Final Report Template
Distributed Battleship

Final Report for the Distributed Systems Course 2025/2026

Viola Leonardo
leonardo.viola2@studio.unibo.it

June 17, 2026

The project consists of a simple, distributed battleship game where two users place their ships on a grid, and the first to sink the opponent's fleet wins.

A distributed architecture is employed, utilizing a peer-to-peer model for communication between the two players during the match, and a client-server model to enable both user matchmaking and the registration of backup servers with the primary server. This registration allows backup servers to receive constant state updates from the primary server, ensuring they are ready to take its place if it becomes unreachable.

Disclaimer (if needed)

During the preparation of this work, the author(s) used LLMs to refine syntax and correct linguistic errors.

After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the final report/artifact.

1 Concept

This project primarily consists of a desktop application equipped with a graphical user interface (GUI).

- **Primary Users:** The main target audience for this application are casual gamers looking to enjoy a match together.
- **Use Cases:** Before playing, a user chooses a nickname and then connects to a random match selected by the system, which has already been started by another waiting player. If no active matches are available, a new one is initiated and will remain in a waiting state until another player joins. For the sake of simplicity, no authentication or authorization mechanisms are provided.

- **Interactions:** Users interact through a GUI, placing their ships on the grid and selecting the cells of the opponent's grid to target via mouse clicks. Both during ship placement and subsequently for selecting the cell to hit, a timer indicates the time limit to make a choice. Specifically, if the timer expires during the placement phase before the match begins, any ships not yet placed will be automatically positioned at random by the system. Similarly, during the match, each turn has a time limit, and when the time expires, a random cell on the opponent's grid will be automatically selected to be targeted.
- **Data:** The system temporarily maintains data related to the game—essentially the state of both players' grids—needed to ensure the correct functioning of the system itself. No data persistence is required between different matches.
- **Geographical Distribution:** The system is designed to operate within a local area network (LAN), such as a home Wi-Fi environment. The system does not support NAT traversal, and multiplayer functionality is restricted solely to local networks.

2 Requirements Elicitation and Analysis

This section provides the functional, non-functional, and implementation requirements for the analyzed system.

2.1 Functional Requirements

1. Core Gameplay
   a) Players must be able to join an available random match (room), meaning a room where exactly one player is already waiting for an opponent. If no rooms are available, the system shall automatically create a new one, and the player will wait for an opponent.
   b) When the second player joins, the room becomes full and the match starts automatically.
   c) The match begins with a ship placement phase where players position their fleet on the grid within a time limit. If a player fails to meet the time limit, the remaining ships will be automatically and randomly placed by the system.
   d) Ships must be placeable anywhere on the grid, either horizontally or vertically, while respecting grid boundaries and preventing ships from being positioned in adjacent cells.
   e) During their turn, a player must be able to interact with the opponent's grid displayed on the interface, selecting the target cell via a mouse click.
   f) The selection of the opponent's cell to target must be made within a time limit; otherwise, the system shall automatically choose a random position to target.

g) During a single turn, a player can fire multiple shots, continuing their turn as long as they do not miss—meaning until their shot fails to hit any part of the opponent's ships and lands in the water.

h) The game shall alternate turns between players automatically.

i) It must be possible to abandon the match at any point during the game.

2. Game Management

a) The system shall generate and assign unique room IDs for each game session.

b) Players may surrender by clicking a dedicated button, immediately terminating the match for both participants.

c) Game rooms shall be removed from memory upon game completion or player disconnection.

d) Users must be able to connect to a server to request entry into an available random room. The server tracks all active rooms—both those still available and those that are full—and upon receiving a join request, it finds an available match and sends the requesting client the necessary data to connect to the waiting opponent.

e) It must be possible to have multiple backup servers; all these backup servers shall remain connected to the primary server, ready to take its place as soon as it becomes unavailable, thereby ensuring system availability for clients who wish to connect and play.

f) The primary server must transmit the updated system state (active clients, active rooms) to all backup servers at predefined time intervals. This state sharing serves both as a heartbeat to indicate the primary's availability to the backup servers and to ensure that the backup server that eventually replaces the primary possesses a state that is as up-to-date as possible.

g) Upon connection to the primary server, each backup server is assigned a failover order (by the primary server), thereby eliminating the need to manage elections among the various potential backup servers when replacing the primary server in the event of its disconnection.

h) All clients must be aware of the existing backup servers and their failover order at all times, so that if they detect a disconnection from the primary server, they know which backup server to connect to after attempting to reconnect to the primary.

2.2 Non-Functional Requirements

1. Availability: The system must provide for the use of backup servers, which guarantee availability by replacing the primary server whenever it becomes unavailable.

2. Usability: The application must be easy to use.

3. State Consistency: A synchronized state must be maintained by the two nodes currently in a match.

4. Robust Error Handling: Invalid or malformed messages must be caught by try/catch blocks; the server replies with an error string and ignores the invalid request.

5. Timeout-Aware User Input: Enforce time-limited user input to keep gameplay responsive and prevent idle blocking.

6. Heartbeat-Based Disconnection Detection: Use heartbeat messages to monitor connectivity and detect disconnections automatically.

7. Configuration Flexibility: Allow easy adjustment of board size, ship sizes, and buffer limits via constants.

8. Performance Logging: Server console must log key events (connections, shots, results, errors) for debugging.

2.3 Implementation Requirements

1. Architecture: The system adopts a hybrid architecture that combines peer-to-peer communication between players during a match with a client-server model for interactions between the peers and the lobby server. This approach ensures consistent gameplay while simultaneously reducing the overall complexity of the system.

2. Programming Language: The project is developed using Java.

3. Communication Protocol: All communication between clients and the server must occur via the TCP protocol. TCP was chosen because, since this is a turn-based game where each turn can last up to several tens of seconds, it was considered preferable to select a protocol that is slightly slower than UDP but guarantees greater robustness and no packet loss, thereby eliminating the need to handle potential message loss and simplifying the system.

4. Version Control: All code shall be version-controlled using Git. A repository is maintained throughout the development cycle to track progress, ensure reproducibility, and support collaboration.

2.4 Relevant Characteristics of the Distributed System

The following are the distributed system characteristics relevant to this project:

- **Transparency:** The system should hide network details from players: they simply join a match and play. However, full transparency is not required. Users may notice some effects of failures (e.g., reconnection messages or delays).

- **Fault Tolerance and Dependability:** Since the system is distributed, components can fail:
  - If the server stops, clients must detect it and try to reconnect.
  - If the reconnection to the primary server fails after a predefined number of reconnection attempts, the clients must try to connect to the backup server with the highest failover priority. If a connection cannot be established with it, they will try the next backup server, and so on.
- **Scalability:** The system must be capable of supporting multiple players connected to the same server. Although the architecture allows for a high number of concurrent users, the expected user base for this system is relatively small.
- **Security and Trust:** For the sake of simplicity, and also because this type of application does not strictly require it, the system does not feature any authentication or authorization mechanisms. Since the application does not handle sensitive data, this does not pose an issue.
- **Heterogeneity of Components:** Although both the GUI application and the lobby server are implemented using the same technology, replacing either of them with a different technology should not represent a problem, provided that the interaction mechanisms and communication protocols are preserved.
- **Evolvability and Maintainability:** The system is designed to be easily extensible, allowing the addition of new gameplay features without requiring significant modifications to the underlying platform.
- **Performance and Concurrency:** The system must support multiple simultaneous communications. Since this is a real-time game, high performance is fundamental to minimize latency and ensure a smooth gaming experience.
- **Economy and Costs:** The project is developed using existing computers already owned by the developer, without relying on external hosting platforms. Consequently, there is no associated cost.

3 Design

This chapter illustrates the design strategies adopted to satisfy the requirements identified during the analysis phase, placing particular emphasis on fault tolerance, session state consistency, and real-time interaction between users.

3.1 Architecture

The system adopts a hybrid architecture that combines Client-Server and Peer-to-Peer (P2P) communication models, while additionally integrating a redundancy mechanism to ensure the high reliability of the backend.

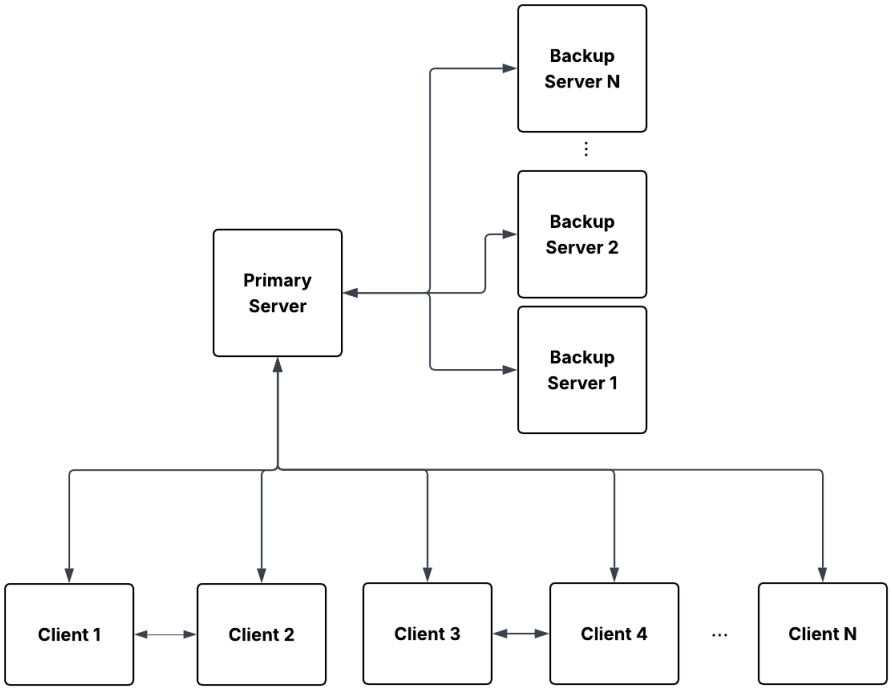

Figure 1: Hybrid system architecture with redundant backup servers.

