

Spaceteam Game - Final Report

Distributed Systems
A.Y. 2023/2024

Riccardo Omiccioli
`riccardo.omiccioli@studio.unibo.it`

October 2024

Contents

1	Domain Analysis	3
1.1	Goal	3
1.2	Game Rules	3
1.3	Usage Scenarios	5
1.4	Questions & Answers	7
1.5	Self-assessment policy	8
2	Requirements	9
2.1	Functional Requirements	9
2.2	Non-Functional Requirements	10
2.3	Implementation Requirements	10
3	Design	12
3.1	Architecture	12
3.2	Structure	14
3.3	Behaviour	16
3.4	Interaction	17
4	Implementation Details	20
4.1	MQTT Broker	20
4.2	Game Application	21
4.2.1	Build Automation & Dependency Management	21
4.2.2	Architecture implementation	21
4.2.3	Design Patterns	23
4.2.4	Serializers/Deserializers	24
5	Self-assessment / Validation	26
6	Deployment Instructions	27
6.1	MQTT Broker	27
6.2	Game Application	28
7	Usage Examples	29
8	Conclusions	36
8.1	Future Works	36
8.2	What did we learned	36

This project focuses on the development of a multiplayer game inspired by the video game *Spaceteam*. In this game, players work together to conduct a spaceship on a journey across the galaxy. Each player has a unique control panel displaying the ship's current state, with a randomized assortment of buttons and other controls labeled with technobabble, symbols, and hard-to-read words.

During each level, players designated as captains receive random orders to activate specific controls, however, the actions often apply to another player's control panel. Each order is timed, and if not completed successfully, the spaceship takes damage. If the damage sustained becomes too severe, the game ends.

Players must communicate verbally, either in person or through online voice chat, to ensure the correct actions are performed.

Successfully completing the required amount of orders within a level advances the team to the next level, where challenges get tougher. The objective of the game is to progress as far as possible.

1 Domain Analysis

Before defining the system requirements and designing a solution, it is crucial to clearly understand the project's goal, expected behaviors and any other relevant considerations that help to frame the problem. This will include, for example, a comprehensive breakdown of the game rules, an overview of how users will interact with the system, and a series of questions and answers that emerged during the project's inception, considering the idea from both a developer's and a potential user's perspective.

1.1 Goal

The goal of this project is to develop a real-time multiplayer game by designing a distributed system that is capable of implementing some logic, in the form of game rules while managing interactions between multiple distinct nodes, regardless of their geographical location. Although the game mechanics and player actions are straightforward, this project provides an opportunity to explore some important challenges of distributed systems, such as ensuring fast and reliable communication between nodes, achieving consensus on critical decisions, and handling system failures.

The expected outcome is a functional game that enables multiple players to play together on different devices. The system should ensure that users can play reliably and efficiently without being aware of the underlying mechanics.

1.2 Game Rules

As previously mentioned, this game draws inspiration from the game *Spaceteam*, however there are some differences emerged both as a necessity to simplify some aspects and to adapt the game to a distributed scenario where there could be more players than in

the original version of the game. The following is a detailed breakdown of the rules for the game that will be implemented, covering only the relevant aspects of "how the game should play", while abstracting from any technical aspect.

Levels Each game begins at level 1 and continues indefinitely. During each level, players must execute a specified number of orders, which will increase as the level number increases.

Orders Each order represents a task necessary for progressing through the level and consists of two components: a control action and a control label. The label indicates which control should be activated, while the action specifies how to manipulate the control. Typically, the label is a single word or brief phrase but could also include symbols or random characters. There are various types of actions; some require just to activate the control, while others may require turning the control on or off, or setting a specific value. Each order also specifies a time limit for successfully completing the order. Failing to complete the order within this timeframe damages the spaceship.

Control Panel Every player has a control panel that displays useful information about the current state of the game as well as having controls for performing orders. Specifically, controls are arranged in a 3x3 grid with nine slots available. Some controls occupy a single slot, while others are larger and can span multiple slots, provided they fit within the panel. The number and type of controls available are randomized for each player and are selected from a predefined set of options.

Players During gameplay, there are two distinct types of players: captains and crew members. The only difference is that captains can see and receive the orders to execute, while crew members cannot. Each captain is assigned one random order at a time, which may refer to any control, whether it belongs to them or to other player's control panel. Given that only captains can check the orders, they must communicate with other players to complete the tasks successfully. The rules do not specify how players should communicate, allowing the freedom to choose their preferred method. Some valid suggestions include gathering in person or using online voice chat. Players may also opt for a more challenging method of communication to increase difficulty and add to the fun of the game. When there are multiple captains, it is possible for some orders issued to be identical across two or more captains. Successfully completing such an order counts as multiple valid orders completed. Receiving a new order resets the timer for completion. There is no limit to the maximum number of players allowed in each game, while the minimum is one, enabling single-player mode. Regardless of the total player count, there must always be at least one captain in every game.

Win condition Technically, there is no definitive win condition since players should aim to progress as far as possible through an infinite number of levels. However, there

are certain conditions that can lead to a game loss.

- If all captains leave the game, the current game will end.
- At the beginning of each level, the spaceship is fully repaired but, if its integrity reaches zero during a level due to incorrect orders, the game is lost.

1.3 Usage Scenarios

After reviewing the game rules, different usage scenarios were identified, and other were expanded by considering how a typical game would be set up and played. This resulted in the creation of the use case diagram shown in Figure 1.

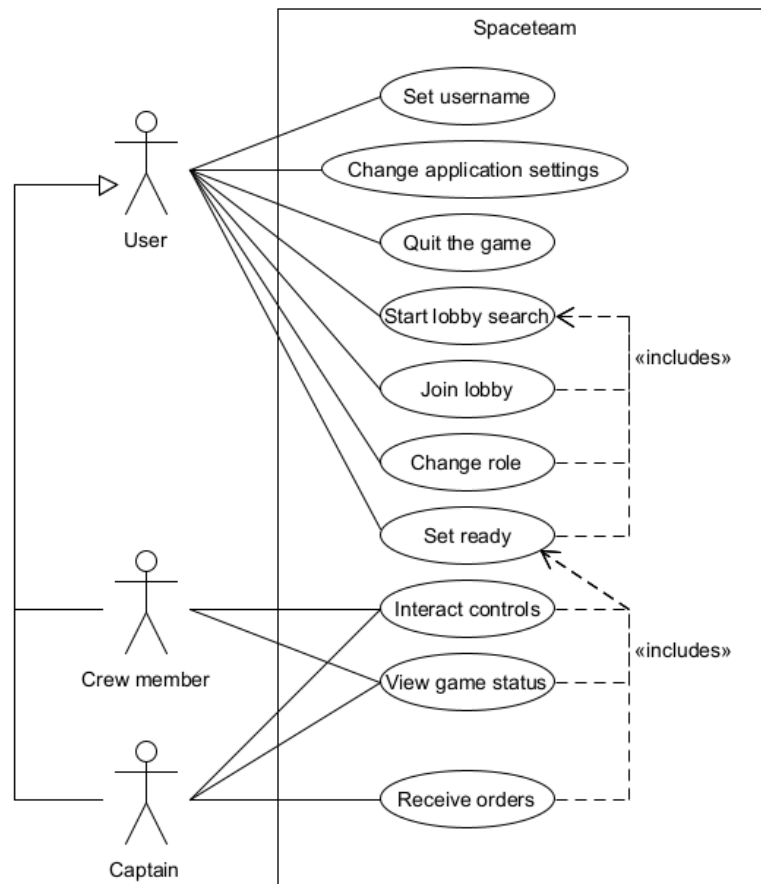


Figure 1: Use case diagram for the game

A user of the application is expected to adjust various settings, set a username, start searching for a lobby, or quit the game. The lobby system is a common feature in many multiplayer video games and serves as a pre-game waiting room where players can find

others and configure game settings. In this game, users are expected to join lobbies, switch between the two roles identified in the game rules, and mark themselves as ready to start a match. From Figure 1, it is also possible to notice that both crew members and captains can input commands and view the game’s current status. However, only captains can receive orders.

