Multiplayer Minesweeper

Fabio Muratori
fabio.muratori2@studio.unibo.it

March 2023

Minesweeper[1], known in Italy as *Campo minato*, is a classic single-player puzzle game developed in the 1990s for the Windows 3.1 operating system.

The game is set on a grid, where players have to uncover squares to reveal whether or not there is a hidden mine underneath. The player's objective is to clear the entire board without detonating any mines. When the game begins, players are presented with a grid of squares, some of which contain hidden mines. The number of mines on the grid depends on the difficulty level chosen by the player. The objective is to uncover all the squares on the grid that do not contain mines. Each square that a player selects reveals a number that indicates how many mines are adjacent to that square. The player must use this information to determine the locations of the mines and avoid uncovering them. If a player uncovers a square with a mine, the game is over, and the player loses. However, if the player successfully uncovers all squares without detonating any mines, they win the game.

Minesweeper quickly became a popular game, and was shipped with most versions of the Windows operating system. It has since been ported to various other platforms and remains a beloved classic among players.

This project aims to develop a version of the game that enables multiple players to collaborate and play in the same match.

Contents

1 Goal/Requirements

The main objective of this project is to implement a distributed system that enables multiple users to play Minesweeper remotely on the same game. The system will consist of three different services that allow users to access the application through a web page, manage game queues, and active game instances. Each of these services will be provided by a specialized server that can communicate with both other servers and the end-user in various ways.

1.1 Requirements

The functional requirements of the system can be summarized as follows:

- users can create a match and wait for other players to join;
- players can see all the available open sessions where other players are waiting to start a game;
- multiple users can play a single match, and at any moment during the game, everyone can see the same game board;
- multiple games and waiting rooms may be active simultaneously, but players can participate in only one game at a time;
- during the creation of a game instance, a player can select one of multiple game modes, each defined by a different map size, number of mines, and maximum number of players;
- players can exit a waiting room or game at any time;
- users cannot join a session or game that is already full or a game that is already in progress.

1.2 Scenarios

The expected use of the application follows approximately the steps listed below:

1. a user opens the web page and is able to view all the available game sessions and a simple form for the creation of a new session;
2. when a user enters a session, if the required number of players are connected, the game can start and all players will be redirected to the game board;
3. once the game board is visible, all players can interact with the map of tiles, and any action taken is propagated to all other players;
4. possible tile interactions include visiting to explore a tile and flagging to signal and keep track of possible mine positions;

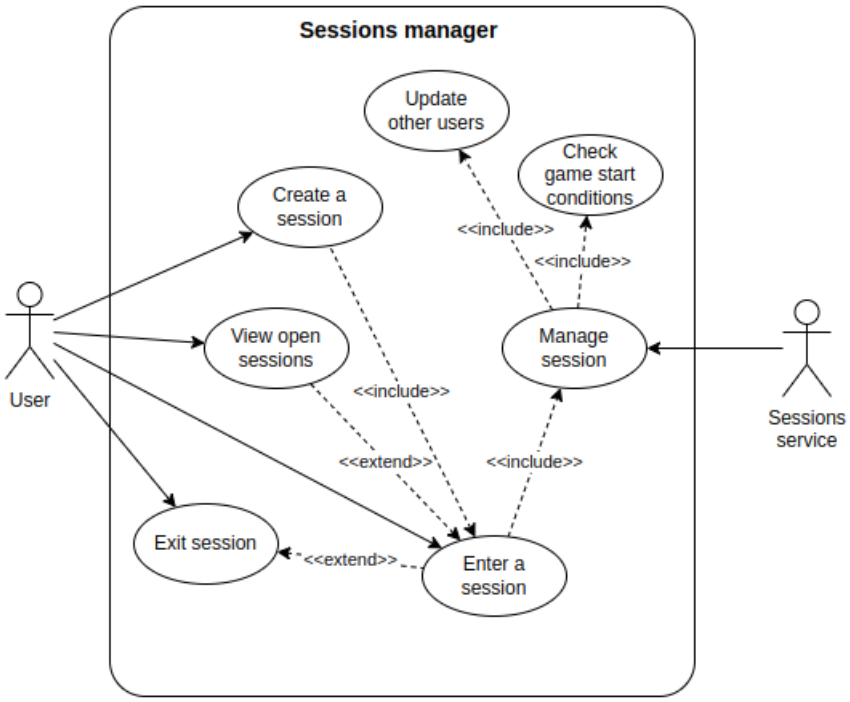

Figure 1: UML Use case diagram of features provided by a session service entity.

5. the game continues until one of three conditions is met: a player triggers a mine, all the tiles without any mine are identified, all the players exit the match;
6. when the game ends, all the players are redirected automatically to the main page.

Here are listed other behaviours the system must implement:

- if a mine is triggered by the first move of the game, the mine is moved to a random position;
- at any time, any player should be able to disconnect from a session or a running game without affecting other players;
- if a player exits a game, he won't be able to re-enter it but if a player exits a waiting room, he can still re-enter if the game has not yet started;
- any player can click on a tile that has already been visited with no effect.

The UML use case diagrams in figure 1 and figure 2 show all the describer functionalities and also identifies a possible subdivision of the solution into two services.

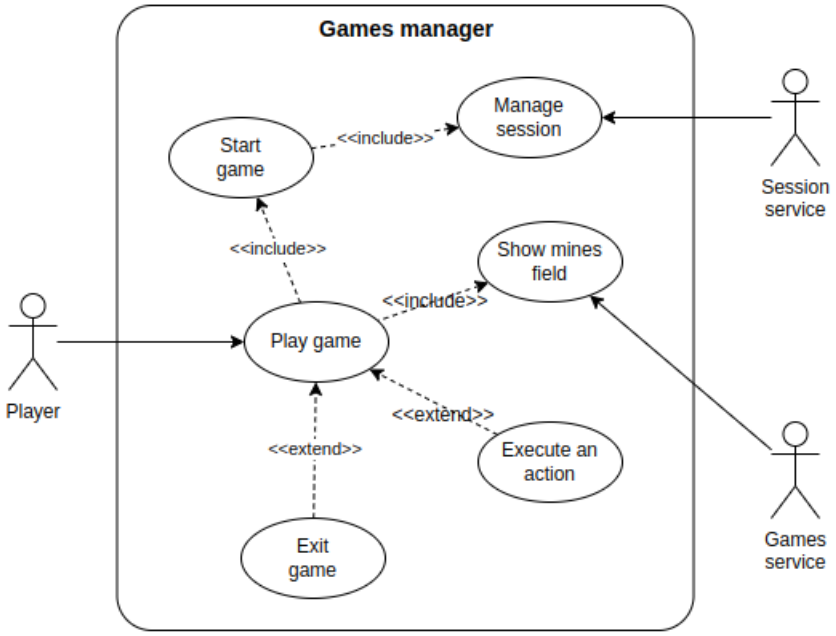

Figure 2: UML Use case diagram of features provided by a game service entity.

1.3 Self-assessment policy

Being the final product composed of different independent services (front-end, sessions management and game management), the software quality is evaluated by means of automated tests and manual tests spanning the entire development process. Automated tests are required for all three components.

Some of the aspects of the system are tested through manual experiments, especially for the integration of the three components. Also, testing of the front-end web page of the application won't be automated but instead executed manually.

Docker will be used during the developmental stage to ensure a standardized testing environment especially for the automated tests and identify potential configuration issues.

1.4 Learning goals

The learning goal of this project is to gain experience and understanding of distributed systems and environments, identify and attempt to solve some of the challenges this topic poses. Shortly, the learning goals may be simplified to:

- gain understanding and implement a distributed system;
- gain understanding and implement a micro-service oriented architecture where different modules are integrated to achieve the desired requirements;

- integrate various web-based technologies and programming languages to enable real-time user interaction;
- practice with CI/CD for the automation and streamlining of the process of building, testing, and deploying of software;
- practice with Docker to realize a consistent testing environment and for easier portability.

2 Requirements Analysis

Based on the necessary requirements and the desire to obtain effective and quality software, it is necessary that, except for fatal errors, both the client and the server are able to resolve their discrepancies without direct user intervention.

We therefore want to have a system that independently resolves errors caused by delays, packet loss or connection, without the result being the application crash. Moreover, the system must be able to handle situations where one or multiple players become unreachable during a game or while waiting for a game to start. Therefore, connections and messages handlers will be introduced to manage situations where users or the game servers become unreachable.

In order to reach the conclusion of a game, various data are sent and received by various components of the system via dedicated channels, therefore, applications must be able to convert user actions into objects in JSON format to be able to serialize them on send and de-serialize them upon receipt.

Furthermore, given that each server will have to be able to handle hundreds of user connections from all over the world, the workload on a single server will have to be as minimal as possible and allowing clients to update their own interfaces based on the new game data sent.

Lastly, multiple users participating in the same game should see the same board at any given moment, the state of the game must be shared across the network. Another kind of shared state across multiple running applications is found in the necessity to show active sessions and update sessions state.

2.1 Assumptions

A user is able to open multiple tabs on the same browser. The expected behaviour should be to mimic the same application status across multiple tabs. The capacity to implement such a feature relies on the implementation of the libraries used in developing the web application and its communication channels. To lower complexity it has been decided to ignore these kind of interactions and it is assumed that a user to open only one tab inside the same browser window. During the test phase, multiple browser windows will be opened to simulate multiple user connections.

To interact with the web application, users do not need to log-in. Furthermore, no data is kept after the end of a game and users are solely identified by the unique connection channels they make with the back-end services. For this reason, and to allow easier human testing, during the creation of a session, users must define the name of the session and the desired game mode.

