Dixit - Distributed System

Author
daniele.dilillo@studio.unibo.it

March 5, 2024

Contents

List of Figures

1 Goals/requirements

The main goal of the project is to create a clone of the so called Dixit card game, in a distributed fashion and based on actor oriented paradigm. The game is made by lot of users interactions, with simple mechanisms explained below.

1.1 Rules

Each card depicts a colorful drawing, more or less imaginative, each with its own context but multiple interpretations. Each player starts with a finite number of cards in its hand. Each player takes turns playing the role of narrator (for that turn), while the other players play as guessers. The narrator (or storyteller) choose a card from its hand of cards, places it "on the virtual table" hidden from everyone, and propose a coherent title that must be revealed to everyone. The title can be any type of string. After that, each player choose a card from its hands that fits as much as possible the title proposed by the narrator. Once all the n-1 (with n number of players) players have chosen their card, all of these n-1 cards plus the narrator's card will be mixed up and revealed to all guessers. From this moment of the turn takes place the guessing phase: all guesser will choose the card they think is the narrator's one, excepting the card proposed by each guesser to match the title (in the first game phase). After that, the points are assigned with precise rules:

- If each player voted the narrator's card, it means that the title was too specific for that card. Each player gains 2 points, while no points for the narrator.
- If no player voted the narrator's card, it means that the title was too incoherent for that card. Each player gains 2 points, while no points for the narrator.
- If some players voted the narrator's card, but not everyone, the narrator gains 3 points, as many as each player who chose the right card. All the other players who didn't guess correctly, will receive a bonus point for each vote received from their proposed card.

The game end when one or more players earns at least 30 points: in case of a tie, the win will be shared.

1.2 Interviews

Below are some short questions on a simulated interview to analyze some detailed questions about the game and the mechanisms to be developed.

- **What are the different roles expected in the game?**
  The narrator is the role of the player who choose the card and the relative title to be guessed by the other players and change each turn . The guesser is the role that tries to guess the card chosen instead, given only the title written by the narrator and the set of cards selected by all guessers to match the title. Obviously, guessers

have to guess between the cards proposed by the guessers plus the card chosen by the narrator.

- **What happens if a player tries to join a full game?**
  If a player tries to join a full game, that means an already started match with the correct number of players, he should be notified that the match is full and kicked out

- **What about title? Are there specific rules?**
  The title should be whatever the narrator wants: any type of string is allowed. Obviously, a too fitted title should be avoided, as it could lead all the guessers to guess correctly the card and no points for the narrator. Otherwise, if the narrator choose a too general title, all the players might feel confused and vote for anything but the right card, again with no points for the narrator. The title should be as specific as mysterious in order to open a good debate about the card to be guessed.

- **What happens if a player choose its own selected card in the guess phase?**
  It must be forbidden: a guesser should not be able to guess his own card, as this would result in a free point for that guesser.

- **Are there any turns order rules?**
  No, there are no order rules, the only thing is that for each turn the narrator must be changed, and a player to embody the narrator role again must wait that all the other players had been narrator, like a circular round robin.

1.3 Scenarios

All users will be able to play together using a simple user interface, that can allow them, in a intuitive way, to create a new match, to visualize the open matches and to join one of them. All the in-game operations can be performed after login, so it is necessary the registration on the system with a unique username and a password. Once the player has chosen the game (or created a new one) will wait in the "lobby" of the match until the correct number of participants will be reached. Once the number of players is reached, the game can begin, in which each player takes turns playing as narrator, therefore choosing their own card and title and letting the other players propose theirs and guess that of the narrator. The game ends when one or more players reach 30 points (or after a defined number of turns).

Use Cases

Below (image 1) are represented the basic use cases between the player client and the system, more precisely the server that is identified as the access point to the system.

A user player, using the client, is able to register a new account and login to an already registered one, see all the open games waiting for players, request to join a match and create a new one. Finally, the user is able to see his total points on the user interface.

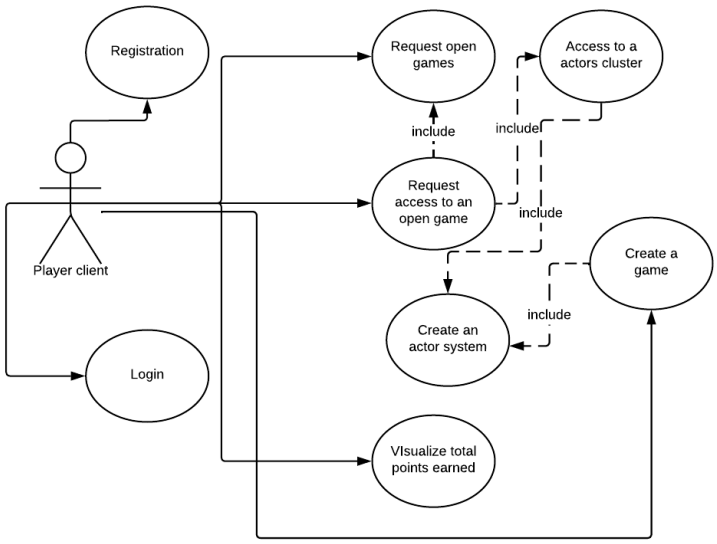

Figure 1: Client use cases

Details on turns

This is a turn-based game, so let's focus on how does the turn work, as anticipated previously. Referring to the image 2:

