Beach Volley Tournament Management System

Final Report

Distributed Systems
A.Y. 2023/2024

Montanari Nicola
nicola.montanari14@studio.unibo.it

December 2024

Contents

This project aims to develop a Beach Volley Tournaments Management Software designed to enhance the setup and execution of beach volley tournaments for private organizers.
The system will improve all aspects of tournament management, beginning with player registration and continuing until the competition's conclusion.
Organizers will have the ability to create new tournaments and customize various elements, such as the number of matches each player will participate in and the overall competition structure. This flexibility is crucial for accommodating different tournament types, ensuring a enjoyable experience for both organizers and participants.
The software will feature a range of functionalities that allow for the delegation of specific roles to tournament members, facilitating efficient management throughout the event.
It will support player registration, match creation, and tracking of match start and end times, including score entry, all while maintaining the integrity of the games.
A key feature of the system will be a real-time leaderboard, providing immediate updates on individual player rankings. This feature enhances audience engagement by enabling spectators to follow the live status of the tournament through graphical user interface (GUI) displays.

1 Domain Analysis

Before specifying the system requirements and designing a solution, it is essential to understand the project's objectives, expected functionalities, and any other relevant aspects that help to define the problem.
This will involve a detailed breakdown of the beach volleyball and tournaments rules, a summary of user interactions with the system, and a set of questions and answers that emerged during the project's initial phase, considering perspectives from both the developers and potential users.

1.1 Goal

The goal of this project is to develop a real-time management system for private organizers that wants to create and manage their beach volleyball tournaments by designing a distributed platform capable of enforcing tournament rules while coordinating interactions among various component of the organization, regardless of their physical location.
Although the tournament structure and user actions are relatively simple, this project presents an opportunity to tackle significant challenges in distributed systems, such as ensuring fast, reliable, and most importantly, consistent communication between users.

The expected outcome is a functional management system that allows multiple users to interact seamlessly across different devices. The platform should ensure that users can organize and participate in tournaments smoothly, without needing to understand the underlying system mechanics.

1.2 Rules

1.2.1 Beach Volley Rules

Each beach volleyball match consists of sets, typically played until one team wins two sets. Matches start with both teams at 0 points, progressing according to standard beach volleyball scoring rules. Each set is played to 21 points, with a team needing at least a 2-point lead to win the set.

Points are awarded by serving and winning rallies. A team scores by either grounding the ball in the opponent's court or when the opposing team commits a fault. Each team may touch the ball up to three times before sending it back over the net, but individual players may not hit the ball consecutively.

Teams are composed of two players. Players are free to strategize and communicate openly during the sets, with no restrictions on their choice of verbal or non-verbal communication.

A team wins the match by securing the required number of sets, usually two out of three. In tournaments, winning conditions may be adjusted based on specific rules set by the tournament organizer, but the basic objective remains to achieve a majority of sets.

1.2.2 Tournament Rules

In each tournament, players participate in a predefined number of sets, established at the time of tournament creation. Instead of fixed teams, players could also be are randomly assigned to different teams at the beginning of each set, ensuring varied team compositions throughout the tournament. This randomized team formation promotes a balanced and dynamic competition.

After each set, the standings are updated based on individual player performance, recording each player's wins, losses, and point differential (points scored minus points conceded). Points from each set are added to each player's individual score, contributing to their overall ranking within the tournament.

The tournament winner is determined by the total number of victories each player achieves across all sets. If two or more players have the same number of wins, the winner is determined by the highest point differential, rewarding players who maintained a stronger overall performance. This structure allows for a clear ranking that reflects both wins and overall contributions to each game.

The tournament concludes once all scheduled sets have been played. Final standings are then determined, with the top-ranking player declared the tournament winner. This format supports both competitive integrity and fairness by mixing teams and basing rankings on consistent performance across multiple randomized matchups.

1.3 Usage Scenarios

This section outlines the primary ways users interact with the system, detailing typical scenarios to illustrate how the application is used to manage tournaments. Each scenario provides insight into user expectations and the functionality required to support a good experience across different user roles.

Creating a New Tournament

- **Description:** A user, typically an administrator or event organizer, wants to create a new beach volleyball tournament. They enter the necessary tournament details, such as name, location, date, number of players, and format.
- **Expected Interaction:** The user accesses the "Create Tournament" option from the main menu, fills out the required information, and submits it. Once submitted, the server registers the tournament, sends a confirmation to the user, and notifies all clients of the new tournament.
- **Purpose:** This feature enables administrators to organize and set up new tournaments efficiently, allowing players to join and participate.

Joining a Tournament

- **Description:** Players who wish to participate in a tournament need to join an existing one. Secretary of the Organization can register the player to desired tournament.
- **Expected Interaction:** A Secretary navigates to the "Register Player" section, and register the new player to the tournament. The server updates the tournament's participant list and sends a notification to all clients.
- **Purpose:** This scenario allows players to sign up for tournaments, keeping the participant list accurate and up to date.

Viewing the Leaderboard

- **Description:** A user wants to check the current standings in a tournament to see the rankings, wins, losses, and points difference of each player.
- **Expected Interaction:** The user selects the "View Leaderboard" option from the main menu and chooses the relevant tournament. The leaderboard is then displayed, showing the most recent scores and rankings.
- **Purpose:** This feature provides real-time updates on tournament progress, enabling spectators to follow the competition closely.

Starting a Match

- **Description:** When a match is set to begin, an administrator starts the match in the system, allowing players to see which teams they are assigned to and start playing.
- **Expected Interaction:** The administrator selects four specific players in the tournament and clicks "Create Match." This action updates the status of the tournament in the system.
- **Purpose:** This interaction initiates the gameplay and team formation, providing players with the necessary information to begin the match.

Recording Match Results

- **Description:** Once a match concludes, the match outcome is recorded in the system.
- **Expected Interaction:** A Referee accesses the live match list, inputs the final scores, and submits them. The system updates the tournament leaderboard based on the results and notifies all clients of the changes.
- **Purpose:** This ensures that all scores and rankings are updated in real time, allowing accurate and transparent tracking of tournament progress.

Receiving Real-Time Notifications

- **Description:** As matches and tournaments progress, users receive real-time notifications about key events, such as a new tournament creation, player registration, match updates, and leaderboard changes.
- **Expected Interaction:** Users are automatically notified by the system through notifications each time a key event occurs, and their local view updates with the latest data.
- **Purpose:** Real-time notifications keep users informed and engaged, ensuring they have up-to-date information without needing to manually refresh or check for updates.