The requirements do not impose any constrain on the possibility to join a running game and what should happen if a session or a game is empty. For the purpose of this project, whenever a session contains no player, it remains open while an empty game will be deleted since it cannot reach a conclusion. Moreover a running game cannot be joined by players who were not previously in the waiting room.

3 Design

The overall system architecture follows a centralized approach over a decentralized approach. A centralized approach in a distributed software system entails all processing and data management being performed by a single server, whereas a decentralized approach requires distributing these tasks across multiple nodes in the network.

The advantages of a centralized approach include easier management and maintenance, as well as faster and more efficient communication because of the presence of a single point of contact. However, the centralized approach is vulnerable to single points of failure, and the capacity for scalability is limited by the central server.

The benefits of a decentralized approach include greater availability due to the absence of a single point of failure, and greater scalability through the addition of new nodes. However, decentralized systems are more complex to manage and maintain, and communication can be slower because of multiple points of contact.

3.1 Microservice architecture

Microservice architecture is a software development approach where a monolithic application of significant size is broken down into smaller, independent services that communicate with each other through APIs. Each service is responsible for a specific functionality, and it can be deployed, updated, and scaled independently from the other services in the application.

The microservice architecture is built on the principles of modularity, loose coupling, and high cohesion. By breaking down the application into smaller services, each service can be developed, tested, and deployed faster, without affecting the other services in the application. This allows for more flexibility and agility in software development, as well as improved scalability, fault tolerance, and resilience.

The microservice architecture also allows for better resource utilization, as each service can be scaled independently based on its specific requirements, rather than scaling the entire application. This improves performance and reduces infrastructure costs of the application.

While microservice architecture comes with several advantages, it also has some drawbacks compared to a monolithic application. One of the main challenges of microservices is the increased complexity of designing, developing, and deploying multiple services that must seamlessly work together. This complexity can result in more difficult testing and debugging, and multiple points of failure, as a failure in one service can potentially affect other services and the overall performance of the application.

Considering the required features proposed in Section 1, we can isolate three different aspects of the architecture, each of which can solve one set of requirements:

- *front-end service* - a front-end that houses the representation layer of the application and is needed to supply the application to the users;
- *sessions service* - a session manager is needed to handle the life cycle of multiple waiting rooms and helps in the division of the game and the match making

dimensions;

- *games service* - a game manager is required to manage all the active games and participating players.

The Figure 3 shows the division of those dimensions and the main interactions between them.

The proposed structure splits the game from the pre-game phase, increasing the complexity of the interactions and opening up the possibility of new points of failure. However, this separation allows for a more modularized architecture and possible future evolutions where multiple game modes or waiting room policies are available and managed independently. It should also be noted that those dimensions require different levels of scalability, which can be achieved with the proposed architecture if we consider the difference in computational complexity between managing a game and managing the waiting room.

The front-end provider service is also decoupled from the other two components of the system. The logic behind this approach is to enable easier management and scaling of the front-end application independently from the back-end services and to allow for a more modular and reusable front-end component.

3.2 Structure

3.2.1 Front-end web application

The structure of the front-end web application closely follows the structure suggested in Figure 3. Figure 4 presents a basic representation of the pages that compose the interface and the transitions between them. The image shows three different panels:

- the sessions list page displays the active sessions that a user can join and a form for creating a custom session for them and other users to join;
- the waiting room page displays a loading screen with an indicator representing the number of users waiting, which can be accessed when the user joins a session;
- he game page displays the mines grid and other game statistics.

Comparing Figures 3 and 4, we can see that the *Sessions list page* defined in the second figure is not directly visible in the microservices partitioning of the entire system. The sessions service, which manages waiting rooms, contains all the necessary data required by the first two screens of the web application. This has implications for managing communication channels between the browser and the sessions service to ensure that the two activities managed by the same service remain decoupled. Specifically, we need to show an updated view of available sessions and manage waiting rooms without these activities interfering with each other. Section 4.1 explores a solution to this problem.

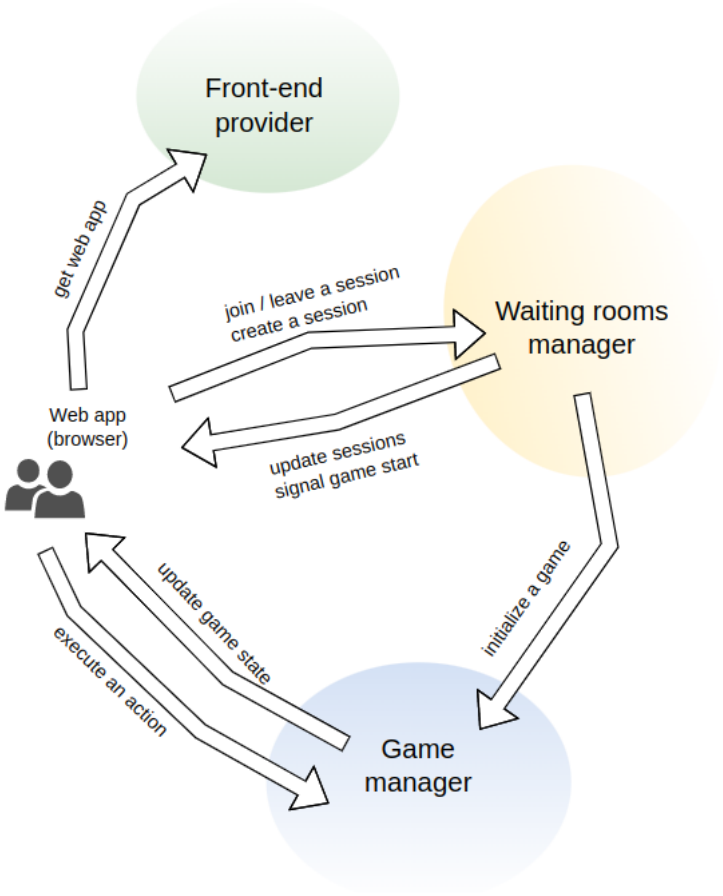

Figure 3: General division of the system into separate services.

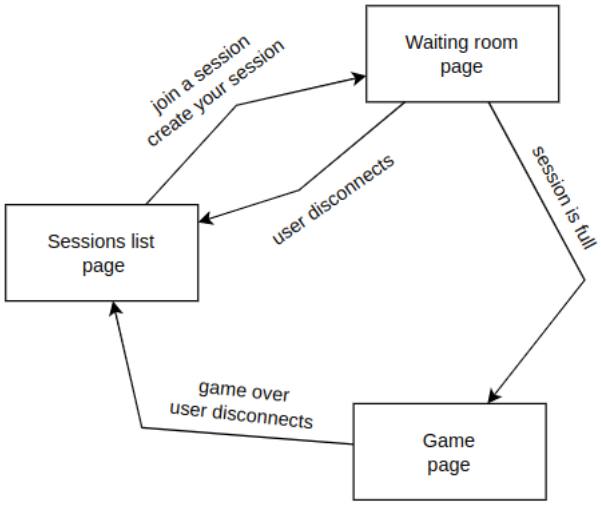

Figure 4: Basic front-end views and transitions between them.

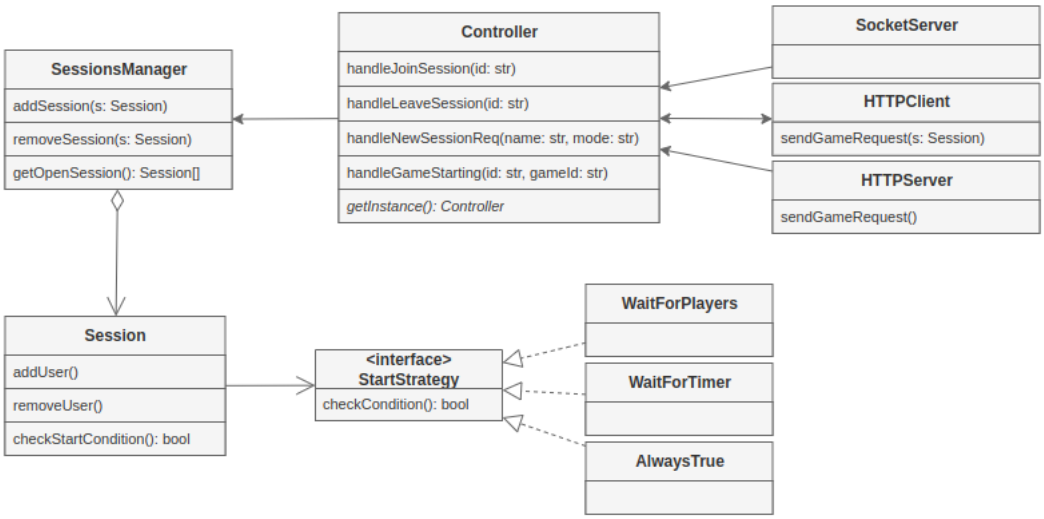

Figure 5: UML Class diagram of the session management microservice.

3.2.2 Sessions service

This component handles the creation and subscribing of players to a session. Whenever some starting conditions are met, the sessions service handles the creation of a game instance by contacting the game service, and as soon as a game is ready, the session manager communicates to the players the identifier for that particular game.