As highlighted in the architectural diagram (Figure 1), the infrastructure is structured across three main logical components:

- **Client-Server for State Management:** The *Primary Server* acts as the central authority of the system. It is responsible for tracking and monitoring the global state, managing the list of connected users and active game rooms. Upon a new user's access, the server registers them and assigns them to a specific room by sending them the network details of their respective opponent.
- **Peer-to-Peer (P2P) for Gameplay:** Once the *Primary Server* has established the matchmaking and provided the contact information, the *Client* nodes establish a direct connection between each other. The data exchange relating to game actions takes place entirely in Peer-to-Peer mode. This approach drastically decreases the computational and network load on the central server, minimizing the latency perceived by the players.
- **Primary-Backup Architecture with State Pushing:** Backend fault tolerance is entrusted to a *Primary-Backup* architecture operating in *push* mode. The communication channel does not rely on silent nodes polling for data; instead, the *Primary Server* actively assumes the responsibility of distributing state snapshots to all registered backups at predefined time intervals. This unidirectional flow

serves a dual function: it ensures the seamless alignment of game sessions and acts as a deterministic *heartbeat* for the timely detection of primary node crashes.

This division of tasks optimizes overall efficiency. The central server acts exclusively as an initial coordinator and guardian of the global state, leaving the dynamic interaction to take place on the P2P channel. Consequently, the system is highly resilient: even in the presence of traffic spikes or temporary backend latency, the direct gaming experience between the two clients remains smooth and uninterrupted.

3.2 Infrastructure

Considering the project objectives and the hybrid nature of the architecture, the technological infrastructure was designed to be extremely lightweight, minimizing deployment complexity and hardware requirements.

3.2.1 Infrastructure Components

The system relies on three main components interacting at the software level:

- **Primary Server (1 active instance):** The central backend process that actively hosts the room management logic and maintains the state of game sessions in memory. It acts as the single entry point for new client connections, handles matchmaking, and coordinates the network topology.
- **Backup Servers (N passive instances):** Redundant nodes configured according to an *Active-Passive* architecture. In the event of a *Primary Server* crash, the system does not require complex distributed election techniques due to a **sequential failover order** strategy. Upon registration with the primary, each backup is assigned an incremental numerical identifier (1, 2, ..., N) that establishes its failover priority. Since the main server promptly notifies all participants (including clients) of the updated network topology, the moment the *heartbeat* transmissions stop, all nodes already know exactly which available backup holds the lowest order. This node takes over instantly, inheriting the last valid state and ensuring service continuity without disconnecting users.
- **Client Nodes (N instances):** The players' machines running the client application. Each client initially operates in a client-server model to interface with the backend during the matchmaking and room assignment phase, and subsequently establishes a direct channel to operate as an independent peer within the network, exchanging data related to actions and interactive gameplay dynamics directly with the opponent.

3.2.2 Network Deployment and Configuration

The components are deployed within a logical Local Area Network (LAN) and communicate using the standard network architecture based on TCP/IP sockets.

In order to bypass configuration challenges related to firewall traversal and Network Address Translation rules (*NAT traversal*), the development and execution of the system assume as a constraint that all nodes are mutually reachable within the same local network segment.

3.2.3 Discovery Mechanisms

Due to the dynamic nature of IP addresses within a LAN, the system implements the following node resolution and discovery protocols:

- **Server Discovery:** The *Primary Server* is launched by listening on a predefined and static network port. When a client application is initialized, the user must explicitly provide the central server's IP address. The client then initiates a direct TCP connection request to the fixed combination of the IP and the predefined port, stably anchoring itself to the authoritative node.
- **Peer Discovery:** The central server acts as a dynamic registry and intermediary to establish the direct connection between nodes. The mechanism develops sequentially during the game request phase:
  1. The first client intending to start a match contacts the *Primary Server* and explicitly communicates its intention to play, specifying the network port on which it will listen while waiting for an opponent.
  2. When a second client connects to the server to request a match, the server responds directly to the latter by sending it the IP address and listening port previously registered by the first waiting client.

  In this way, thanks to the information conveyed by the primary server, the second client knows the opponent's network details and is able to initialize the opening of the direct TCP communication channel for P2P gameplay.

3.3 Modelling

3.3.1 Domain Entities and Infrastructure Mapping

Below is a detailed explanation of the system's class structure, based on the UML entity diagrams. Each class encapsulates a specific responsibility within the global client-server game flow, helping to maintain the modularity and clarity of the overall architecture.

- **Server:** Represents the backend component of the system, responsible for managing the network infrastructure, matchmaking, and fault tolerance mechanisms.

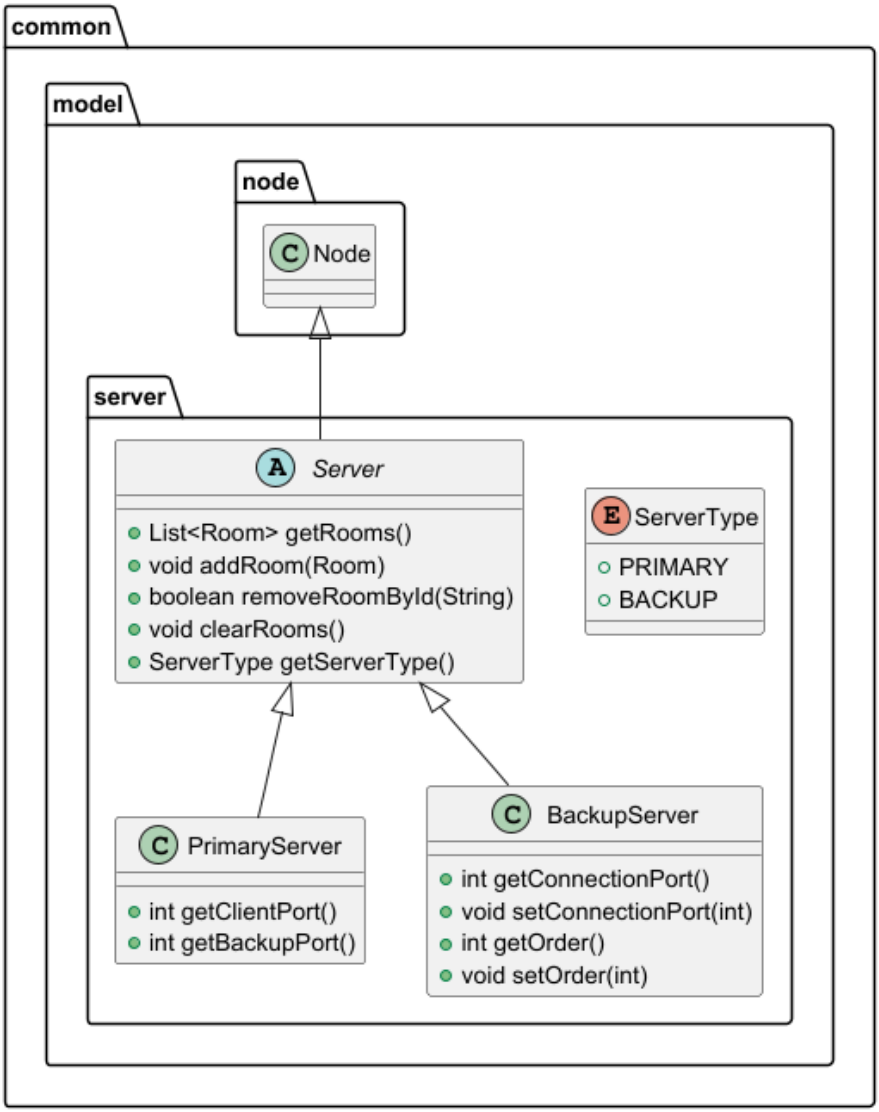

Figure 2: Class diagram relating to Server-side entities and nodes.

Within this module, the class hierarchy and responsibilities are distributed as follows:

- **Abstract Class *Server*:** Extends the generic `Node` entity and defines the common behavior and interface for any backend instance. It exposes the fundamental methods for managing the lifecycle of game rooms and active users. The identification of its specific role within the network is handled via the `ServerType` enumeration.
- **Class *PrimaryServer*:** Specializes the abstract *Server* class. It represents the active central node that handles the ports dedicated to direct communi-

cation, allowing it to accept matchmaking requests from players and accommodate connections from backup nodes.

- **Class *BackupServer*:** Represents the peer entity to the primary within the backend. It encapsulates the sequential failover mechanism logic by storing the fixed, numerical backup takeover order.

- **Client:** Represents the frontend component and the distributed peer-to-peer logic of the application installed on the player's machine, integrating both connectivity management and the local simulation of the game state.