1. The narrator starts selecting a card and a title;
2. then he will show up only the title to all players, that impersonate the guessers for that specific turn;
3. all the guessers propose a card from theirs that is as similar as the title meaning;
4. the narrator will collect all the proposals and mix them up with the chosen card, showing them to the guessers;
5. finally, each guesser tries to guess the narrator's card, except their proposal, and points are updated;
6. the next turn will change narrator in a circular way.

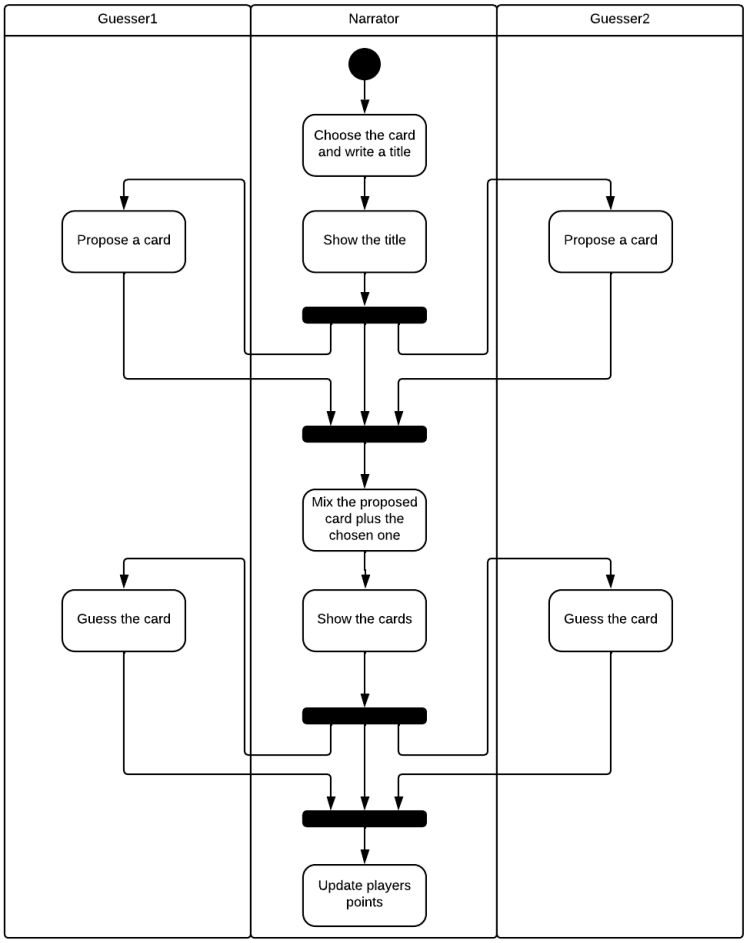

Figure 2: Macro activity diagram

1.4 Self-assessment policy

- How should the *quality* of the *produced software* be assessed? With regard to the quality of the project, it has been used the SBT build automation tool, that permits to automate all the dependencies and tasks declarations, in a declarative way. In particular, the following functionalities were used:
  - Plugin for Conventional Commits, in order to force a correct commit message format, following the convention: if the message does not comply the standard rules, it is canceled. This mechanism guarantees a better story readability of the project and give a fast and clear idea of the changes applied
  - Continuous Integration/Continuous Deployment: I chose to use the CI/CD GitHub's pipeline, the GitHub Actions, to run all the test and release jobs in an automatic way on the repository
  - Plugin Scoverage: this plugin permits to have a clear idea of the global coverage level derived from all the tests produced inside the project. This is made available through a GitHub Page specifically deployed for the repository, as we will see later
- How should the *effectiveness* of the project outcomes be assessed? The project outcomes will be assessed through a test suite composed by automatic and manual tests:
  - Automatic Tests: the automatic tests are designed to verify the system operation in different execution scenarios, as we will see later. These tests are aimed to verify the correct execution and the production of the expected results, especially with regard to the actors behavior and their reactions to received messages during a scenario execution
  - Manual Tests: manual tests are fundamental to verify the correct operation for components that are not automatable, such as user interfaces and the global operation with real user inputs

2 Background

This section will discuss some initial questions about the paradigm and the approach to choose for the distributed design model to use and the requirements of the project. The main goal is to experiment the actors distributed paradigm and apply it to model a turn-based game, analyzing the pros and cons of such approach applied to highly interactive games.