1.4 Questions & Answers

The following is a list of questions and answers that emerged during the initial exploration of the tournament management's domain.
While many of these questions have been refined and integrated into the tournament rules and domain analysis, they are included here to reinforce key concepts and provide clearer explanations.

- **Question:** *Is the username of the player unique?*
  Answer: Yes, each player nickname is unique to avoid issues with players called the same way
- **Question:** *Can a user join a tournament that has already started?*
  Answer: Yes, there is no restriction of when a player can join a tournament. Practically, a tournament should register a player before the start of a Tournament and of course not at the end of it.
- **Question:** *When should a match begin?*
  Answer: A set begins once all players assigned to a match have confirmed their availability and are ready. The system verifies that each match has the minimum number of players before allowing it to proceed.
- **Question:** *What happens if a referee is unable to continue during a match? (disconnects)*
  Answer: The match will proceed as long as there is at least one referee available (connected). If a referee is unavailable during the match, a substitute referee can still manage the points system to ensure the match continues without disruption.
- **Question:** *What happens if a server fails?*
  Answer: Multiple "Node" servers keep the Master Server monitored. In any case of Master server failure, an election will be initiated between the Node servers. Once the election is complete, an elected Node Server will replace the failed Master Server.

1.5 Self-assessment policy

- *How should the quality of the produced software be assessed?*
  The quality of the system will be evaluated mainly based on non-functional requirements, which will be defined and refined during the design phase. Key quality attributes to be considered include performance and scalability.
- *How should the effectiveness of the project outcomes be assessed?*
  The effectiveness of the software will be evaluated primarily through its functional aspects. Clearly defined functional requirements will form the basis of this assessment, ensuring that all specified features are implemented as expected. Manual integration testing will be conducted to assess the overall system behavior.

2 Requirements

After the domain analysis phase, the concepts and scenarios identified are translated into formal requirements. These requirements will specify the actions that need to be taken, providing a roadmap for the design and implementation stages. They will also serve as a reference to ensure that all necessary tasks for developing a fully functional system are completed.

2.1 Functional Requirements

The following are requirements that define the functionalities that the system must provide. These are the features that needs to be implemented to consider the system complete.

- Create a Tournament
  - Set a name
  - Set a Organization name
  - Set Number of individual matches
- View List of Tournaments created
  - View Tournament Info
- Player Registration
  - Set a name
  - Assign the player to a specific tournament
- View Player Leaderboard of a specific tournament
- Manage Live matches (referee) of a specific tournament
  - View list of current matches
  - Set points of a match and set it as completed
- Manage a specific Tournament
  - View Player Leaderboard
  - Create new matches for the tournament

2.2 Non-Functional Requirements

The following requirements outline the expected performance and quality standards the system must meet:

- The system should include fault tolerance features, specifically:
  - It must continue to function normally if one or more users disconnect.
  - It must continue to function normally if one or more servers fail.
  - It should be able to handle brief connectivity interruptions (up to 5 seconds) from users without affecting real time updates data consistency.
- The application must function smoothly and maintain full functionality across different operating systems
- The system must operate correctly without the need for any additional configuration, especially regarding network settings.

2.3 Implementation requirements

The following requirements describe the environment in which the system will be developed and deployed, including details on the tools and technologies that will be used, as well as the necessary conditions for execution:

- The system must utilize a *client-server* architecture, where the server handles application logic and manages communication with the clients.
- The master server will be responsible for managing the state of the tournaments and synchronizing updates across clients, ensuring real-time interaction.
- The master server will send real time notifications to every connected client every time a new update has been received.
- The node servers will monitor the status of the master server and replace it if necessary.
- The user interface must be developed using *Java Swing* to ensure a usable and responsive graphical user interface.
- The project should incorporate tools for build automation and dependency management, such as Maven or Gradle.
- The project code must be managed using Git for version control.

3 Design

The design phase outlines the core components of the system and offers a detailed analysis of the decisions made regarding its functionality. For each design choice, an explanation is provided, along with a discussion of other potential alternatives that were considered. This approach ensures that the system fulfills its requirements while balancing factors such as efficiency, scalability, maintainability, and development time.

3.1 Architecture

This project is implemented using a client-server architecture, which provides a clear separation of responsibilities between the client applications and a centralized server. While other distributed architectures could potentially support similar functionalities (such as peer-to-peer or decentralized models) the client-server structure offers several advantages that align well with the project's requirements.

In the client-server model, the master server acts as the central authority responsible for managing match states, handling tournament data, and ensuring the consistent application of game logic across clients. This setup allows for a high level of control over data integrity and synchronization, as the server maintains a single source of truth for all connected clients. Each client interacts with the master server to perform actions like creating matches, recording set results, or viewing the leaderboards, while the server processes these requests and distributes updates back to all connected clients as needed.

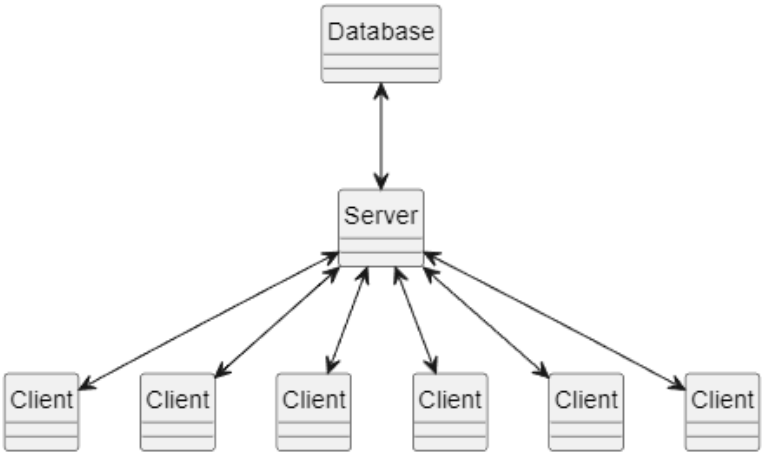

Figure 1: Client Server Architecture Scheme

This architecture also simplifies scalability and security. Adding additional clients does not require complex networking between each client; instead, each client only connects directly to the master server, making the system easier to scale as the user base grows. Additionally, security measures are more manageable, as sensitive data and game logic are centralized, allowing for more straightforward implementation of access controls and secure communications.

Using a client-server model reduces the potential for errors and inconsistencies that can arise in fully distributed systems, where nodes may struggle to maintain consistent data across the network. In this design, the master server is responsible for key state changes, which eliminates the need for clients to coordinate with each other directly and ensures that updates are efficiently distributed. This design provides a robust, maintainable, and performing structure well-suited to the real-time demands of the application.