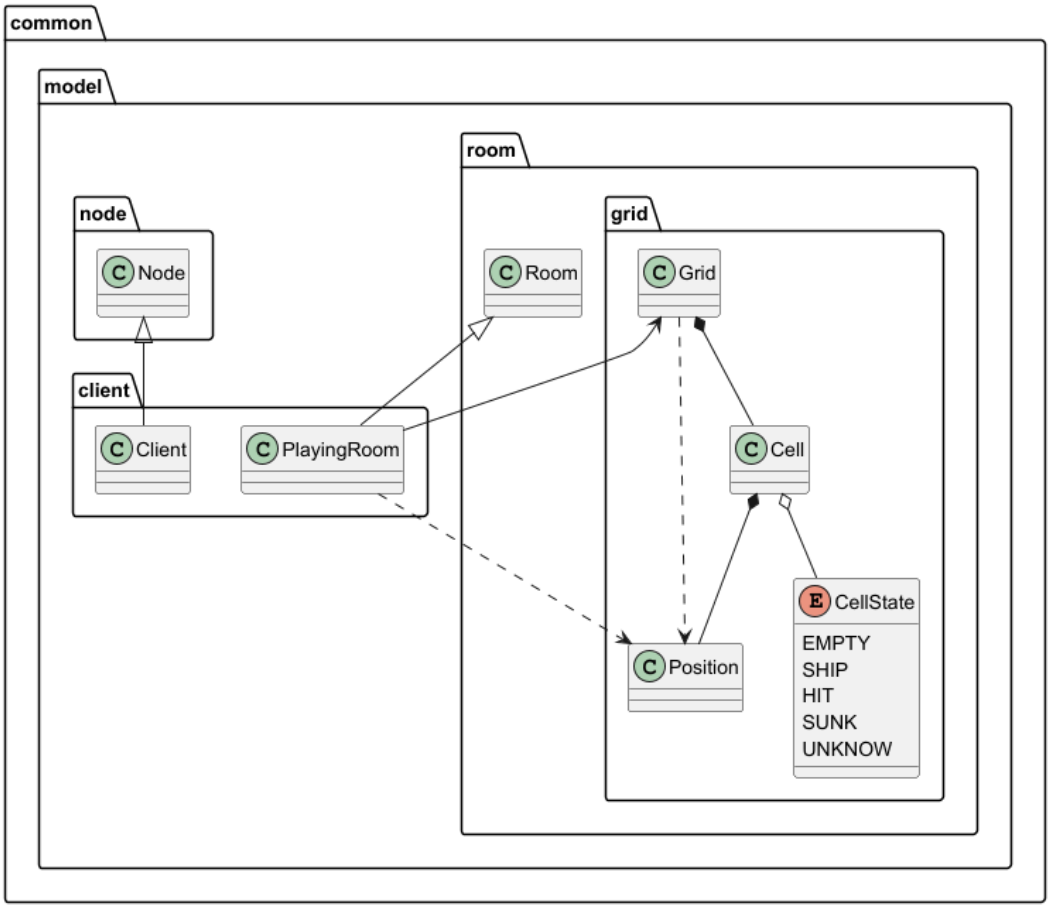

Figure 3: Simplified class diagram relating to Client-side entities.

Within this module, structural relationships and responsibilities are distributed as follows:

- **Class *Client*:** Extends the `Node` entity and models the local player instance. It maintains the references required to anchor both the backend server and the dynamic address of the opponent peer for the direct session.

- **Class *Room*:** Located in the common package, it is used to track the identity of the two paired players and the initial activation state of the room at a base level.
- **Class *PlayingRoom*:** Directly extends the `Room` class. It resides exclusively on the client and encapsulates the entire interactive peer-to-peer gameplay logic. It coordinates the progression of turns and contains direct references to the game grids.
- **Grid Model (*Grid*, *Cell*, *Position*, *CellState*):** The `PlayingRoom` class controls the match by acting on the two-dimensional map defined by `Grid`, which aggregates a matrix of `Cell` objects. Each cell is positioned in space via `Position` coordinates and assumes one of the logical states provided by the `CellState` enumeration (*EMPTY, SHIP, HIT, SUNK, UNKNOWN*).

- **Controller:** Represents the logical component responsible for application orchestration, lifecycle management of individual nodes, and interfacing with control flows. They never manipulate network sockets directly, but instead delegate all communication management to the underlying services.

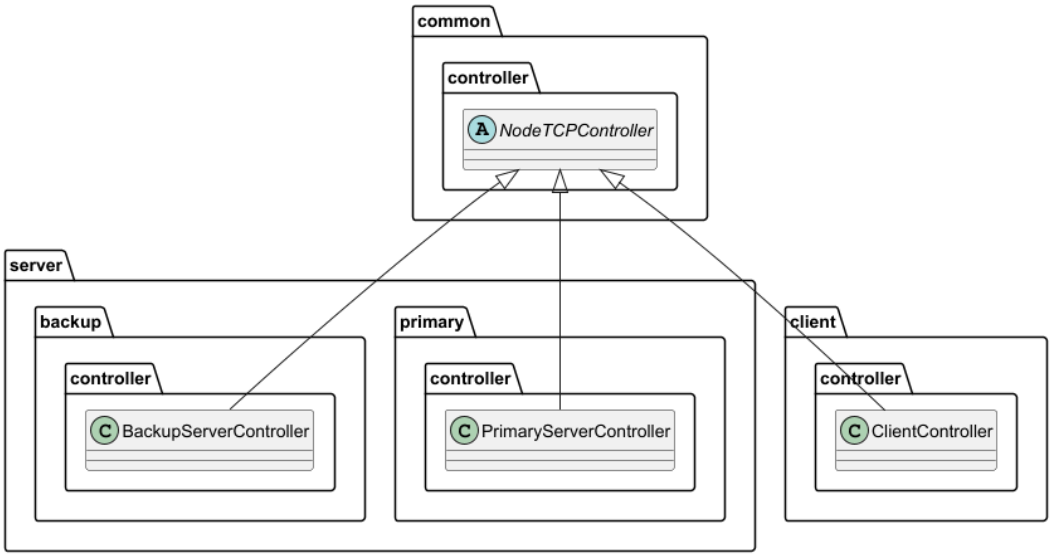

Figure 4: Diagram of the control components for the Client, Primary Server, and Backup Server.

The control logic is distributed across three main actors, each specialized for a specific role within the architecture:

- ***ClientController*:** Coordinates the behavior of the user-side application. It manages menu navigation, game initialization, and routes responses both to the primary server (matchmaking phase) and to the opponent peer (gameplay phase).

- ***PrimaryServerController*:** Oversees the lifecycle of the active backend. It accepts connection requests from new clients and external backup servers, places players into logical rooms, and coordinates the local network topology.
- ***BackupServerController*:** Monitors the state of the resilient backend. It manages the initial handshake connection to the primary node, passively receives data synchronization streams, and triggers the automatic promotion procedure to *Primary* in the event of a crash.

- **Connectivity Services:** Define the infrastructural network components dedicated to opening, maintaining, and structuring TCP communication channels. The introduction of this intermediate layer completely abstracts the network logic and the bidirectional interaction between specific nodes, relieving the respective *Controllers* of the task of manually managing sockets and the state of I/O channels. Each specific service is specialized in communicating with a single type of counterparty (e.g., client-server or peer-to-peer), encapsulating the rules of engagement and the exchange of byte streams between the two entities.

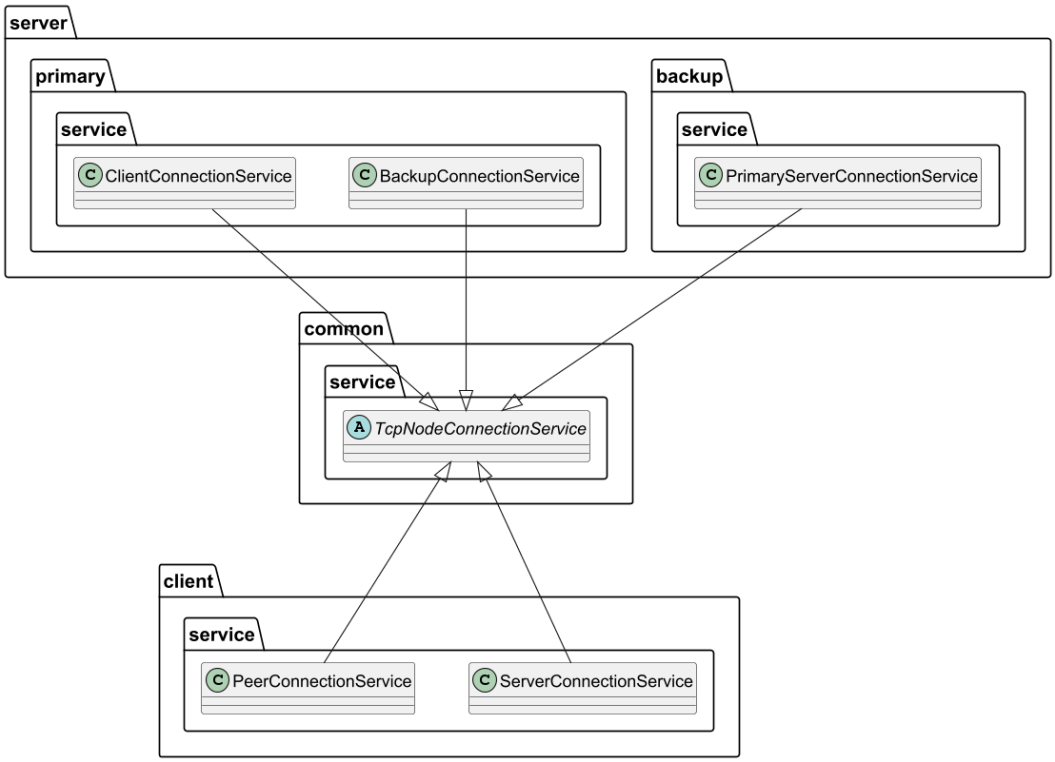

Figure 5: Class diagram of the Service layer focused on TCP connectivity hierarchies.