A *Controller* handles all the requests received and actions forwarded from and to the communication channels. Each message received is translated to the *SessionsManager*, which holds the reference to all the active sessions,sessions represented by the class *Session*. A session is then characterized by the list of players connected to it and a particular *StartStrategy*. The logic behind a game starting condition is defined by using a strategy design pattern, and in Figure 5, three examples are provided:

- with *WaitForPlayers*, a game is started only when the maximum number of players is connected to the session;
- with *WaitForTimer*, a game is started only when at least one player is connected and a certain amount of time has elapsed;
- with *AlwaysTrue*, a game may start as soon as the check is executed.

This last behaviour is used, for example, when a single player game mode is selected. Once the player who created the session joins the waiting room, the game starts immediately without checking any other conditions.

Three different classes (*SocketServer*, *HTTPClient*, and *HTTPServer*) are dedicated to managing communication channels between this and other services and the end-user. More information about this implementation can be found in Section 3.4.

3.2.3 Games service

The structure of the game service resembles the architecture of the sessions service where some communication channels and a *Controller* class are defined. Two different classes (*SocketServer* and *HTTPServer*) are dedicated to the management of communication channels between this and other services and the final user. More information about this implementation is explored in 3.4.

A *Controller* object also demands the implementation of the management of multiple game instances to the *GamesManager* object. The expected behaviour and logic behind a single running game is then represented by a *Game* interface. The *toString* method returns the representation of the game board. With this approach entirely different games could be implemented. To handle the different game modes required, the *Controller* class uses a simple *factory design patter* which provides default configurations.

In details, a Minesweeper game is characterized by a rectangular board of tiles. At any moment in time a *Tile* object holds two distinct features: a state that tells us if a tile has been interacted with by a player, and a content, what the specific tile contains. Each action requested, represented by the enumeration *ActionType*, produces an outcome represented by the enumeration *ActionResult*.

3.3 Behaviour

Figures 7, 8, and 9 present the UML state machines that represent the expected behavior of the three main aspects of the architecture:

- Management of active sessions.
- Management of a single waiting room.
- Management of a single game instance.

These figures illustrate the expected behavior from the end-user interface's point of view. In fact, some of the consequences of actions are directly linked to the structure of the front-end web application, which is explored in Section 3.2.1.

When the user enters one of those pages, a connection is attempted with the appropriate back-end service. The main behavior is initialized only after receiving a successful connection. This approach forces a robust solution with particular emphasis on network-related errors. In most cases, as shown in Figures 7 and 8, reestablishing the connection is attempted as soon as it is lost. However, in Figure 9, which represents the behavior of a specific user interaction with the game back-end, a sudden disconnection from a game is treated as a "catastrophic failure" by exiting the current game and redirecting the player to the initial page.

As explained in Section 1, once a game starts and all initial players are connected, no new connections can be established. This is because connections are defined by an unequivocal bidirectional and stateful channel, and the system cannot differentiate between an old player and a new external one attempting to connect. When a player refreshes the web page or loses their connection temporarily, a new channel is opened,

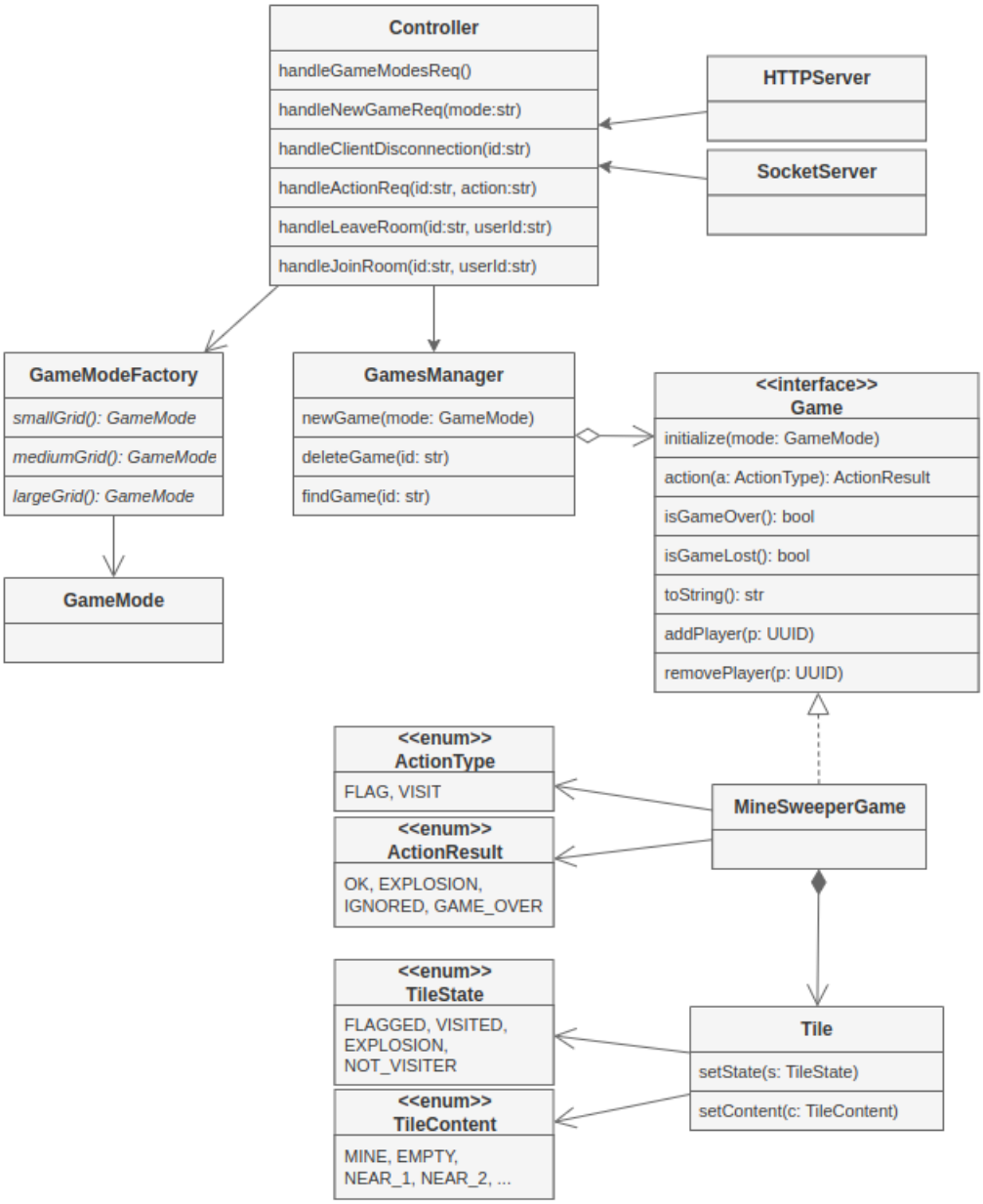

Figure 6: UML Class diagram of the game management microservice.

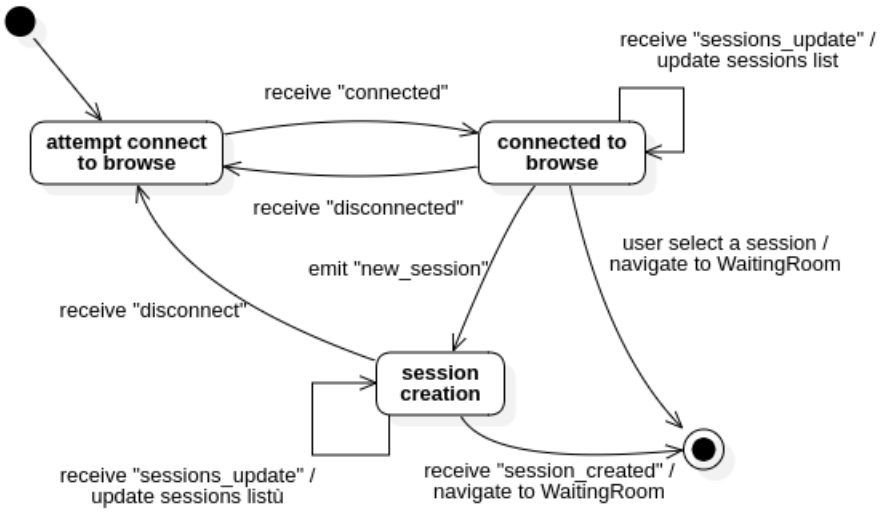

Figure 7: UML State diagram of the sessions list interface behaviour.

and the previous identifying code is lost. In situations where a re-connection mechanism must be implemented, alternative means of user identification should be used, which are not covered in this project.

3.4 Interaction

In a microservice-oriented architecture, each service is designed to be autonomous and independent, which means that it must communicate with other services to perform its functionality. However, managing communication between microservices can introduce complexity, as each service may use a different communication protocol, data format, or interface.

Moreover, in a distributed system, there may be network packets latency and data loss, which can cause delays in communication and affect the overall performance of the application. Additionally, services may need to handle different levels of traffic, which can result in scalability issues if not managed properly.

To manage these complexities, microservices must communicate through a well-defined API, which specifies the interface, protocols, and data formats used for communication. APIs can help ensure that services can communicate effectively and efficiently, while also providing a level of abstraction that can make it easier to update or replace services without affecting the rest of the system.