2.1 Analysis on paradigms

Let's now focus on the paradigms that can be used to achieve the project goal. Firstly, different patterns and paradigms has been taken into account before the proposal:

- Using Java Sockets to create a P2P interaction network: this is probably the simplest interaction mechanism to use between clients, but it may bring to a very complex system, because each player should have a double connection established between him and the other players, with no event manager shared channel, and raw data transmission (low-level abstraction). This mechanism does not model any type of encapsulation of behaviors, requiring more implementation effort.

- MQTT protocol based system with publish/subscribe pattern: a better choice than sockets, topics and messages can be used to simulate events on different channels, requiring duplex channels between each player and the central foreman. A valid choice, even if it will result in a more complex system for a turn-based game, because of the more "event notification" oriented style and not asynchronous request/response. Moreover, the strong decoupling offered by a pub/sub protocol doesn't fit very well a turn-based game, where actors are strongly coupled

- Actor based system: this should be a perfect trade off between the level of complexity of system (in terms of organization/communications) and the entity's states management, using a behavioral pattern and encapsulation of both state and behavior. Moreover, it offers a better scalability, distributing actors across nodes. Finally, we have better built-in fault tolerance mechanisms, such as actors supervision

Before starting analyzing design and implementation aspects, let's focus a bit on the background of the project and the technologies that have been used to accomplish the project goals. Why using the Actor programming paradigm? The idea is to try to make a game using a message oriented paradigm, where all type of interactions and entities are modelled as actors, so that everything is an actor. The main concept is that an actor can independently:

- define behaviors able to receive a subset of all the available messages it could receive (filtering) and send messages to other actors it knows

- create new actors

- do something according to a determined message received and a determined behavior

This could certainly bring a bit more complex data structures and entity organization, but the way each entity interact should be perfect for this type of game, where all players, for example, are modelled as actors, and the interactions between turns as messages. This should also bring to a more distributed system, based obviously on asynchronous interactions and non-blocking behaviors. The development process of the project is made of a mix of different technologies, explained in the following:

- Akka Typed Actors: the framework used to create actor systems in a simple and transparent way, with many mechanisms to facilitate actors discovery and clusters managing

- Akka gRPC framework: the framework used to create a client-server application using the Protobuf protocol
- Akka Asynchronous Testing: framework used to create testing environments where to test predefined scenarios between actors
- Docker and MySQL: used to create a containerized MySql environment server side and the relative phpMyAdmin application
- Test Container: framework used to test the correct operation of the container used for the database
- Clean Architecture: design pattern used to organize the server side business logic and separate it from technical aspects

3 Requirements Analysis

Let's analyze now the requirements of the project, from functional ones to non-functional, hypothesis and assumptions to be taken into account to design a game using an actor paradigm and a possible evaluation with different paradigms.

3.1 Functional Requirements

- Client-server architecture to manage the creation of a new game, the possibility to join open games with other players entering its lobby, store data in a persistent way
- Implement a simple login/registration mechanism, with a unique username and a simple password
- Cluster of actors systems to organize and manage a single match, composed by a foreman actor (that is the turns manager) and all the players
- A player must communicate only with the foreman, who orchestrate the game, and not with the other players
- Failure management: if a player exit the game, for whatever reason, the game should wait a defined amount of time in order to give the exited player the possibility to rejoin (invalidating the current turn) or terminating the game if timeout occurs
- If the foreman player (the one who created the game) exits, the match must be terminated
- When the match ends, the foreman player communicate the points gained to each player, and each player should then communicate with the server to update its points
- A single game has the name of the player who created it as its access address

3.2 Non-functional Requirements

- High scale performance
- Good performance in events listening, such as players exit reaction or join request
- There are no security requirements, as the project does not share personal data or does any kind of personal operation, the main goal is to develop a simple distributed game with the focus on the distributed communications
- The system must be designed so that maintenance and evolution can be easily managed
- Intuitive and good usability of the user interface

3.3 Architectural Requirements

The system will be a mix of client-server architecture and actors paradigm based architecture.