- **Message Handler Service:** Represents the layer responsible for handling in-

coming messages. To ensure a clear separation of concerns, the logic for handling individual received messages is extracted from the connectivity services and centralized within dedicated modules that implement the `MessageHandlerService` interface. In this way, each node type (Primary Server, Backup Server, or Client) has its own specialized handler to consume and process exclusively the set of messages that specific system role is authorized to receive.

Figure 6: Class diagram focused on the common interface and implementations of the Message Handlers.

- **Controller-Service Interaction:** Controllers instantiate and maintain direct references to their services to delegate I/O operations and process messages asynchronously.

  A detailed analysis of these interactions highlights how each specific system role maps its own dependencies:

PrimaryServerController Flow: Given the centralized nature of the active backend, this controller orchestrates concurrent collections of services to manage multiple nodes. It maintains the `clientConnectionsByNodeId` map composed of `ClientConnectionService` instances to communicate with individual players, and the `backupConnectionsByNodeId` map which stores `BackupConnectionService` objects to interact with the various backup servers. Additionally, it maps the corresponding message handlers via the `backupMessageHandlersByNodeId` collection.

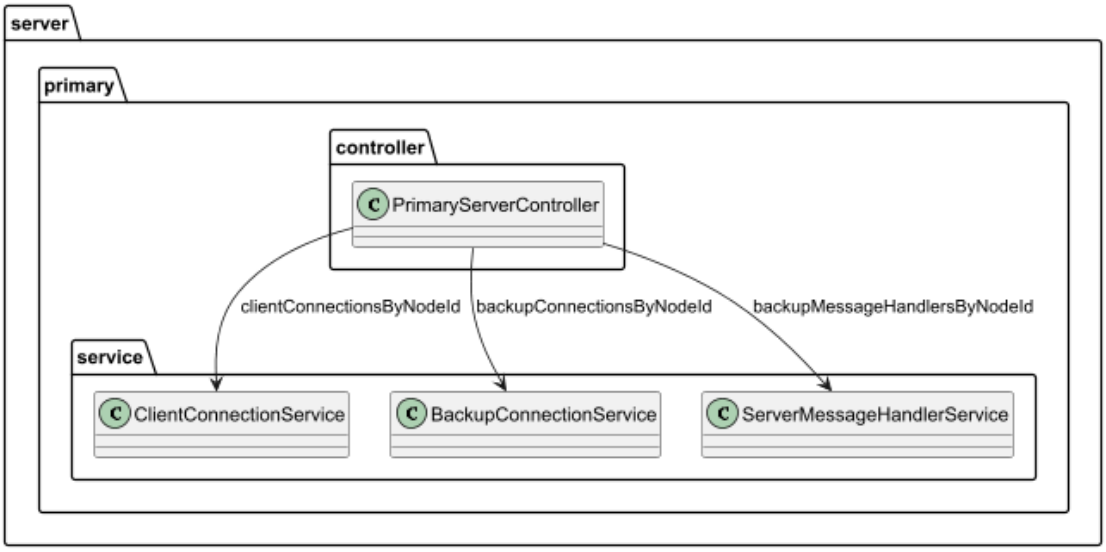

Figure 7: Association and dependency relationships of the PrimaryServerController toward the dedicated Service modules.

BackupServerController Flow: The backup server controller relies on two well-defined channels. It utilizes the `primaryServerConnectionService` as an outbound connection channel to the primary server for receiving state updates, while delegating to the `backupServerMessageHandlerService` the task of decoding and processing incoming packets to keep its local memory perfectly synchronized.

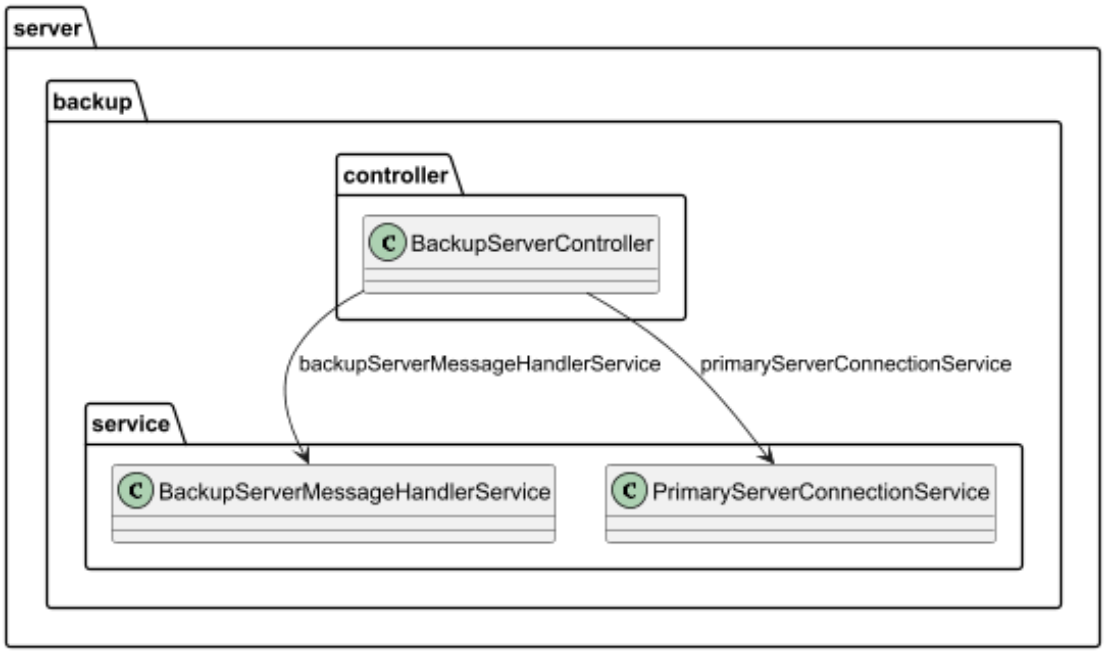

Figure 8: Association and dependency relationships of the BackupServerController for synchronization and failover management.

ClientController Flow: The player's controller adopts a three-way configuration to balance the hybrid architecture of the system. It uses the `serverConnectionService` to transmit matchmaking requests and detect room closures on the primary server; it interfaces with the `peerConnectionService` to exchange shot coordinates directly with the opponent in P2P mode; and it centralizes the listening and dispatching of network messages through the `messageHandlerService`.

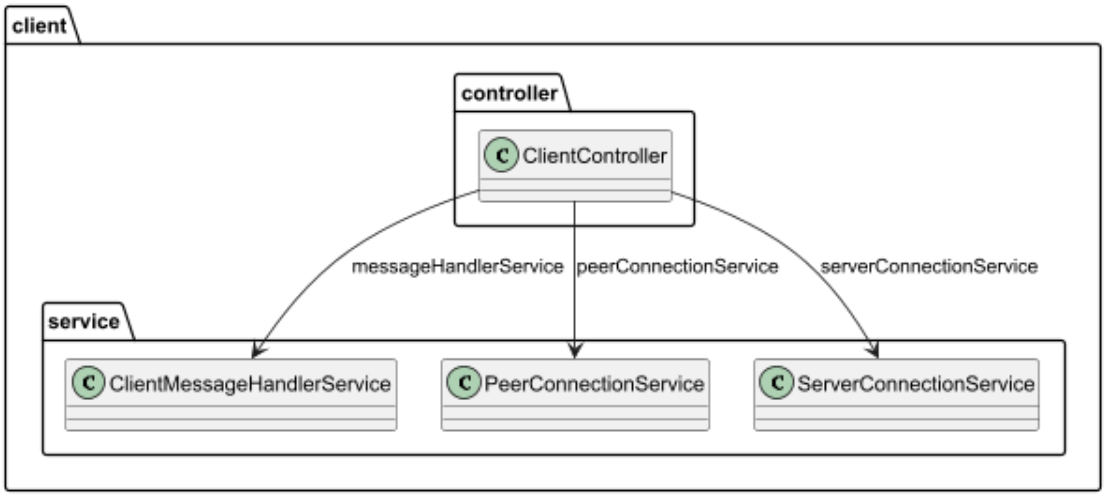

Figure 9: Association and dependency relationships of the ClientController within the hybrid P2P/Server architecture.

3.3.2 Domain Events and Messages

This section illustrates the message format and the communication protocol events exchanged between the various nodes of the system. **Messages between Primary and Client**

- `CS_REQUEST_JOIN_ROOM`: Sent by the client to request matchmaking and join any available random room.
- `CS_RESPONSE_CREATE_ROOM`: Sent by the server as a direct response to the client's request when no available rooms are found, resulting in the creation of a new room and returning its unique ID.
- `CS_RESPONSE_JOIN_ROOM`: Sent by the server to the requesting client when an available room is successfully found. It contains a randomly selected initial turn parameter.
- `CS_CLIENT_EXIT`: Sent by the client to notify the server of its graceful disconnection from the system.
- `CS_ROOM_OPENED`: Sent to Primary by the last player joining the match to confirm that a direct P2P connection with the opponent has been established.
- `CS_ROOM_CLOSED`: Sent by the winner at the end of the match to notify the server that the room can be officially closed and deallocated.
- `CS_ROOM_INTERRUPTED`: Sent by a peer to notify the server of a voluntary, premature disconnection from a match.