This project will be modeled using a single master server and a single database. To mitigate the risks of a "single point of failure", Multiple "Node" servers will be modeled that will keep the Master Server monitored. In any case of Master server failure, an election will be initiated between the Node servers. Once the election is complete, an elected Node Server will replace the failed Master Server. To choose which Node server will become Master Server during an election, each process will compare its "*serverId*" with all other servers. The server with the biggest ID will be declared the Master Server. By deploying multiple servers, the system gains resilience against server failures; if one server encounters issues, the others can handle client requests without impacting the user experience. This approach enhances the system's fault tolerance, contributing to greater stability and reliability for large-scale or mission-critical applications.

3.2 Structure

The system is composed of distinct components that work together to handle the different functionalities required for managing tournaments and matches in real-time.

At the core of the system is the Master Server, which is responsible for managing the business logic, processing user requests, and ensuring synchronization between all connected clients. It acts as the central point of control. The server also pushes real-time updates to clients, ensuring that all users are notified of any changes or actions that need to be reflected immediately.

The Client components are the interfaces through which users interact with the system. All users, **Admin**, **Secretary**, **Referee**, and **Public (Leaderboard)**, are clients in this architecture.

Each type of user has specific roles:

- **Admin:** Can create, manage, and control tournaments and matches.
- **Secretary:** Subscribe players to tournaments.
- **Referee:** Update the system with match results.
- **Public (Leaderboard):** Views the tournament leaderboard.

All clients communicate with the server for sending requests and receiving responses. Additionally, clients receive real-time updates, which notify them of changes like tournament creation, match updates, and player actions. This real-time communication ensures that all connected clients have up-to-date information.

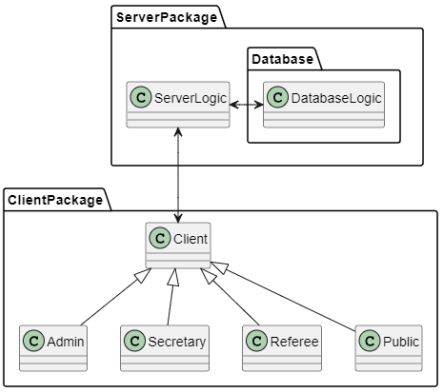

Figure 2: UML Class Diagram

3.3 Behavior

The behavior of each entity is summarized in their relative state diagram. The diagrams only considers the point of view of a single entity while interaction between entities are omitted.

Secretary:

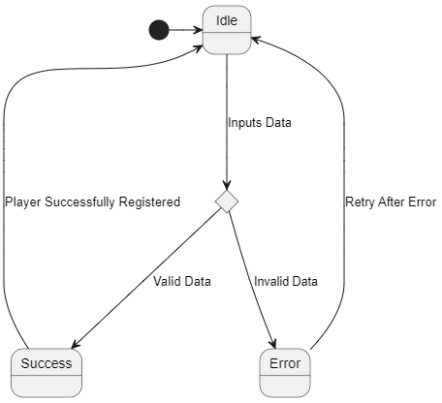

Figure 3: Secretary State Diagram

Admin:

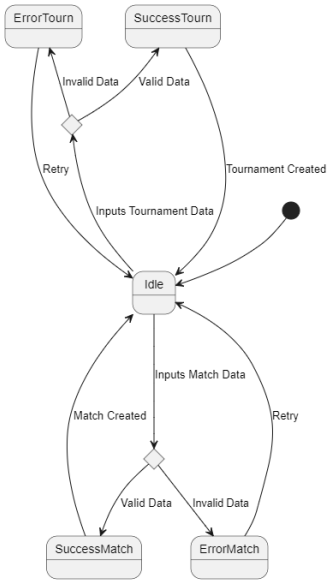

Figure 4: Admin State Diagram

Referee:

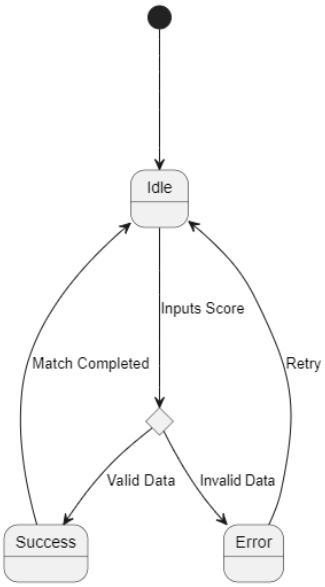

Figure 5: Referee State Diagram

Server:

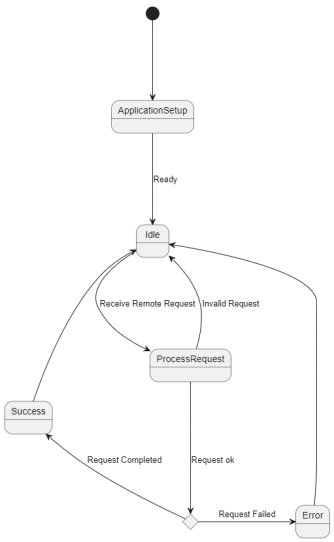

Figure 6: Server State Diagram

3.4 Interaction

The interaction between clients and the server in this system follows a clear and structured process that ensures smooth and efficient communication.
Each client type (Admin, Secretary, Referee, and Public) interacts with the server to perform specific tasks based on their specific role, while the server manages the game state, processes requests, and synchronizes updates across all connected clients.

1. Client-Server Communication via RMI
Clients use Remote Method Invocation (RMI) to send requests to the server and invoke methods remotely. For example, when an Admin creates a new tournament, they will send a request to the server via RMI. The server processes this request, creates the tournament in the system, and then responds to the client with confirmation.

2. Real-Time Updates via Webhooks
While RMI handles the primary interactions, webhooks are used for real-time communication between the server and clients. This allows the server to push updates to all connected clients whenever a change occurs in the system, such as a new tournament being created, a player joining a tournament, or match results being recorded. Upon receiving the update, each client evaluates the notification and updates its view accordingly.

Through RMI and webhooks, the server maintains data consistency across all clients, ensuring no client is left behind.

