates a functional and concurrent multiplayer game using Java's TCP networking. It ensures reliable communication, proper synchronization, and a smooth user experience.