- CS_ROOM_TIMEOUT: Sent by Client A to the server when it detects an unresponsive peer during gameplay (e.g., after a missing P2P response like PP_HITTED or PP_MISSED). The server then forwards this timeout notification to Client B to verify its connection status.
- CS_ROOM_TIMEOUT_ACK: Sent by Client B back to the server to acknowledge the timeout inquiry and confirm its active presence in the game. If this acknowledgment is not received within a strict 5-second window, the server presumes Client B is disconnected.
- CS_BACKUP_JOINED: Sent by the primary server to notify active clients about the registration of a new backup server.
- CS_BAKUP_EXIT: Sent by the primary server to notify active clients that a backup server has disconnected.
- CS_RECONNECT: Sent by the client to either the new primary server (if a failover occurred) or the same primary server to announce a reconnection, specifying that it is a continuing session rather than a new client connection.

Messages between Primary and Backup Servers

- SS_HELLO_FROM_BACKUP: Sent by a backup server upon joining the network to register with the primary server and request periodic state synchronization.
- SS_RESPONSE_HELLO: Sent by the primary server to acknowledge the backup's registration and assign its relative replication order.
- SS_SEND_STATE_TO_BACKUP: Sent periodically by the primary server to all registered backups with the full system state. This message also acts as an implicit heartbeat.
- SS_ACK: Sent by a backup server to acknowledge a state update, allowing the primary server to verify the backup's continuous availability.
- SS_BACKUP_EXIT: Sent by a backup server to notify the primary server that it is gracefully shutting down.

Peer-to-Peer Messages (P2P)

- PP_CONNECT: Exchanged between the two peers during the initial connection phase to share player details, such as their chosen nicknames.
- PP_READY: Exchanged between peers after a successful connection to trigger the start of the match and initiate the time-limited ship placement phase.
- PP_START: Sent to the opponent once a player finishes placing their fleet. If the player is chosen to start, they wait for this message (if not already received) and then send their first move. If they do not start, they wait for this message followed by the opponent's first shot.
- PP_SHOT: Sent by a peer to communicate the grid coordinates of an attack. Parameters include: x, y.

- `PP_HITTED`: Sent by a peer to notify the opponent that their previous shot successfully hit a ship.
- `PP_FAILED`: Sent by a peer to notify the opponent that their previous shot missed (water).
- `PP_WIN`: Sent by a peer to notify the opponent of their victory and end the match.
- `PP_EXIT`: Sent by a peer to notify the opponent that they are leaving the match room before the game naturally concludes.

3.4 Interaction

This section analyzes the dynamic behavior of the system through sequence diagrams that orchestrate the three fundamental macro-interactions of the architecture: matchmaking between Client and Server, the game loop execution between Peers in P2P mode, and the lifecycle (registration, synchronization, and deregistration) of a Backup Server with the Primary Server.

3.4.1 Connection and Matchmaking between Client and Primary Server:

The lifecycle of a game session begins when a client decides to enter the system by requesting placement in a room. This preliminary handshake and negotiation phase leverages the centralized infrastructure of the Primary Server to find an available opponent.

The flow, described in Figure 10, involves the following logical steps:

- The client initiates the procedure by sending a `CS_REQUEST_JOIN_ROOM` message to the Primary Server to request random matchmaking.
- The server checks the internal state of the rooms. If there are no rooms waiting for a second player, the server instantiates a new logical room and responds with a `CS_RESPONSE_CREATE_ROOM` message, providing the client with its respective unique `room_id`.
- If, on the other hand, an existing room with only one player inside is found, the server assigns the new client to it, completing the matchmaking process. In this case, the server sends a `CS_RESPONSE_JOIN_ROOM` message that includes, in addition to the `room_id`, the random parameter that determines which player takes the initial turn of the match.

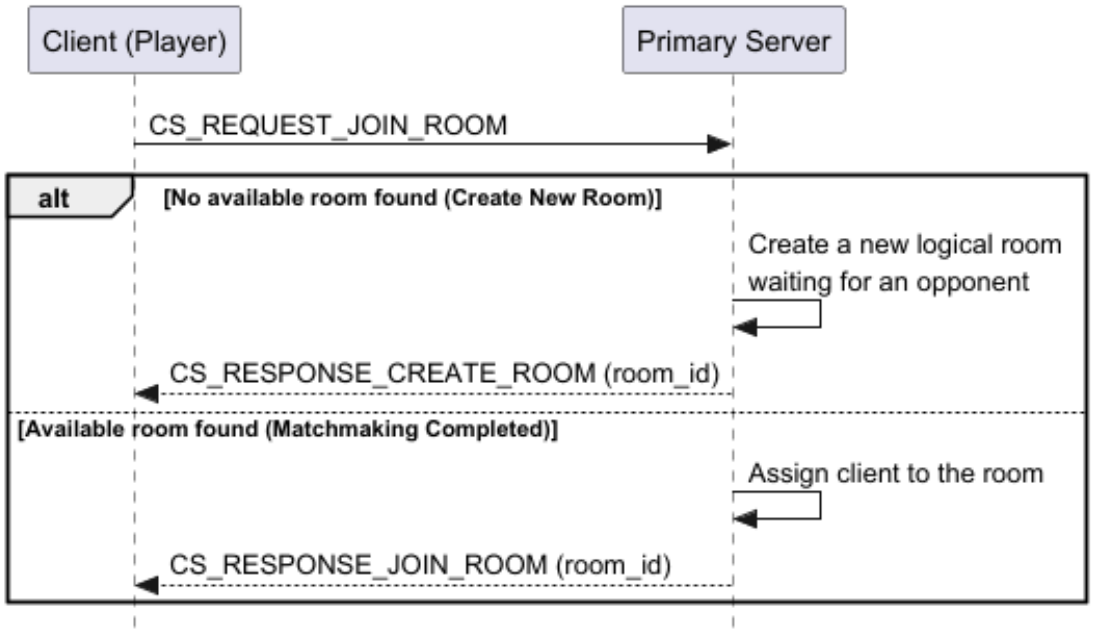

Figure 10: Sequence diagram of the handshake and matchmaking phase between Client and Primary Server.

3.4.2 Peer-to-Peer (P2P) Game Loop:

Once the Primary Server has notified the successful matchmaking, the two clients open a direct TCP communication channel between each other, excluding the server from the management of the match to avoid overloading the backend.

As illustrated in Figure 11, the interaction is structured into three distinct phases:

- **Initialization:** The two clients exchange the `PP_CONNECT` message to share their respective metadata (such as nicknames) and subsequently send `PP_READY` to trigger the timer for the fleet placement phase on the local grid. Once deployment is complete, each node sends `PP_START`.
- **Gameplay Loop:** The game progresses through strict turns. The peer whose turn it is transmits the coordinates of the attack via `PP_SHOT(x, y)`. The receiving peer verifies the outcome on its own grid and responds asynchronously with either `PP_HITTED` (in the event of a successful hit, allowing the attacker to maintain their turn) or `PP_FAILED` (in the case of a miss, resulting in the turn passing to the opponent).
- **Match Termination:** In a natural conclusion scenario (Scenario A), the player who destroys the last opponent ship locally detects the victory and sends a `PP_WIN` message to notify the defeated peer. In the case of a voluntary

and premature exit (Scenario B), the node that disconnects transmits the `PP_EXIT` message.

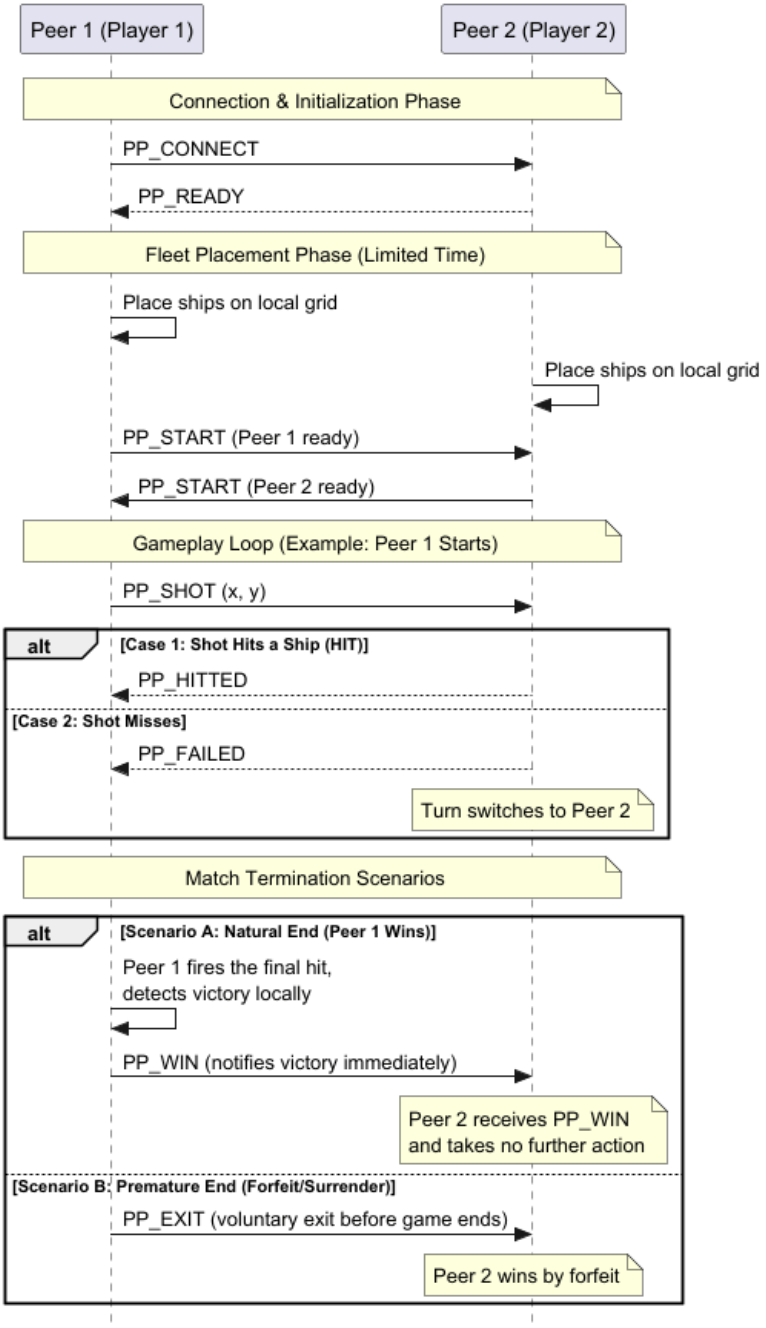

Figure 11: Sequence diagram of the game loop and message exchange in Peer-to-Peer mode.