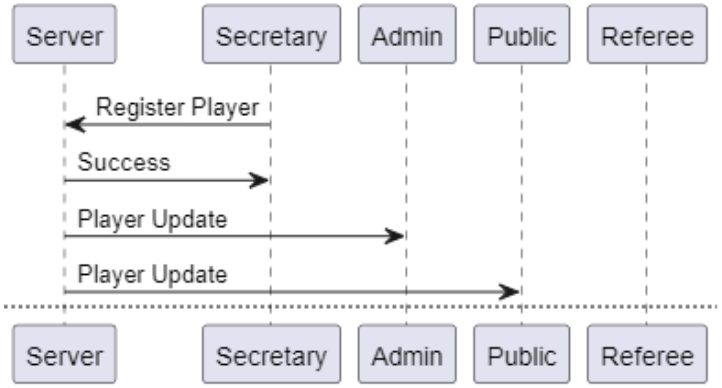

Figure 7: UML Sequence Diagram Interaction with Secretary

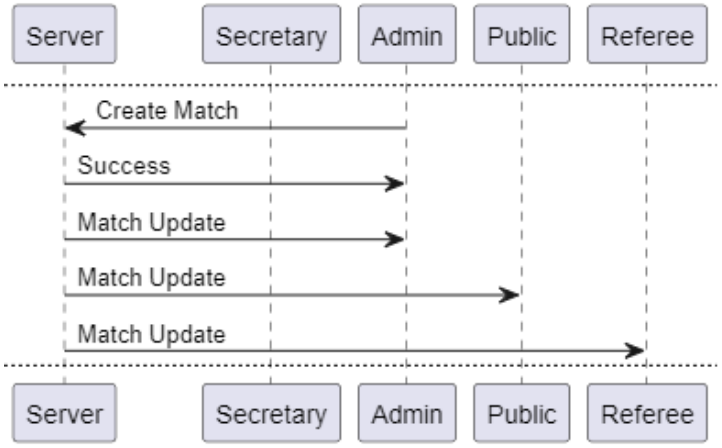

Figure 8: UML Sequence Diagram Interaction with Admins (Match)

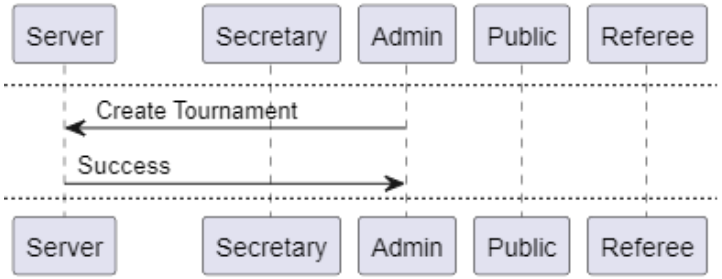

Figure 9: UML Sequence Diagram Interaction with Admins (Tournament)

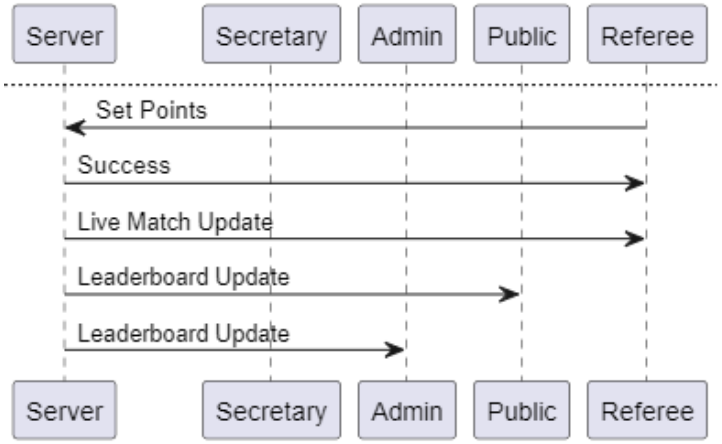

Figure 10: UML Sequence Diagram Interaction with Referee

4 Implementation details

After the design phase, the implementation phase focuses on translating the chosen design into code. The emphasis will be placed on key or complex implementation details, while more straightforward aspects will be omitted.

4.1 Remote Method Invocation

Remote Method Invocation (RMI) is used to establish the main communication between the client and the Master server. RMI allows clients to call methods on the server as if they were invoking them locally, enabling distributed interactions while maintaining a clear separation between the client and server components.

The Master server exposes a set of remote interfaces that define the methods available for clients to call. These remote interfaces are implemented on the server side, allowing the server to perform specific actions, such as updating tournament standings, retrieving player information, or managing tournament state.
The client-side application accesses these remote methods through RMI stubs, which act as proxies for the actual methods on the server.

When a client needs to perform an action, it calls a method on the remote interface. For instance, if a client wants to update match results, it invokes the corresponding remote method. These calls are transmitted over the network, and the server processes the request and sends back the appropriate response. The use of RMI ensures that the communication is transparent to the user, allowing them to interact with the system without needing to worry about the underlying network complexities.

To facilitate this communication, when the Master server starts, it binds the remote objects to a specific registry, making them accessible to the clients.

Listing 1: Master Server-Side RMI connection Setup

```
1 public void declareMaster(int serverId) throws RemoteException {
2     if (serverId == this.serverId) {
3         this.isMaster = true;
4         Registry registry = LocateRegistry.createRegistry(1099);
5         registry.rebind(MASTER_NAME, this);
6         System.out.println("Server " + this.serverId + " is now the master.");
7 
8         [...]
```

The client, upon starting, locates the Master server using the registry and obtains a reference to the remote object, which it then uses to invoke methods.

Listing 2: Client-Side RMI connection Setup

```
public class ClientRMISetup {
    ServerLogic serverLogic;
    public ClientRMISetup(String address, int port) throws RemoteException,
        NotBoundException {
        Registry registry = LocateRegistry.getRegistry(address, port);
        ServerInterface  server = (ServerInterface) registry.lookup("MasterServer"
            );
        this.serverLogic = server.getServerLogic();
    }
    public ServerLogic getServer() {
        return serverLogic;
    }
}
```

This approach makes it easy to extend the system with additional features or remote services, as new methods can be added to the server's remote interfaces without disrupting the client-side functionality.

RMI also handles object serialization, which is crucial for transferring complex data structures, such as tournament data or match results, between the client and the server. This allows the system to exchange large amounts of data in a consistent and reliable manner.

Although not directly implemented in this project, RMI allows the creation of security mechanisms that enable the server to restrict access to certain methods or requests for specific clients. This feature provides the ability to control access to sensitive operations, ensuring that only authorized clients can perform certain actions, thereby enhancing the overall security of the system.

4.2 Webhooks

Communication between the master server and the clients is facilitated through the use of webhooks. Whenever a client sends a request (using RMI) that causes the server to update the state of a tournament (for example, registering a player or creating a new match), the server triggers a webhook notification to inform all connected clients about the update. Upon receiving this notification, each client updates its view accordingly to reflect the latest changes.