The use cases gets refined by developing mockups for the application, which helps to quickly prototype and visualize how users will interact with the system. While this approach does not directly contribute to defining the internal design, it provides valuable visual insight into how the identified use cases will be presented to users. By visualizing user interactions, it helps clarify important details that can be later formalized into requirements, which will ultimately guide the design process to meet those requirements. The game is expected to feature a main menu, serving as a starting point when the users load the game, as shown in Figure 2a. From the main menu, users can adjust application settings, quit the game, set a username, and initiate a lobby search.



Figure 2: Menu mockups

Figure 2b shows the lobby feature, which has its own dedicated menu. Given the game’s multiplayer nature, this element is crucial, requiring a specialized view to effectively interact with the feature. In this view, users can search for lobbies, view the current lobby status, switch their role for the game, and mark themselves as ready to

start. Finally, Figure 3 presents a more refined visual representation of the game’s interface. Most elements outlined in the game rules are depicted here. For example, players can see key information such as the current level, ship integrity, active orders, and the remaining time for completion. Additionally, the game features a control panel, illustrated in Figure 3, which showcases a 3x3 grid layout for controls. This grid presents different concepts for the controls, featuring labels and the corresponding actions that players can execute.



Figure 3: Mockup for the game

1.4 Questions & Answers

The following is a list of questions and answers that emerged while discovering the project domain. While most of these questions have been refined and incorporated into the game rules and domain analysis, they are included here to reinforce key concepts and provide clearer explanations.

- **Question** What happens if a player disconnects during a game?
Answer The game continues as long as there is at least one captain. If the player that left had a control that was relevant for a currently issued order, the order will fail on timeout, the ship will take damage, and if possible a new one will be issued.
- **Question** Can a user join a lobby during a game?
Answer Yes, if the players knows the lobby identifier.
- **Question** Is the username of the player unique?
Answer No, the username of a player has the only purpose to make it easier to differentiate players inside a lobby.
- **Question** When should a game start?
Answer A game starts after all players in a lobby have marked themselves as ready, as long as there is at least one player with the role of captain.
- **Question** How does a level play out?
Answer Each captain receives a random order. If the order is relative to their control panel, they execute it; if not, they communicate it to other players and wait for it to be executed by them. The level is considered complete once the required number of orders has been successfully executed.

1.5 Self-assessment policy

- How should the *quality* of the *produced software* be assessed?
The quality of the system will be evaluated primarily based on non-functional requirements, which will be formalized during the design phase. Quality attributes will include performance, scalability, and compatibility across different platforms.
- How should the *effectiveness* of the project outcomes be assessed?
The effectiveness of the software will primarily be evaluated based on the functional aspects of the application. Formally defined functional requirements will serve as the foundation for this assessment, ensuring that all specified features are met. Manual integration tests will be carried out in order to test the overall system behavior. When necessary, formal tests will be implemented to target critical sections of the application's code.

2 Requirements

Following the domain analysis phase, the identified ideas and scenarios are formalized into requirements. These requirements will outline what needs to be done, in order to guide the design and implementation process, serving as a reference to ensure that all tasks required for developing a complete system are addressed.

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.

1. Set a username
2. Change application settings:
 - 2.1. Change application resolution
 - 2.2. Change application frame rate
3. Quit the application
4. Manage lobbies:
 - 4.1. Join a random lobby
 - 4.2. Join a lobby with an identifier
 - 4.3. View players information in a lobby
 - username
 - role
 - status
 - 4.4. Change player role
5. Start a game:
 - 5.1. Set ready status for a player
 - 5.2. Start a game when all players are ready
 - 5.3. Join an ongoing game
6. Play a game:
 - 6.1. View current game status:
 - View current level
 - View spaceship integrity
 - View number of players in the game
 - View orders as a captain

- View order's timer
 - View completed orders number
 - View required orders number
- 6.2. Generate a random control panel
 - 6.3. Proceed to the next level
 - 6.4. End a game on loss conditions

2.2 Non-Functional Requirements

The following requirements specify how the system should perform. This includes quality attributes and performance standard that the system must meet.

- The system should incorporate fault tolerance features, specifically:
 - It must function normally even if one or more users disconnect;
 - It should effectively handle brief connectivity losses from users, with a maximum duration of 5 seconds.
- The application must operate seamlessly and without any loss of functionality across multiple operating systems, specifically on Windows (versions 11) and Linux (distribution Fedora);
- The system must work correctly without requiring any configuration, particularly regarding network settings;
- All interactive elements of the graphical user interface should comply with the WCAG (Web Content Accessibility Guidelines) AA standard regarding font size and contrast ratio.

2.3 Implementation Requirements

The following requirements focus on the environment in which the system will be developed and operated, providing details on the languages, tools and technologies that will be used for implementation, as well as the conditions necessary for execution.

- The system must utilize a *peer-to-peer architecture* to eliminate the need for a centralized server that handles the application logic;
- The application should be developed using *Java 21*;
- The user interface must be built using *JavaFX* to provide a usable and responsive graphical user interface;
- The system should utilize the *MQTT* protocol for efficient and reliable communication between peers;

- The *MQTT broker* must be containerized to simplify installation and deployment;
- The game application must be delivered as a bundled executable;
- The project should incorporate tools for build automation and dependency management;
- The project code should be managed using *Git* for version control.

3 Design

The design phase defines system's core components and provides a detailed review on the decisions made to define how the system should function. For each design choice, an explanation is provided, highlighting the reasoning behind it, while also considering other potential alternatives that were evaluated. This ensures that the system meets its requirements while balancing factors such as efficiency, scalability, maintainability, and development time.

3.1 Architecture

The architecture chosen for this system is peer-to-peer using a publish-subscribe communication pattern facilitated by a remote message broker. Figure 4 depicts such architecture at a high level of abstraction.

The peer-to-peer architecture offers several advantages for this project, particularly in terms of decentralization and scalability. Since each node in a peer-to-peer system communicates directly with others, there is no single point of failure that could disrupt the system's normal operations in the event of errors. Additionally, the workload is distributed across nodes, making the system scalable and cost-effective. As long as the broker can handle the network traffic, the system should scale efficiently with minimal computational overhead. However, peer-to-peer architectures require careful consideration regarding decision making to ensure the system remains reliable and consistent. In contrast, client-server architecture simplifies control but introduces a single point of failure.