3.4.3 Registration and Deregistration of the Backup Server:

To guarantee high reliability and fault tolerance, the system allows for the dynamic integration of Backup nodes that replicate the state of the Primary Server. This flow covers backup registration, data alignment, and system exit.

The process, detailed in Figure 12, unfolds as follows:

- **Registration Phase:** The Backup Server connects to the Primary Server by sending an `SS_HELLO_FROM_BACKUP` message. The Primary responds with `SS_RESPONSE_HELLO`, confirming successful registration and assigning the backup's replication order. Immediately after, the Primary Server sends a `CS_BACKUP_JOINED` message to all active clients in the network to update their local list of available redundant servers.
- **Periodic Synchronization Loop:** A continuous cycle is initiated in which the Primary Server periodically transmits the `SS_SEND_STATE_TO_BACKUP` message containing the updated system state. The Backup processes the data and replies with an `SS_ACK` message, which also serves as an implicit heartbeat to confirm its active presence.
- **Shutdown Scenario:** When the Backup Server undergoes a controlled shutdown, it sends an `SS_BACKUP_EXIT` message to the Primary. The latter removes it from the backup list and promptly notifies all active clients via the `CS_BACKUP_EXIT` message, allowing them to update their failover list and avoid empty connection attempts toward a deactivated node.

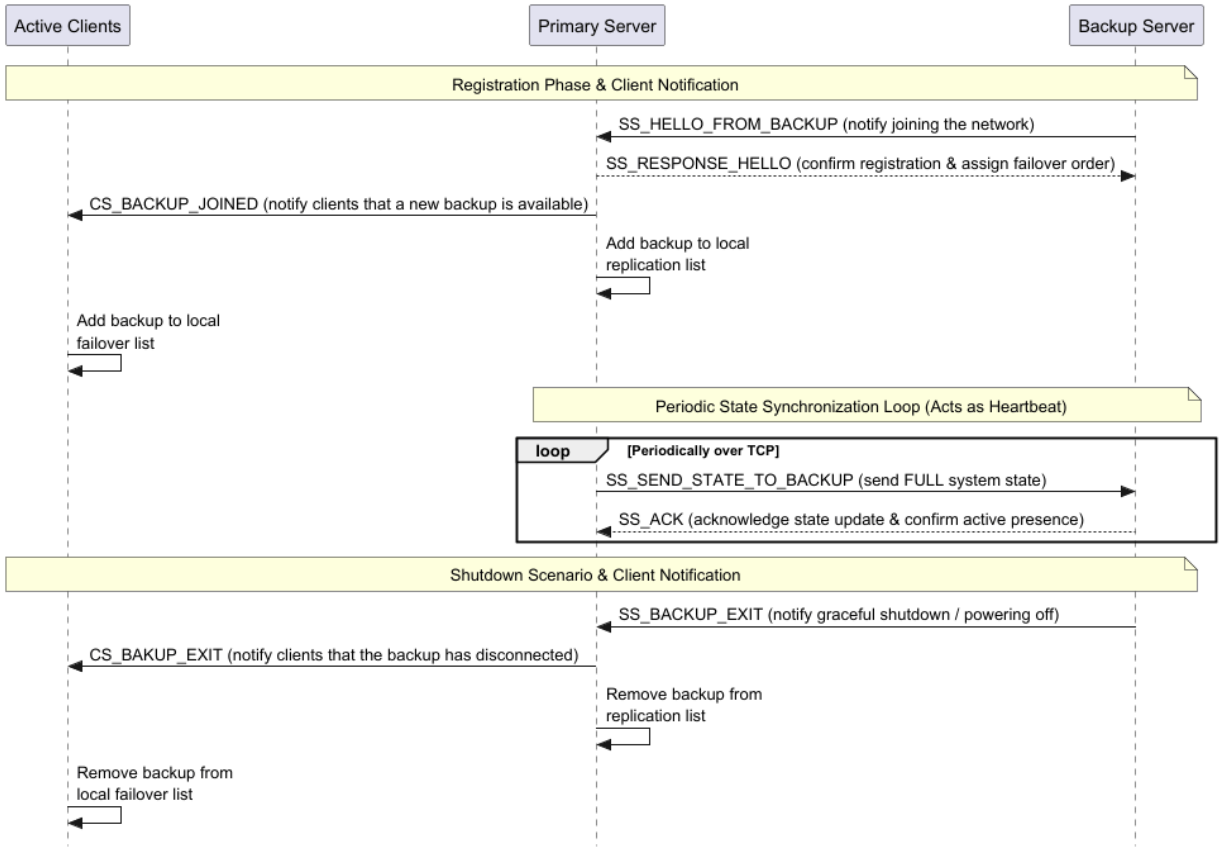

Figure 12: Sequence diagram for the Backup Server lifecycle management and state synchronization.

3.5 Data and Consistency Issues

The decentralized and hybrid architecture of the system introduces specific dynamics and trade-offs regarding data management, game state consistency, and the security of interactions between nodes.

Absence of Data Persistence: One of the fundamental characteristics of the system is the total absence of a supporting persistence layer (such as relational or non-relational databases). All information necessary for the application's operation is maintained exclusively in central memory.

– **Server Side:** The Primary Server (and the Backup nodes) maintains the volatile global state, limiting itself to tracking the real-time list of connected users and active rooms for matchmaking purposes. In the event of a simultaneous crash or a complete restart of the backend infrastructure, this

information is entirely lost, delegating to the clients the task of announcing their presence again via reconnection messages (`CS_RECONNECT`).

- **Client Side:** Player nodes store only the state of the current match and the session parameters, both of which expire as soon as the match is concluded or the application is closed.

Fleet Exchange and GUI Updates: To optimize the user experience and eliminate perceived latency during critical phases of the gameplay loop, the protocol requires that, at the end of the fleet placement phase, the two peers transmit the entire layout of their respective ships to each other. Consequently, each client acquires complete knowledge of the opponent's grid right from the start of the match. Naturally, this information remains confined within the logical layer of the software and is strictly hidden from the graphical user interface displayed to the user.

This approach offers a crucial advantage in terms of responsiveness: the moment a player launches an attack by sending the `PP_SHOT(x, y)` message, the local application is able to immediately determine whether the shot was a hit (*Hit*) or a miss (*Miss*). The GUI can thus update instantaneously, without having to wait for network propagation delays, processing by the remote peer, and the reception of the corresponding response message.

Prevention of Arbitrary Byzantine Nodes: In addition to the performance benefits related to the GUI, the preemptive exchange of grids addresses a specific security requirement against potential opportunistic or malicious behaviors (*Byzantine* nodes).

In a traditional P2P model, where a node does not know the opponent's grid and waits for the other player to validate the outcome of each individual shot, a modified or compromised client could systematically lie. A malicious user could indeed instruct their software to always return a negative outcome, denying that a shot landed even if one of their ships was actually struck, effectively rendering themselves invincible.

3.6 Fault-Tolerance

System resilience and service continuity are guaranteed by two complementary fault-tolerance mechanisms designed to handle both the crash of centralized infrastructure nodes and the connectivity instability of individual players.

Redundancy and Failover of Backup Servers: As previously introduced, the high availability of the backend relies on the presence of one or more Backup Server nodes. These nodes maintain a copy of the Primary Server's state through

a continuous synchronization cycle. In the event of a failure or sudden disconnection of the Primary Server, active clients (which locally store the list of registered backups) can initiate an immediate failover procedure. Clients migrate their session requests to the designated Backup Server using the `CS_RECONNECT` message, minimizing downtime and preserving the integrity of ongoing matchmaking rooms.

Peer Timeout Arbitration Mechanism: In a hybrid architecture where the match takes place over a direct P2P channel, the crash or silent disconnection of a player could cause the opponent's client to freeze. To resolve this critical issue without constant centralized monitoring, a reactive arbitration mechanism driven by the Primary Server has been implemented:

- **Local Suspicion and Reporting:** If the current client does not receive a response from the remote peer within the specified timeout limit, it assumes that a disconnection has occurred. It then displays a disconnection message to the user and notifies the Primary Server by sending a `CS_ROOM_INTERRUPTED` message.
- **Primary Server Verification:** Upon receiving the report, the Primary Server sends a `CS_ROOM_TIMEOUT` control message to the node suspected of being disconnected.
- **No Response (Crash):** If the suspected peer fails to respond within the timeout limit, it is also assumed disconnected by the Primary Server, which will proceed to permanently remove it from the system's active node list.
- **Response Received (False Alarm):** If the peer is still active, it receives the `CS_ROOM_TIMEOUT` from the primary and realizes that the opponent suspected its disconnection and has therefore left the match. Consequently, it responds to the Primary Server with a `CS_ROOM_TIMEOUT_ACK` message to confirm it is still connected, and then displays a message to the user warning them that the opponent has disconnected.