When the master server is started, it launches a dedicated webhook server that listens for incoming requests from clients.

Listing 3: Client-Side Webhook connection Setup

```
public class WebHookServer {
    private static final List<String> registeredClients = new CopyOnWriteArrayList
        <>();
    public WebHookServer() {
        port(8080);
        post("/register", (request, response) -> {
            String clientUrl = request.body();
            registeredClients.add(clientUrl);
            return "Client registered: " + clientUrl;
        });
    }
}
```

Similarly, when a client starts, it searches for the server by sending a registration request to the webhook server.

Listing 4: Client-Side Webhook connection Setup

```
public class WebHookClient {
    private ClientController clientController;
    public WebHookClient(String localAddress, String serverAddress){
        [...]
    }

    private static void registerWithServer(String serverUrl, String clientUrl) {
        try {
            URL url = new URL(serverUrl);
            HttpURLConnection connection = (HttpURLConnection) url.openConnection
                ();
            connection.setRequestMethod("POST");
            connection.setRequestProperty("Content-Type", "application/json");
            connection.setDoOutput(true);
            try (OutputStream os = connection.getOutputStream()) {
                byte[] input = clientUrl.getBytes(StandardCharsets.UTF_8);
                os.write(input, 0, input.length);
                connection.getResponseCode();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Each time a client registers, the server stores the client's address in memory so it can send updates to that specific client when necessary.

There are four different types of notification that the server can send to clients:

- **tournament:** Sent after a new tournament is created.
- **player:** Sent when a player joins a tournament.
- **match:** Sent when a match is created within a tournament.
- **points:** Sent when a match ends and the points are recorded in the system.

Once the server sends a notification to all connected clients, each client evaluates the notification and updates its local view accordingly.

The webhook system is **not the primary connection** between the client and the server; rather, it is used exclusively for real-time notifications. The decision to use webhooks instead of continuing with Java RMI for this solution was driven by several factors.