The use of a remote broker allows seamless communication between nodes from different locations without requiring network configurations. Alternative communication patterns, such as a brokerless model or direct publish-subscribe with a message bus were considered, but these options would require additional configuration when nodes are not on the same network, making the chosen solution more practical and flexible.

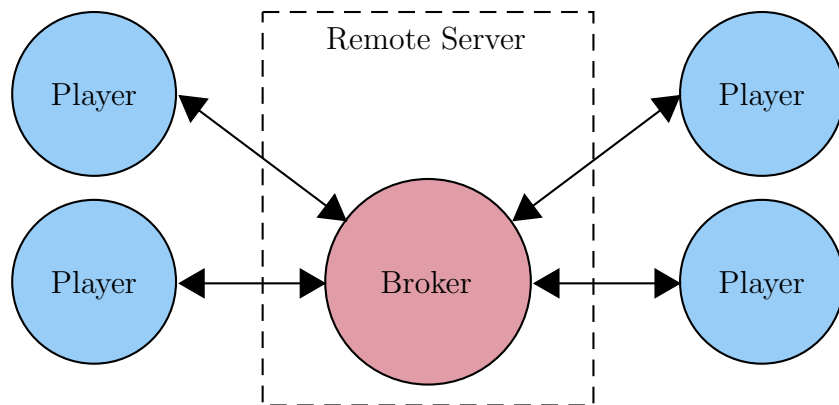


Figure 4: General architecture of the system

For this project, the *MQTT 5.0* protocol has been chosen to manage communication,

using the *EMQX* self-hosted open source broker. MQTT is a lightweight protocol ideal for peer-to-peer setup with minimal overhead during communication.

The EMQX broker, in particular, offers remarkable scalability and flexibility even in the open source and free version. A vastly superior version is available as an enterprise product, designed to manage a significantly larger volume of connections and messages. Both versions excel in minimizing latency for message routing to topics, adding only a few milliseconds on top of the inherent network latency.

The table 1 presents some statistics on EMQX self-hosted broker.

	EMQX Open Source	EMQX Enterprise
Scalability	3 nodes cluster/100,000 connections	100 nodes cluster/100 million connections
Performance	100,000 messages per second	5M+ messages per second
Latency	1 to 5 milliseconds	1 to 5 milliseconds

Table 1: EMQX Self-Hosted broker scalability and performance

MQTT also offers a flexible and straightforward solution for managing various communication scenarios. It does so by defining multiple topic on which many nodes can publish or subscribe. For example, Figure 5 shows a simple one-to-one scenario where one node communicate directly with another. This allows for efficient direct communication by targetting directly the intended node without overwhelming the network with unnecessary messages.

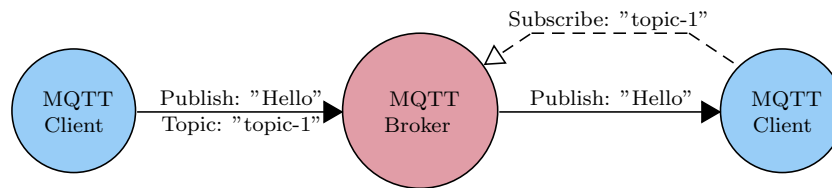


Figure 5: Simple one-to-one MQTT interaction

MQTT can also effectively manage a wide range of situations, such as many-to-one, one-to-many, and many-to-many scenarios. For example, Figure 6 demonstrates how multiple nodes can simultaneously publish data to many distinct listeners. This could be particularly useful for broadcasting updates about the application's status. Finally, MQTT offers different Quality of Service (QoS) levels, which ensure automatic and reliable message delivery.

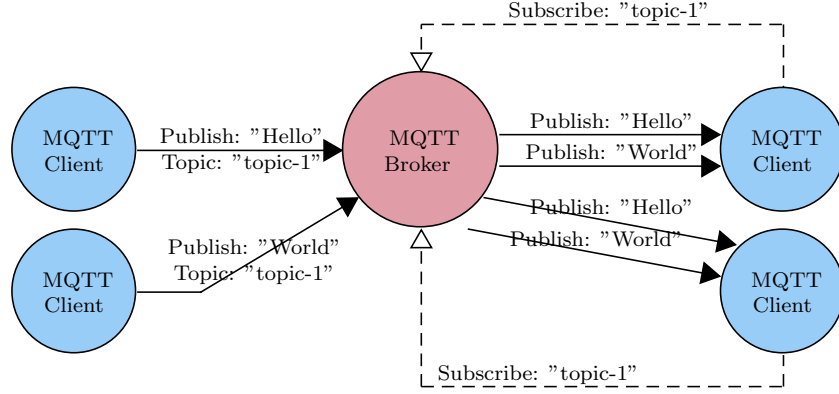


Figure 6: Many-to-many MQTT interaction

3.2 Structure

The structure of this system is composed of two modules, a MQTT broker and a game application. The detailed architecture, derived from the general architecture shown in Figure 4, is illustrated in Figure 7, which shows the system's deployment.

To meet the implementation requirements, the MQTT broker will be hosted in a containerized environment, simplifying deployment. This environment will be provided by Docker and can be either local or remote; however it is expected to use a remote solution after initial development. The game application is executed within a JVM environment and utilizes the MQTT protocol to exchange information with other nodes through the broker.

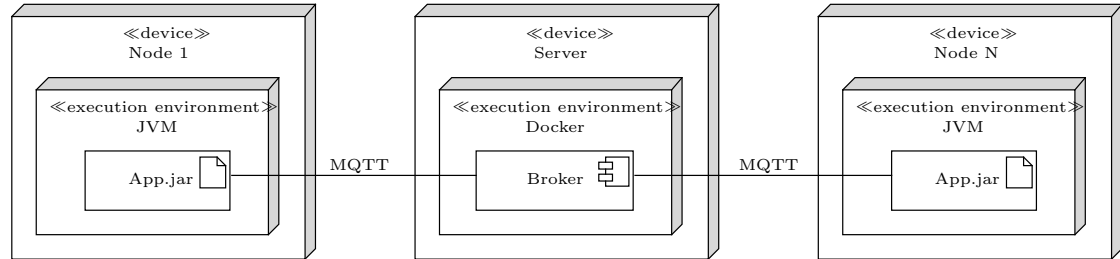


Figure 7: Deployment diagram

The game's application architecture follows a modular design inspired by the *MVC* pattern. Figure 8 illustrates the main components of the application and their interactions. As in MVC, there are View, Controller, and Model components, represented by blue, red, and green respectively in Figure 8. The View interacts with the user, providing a graphical interface, handling inputs, and rendering the game. User inputs are passed to the Controller, which initiates actions that change the state of the application. Following an update the Controller calls for a render on the View. However, unlike the traditional MVC pattern, the Controller does not directly interact with the Model. Instead, it generates actions that the Game Engine component executes when appropriate.

It's the Game Engine that updates the local game state modifying the model. Similar to how the View and Controller handle interactions with a user the Multiplayer Service and Mqtt Service interact with the Game Engine to provide communication with other nodes. The Multiplayer Service manages high-level subscriptions and publications, while the Mqtt Service implements the effective communication with the MQTT broker. Additionally, the Multiplayer Service interacts with a Vote Service, which handles voting for critical shared decisions in the game.