4 Implementation

4.1 Data Representation and Protocol Messages

Data in transit within the network is entirely represented and exchanged in the form of JSON payloads encoded in UTF-8 format. This approach guarantees interoperability and maximum flexibility when parsing raw data. Within the system, byte streams received from TCP sockets are deserialized into high-level logical objects, and conversely serialized into bytes prior to transmission.

4.2 Technological Details

The system was developed by minimizing invasive external frameworks, relying instead on core Java libraries alongside lightweight components for serialization and testing. The technological choices adopted are summarized below:

- **Build and Dependencies (Gradle):** The management of the software lifecycle and external libraries is entrusted to **Gradle**. The `fatJar` task enables the generation of an autonomous, self-contained JAR archive for each node.
- **Networking and Communication (Java TCP Sockets):** Communication between nodes (client-server and peer-to-peer) is carried out via native TCP sockets (`java.net.Socket` and `java.net.ServerSocket`) without intermediate middleware. The message exchange leverages a *newline-delimited* protocol managed through `PrintWriter` and `BufferedReader`.
- **Serialization (Gson 2.11.0):** The transparent conversion between Java records (the protocol tuples) and network payloads is entirely delegated to the **Gson** library, which handles JSON serialization and deserialization.
- **Graphical User Interface (Java Swing):** The visual interaction layer is developed natively using **Java Swing**. The architecture separates logical views into `GameView` and `MenuView` for the clients, and `SimpleServerView` for monitoring the internal state of the servers.
- **Concurrency and Thread-Safety (`java.util.concurrent`):** The asynchronous nature of the system is coordinated using standard Java primitives:
  - `ScheduledExecutorService` for timed transmission of the state updates to the backups.
  - `FutureTask` to handle connection timeouts toward peers.
  - `ConcurrentHashMap` for the thread-safe tracking of active connections.
  - Dedicated *Daemon Threads* (one thread for each connected node) for continuous network listening.
  - `volatile` variables, atomic structures (`AtomicInteger`), and flags for the secure exchange of synchronization events between threads.
- **Automated Testing (JUnit 5):** The validation of domain models, game constraints, and the correctness of JSON serialization streams is managed by a suite of unit tests written in **JUnit 5**.

5 Validation

5.1 Automatic Testing

As mentioned in the implementation section, testing was executed utilizing the JUnit framework. The following components of the system are subjected to automated tests:

- **Game Logic:** Classes related to core gameplay, including entity creation, component behaviors, and system operations.
- **Game Grid:** Validation of the correctness of the game's underlying data structures, specifically the grid that manages and tracks ship placement.
- **Messages:** Comprehensive coverage of both the serialization and deserialization processes of network messages.
- **Lobby HTTP Server:** Extensive testing of the lobby server, ensuring that every possible interaction via HTTP methods yields the expected results.

Automated tests can be executed from the command line using the `gradlew test` command, which leverages the Gradle build automation tools.

5.2 Acceptance Testing

While automated testing validates the fundamental business logic, the inherent complexity of distributed systems demands manual verification within a real operational environment. I conducted manual end-to-end tests across multiple personal computers within a local home network to validate the system's runtime behavior. This evaluation focused on scenarios that automated tests could not natively cover.

6 Deployment

The complete source code of the project is publicly available and can be cloned from the official GitHub repository at `https://github.com/leonardo284/Distributed-Battleship`.

However, compiling the source code locally is not strictly required to execute the system. For a rapid and straightforward deployment, a pre-compiled standalone executable archive (*fat JAR*) is available for download on the official GitHub Release page at `https://github.com/leonardo284/Distributed-Battleship/releases/tag/1.0`.

Once the executable is downloaded, users can refer directly to the detailed instructions provided in Section 7 to launch the application multiple times, properly configuring each specific system node (Primary Server, Backup Server, or Client) depending on the desired architectural role.

7 User Guide

The application is designed to be simple to use. Below a user guide is provided:

1. Start the primary server (only one): open a terminal and execute the following command: ***java -jar DistributedBattleship-1.0.jar primary***
2. Start the peer application (as many as you want): open a terminal and run the following command in a terminal: ***java -jar DistributedBattleship-1.0.jar client*** to launch a client on the same machine where the primary server was started. Alternatively, if you are on another node within the local network, start the client using the command ***java -jar DistributedBattleship-1.0.jar client ipAddress*** (for example, *java -jar DistributedBattleship-1.0.jar client 192.168.178.21*).
3. Start the backup server (as many as you want): open a terminal and run the following command in a terminal: ***java -jar DistributedBattleship-1.0.jar backup*** to launch it locally. Otherwise, if located on a different node of the local network, start the backup server with the command ***java -jar DistributedBattleship-1.0.jar backup ipAddress*** (for example, *java -jar DistributedBattleship-1.0.jar backup 192.168.178.21*).

In short, when starting the client or the backup without specifying an IP address via the command line, the application assumes that the primary server is running on the same node; otherwise, if an IP address is specified, it will attempt to communicate with that specific IP.

Furthermore, upon launching the client, you can append the optional *debug* parameter to launch the client in debug mode—a functionality introduced to streamline and accelerate application development.

When launching the client, the following window will be displayed, allowing you to enter your nickname:

Figure 13: Home screen of the Client graphical user interface.

If no game room is currently available, the system will automatically create a new one, and the user will wait for an opponent while viewing this window:

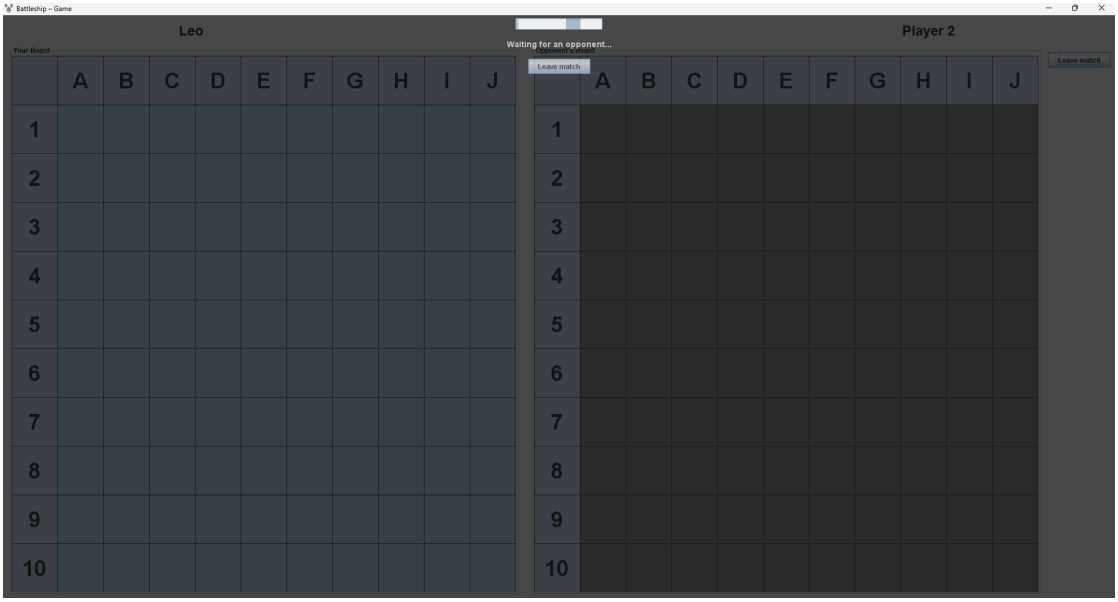

Figure 14: Waiting interface for the first player inside the matchmaking room.

As soon as the opponent connects, the match begins automatically, starting with the fleet placement phase. Ships can be placed by choosing between horizontal or

vertical orientation using the two buttons at the top right, and then clicking on the grid cell where you want to position the ship. A timer is also visible in the upper right corner; once it expires, any remaining unplaced ships will be positioned randomly.

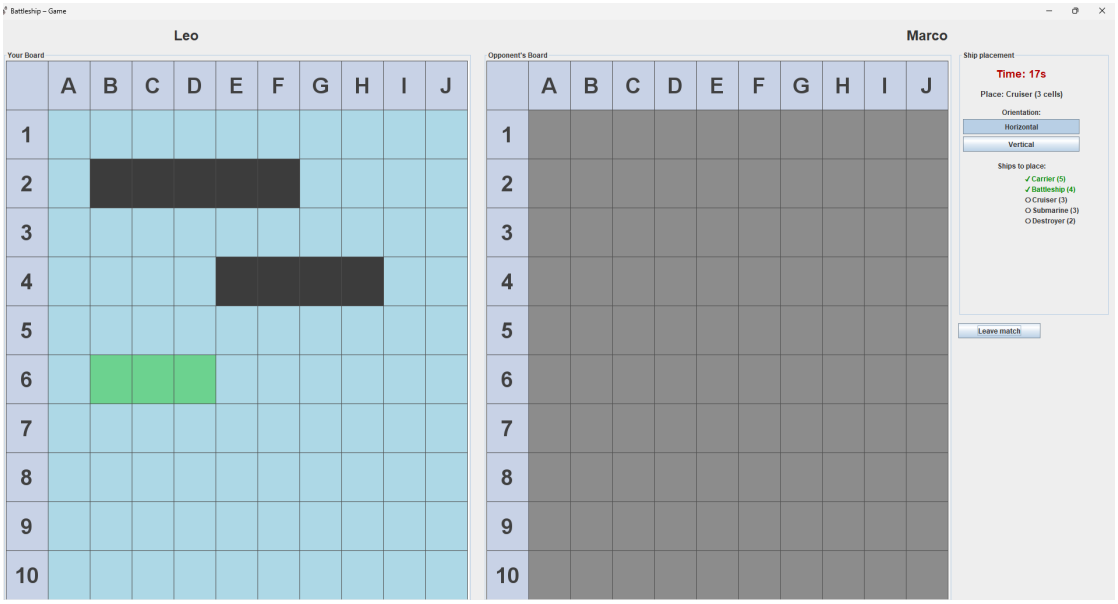

Figure 15: Initial phase of placing ships on the game grid.