In the context of this project, different approaches are needed to manage the communication between the end user's web browser and the microservices described. A channel is needed for instance during the request of an action by some player and the following update of the new game state to all the other players. In this case the required channel must be bi-directional and state-full since both the player and the game manager must be able to send data independently and selectively (only the users playing the same game

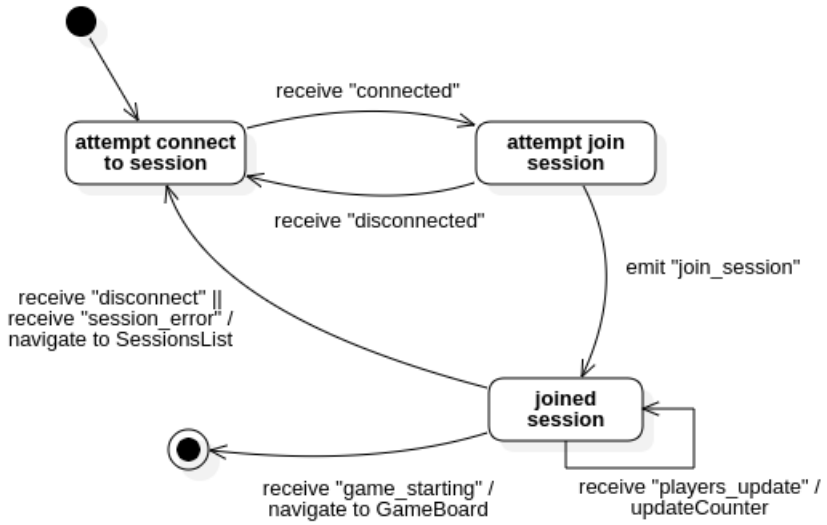

Figure 8: UML State diagram of the waiting room interface behaviour.

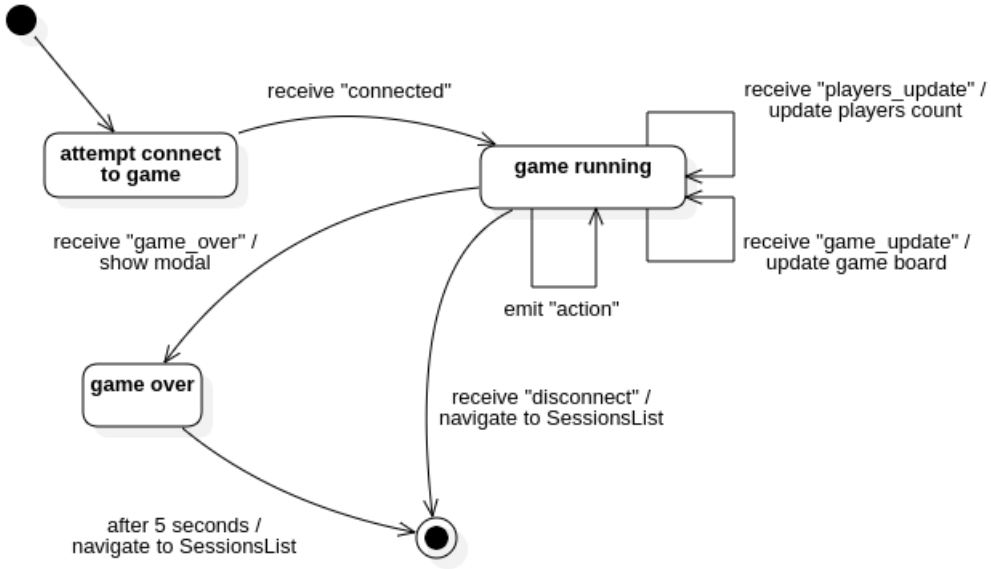

Figure 9: UML State diagram of the game interface behaviour.

must be updated with the state).

Another type of channel is used for instance during the first access to the application when an user wants to see all the available game modes or while the sessions service requests for a new game mode to be created. In these cases, the micro-services will never spontaneously send data but will always be contacted and respond accordingly. This is a typical request-response kind of communication where only one part will send a request and another will respond.

3.4.1 Game state management

Finally, we need to decide what information from a single game should be shared with players and determine the approach that best fits the requirements. During the design of the solution, two options were found:

- *Option 1* - The server sends a validated action, and the game state is defined as a sequence of these events;
- *Option 2* - The server receives an action, computes the new game state, and sends the latest updated game board.

Both options have their pros and cons. Option 1 allows for a more generalized server-side implementation and has low computational needs since the front-end web applications need only implement the game logic and compute the evolution of a game. On the other hand, option 2 seems to be more secure since each web browser does not have knowledge of the mine's location.

However, both options have issues in case a message is lost. In option 1, the loss of a message could lead to inconsistent game states. If the web app does not detect the loss of a message, it cannot recover the current configuration. Option 2 could also lead to inconsistencies, but each new message coming from the game service would unequivocally update the player on a correct game state.

Overall, the first solution seems to perform better, and it could be argued that most of the proposed problems could be solved. However, given the limited amount of time and the decrease in complexity, it has been decided to implement the second solution.

3.4.2 Sessions and game start

One of the most critical sequences of events are found during the creation of a game following the completion of a waiting room condition. The expected behavior follows the representation shown in Figure 10, the UML Activity Diagram. Basically, after the initial connection of a player to the waiting room, we expect the following sequence of actions:

1. the user attempts to join the session previously selected;
2. the session socket in the sessions service requests the resolution of the request;

3. the game starting condition is checked;

4. the player is informed of the successful connection by receiving the number of currently connected players.

If the condition checked at the third point of the previous list is met, the sessions service asks the games service to create a new game instance. Upon receipt of the game instance id, the sessions service locates the waiting room and updates all the user with a *"game_starting"* message. Then, the web app can update the interface by redirecting the players to a game screen. Interestingly, whenever the web application requests an action by emitting messages into the Socket.IO channel, the sender does not wait for a response. Whenever a response message is received, a new message handler is executed inside the front-end environment. Interestingly, whenever the web application emits messages into the Socket.IO channel requesting an action, the sender does not wait for a response message. When a response message is received, a new message listener is executed inside the front-end environment. The Socket.IO implementation uses an event-driven programming paradigm under the hood, in which the flow of the program is determined by events or messages rather than sequential instructions.

3.4.3 Game execution

When the waiting room is terminated due to the game start requirements being met, the sessions service sends a new game request to the games service via a POST REST API call. Figure 11 illustrates this process. The games service generates a unique `gameID` for the game, which is then linked to the specific game instance and propagated to the sessions service. The sessions service can then inform all users about the specific game identifier. Each client can create a Socket.IO channel with the game server, allowing them to request actions and receive updates about the game state.

Figures 12 and 13 present the sequence of activities that occur when a player joins a game instance and requests an action, respectively. Both activities are managed by the game service. The main difference between these figures is the execution of different tasks on a single *Game* object, as described in Figure 6.

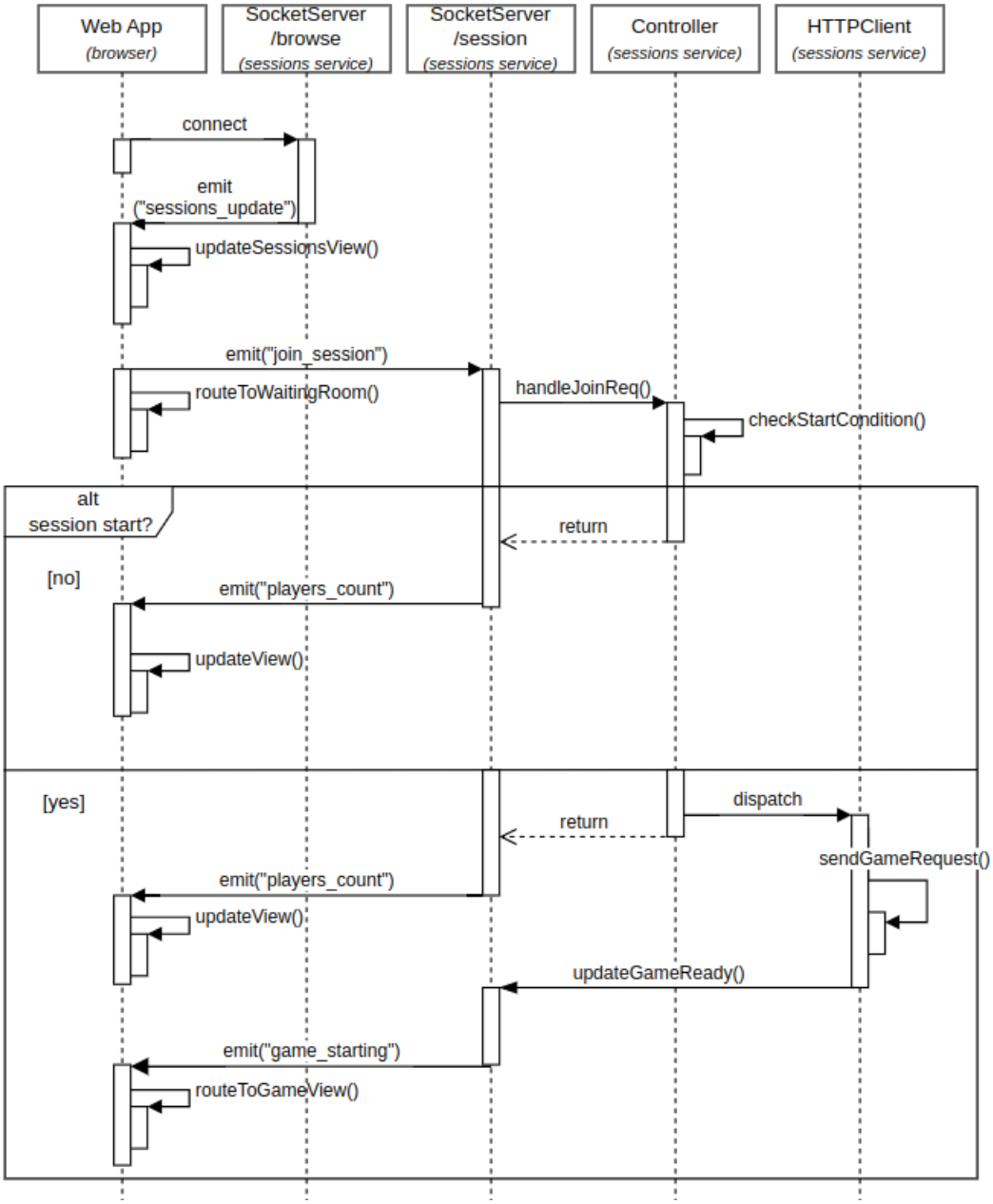

Figure 10: UML Activity diagram of the sequence of activities executed following the connection of a player to a waiting room.