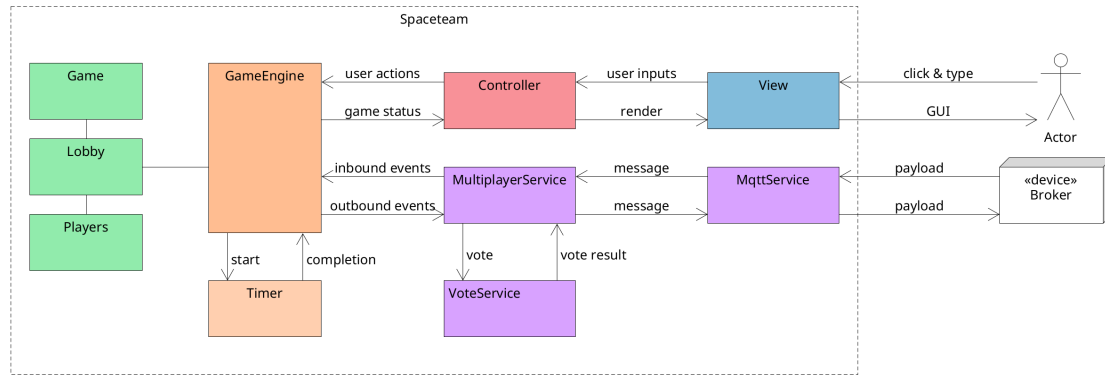


Figure 8: Architecture of the application

The detailed class diagram for Model of the application, briefly introduced in the architecture of Figure 8, is fully illustrated in Figure 9. At the core of the Model is the Lobby class, which represents a game lobby, including all joined players and game status. The class diagram of Figure 9 highlights some key aspects already identified during the domain analysis phase, such as the game status, controls, and orders. Many of these classes are in fact directly derived from the domain analysis.

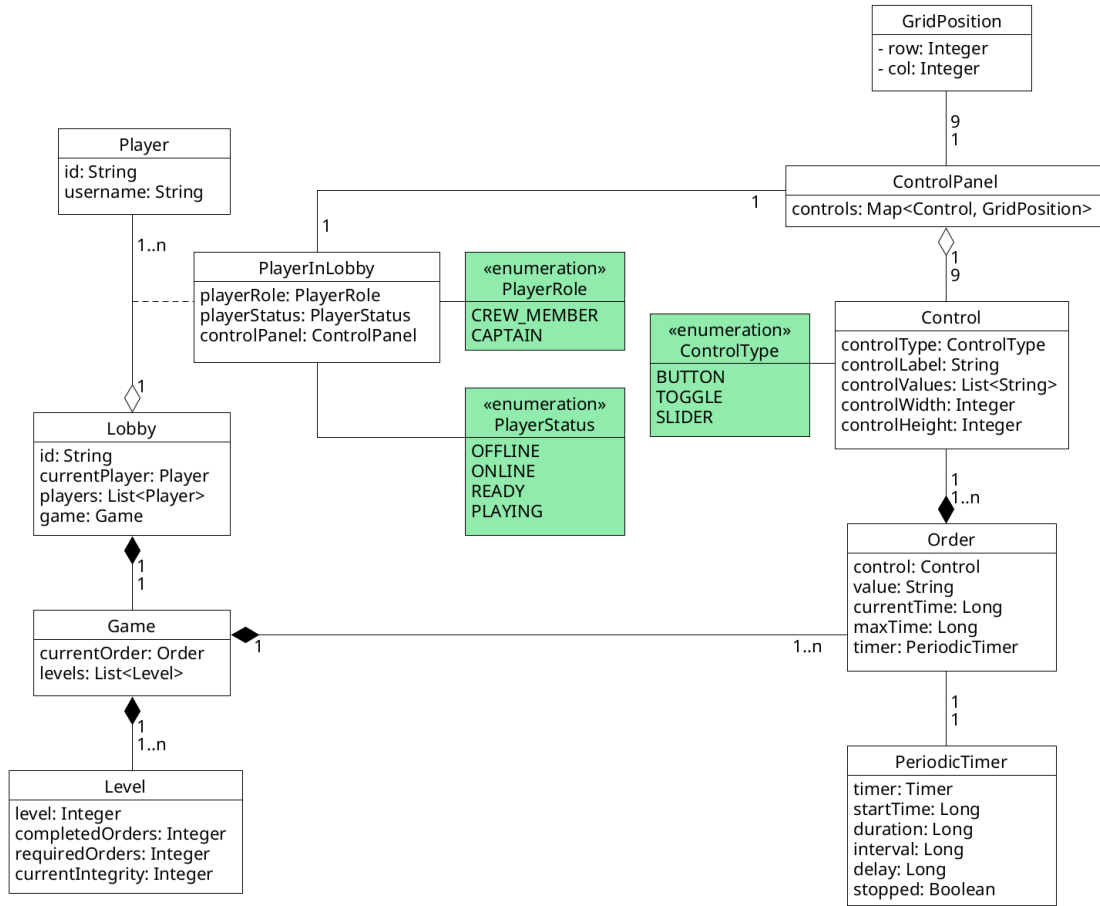


Figure 9: Class diagram of the model

3.3 Behaviour

The behavior of each node is summarized in the state diagram shown in Figure 10. This diagram outlines the main logic of the entire application, from initialization and setup to the conclusion of a game. It highlights two macro states: the behavior of the game during a lobby setup and during gameplay. The state diagram formalizes the game rules into specific conditions and actions that must be fulfilled to progress through the game and interact with the application's functionalities. The diagram only considers the point of view of a single application while interaction patterns are omitted.