Webhooks are much more scalable and can handle a larger number of simultaneous users. In addition, they allow notifications to be sent rapidly and frequently, even multiple times per second, if needed. This is critical for real-time updates, as in the case of a tournament system, where timely and efficient communication is essential to maintain up-to-date information across all clients.

4.3 Project Application

The project is implemented in Java using the following technologies:

- Gradle to manage dependencies
- Gson to serialize and deserialize data in the JSON format
- Java RMI for the client-server main connection
- Webhooks for the server notifications to the clients

4.3.1 Build Automation / Dependency Management

This project leverage Gradle for build automation and dependency management.

4.3.2 Architecture Implementation

The diagram in Figure 11 shows how the main application architecture is built, closely following the design outlined in Figure 2.

An MVC architecture was used for both the client and the server.
For the client, the model, view, and controller were all implemented, allowing for proper separation of concerns.
However, since the server does not require a graphical user interface (GUI), there was no need to implement the view and controller for the server.
The server logic communicates directly with the database logic.

Client Architecture:

- **View:** These components manage the data and user input through the GUI. Each component of the organization has its own dedicated View.
- **Controller:** These interact with the View to translate user input into actions and send requests to the Server. Each menu has a single generalized controller.
- **Model:** These represent the tournament structure, providing methods to modify it in a controlled manner. These components focus on contacting the remote server and invoking its methods.
- **RMI:** Components to manage the Remote Method Invocation Client-side.
- **Webhook:** Components to manage the Webhook registration and updates for the Client.

Server Architecture:

- **Database:** Components necessary to connect and manage the Database Comunication.
- **Model (Logic):** The main logic of the entire system: handles remote invocations from clients and updates the state of the tournaments if the requests are valid.
- **RMI:** Components to manage the Remote Method Invocation Server-side, the Master election process and the Master monitoring system from the Node Servers.
- **Webhook:** Components to manage the Webhook notification system from the Server.

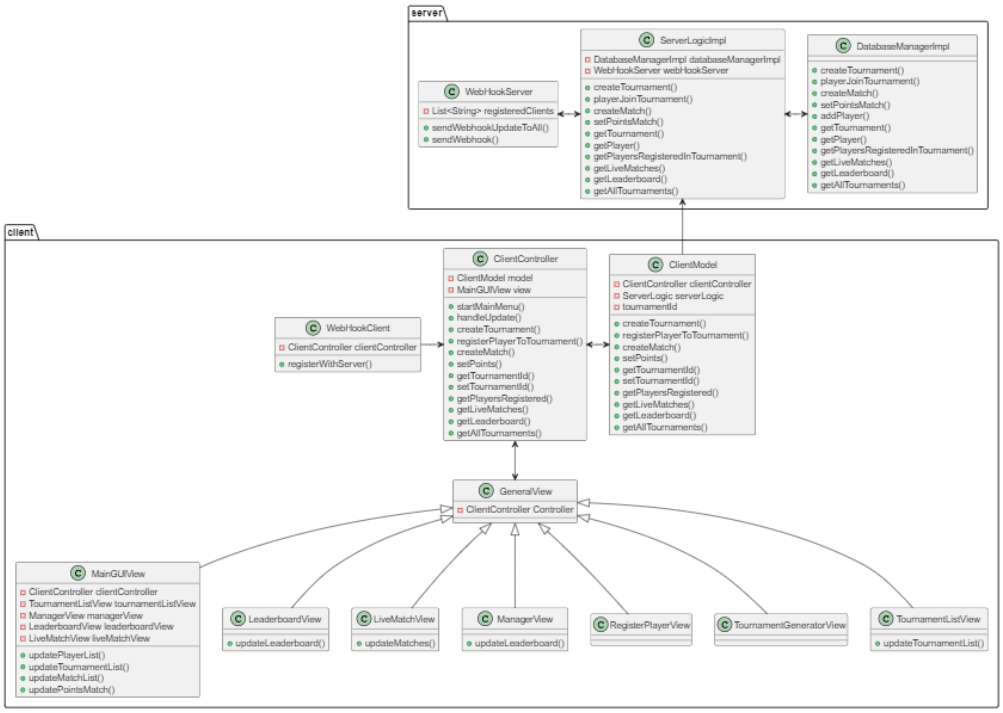

Figure 11: UML Class diagram of the architecture implementation

5 Self-assessment / Validation