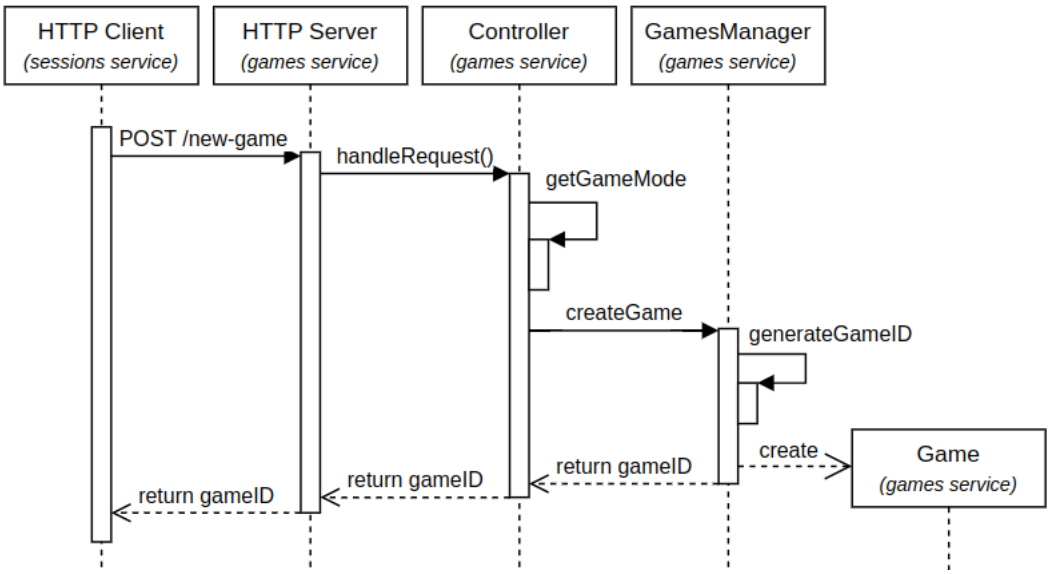

Figure 11: UML Activity diagram of the sequence of activities executed following the request of a new game instance.

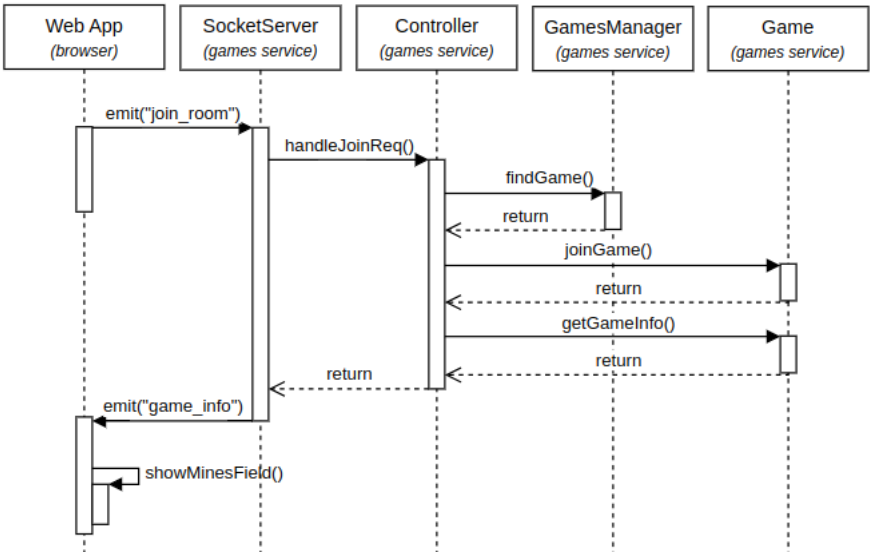

Figure 12: UML Activity diagram of the sequence of activities executed following the startup of a new game instance.

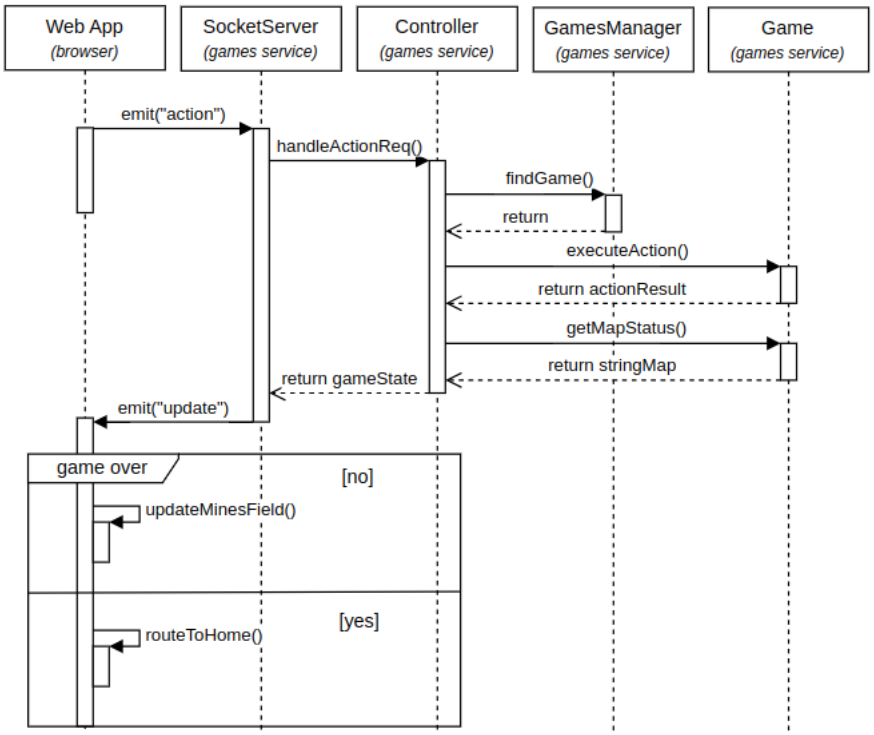

Figure 13: UML Activity diagram of the sequence of activities executed following the request of a new game action.

4 Implementation Details

The implementation of the three services introduced in Section 3.1 uses different programming languages, frameworks and technologies. For the front-end service we use:

- Node.js[14], a server-side runtime environment for executing JavaScript code outside of a web browser;
- npm[9], a package manager for Node.js that allows developers to easily install and manage third-party libraries and packages;
- JavaScript[7], a popular programming language used for building dynamic web applications that run in web browsers;
- TypeScript[15], a superset of JavaScript that adds static typing and other features to improve code maintainability and developer productivity;
- React[12], a popular JavaScript library for building user interfaces that enables developers to create reusable UI components.

For the sessions and games services we use:

- Java[10], a popular programming language that is class-based, object-oriented, and designed to run on any platform with a Java Virtual Machine (JVM);
- Gradle[5], a build automation tool widely used for building and managing projects in various programming languages;
- JUnit[6], a popular unit testing framework for Java that allows developers to write and run automated tests to ensure the quality of their code;
- Vert.X[3], a lightweight and high-performance framework for building reactive, event-driven applications.

To implement bi-directional and state-full communications between clients and servers Socket.IO[13] will be used, a library built on top of WebSocket[20], a low-level protocol that provides a persistent, bi-directional communication channel between a client and a server. Socket.IO provides additional features, such as support for fallback options, automatic re-connection, and room-based communication. Some of the features used in this project are explored in Section 4.1.

This project will use RESTful APIs [18], which use HTTP [16] methods to provide a uniform interface for accessing and manipulating resources over the web, and usually return data in a common format such as JSON [17].

4.1 Socket.IO features

Socket.IO implements the concept of *namespaces*, in which a single server can support multiple endpoints, each with its own pool of sockets connected under a given scope identified by a path name. As previously mentioned in Section 3.2.1, the session manager service is responsible for managing multiple waiting rooms and the state of each room, providing functionalities for both the sessions list screen and the waiting room screen. This feature can be used to manage the following scenarios:

- when the player is exploring open sessions, a dedicated `browse` endpoint provides updated state across all clients;
- when the player is inside a waiting room, they do not need to be updated on other sessions, and a dedicated `session` endpoint provides only the current session state changes.

Socket.IO also provides a *room* mechanism that allows for the grouping of different sockets into a single channel. This enables the server to broadcast messages directly without the need to contact every single user sequentially. This feature has been extensively used at various points in the system:

- to group players of a single waiting room session;
- to group players of a single game instance.

To clarify, the session service deploys both mechanisms to effectively target different sets of users: those who are on the main page and those who are waiting in a session. At no point in time does any user receive unnecessary data.

4.2 Handling concurrency

The term concurrency refers to the ability to handle multiple tasks or requests simultaneously. This project requires the handling of simultaneous access of shared data for:

- the management of multiple sessions inside the sessions service;
- the management of multiple games inside the games service.