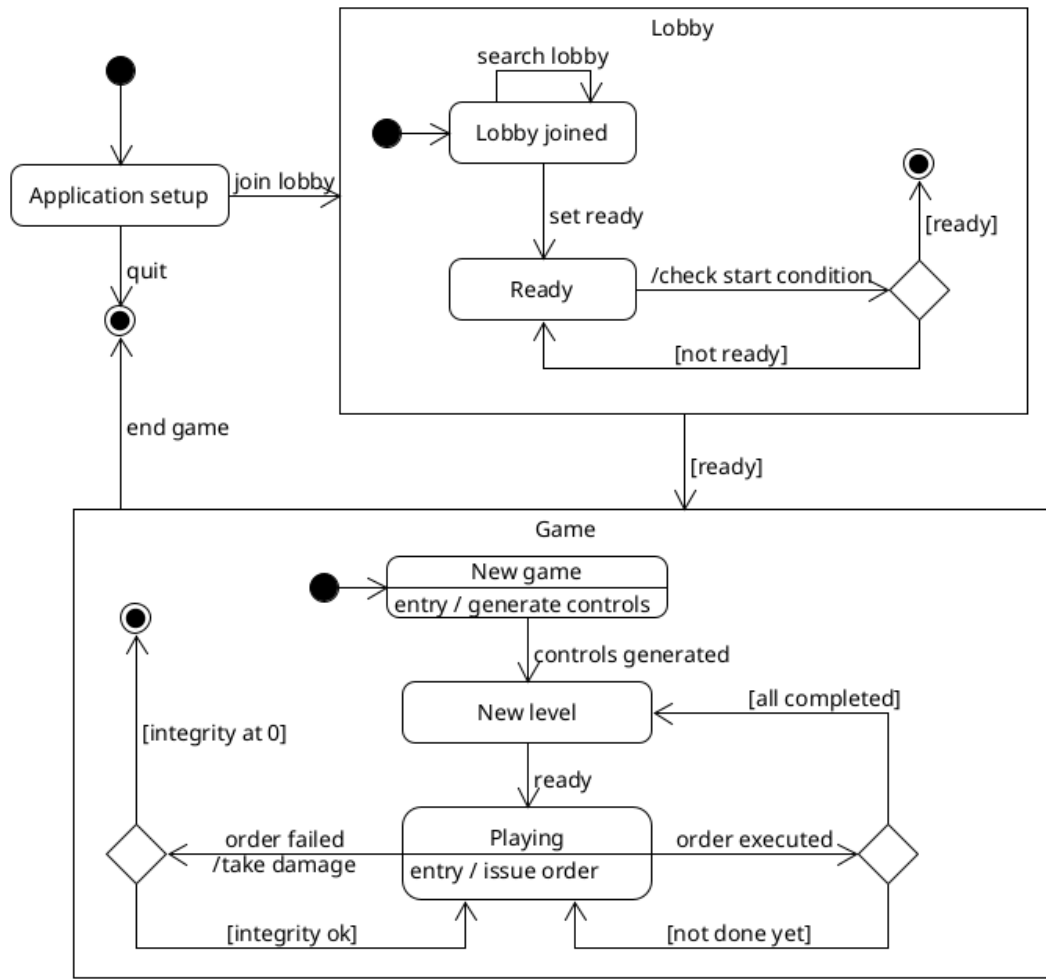


Figure 10: State diagram of the game application

3.4 Interaction

As previously mentioned, interactions between the nodes of the application are managed using a publish-subscribe pattern. Utilizing the MQTT protocol brings several advantages for nodes communication in this application. The most relevant features for this project are QoS (Quality of Service) levels, wildcards subscriptions, retained messages, and last will messages.

Each message can have a specified QoS level that determines how reliably it should be delivered to other nodes. At QoS level 0, no delivery guarantees are provided, and the messages are sent with a best-effort policy. This is the least resource intensive option but does not ensure that all subscribers will receive the message in case of network faults. QoS level 1 enables the delivery of messages at least once for every node, at a

cost of increased overhead. Finally, QoS level 2 ensures exactly-once delivery, making it the most reliable but also the most expensive in terms of resources. Importantly, these QoS assurances are managed by the MQTT protocol itself, greatly simplifying reliable communication between nodes and handling some network failures automatically.

Wildcard subscriptions are used in this project to allow automatic subscription to multiple topics without manually specifying each one. This simplifies managing various communication topics within the application, to enable different interactions on each of them. Figure 11 illustrates the topics used during a game, highlighting wildcard topics and providing examples of the data expected for each one. Each lobby has a base topic, consisting of a "spaceteam/lobby" prefix followed by the lobby identifier string. Within each lobby, different topics manage communication for players information, votes, controls, and orders validation. Most of the wildcard subscriptions are used to differentiate subtopics depending on the current level of the game, separating the flow of messages in appropriate topics in order to simplify the interactions during a game. In addition, the player topic contains all information about players, with individual subtopics for each player based on their identifier.

spaceteam/lobby/X1A67GH/player/#	{ "playerId": "c1d8dda1", "username": "player 1", "role": "CAPTAIN", ... }
spaceteam/lobby/X1A67GH/vote/level/#	{ "voterId": "c1d8dda1", "voteValue": true }
spaceteam/lobby/X1A67GH/vote/end	{ "voterId": "c1d8dda1", "voteValue": true }
spaceteam/lobby/X1A67GH/control/#	{ "playerId": "c1d8dda1", "controlLabel": "BUTTON", "controlValue": "PUSH" }
spaceteam/lobby/X1A67GH/order/#	true / false

Figure 11: Topics with some example payloads

This topic uses retained messages, meaning that the MQTT broker stores the last message sent to each player topic. This feature allows new players joining the lobby to retrieve up-to-date information about other players, even if they were not connected when the original messages were sent.

Another important feature for this project is the last will message. This is a message configured before connection that gets published to a specified topic if a client unexpectedly disconnects. Using this feature, the system can overwrite retained messages with players information, preventing stale data from being sent to newly joined players. Additionally, it simplifies detecting disconnections, allowing the system to react appropriately when a player leaves mid-game.

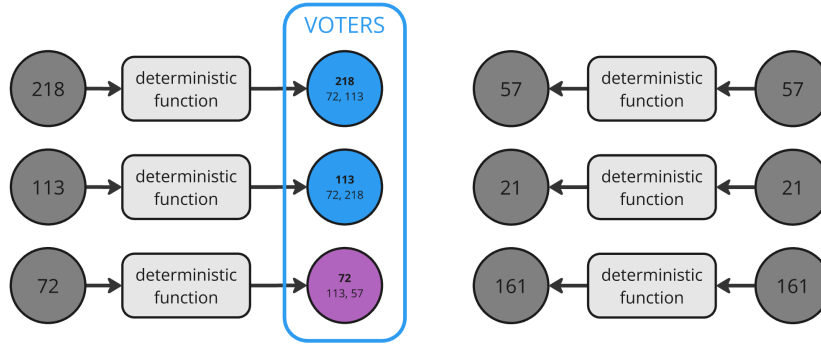


Figure 12: Example of voters identification

While some interactions between nodes rely on the built-in assurances of the MQTT protocol, other more critical and shared decisions are handled using a voting system designed for this application. This voting mechanism is implemented to mitigate incorrect decisions by individual nodes and to prevent system failure or slowdown caused by a slow or faulty node. Given the peer-to-peer nature of the application, there is not a central node responsible for the decision making of the system, and each node casts and verifies votes when making critical decisions. However, in scenarios with a large number of nodes, having every node participate in voting can generate an exponential exchange of messages, overwhelming the system and potentially causing delays.

To solve this, voting is limited to a restricted group of nodes that gets dynamically selected. As depicted in Figure 12, only a subset of nodes are chosen as voters. This selection is made by using a deterministic function, which identifies eligible voters based on information such as the node ID and game state data. The number of voters is configurable, allowing scalability depending on the system requirements. The designated voters are responsible for sending their decisions and informing the rest of the nodes. Every other node listens for these votes and validates the results. Depending on the needs, different voting policies can be applied, such as waiting for just one vote, a majority, or all votes from the selected group before committing to a critical state change.

4 Implementation Details

Following the design phase, the implementation realizes the application writing code that follows the chosen design. Attention will be focused on interesting or non-trivial implementation details while obvious implementation aspects will be omitted.

4.1 MQTT Broker

This project utilizes an MQTT 5.0 compliant broker provided by EMQX, which can be easily instantiated using the official EMQX Docker image. To streamline deployment, this project uses a *Docker Compose* setup alongside configuration and `.env` files, enabling the broker to be set up and run with a single command.