Following the placement phase, the actual match begins. The game is strictly turn-based; when it is your turn, a window similar to this one will appear, showing the current player's name (Leo) on the left and the opponent's grid (Marco) on the right. On the current user's grid (left), cells containing intact ships are colored black, ship segments hit by the opponent are colored red, and the opponent's missed shots are marked in blue. Meanwhile, on the opponent's grid (right), you can view your own successful hits on opponent ships in red, and missed shots in light blue.

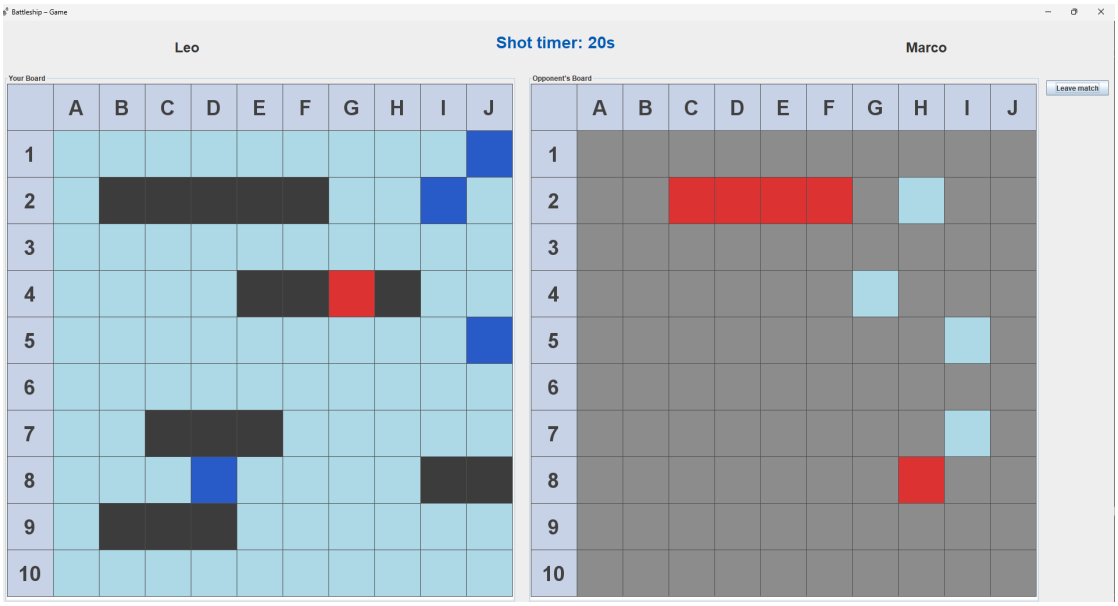

Figure 16: Graphical user interface during your own game turn (cell selection).

Conversely, when it is the opponent's turn, the following waiting screen will be displayed:

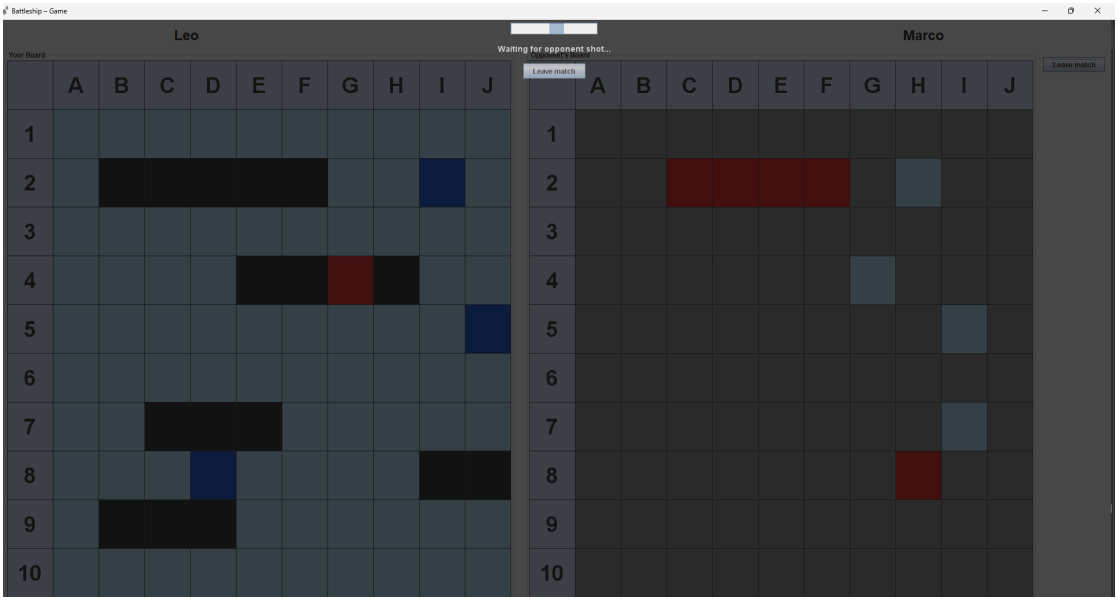

Figure 17: Game screen displayed during the opponent's turn.

At the conclusion of the match, a message appears indicating the winner:

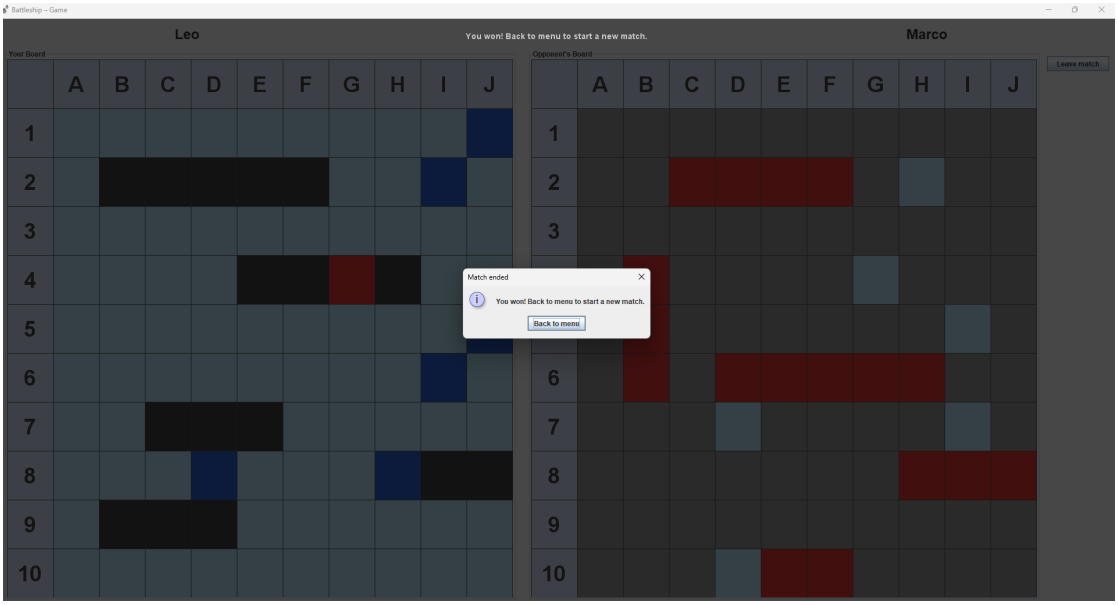

Figure 18: Final summary screen displayed in case of a match victory.

8 Self-Evaluation

The project was entirely conceived, designed, and implemented as an individual effort. Developing the system single-handedly represented a significant opportunity for engineering growth, forcing me to personally face and manage every single aspect of the software lifecycle: from defining the communication protocol and concurrency models to handling all corner cases and typical anomalies that arise within a distributed environment.

Given the project's timeline and individual scope, certain architectural trade-offs were necessary, such as forgoing the implementation of *NAT Traversal* (thereby requiring all nodes to reside within the same local network segment) and relying on the complete volatility of the in-memory state on the Primary Server, which was partially mitigated by the failover system across the Backup Servers.

Regarding the graphical user interface developed in Java Swing, although it is linear and minimal, I believe it turned out remarkably well and is entirely functional relative to the predefined objectives. An advanced aesthetic design for the GUI was not, after all, the primary focus of this thesis, which centered strictly on the core challenges of distributed systems. In light of the challenges addressed and the solutions implemented to guarantee fault tolerance and synchronization, I am highly satisfied with the work completed and the results achieved.

9 Future works

Although the current implementation fully satisfies the project requirements, the system could be further enhanced with the following future improvements:

- **In-Game Chat System:** Implementing a real-time chat functionality would enable direct text messaging between the two players during a match, enhancing the interactive experience.
- **Private Matchmaking (Play with Friends):** While the current room assignment is completely random, a valuable future enhancement would allow users to create private rooms. A player could host a private session, generating a unique room identifier that a friend could explicitly enter to join the match.