In order to manage concurrency, servers use various techniques such as multi-threading or event-driven programming. These techniques allow servers to process multiple requests at the same time, improving their overall performance and responsiveness. The implemented distributed services provide features by means of REST APIs and Socket.IO channels and each of those gates must be able handle parallel execution of requests.

Both Socket.IO and Vert.X use the same approach to handle concurrency, by means of an event driven architecture, the event loop. The event loop is able to handle concurrent executions by delegating the execution of each event to a separate task, which is executed

asynchronously in the background, allowing the event loop to handle the next event in the queue without blocking. This way, even though each event is executed sequentially by the loop, the system can handle multiple events concurrently by switching between tasks in a non-blocking way.

In a system where data is accessed by only one event loop, the problem of concurrency would be easily handled by the execution of handlers inside the same thread, as stated above. However, this approach is not sufficient in the scope of this project. It cannot provide thread safety if multiple event loops (one for the Vert.X server and one for the Socket.IO server) are used concurrently. More specifically, both the Vert.X and Socket.IO Java implementations use Netty[11], a networking framework used for building high-performance and highly scalable network applications in Java, but each server initializes its own event loop.

To resolve the issue, the Java implementation uses two mechanisms: the *Promises* and *Monitors* approach. With Promises, servers demand the execution of asynchronous behaviors by a separate entity, in this case, a *Controller* object. The real execution is delegated to a worker thread managed by the object itself, and as soon as the task is completed, the Promise is resolved.

The following example shows the simplified execution of an action received by a player through a socket channel. This same process is explained in Section 3.4.3 and Figure 13.

Firstly, the player action is detected in the front-end web application:

```
gameSocket.emit('action',
  {xCoordinate: i, yCoordinate: j, action:'VISIT'})
```

This request is received by the game service Socket.IO server, a `SocketServer` object. In the following snippet, the server asks a `Controller` object to handle the request. The `Controller` returns a *CompletableFuture* object and as soon as the promise is resolved, the handler implemented by the method `thenAccept` is executed.

```
private void handleActionRequest(
  SocketIOClient client, ActionMessage d) {
  Controller.get().handleActionRequest(
      roomId, d.getAction(),
      d.getxCoordinate(),
      d.getyCoordinate()
    ).thenAccept((Map<String, Object> obj) -> {
      var status = (String) obj.get("status");
      var result = obj.get("actionResult");
      var map = (String) obj.get("map");
      switch (result) {
        // omitted GAME_LOST, GAME_WON, ...
        // IGNORED action results
        case "OK":
          server.getRoomOperations(roomId)
          .sendEvent("game_update",
```

```
            new GameUpdateMessage(...));
          break;
        }
      }
    });
}
```

The following snippet shows the behaviour implemented by the `Controller` object which links the network requests to the game core instances.

```
public CompletableFuture<Map<String, Object>>
    handleActionRequest(String roomId,
    String action, int xCoord, int yCoord) {
  var res =
    new CompletableFuture<Map<String, Object>>();
  executor.execute(() -> {
    var output = new HashMap<String, Object>();
    ActionType act = ActionType.valueOf(action);
    var game = manager.getGameInstance(roomId);
    output.put("status", "EXECUTED");
    output.put("actionResult",
        game.action(xCoord, yCoord,  act));
    output.put("map", game.toString());
    res.complete(output);
  });
  return res;
}
```

Finally, the front-end web app receives the message and updates the page content with the new map configuration.

```
gameSocket.on('game_update',
  (data: { map: string }) => {
    setGameData(gameData => ({ ...gameData,
      map: data.map }));
});
```

To further improve the implementation's performance, a pool of control flows is used instead of a single worker thread. This allows for effective concurrent management of multiple games, but it raises the issue of what happens when two players try to execute an action in the same game at the same time. In fact, this approach alone does not solve the problem when multiple worker threads concurrently use the same resource. For instance, a single *Game* object may be accessed by multiple players following a request received from each one, or the *GamesManager* object needs to be interacted with via an API request whenever a game needs to be accessed. To solve this problem, we use the `synchronized` keyword in Java to ensure mutual exclusion among objects. Both the

GamesManager and *Game* objects are passive entities, and the end users are the active entities of the system. In this implementation, users access the data remotely through dedicated communication channels.

4.3 Game core optimizations

Efficient implementation of the core of games is crucial to ensure fast response times and low latency during gameplay. For this project, careful consideration has been given to the key aspects of a game's core functionality. The approach used is to pre-compute as much of the game content as possible. While this may cause longer initialization times, it enables faster execution of actions during gameplay. This results in the following sequence of actions being executed during the initialization of a game.

1. a grid of *Tile* objects is generated, each tile is characterized by a `state` (which could evolve during the execution of actions) and `content`;
2. set of mines is randomly generated across the grid;
3. for each mine, the corresponding tile `content` field is set to mine;
4. the number of empty tiles in the grid is tracked;
5. for each empty tile, we compute the `content` field as a number between 0 and 8 (the minimum and maximum number of mines adjacent to the tile).

This method prevents us from having to compute the number of mines each time an action is performed. Additionally, when an action is executed, a graph traversal algorithm is used to explore the grid and uncover zones of clear tiles. The Depth First Search recursive algorithm is utilized for this purpose. During the exploration of the *Tile* object grid, the `state` field of each reached tile is set to "visited". This helps prevent inefficient future explorations on tiles that have already been visited, and also allows the system to update the user on newly visited tiles. Finally, we keep track of the number of visited cells while executing the algorithm. As soon as this number reaches the expected number of empty tiles (which is pre-computed during the initialization process), the game ends. This avoids the need for expensive checks during the execution of an action.

4.4 Project structure

The concrete implementation of the discussed solution, as visible in the GitLab repository [8] is articulated into two main folders:

- the `frontend` folder contains a simple implementation of a web frontend service which deploys the front-end application through a web browser;
- the `backends` folder contains a single Java Gradle project composed of two sub projects, respectively the sessions and games services.

5 Self-assessment / Validation

5.1 Automated tests

The role of automated tests in a software system is to ensure that the system behaves as intended and to detect any potential issues early on in the development process, thereby helping to improve the overall quality of the software. Automated tests can also help to increase confidence in the system by providing a reliable way to verify that the system is working correctly, even as it evolves over time.

Most of the automated tests implemented are *unit tests* done on individual units or components of code to ensure they work correctly in isolation. Some test focuses also on the integration of multiple components but always inside the boundaries of a single service of the micro-service architecture.

Some of the Unit tests implemented for the sessions service are:

- `TestController` focuses on checking the high level requirements of the interfaces offered to network related calls, thus simulating a remote action call;
- `TestSession` verifies the correctness of various sessions configurations and lifecycle;
- `TestSessionsManager` assures that multiple sessions instances can be managed simultaneously.

The sessions service implements 13 tests across multiple files and most of the tests are focused on the verification of the core functionalities of the service. Overall, the code testing coverage is 27% but if we ignore the web interfaces (Socket.IO and Vert.X implementation) and the behaviours related the percentage grows to 80%. We can clearly see how this service main tasks focuses on the handling of network related requests and logic.

Some of the Unit tests implemented for the games service are:

- `TestController` focuses on checking the high level requirements of the interfaces offered to network related calls, thus simulating a remote action call;
- `TestGameConcurrent` focuses on checking the consistency of multiple game states when put under pressure with rapid requests;
- `TestGameCore` verifies the correctness of various game configurations and user actions;
- `TestGamesManager` assures that multiple games instances can be managed simultaneously.

The games service implements 25 tests across multiple files and most of the tests are focused on the verification of the core functionalities of the service. Overall, the code testing coverage is 64% but if we ignore the web interfaces (Socket.IO and Vert.X implementation) the percentage grows to 92%.

No automated test has been implemented for the front-end service.

5.1.1 CI/CD

In modern software development, Continuous Integration and Continuous Delivery are widely adopted methodologies that streamline the software development process and enable faster, more frequent, and more reliable software releases. In a distributed system, where multiple components must work together to provide a seamless and efficient service, CI/CD becomes even more critical.

For the scope of this project, only the CI functionalities have been explored, and no deployment phase has been implemented. The project repository is hosted on GitLab [4], and the configuration of the CI pipeline is visible in the file `.gitlab-ci` inside the project's main folder. Basically, this file orchestrates the execution of independent pipelines, each representing a single microservice, via a trigger-based mechanism. The configuration for those pipelines is visible inside the `.gitlab-ci` file of each sub-folder. This allows each microservice to implement its own configuration independently of the others.

5.2 Manual tests

Manual tests are performed by a human tester who executes the software system or application under test, observes its behavior, and compares the results with the expected outcomes. Unlike automated tests, manual tests rely on the tester's knowledge, experience, and judgment instead of being performed by a machine or tool.

By using a manual approach, the entire system was tested as a whole to ensure it meets the required specifications and functions correctly in different scenarios. Additionally, it was possible to test how the system performs under different loads and stresses to ensure it can handle expected usage levels.

However, it was not possible to test the behavior of the architecture when a large number of clients are interacting with the system simultaneously. Moreover, the execution of manual tests has been carried out inside the boundaries of a single host without taking into account external hosts interactions. This decision was taken to simplify the development process. Having used Docker extensively in the deployment process, to allow remote hosts access to the implemented services would have required for an ulterior micro-service in the role of proxy between the external word and the Docker defined network.