The following listing is an extract from the Docker Compose file, highlighting the basic configuration used for the broker.

```
1 services:
2   emqx:
3     image: emqx/emqx:5.8.0
4     container_name: emqx
5     restart: always
6     ports:
7       - "1883:1883"    # MQTT TCP port
8       - "8083:8083"    # MQTT WebSocket port
9       - "8084:8084"    # MQTT WebSocket Secure (SSL) port
10      - "8883:8883"    # MQTT Secure (SSL) port
11      - "18083:18083"  # EMQX Dashboard
12     environment:
13       EMQX_DASHBOARD__DEFAULT_PASSWORD: "${EMQX_DASHBOARD_PASSWORD}"
14
```

The EMQX broker can also be configured in a cluster setup, deploying multiple instances to enhance both the availability and performance of the broker.

Below is a simplified example of how this can be achieved using Docker Compose to run multiple instances on a single server. However, a better use of this feature would be to deploy the cluster on different remote servers to further increase performance and availability.

```
1 services:
2   emqx1:
3     image: emqx:5.8.0
4     container_name: emqx1
5     environment:
6       - "EMQX_NODE_NAME=emqx@node1.emqx.io"
7       - "EMQX_CLUSTER__DISCOVERY_STRATEGY=static"
8       - "EMQX_CLUSTER__STATIC__SEEDS=[emqx@node1.emqx.io,emqx@node2.emqx.io]"
9     healthcheck:
10       ...
11     networks:
12       emqx-bridge:
13       ...
```

```

14     ports:
15         ...
16
17     emqx2:
18         image: emqx:5.8.0
19         container_name: emqx2
20         environment:
21             - "EMQX_NODE_NAME=emqx@node2.emqx.io"
22             - "EMQX_CLUSTER__DISCOVERY_STRATEGY=static"
23             - "EMQX_CLUSTER__STATIC__SEEDS=[emqx@node1.emqx.io,emqx@node2.emqx.io]"
24         healthcheck:
25             ...
26         networks:
27             emqx-bridge:
28                 ...
29 networks:
30     emqx-bridge:
31         driver: bridge
32

```

4.2 Game Application

The game application is implemented using the following technologies:

- *Java 21* as the main language to write the application code
- *JavaFX 23* to implement the Graphical User Interface
- *Gradle* to automate tasks and manage dependencies
- *Gson* to serialize and deserialize data in the *JSON* format
- *Eclipse Paho* for the MQTT client code

4.2.1 Build Automation & Dependency Management

This project uses Gradle for build automation and dependency management, following a flat-project structure. Beyond typical configurations in a standard Gradle setup, this project includes the *Shadow* plugin, which enables the creation of a `shadowJar` task to create an Uber JAR. This bundled JAR contains all application code and dependencies, allowing the application to run without additional configurations.

4.2.2 Architecture implementation

The diagram in Figure 13 illustrates how the core application architecture is implemented, closely following the overall design outlined in Figure 8. The diagram in Figure 13 shows the components using different colors depending on their role:

- **View** components (blue color): These handle the display and input handling through the GUI, using *FXML* files for layout definition, allowing for simple and modular designs that can be edited using external tool such as *Scene Builder*.
- **Controller** components (red color): These interact with the View to translate user inputs into actions and send updates the GUI. Each menu has a dedicated view and controller, modularizing the code for different interfaces.
- **Model** components (green color): These represent the game's data structure, maintaining the state of the game and providing methods to change the state in a controller manner. The core class for the model, *Lobby*, is reported in the diagram, while other classes are omitted and can be found in Figure 9.
- **Distributed** components (purple color): These include the Multiplayer Service, managing subscriptions and publishing at a high level of abstraction; the Mqtt Service, which establishes communication with the broker using the Eclipse Paho client; and the Vote Service, which tracks votes and voting members for critical decisions.
- **Game** components (orange color): This component serves as the central logic handler, interacting with the Model and interfacing both the GUI and broker through the Controller and Multiplayer Service. The Game Engine incorporates an Executor Service to run tasks from both GUI inputs and node events, which are processed as Action and Event objects queued in response to external inputs. This ensures a modular and adaptable handling of the game logic in response to external inputs, whether from the user or other nodes.

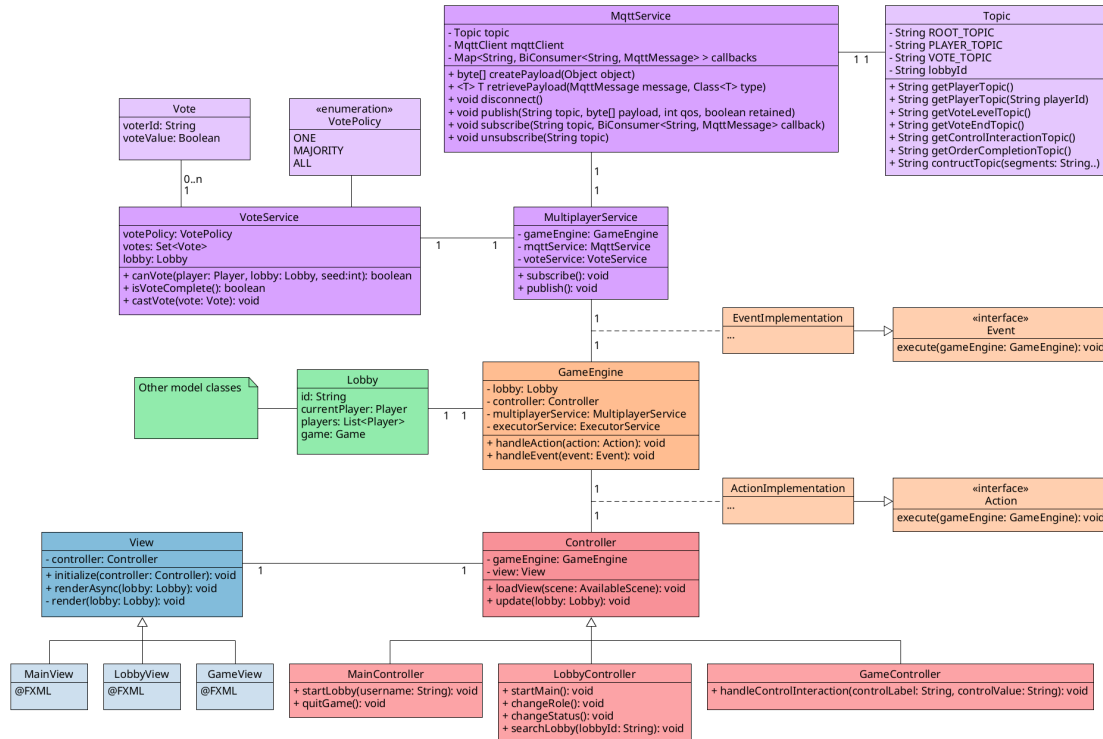


Figure 13: Class diagram of the architecture implementation

4.2.3 Design Patterns

During implementation, design patterns are used to improve the code quality, modularity and maintainability.