The entire system underwent thorough self-assessment checks during and after its implementation to ensure its proper functionality. As outlined in the self-assessment process, both quality and effectiveness were evaluated when validating the system.

Regarding quality, the system meets all the specified non-functional requirements. The application functions seamlessly and operates efficiently in a client-server environment. It does so without requiring additional configuration (besides Server setup), particularly for network settings.
The Graphical User Interface (GUI) is responsive, providing user feedback within a few milliseconds and always within one second of user input.
All implementation requirements were successfully met, and the technologies identified during the design phase were effectively utilized.

In terms of effectiveness, all functional requirements were verified and fulfilled. Manual integration tests were performed to check the system features and that everything is managed correctly.
The following scenarios were tested:

- Joining a tournament in which the player has already joined.
- Disconnecting an organization component while a tournament is running.
- Joining a tournament after some sets have started
- Check the leaderboard while some matches are completed.
- Disconnecting a Master server while a tournament is running.

For critical components, automated tests were implemented using *JUnit Jupiter* to streamline the testing process and validate the code under various scenarios.

6 Deployment Instructions

Below are the instructions for setting up the entire application to fully utilize the system's functionalities.

1. Build the Docker image for the database using:
   - docker build -t mysqldatabase .
2. Start the docker container
   - docker run -d -p 3306:3306 –name mysql-container mysqldatabase
3. Start the Servers using the Main present in **server/main/StartServer.java**
4. Start the Clients using the Main present in **client/main/StartMainClient.java**

7 Usage Examples

Upon launching the application, users are presented with a main menu that displays all available functionalities, providing quick and easy access to each feature offered by the system. This menu serves as the starting point for navigating the features of the application.

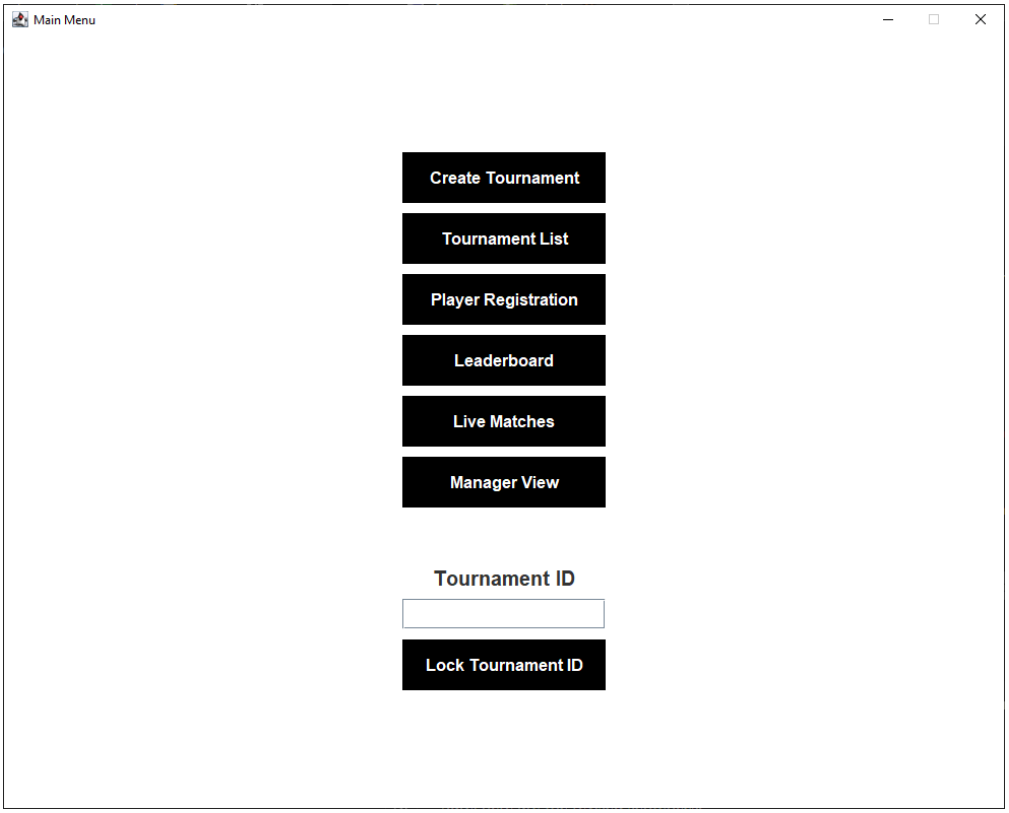

Figure 12: Main Menu Page

Creating a New Tournament:
This screen allows admins to create a new tournament, defining its parameters.

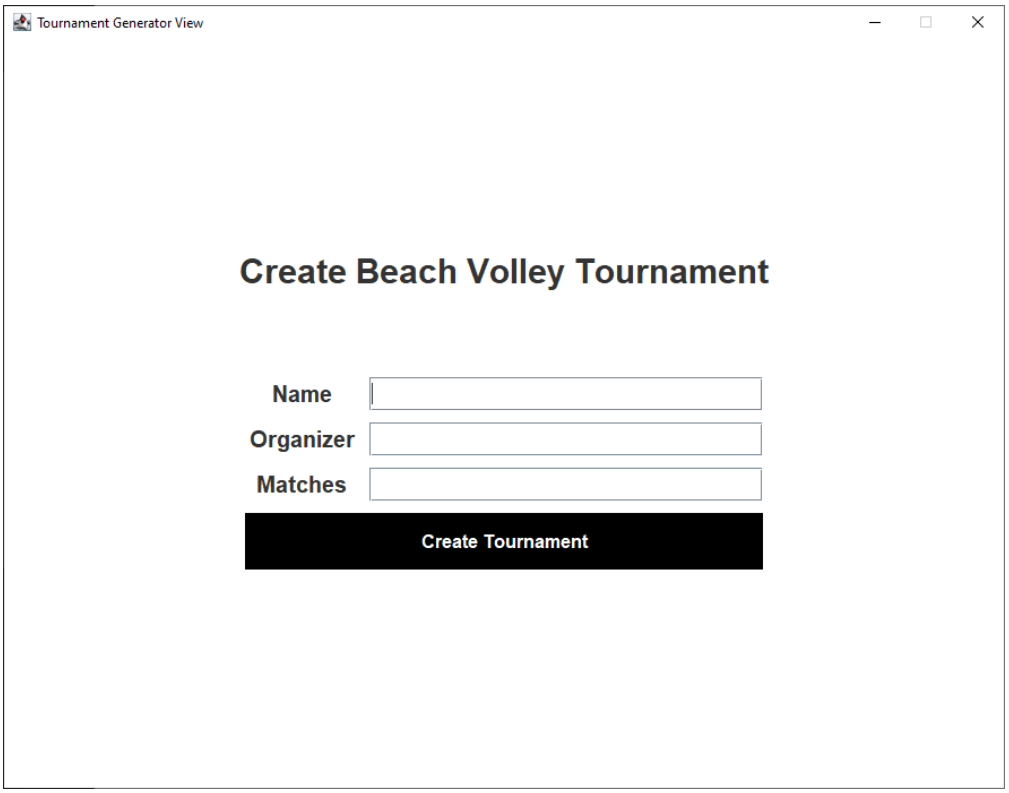

Figure 13: Tournament Generator page

Joining a Tournament:
On this page, secretaries can add players to an existing tournament.