6 Deployment Instructions

6.1 Build automation tools

The system can be deployed in various ways. During the project's initial stages, automated tools such as Gradle and npm were introduced to deploy the necessary components to testing environments. This approach is still available in later versions and is used to verify that all required microservices can be built and tested rapidly.

6.2 Manually initialized Docker containers

To ensure a more standardized and effective testing and development environment, Docker was also introduced. This also allowed for the standardization of network resource configurations. Each microservice must be accessible from outside its boundaries, and Docker also enabled the configuration of user-defined networks and automatic container naming resolutions.

For the system deployment, three different Docker containers must be initialized. First, a user-defined bridge network must be created using the following command:

```
docker network create \
    --driver bridge \
    --subnet=172.18.0.0/16 \
    mmnetwork
```

A docker image must be built for each micro-service. From the project main folder execute the following commands:

```
# build frontend service image
docker build -t mmfrontend ./frontend --no-cache

# build sessions service image
docker build -t mmsession ./backends/sessions --no-cache

# build games service image
docker build -t mmgame ./backends/games --no-cache
```

Docker containers can now be deployed on the current host:

```
# runs frontend service container
docker run \
    -p 3000:3000 \
    -it \
    --rm \
    --network mmnetwork \
    --hostname mmfrontend \
    mmfrontend
```

```
9
10 # runs sessions service container
11 docker run \
12     -p 8001:8001 \
13     -p 8002:8002 \
14     -it \
15     --rm \
16     --network mmnetwork \
17     --hostname mmsession \
18     mmsession
19
20 # runs games service container
21 docker run \
22     -p 8003:8003 \
23     -p 8004:8004 \
24     -it \
25     --rm \
26     --network mmnetwork \
27     --hostname mmgame \
28     mmgame
```

Finally, opening a web browser and navigating at the following URL, the first page of the application should be visible:

```
1 http://localhost:3000/
```

6.3 Docker compose

For convenience, a Docker Compose configuration file has been provided to allow for rapid deployment of all components simultaneously. To use this file, navigate to the project home folder and execute the following commands:

```
1 docker compose build --no-cache
2 docker compose up
```

Docker creates images for each service by following the steps specified in the corresponding Dockerfile. It then creates a container based on each image and executes the commands specified in the docker-compose file. For example, it exposes port 3000 of the web service container, defines environment variables, and connects all containers to a bridge-type network to allow them to communicate.

Finally, by opening a web browser and navigating to the following URL, you should be able to see the first page of the application.

```
1 http://localhost:3000/
```

7 Usage Examples

The provided example show a typical interactions with the system. For the realization of this example, 2 browser tabs were opened inside the same host. Of particular importance is the fact that Socket.IO forces the use of a single socket even across multiple web pages, feature that could bring the system to an inconsistent state. To force the creation of multiple sockets, an instance of the application is launched on an **anonymous web page** and the other one on a **standard web page**.

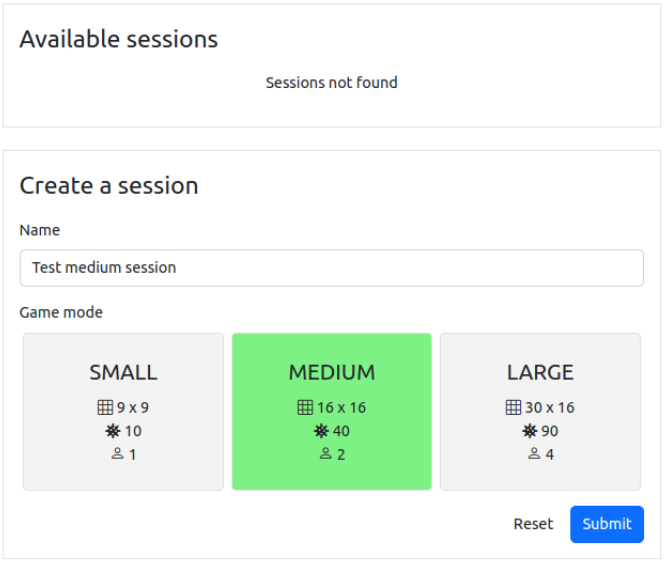

Figure 14: First access to the web application. No sessions are present. The bottom part of the image shows an example of session creation.

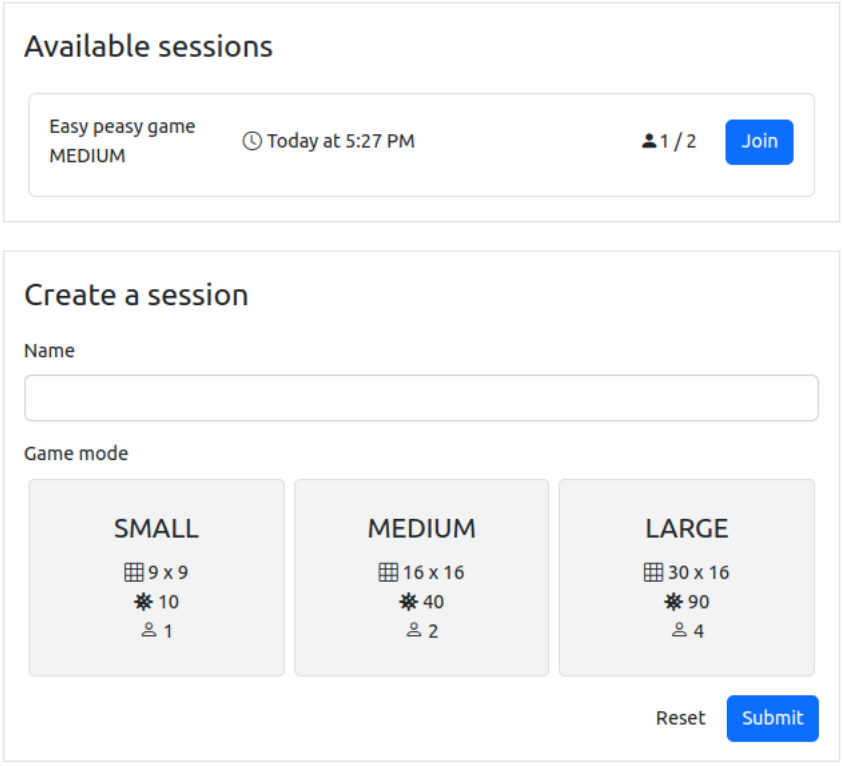

Figure 15: First page content when a session is present. A player can select said session to join it.

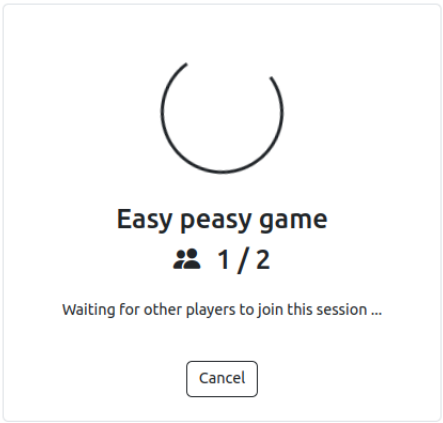

Figure 16: Waiting room screen. Every player is able to see the current number of participants. The game starts as soon as the required amount of players are ready.

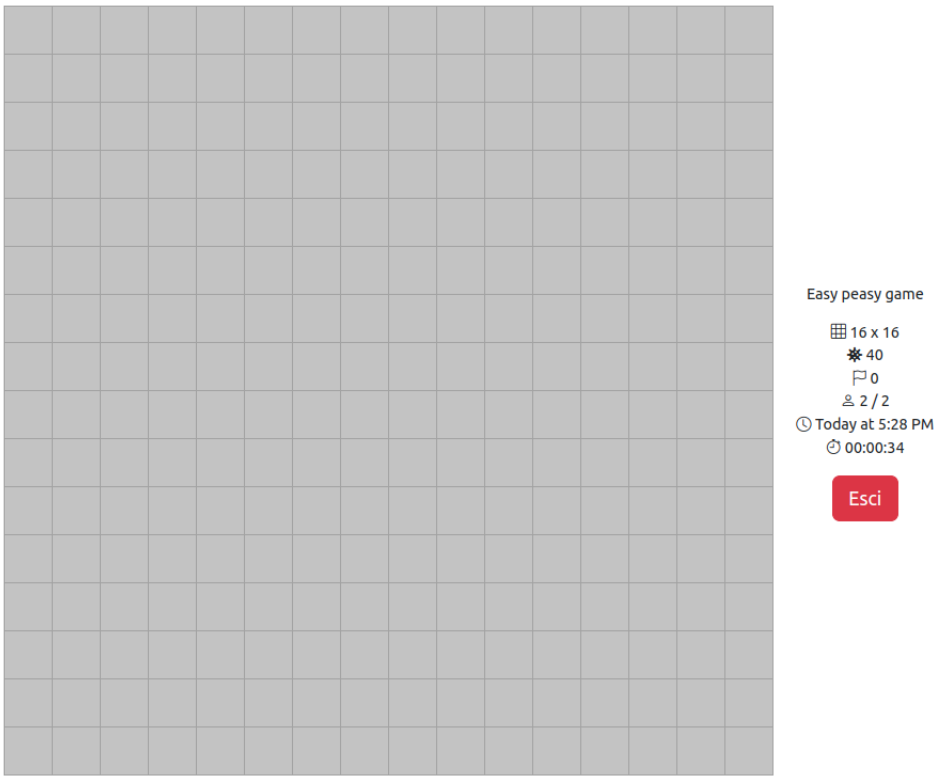

Figure 17: Game board visible when a game is just started, no tile is explored.