The Command pattern, for instance, is used to implement multiplayer Event and user Action classes. An example of an Action is the "join lobby" functionality, which is executed in response to a user input of a lobby identifier. The following code snippet gives an overview on how an Action is constructed using the Command pattern.

```

1 public class JoinLobby implements Action {
2
3     private String lobbyId;
4
5     public JoinLobby(String lobbyId) {
6         this.lobbyId = lobbyId;
7     }
8
9     @Override
10    public void execute(GameEngine gameEngine) {
11        Lobby lobby = new Lobby(gameEngine.getLobby().getCurrentPlayer(),
12                                lobbyId);
13        // Code to load the View and Controller for the lobby menu
14        // Subscribing to joined lobby topics
15        // Publishing the current player information in the lobby
  
```

```
15     }
16
17 }
18
```

Another pattern used in this application is the Builder pattern, which is applied to dynamically construct different GUI elements required to provide a randomized control panel for each different game. This approach streamlines customization of JavaFX Nodes, reduces code duplication, and automatically applies default properties consistently across the graphical user interface elements, resulting in a more organized and efficient setup of the UI.

The following code shows how the pattern is used to create a customized Label to use in the Graphical User Interface.

```
1 public class FXLabel {
2
3     private String text;
4     private String styleClass;
5
6     public FXLabel() {
7         this.text = "";
8         this.styleClass = "";
9     }
10
11     public FXLabel setText(String text) {
12         this.text = text;
13         return this;
14     }
15
16     public FXLabel setStyle(LabelStyle style) {
17         this.styleClass = style.getStyleClass();
18         return this;
19     }
20
21     public Label build() {
22         Label label = new Label(text);
23         label.getStyleClass().add(styleClass);
24         label.setEllipsisString(".");
25         return label;
26     }
27 }
28
29
```

4.2.4 Serializers/Deserializers

The game application uses extensively serializers and deserializers to load all available command from a file and encode messages between nodes. By using Gson `GsonBuilder`, custom type adapters can be registered, enabling seamless conversion between Java objects and JSON. This ensures that data is correctly formatted and interpreted when

converting objects to and from the JSON format. The following code demonstrates how these type adapters are registered to achieve consistent serialization and deserialization.

```
1 public static Gson createGson() {
2     return new GsonBuilder()
3         .excludeFieldsWithoutExposeAnnotation()
4         .serializeNulls()
5         .registerTypeAdapter(Player.class, new PlayerSerializer())
6         .registerTypeAdapter(Player.class, new PlayerDeserializer())
7         ...
8         .create();
9 }
10
```

The code below provides an example on how to implement a custom serializer and deserializer for the Player class of this application. This approach ensures that specific properties of the Java Objects are accurately interpreted when converting from and to JSON.

```
1 public class PlayerSerializer implements JsonSerializer<Player> {
2     @Override
3     public JsonElement serialize(Player player, Type type,
4     JsonSerializerContext context) {
5         JsonObject jsonObject = new JsonObject();
6         jsonObject.addProperty("id", player.getId());
7         jsonObject.addProperty("username", player.getUsername());
8         ...
9         return jsonObject;
10    }
11 }
12 public class PlayerDeserializer implements JsonDeserializer<Player> {
13     @Override
14     public Player deserialize(JsonElement jsonElement, Type type,
15     JsonDeserializationContext context) throws JsonParseException {
16         JsonObject jsonObject = jsonElement.getAsJsonObject();
17
18         String id = jsonObject.get("id").getString();
19         String username = jsonObject.get("username").getString();
20         ...
21         return new Player(id, username, ...);
22    }
23 }
24
```

After defining and registering these serializers and deserializers, they can be used with the methods `toJson(object)` to convert an object to its JSON representation, and `fromJson(jsonString, classOfT)` to reconstruct an object from a JSON string.

5 Self-assessment / Validation

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

Regarding quality, the system meets all the non-functional requirements specified. The game application successfully operates across all designated operating systems, and it works seamlessly in a distributed environment without requiring additional configurations of the game application, particularly for what concerns network configurations. The Graphical User Interface is responsive on moderately performing machines, providing user feedback within a few milliseconds, and always within one second from the user inputs.

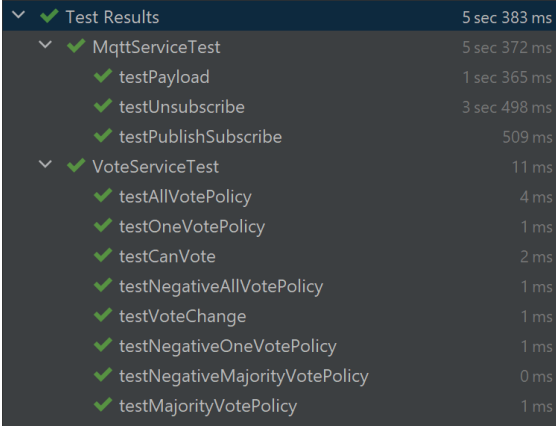
All implementation requirements are fully satisfied, using successfully all the technologies identified during the design phase.

In terms of effectiveness, all functional requirements are checked and met. Manual tests were conducted to evaluate fault tolerance features through integration testing, demonstrating that node failure were manages correctly according to predefines rules.

The scenarios tested included:

- Joining a lobby after other players were already present;
- Disconnecting a node during lobby setup;
- Joining a lobby after the game had started;
- Joining an already active game;
- Disconnecting a node during gameplay.

For critical components, automated tests were implemented using JUnit Jupiter to streamline the testing process and validate the code under different scenarios. Figure 14 presents the results of these tests, which confirm the correctness of both the Voting Service and the Mqtt Service for the analyzed functionalities.



Test Results	5 sec 383 ms
MqttServiceTest	5 sec 372 ms
testPayload	1 sec 365 ms
testUnsubscribe	3 sec 498 ms
testPublishSubscribe	509 ms
VoteServiceTest	11 ms
testAllVotePolicy	4 ms
testOneVotePolicy	1 ms
testCanVote	2 ms
testNegativeAllVotePolicy	1 ms
testVoteChange	1 ms
testNegativeOneVotePolicy	1 ms
testNegativeMajorityVotePolicy	0 ms
testMajorityVotePolicy	1 ms

Figure 14: JUnit tests

6 Deployment Instructions

This project requires two components to operate: an MQTT 5.0 broker and the game application. Below are the instructions for setting up and using these components to fully utilize the system's functionalities.

6.1 MQTT Broker

There are two options for the MQTT broker component: use an already available MQTT 5.0 compatible broker, or self-host the MQTT broker.

- **Using an Available Broker**

Several brokers are available online for free testing purposes. Choose one that supports the MQTT 5.0 protocol and note its address.

This project utilizes the standard MQTT port, 1883 over `tcp`; however, alternative connection methods that provide enhanced security are also available.

For this project, a suitable remote broker is configured and set up for testing the game. To gain access, send an email to `riccardo.omiccioli@studio.unibo.it`.

- **Self-hosting the broker**

Requirements:

- Docker
- Docker Compose

To start the MQTT broker, navigate inside the `emqx` directory and run the command `docker-compose up`. Optionally, the `-d` flag can be added to run the container in detached mode.

(Optional) Remote deployment: For a remote deployment, log-in into a remote server, clone the project repository and follow the same instructions used for local deployment.