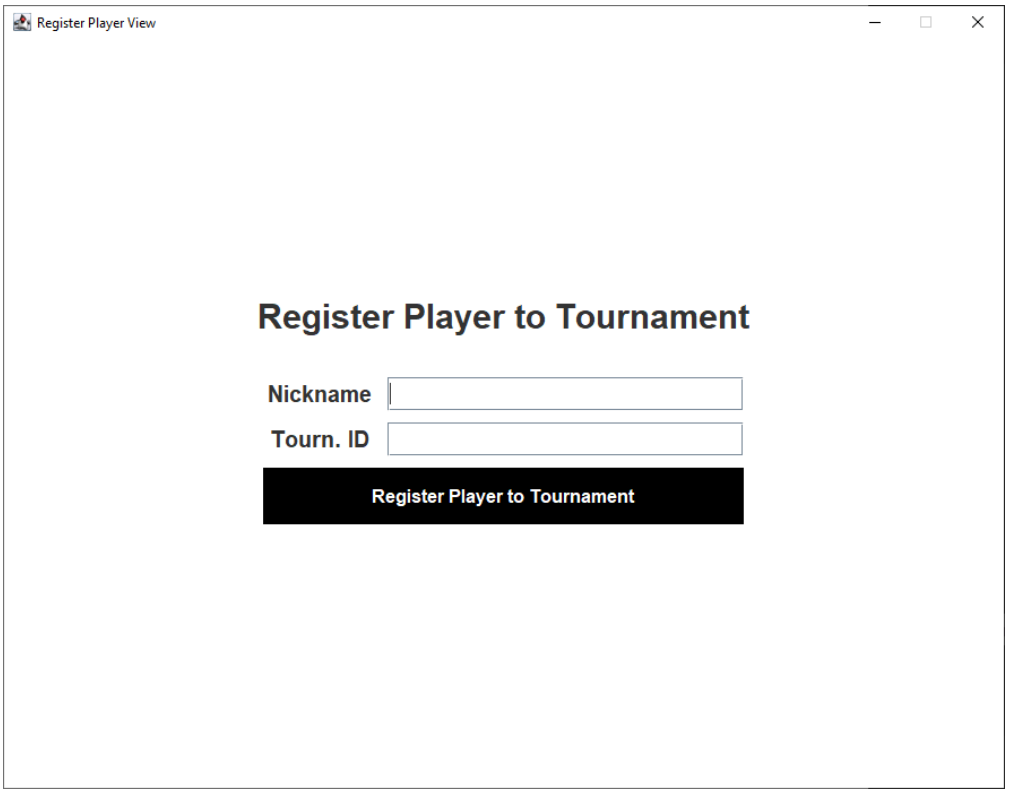

Figure 14: Player Registration page

Viewing the Leaderboard:
This view displays the current leaderboard, showing player rankings and match statistics.

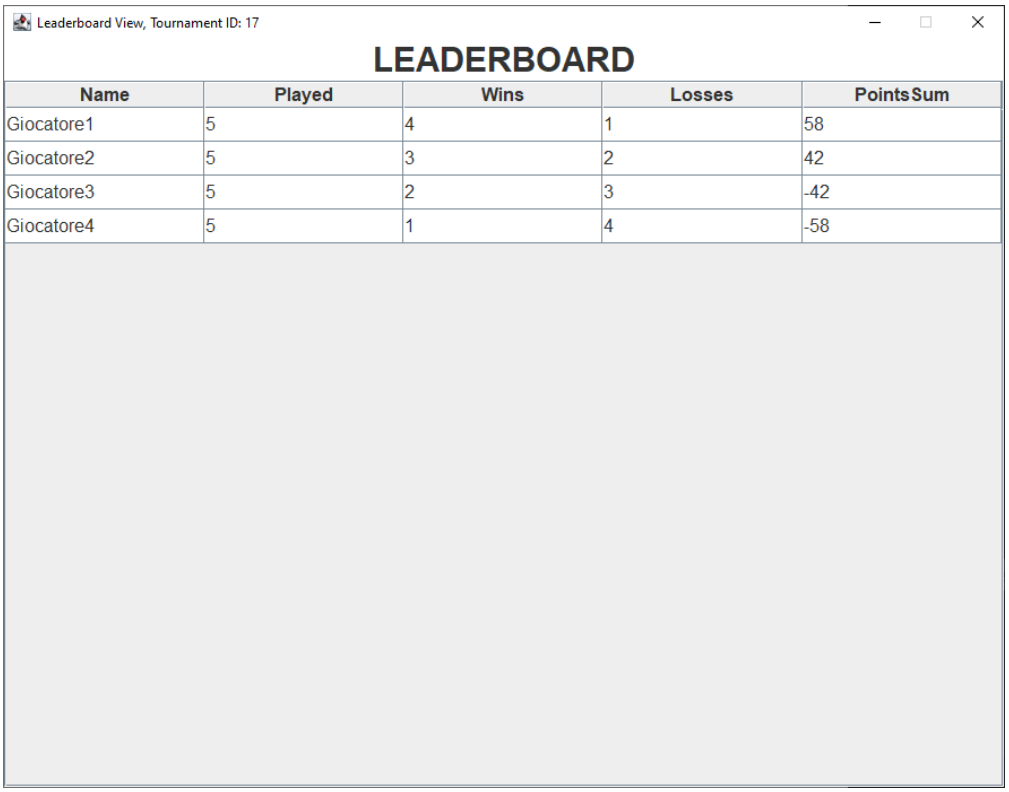

Figure 15: Leaderboard Page

Starting a Match:
From this screen, an admin can initiate a new match within the selected tournament.

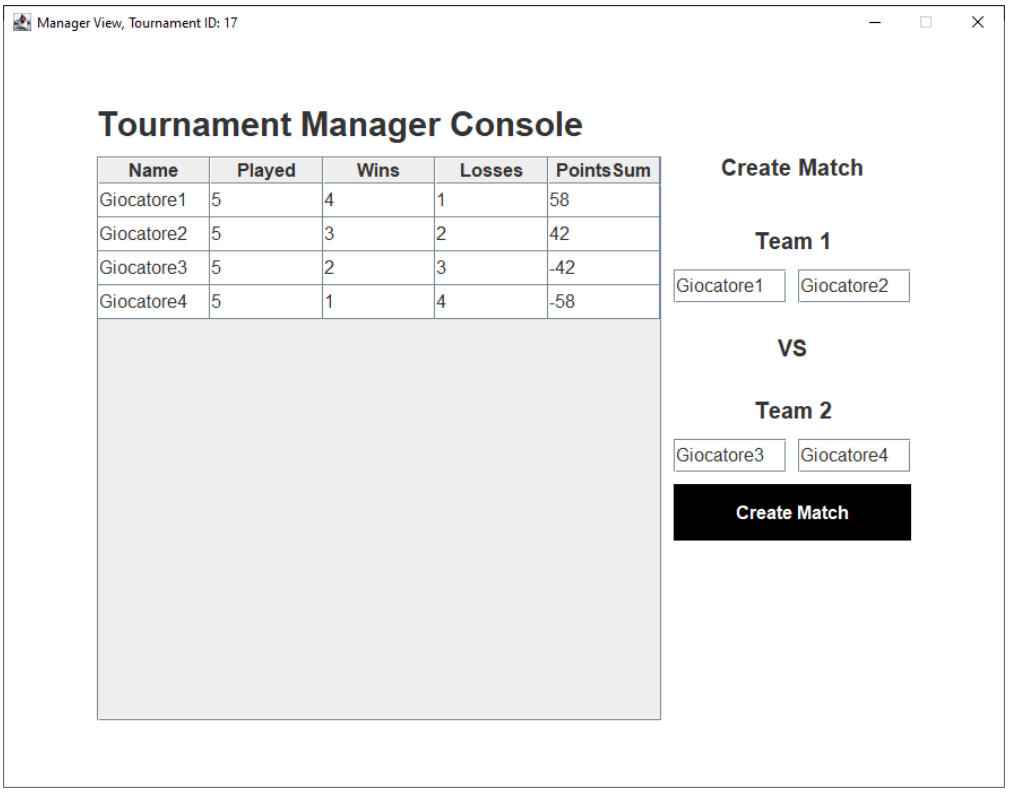

Figure 16: Manager View Page

Recording Match Results:
This view allows referees to input and confirm the results of live matches, updating the tournament standings.

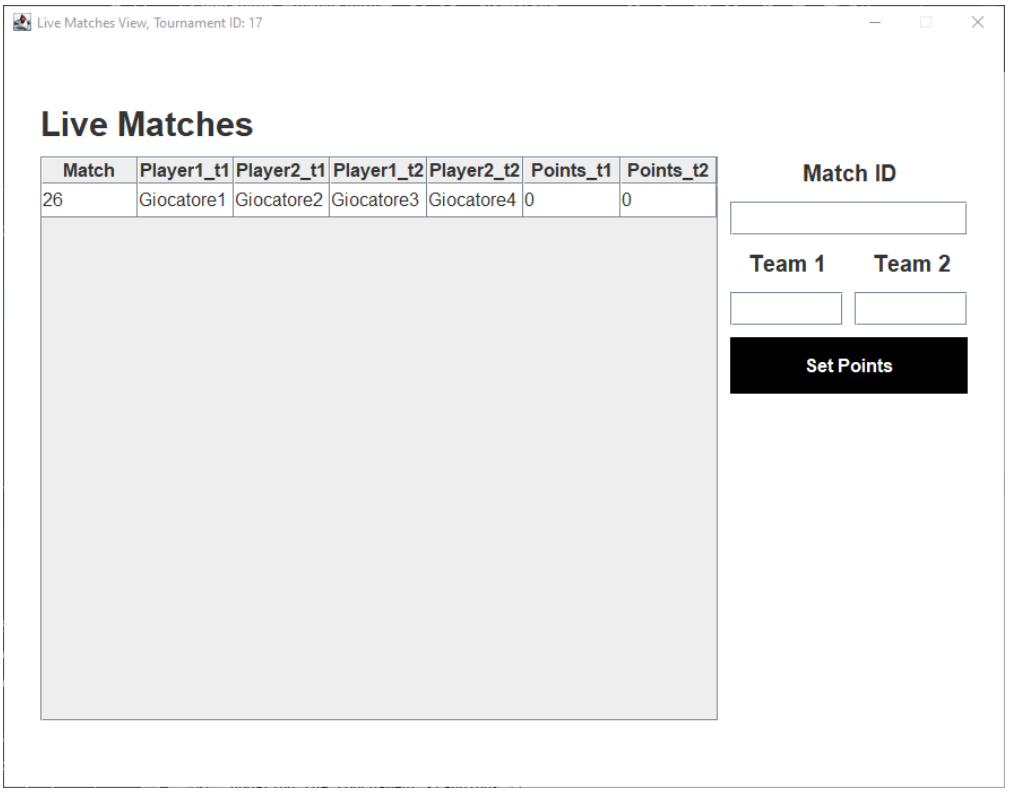

Figure 17: Live Matches Page

8 Conclusions

In this project, a fully functional distributed system was developed, enabling real-time communication between different clients while managing tournament data with a shared status.

Although the project focuses on a beach volleyball tournament management system, the concepts and technologies used (such as the structure of the client-server project and communication) can be easily applied to other projects in various domains, allowing for scalability and efficient handling of multiple users and requests.

8.1 Future Works

Although the project met its main goals, there are several areas that could be improved in the future. An aspect not explored in this project is a detailed performance test when a large number of clients are connected at the same time. This test could provide useful information about the system's ability to handle many users and its performance under heavy load.

In the future, to avoid problems with a "*single point of failure*", it would be helpful to implement several synchronized databases. These databases would handle requests while keeping the data consistent, and the servers would manage multiple database instances to ensure the system remains available and reliable.

8.2 What did I learn

The development of this project provided valuable insights into the principles of distributed systems, particularly in managing communication between multiple clients and servers. I gained a deeper understanding of how different components interact within a distributed environment and how to ensure seamless communication across them. Specifically, I learned how technologies like webhooks for server-to-client notifications and RMI for remote method invocation can significantly improve interactions between clients and servers.
Additionally, the project highlighted the importance of choosing the right communication patterns to optimize performance and maintain system reliability.