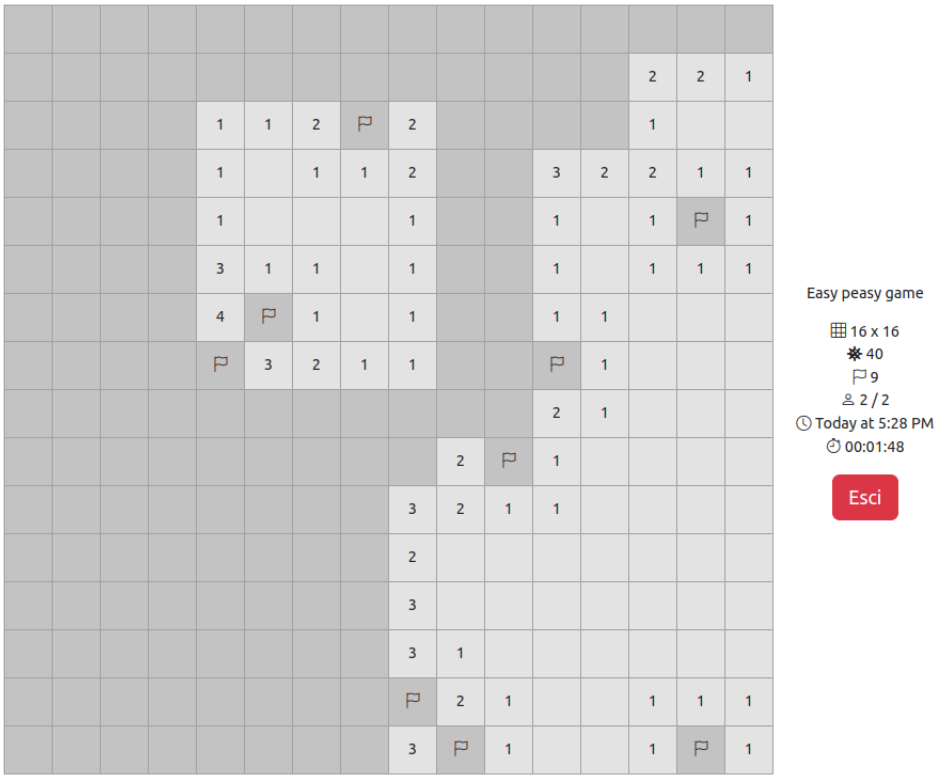

Figure 18: A running game example. Every player is able to see the same board and interact with it.

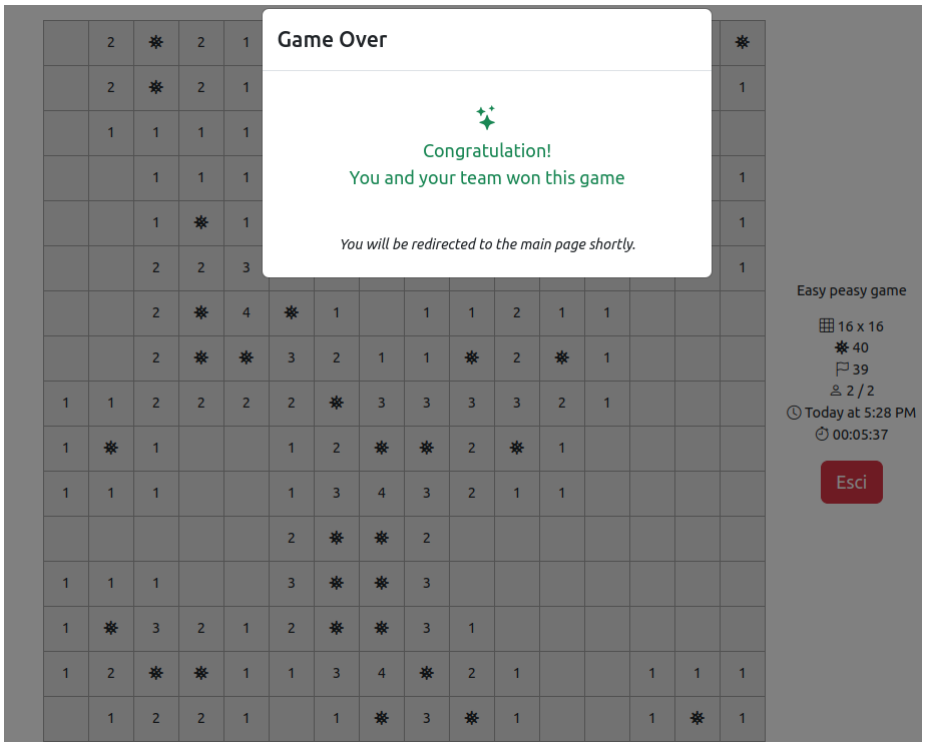

Figure 19: Screen visible when a game is won by a team. After some tile, players will be automatically redirected to the sessions list page.

8 Conclusions

The project I completed allowed me to implement a truly distributed system for the first time, and I am somewhat satisfied with the result. The decision to implement a distributed version of Minesweeper, a game with a simple logic, proved successful, as it allowed me to focus most of my time on studying and managing aspects related to distributed programming, such as managing client connections and disconnections using Socket.IO and handling a shared state across multiple entities.

By choosing a microservices architecture instead of the classic monolithic architecture, I gained a better understanding of various tech stacks and was able to tackle some of the difficulties in creating a hopefully solid solution. In conclusion, I believe that I have achieved a good result, especially considering the challenge of integrating multiple new technologies into the project.

8.1 Future Works

In a future version of the application, an improvement of the graphical interface will be necessary. Currently, it is deliberately simple, but we aim to make it more engaging and visually appealing. Additionally, we plan to implement real-time notifications of player disconnection during the game and introduce new methods for showing each player's actions, such as detailed game logs or interactive icons to improve communication. To achieve this, we will partially refactor the games service, making it more extensible and enabling multiple game modes, competitive matches, and entirely different real-time games.

For educational purposes, it is also necessary to implement a proper deployment process. Currently, the application can only be run locally and cannot be accessed from outside a private network.

8.2 What did we learned

The development of this software has helped me gain a better understanding of numerous aspects related to the development of distributed systems. First and foremost, I have gained a deeper understanding of the roles of clients and servers, and the different ways in which they can communicate and exchange messages. I have also analyzed various network architectures and socket connections with clients, which are present on different devices with different nature and technologies.

Another important aspect that I learned is programming with new frameworks, services, and technologies that I had not had the opportunity to explore before this course. This project is the first to use a large stack of different technologies, and I learned how to interface these various environments to make everything work cohesively as one large distributed system. The size of this project also allowed me to understand how to approach a potentially onerous job that requires organization, understanding, and time management.

In conclusion, completing this project has helped me better understand the functioning

of a distributed system. Together with the course, it has provided me with concrete knowledge that I consider essential for those working in our field and for understanding topics that develop on these concepts.

References

[1] Microsoft Corporation. Minesweeper. https://www.microsoft.com/en-us/p/minesweeper/9wzdncrfhwcn, 1990. Accessed: 2022-03-22.

[2] Inc. Docker. Docker. https://www.docker.com/, 2013. Accessed: 2022-03-22.

[3] Eclipse Foundation. Vert.x. https://vertx.io/docs/vertx-core/java/, 2011. Accessed: 2022-03-22.

[4] Inc. GitLab. Gitlab. https://gitlab.com, 2011. Accessed: 2022-03-22.

[5] Gradle Inc. Gradle. https://gradle.org/, 2007. Accessed: 2022-03-22.

[6] JUnit contributors. JUnit. https://junit.org/junit5/, 2022. Accessed: 2022-03-22.

[7] MDN Web Docs contributors. JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript, 2022. Accessed: 2022-03-22.

[8] Fabio Muratori. Multiplayer minesweeper project. https://dvcs.apice.unibo.it/pika-lab/courses/ds/projects/ds-project-muratori-ay2021, 2023. Accessed: 2022-04-19.

[9] npm contributors. npm. https://www.npmjs.com/, 2022. Accessed: 2022-03-22.

[10] Oracle Corporation. Java. https://www.oracle.com/java/, 1995. Accessed: 2022-03-22.

[11] Netty Project. Netty. https://netty.io/, 2011. Accessed: 2022-03-22.

[12] React contributors. React. https://reactjs.org/, 2022. Accessed: 2022-03-22.

[13] Socket.IO contributors. Socket.IO. https://github.com/socketio/socket.io, 2022. Accessed: 2022-03-22.

[14] The Node.js Foundation. Node.js. https://nodejs.org/, 2009. Accessed: 2022-03-22.

[15] TypeScript contributors. TypeScript. https://www.typescriptlang.org/, 2022. Accessed: 2022-03-22.

[16] Wikipedia contributors. HTTP - Hypertext Transfer Protocol. https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol, 2022. Accessed: 2022-03-22.

[17] Wikipedia contributors. JSON. https://en.wikipedia.org/wiki/JSON, 2022. Accessed: 2022-03-22.

[18] Wikipedia contributors. REST - Representational state transfer. https://en.wikipedia.org/wiki/Representational_state_transfer, 2022. Accessed: 2022-03-22.

[19] Wikipedia contributors. UUID - Universally unique identifier. https://en.wikipedia.org/wiki/Universally_unique_identifier, 2022. Accessed: 2022-03-22.

[20] Wikipedia contributors. WebSocket. https://en.wikipedia.org/wiki/WebSocket, 2022. Accessed: 2022-03-22.