However, certain network configurations on the server may be necessary to enable internet access to the broker, especially for accessing the dashboard functionality. In this project the broker is deployed on a remote Linux server, with *NGINX* configured as a reverse proxy to route connections to the appropriate container. The following is a template for the configuration used to enable external access to the dashboard.

```
1 server {
2     server_name emqx.domain.com www.emqx.domain.com;
3     listen 443 ssl;
4     ssl_certificate ../../certificate.cer;
5     ssl_certificate_key ../../private_key.key;
6     location / {
7         proxy_pass http://localhost:18083;
8         proxy_set_header Host $host;
9         proxy_set_header X-Real-IP $remote_addr;
```

```
10     }  
11 }  
12
```

Once, configured, the dashboard can be accessed remotely via the specified domain. However, this feature is not essential for the system's operation and serves primarily as a monitoring tool.

6.2 Game Application

Requirements:

- Java 21 Runtime Environment

There are two options for running the game application: use the provided bundled *JAR*, or run the project using *Gradle*.

- **Using the provided JAR**

To start the game application, navigate inside the `spaceteam` directory and run the command `java -DmqttBrokerAddress=tcp://127.0.0.1:1883 -jar Spaceteam.jar`.

When using a remote broker, replace the loopback address `127.0.0.1` with the appropriate domain.

- **Using Gradle**

To start the game application, navigate inside the `spaceteam` directory and run the command `./gradlew run` on Linux or `.\gradlew.bat run` on Windows.

To specify a remote broker address create a `gradle.properties` file following the example provided in the `gradle.properties.template` file, replacing the loopback address `127.0.0.1` with the appropriate domain.

7 Usage Examples

Upon launching the game, the main menu is displayed, as shown in Figure 15a. In this menu, users can modify their username through the provided input field, adjust the game window size, and set the maximum frame rate for game animations.

Random controls are available for testing and familiarization with the game controls. Additionally, buttons for initiating a lobby search and quitting the game are provided respectively in the center top and bottom positions. Once a lobby search is initiated, the game automatically connects the current player to a random lobby, with the lobby identifier visible in its text field, as illustrated in Figure 15b. Players can join other lobbies by entreing the same identifier used by other players in this text field.

The lobby menu also includes buttons for returning to the main menu, changing the player's role, and setting their status to ready in order to start a game. In the center of the lobby menu, a list displays the players currently in the lobby, summarizing the lobby's status, including the total number of players and how many are ready to play.



(a) Main menu



(b) Lobby menu

Figure 15: Main menu and lobby menu of the game

Figure 16 illustrates two instances of the game in which two different players have joined the same lobby. The image also shows the distinct roles chosen by each player, as well as their readiness status.



Figure 16: Example of two players in the same lobby

Once both players have set their status to ready, the game begins. Figure 17 illustrates the game interface, which displays the current game status along with the randomized control panels unique to each player.

The figure also presents the different view for the two roles: the gampleay perspective of the captain role on the left and that of a crew member on the right. The captain receives random orders, which may refer to their control panel or those of other players. Below each order, the remaining time to complete it is displayed. Additionally, various information regarding the current status are available, including the current level, the spaceship's integrity, the total number of players, and the count of completed orders followed by the required number for level completion.



Figure 17: Example of two players in the same game

Figure 18 shows the game over status following a loss. In this instance, spaceship's integrity has fallen to 0%, caused by inability to successfully complete orders within the available time.



Figure 18: Example of game over screen after a game loss

Figure 19 provides an interesting view of the game, featuring a third player joining a lobby where two other players are already engaged in a game. The image displays the views of the two active players, while the third can observe the lobby status, indicating that two players are currently in the gameplay phase.



Figure 19: Example of three players in the same lobby, after two players started a game

Once the third player sets their status to ready, they are placed in the game, awaiting the start of the next level before being able to participate. Figure 20 illustrates this scenario.



Figure 20: Example of a third player joining a started game

Finally, when a new level starts the third player can join the gameplay. This is depicted in Figure 21. It is important to note that the player on the left and the player on the right are both captains, while the player in the center serves as a crew member. The game allows players to select their desired roles, provided there is at least one captain for each game.



Figure 21: Example of three players game after joining a started game

Figure 22 and Figure 23 display information from the broker dashboard, illustrating the status during multiplayer games. In addition to the provided examples, additional details are available for each connected client, as well as for every topic, subscription, and retained message, enabling comprehensive monitoring of the system.

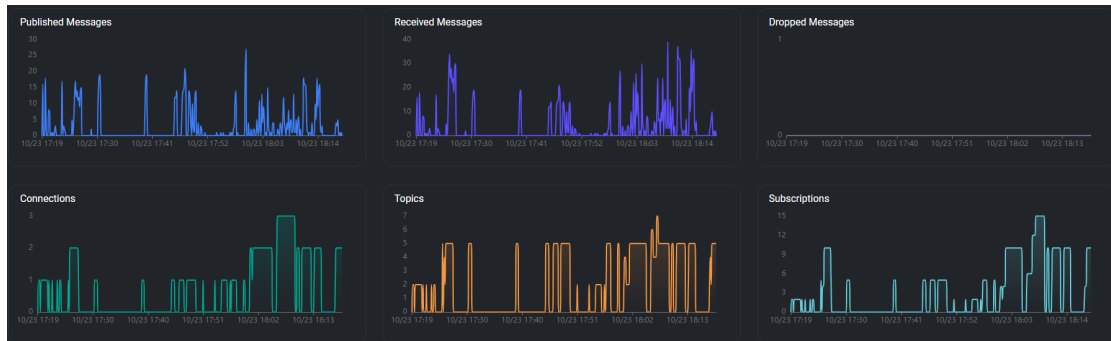


Figure 22: Dashboard view for the remote broker

Bytes		Messages	
Bytes Received	5662	Published Messages	12
Bytes Sent	9793	QoS 0 Messages Published	3
		QoS 1 Messages Published	9
		QoS 2 Messages Published	0
		Dropped Published Messages	0
		Dropped Published Messages (Expired)	0
		Messages Received	38
		QoS 0 Messages Received	9
		QoS 1 Messages Received	29
		QoS 2 Messages Received	0
		Dropped Incoming Messages	0
		Dropped Incoming Messages (Expired)	0
		Dropped Incoming Messages (Queue Full)	0
		Dropped Incoming Messages (Oversize)	0

Figure 23: Client details available in the broker dashboard

8 Conclusions

In this project, a complete and working distributed system was developed featuring real-time communication between distinct applications while implementing some logic with a shared status. Even if this project implements a simple multiplayer game, the concepts and technologies employed are easily adaptable to other projects in different fields.

8.1 Future Works

While the project completed its primary goals, there are several areas for future improvements. One aspect that was not explored during this project is an in-depth performance analysis when a large number of clients are connected simultaneously. This evaluation could provide interesting data into the system's scalability and availability under challenging scenarios. Additionally, the gameplay experience could be improved by expanding the data exchanged between nodes, which would allow for the creation of more complex and engaging situations for the players.

8.2 What did we learned

The development of this project has proven useful to understand and applying the principles of distributed systems, particularly through the use of a publish-subscribe model enabled by the MQTT protocol. This well-established and effective technology is very useful for creating reliable and lightweight communication frameworks to allow communication between multiple clients. Additionally, challenges encountered in managing a shared status among distributed nodes provided a valuable insight into the complexities of distributed systems.