- The client-server is composed by the server and docker services from one side, and the client from the other side, that are responsible of the match creation and join operations. The client will contact the server to receive the open games addresses and to update its points after a game (and obviously login/registration). The join request is sent directly to the cluster of actor systems instead, to join as a new member
- The cluster of actor systems is based on an actors based architecture, where multiple actors spatially distributed are grouped together to compose a match. A match, in fact, is a group of actors that communicates and play together, without interactions with the server, that manages only the persistent data

Let's now identify the main roles to model in a first analysis:

- Foreman: the actor that is responsible to manage the turns and orchestrate the players, for example requesting to select a card or sending the set of cards
- Player: the effective player, that is able to impersonate the narrator or the guesser during a turn
- Narrator: the narrator must choose the card to be guessed by the other players (guessers) and the relative title for that turn
- Guesser: a guesser must select a card from its hand that is as much as similar as the meaning of the title chosen by the narrator. After that, he vote the card (from all the cards voted by the other players plus the right one) that he think it is the narrator's one

4 Design

This section will talk about the design of the project, the pattern of all the components that are part of it and their interactions. Let's start analyze some user stories to define, with an informal and natural language, what are the main entities to investigate further and the basic interactions. As we can see, we can identify two entities firstly, the client

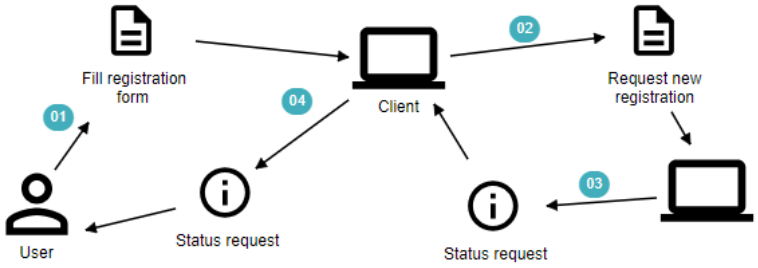

Figure 3: User registration user story

and the server: a user, with the means of his client, is able to perform a registration request within the system creating a new registration request document and send it to the server, that will notify if some errors occurred. Then, the user should be able to login to the server, in order to access the system. When the user is logged, he can

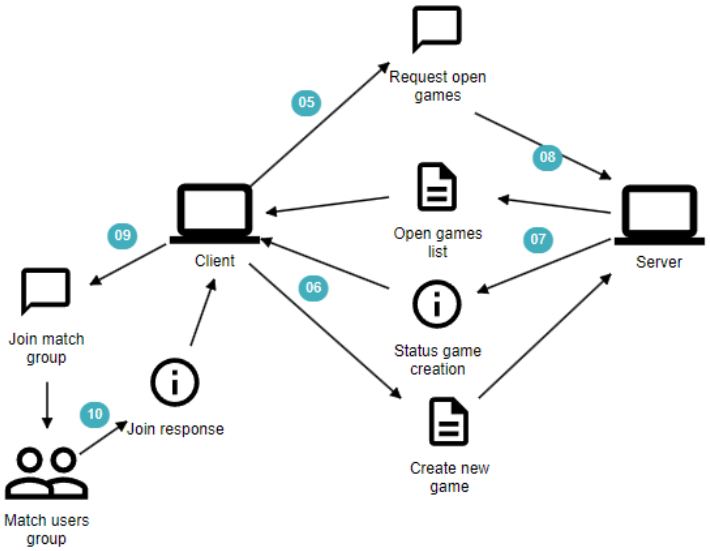

Figure 4: Create/join game user story

request the open games list for example, receiving them and let the user choose to join a game or create a new one. The game is represented as a unique users group, that will communicate independently from the server, in order to play the match.

4.1 Client-Server Design

Let's now analyze the design of the client-server architecture, that is essentially based on two components, literally the server and the client.

4.1.1 Structure / Domain Entities

The starting point is the structure of the components and the technologies to develop and integrate in order to allow the client to access the system by means of the server. Firstly, the structure of the server is modelled using a domain driven development, while the client is a simple component that sends requests and intermediate the user commands. The server design, that is the access point of the system, is based on the Clean Architecture (also called Ports&Adapter), modelling two nested layers and loosely-coupled interchangeable components:

- Business Logic layer: this layer, independent from the others, is responsible to model the business entities and logic, without any technical or technological implementation references, with a pure domain oriented core describing the model and the business services
- Application adapters layer: this layer contains all the technical components that implements a specific technology and does something aimed to exit from the core layer (going outbound) or get something in (going inbound). These adapters are connected to the core using so called "ports", that are specific protocols that permit inbound and outbound communications. Ports can be modelled as interfaces in an object-oriented language programming, that are a sort of contract to respect.

The hexagonal architecture design is visible in the image 5. We can distinguish different adapters:

- gRPC Akka service inbound: this is an inbound adapter, as the main role of this component is to redirect clients incoming requests to the domain logic, through the use case interface (inbound port), that is sort of implemented by this adapter. The port let the adapter do operations like login, registering a new account, requesting the list of open games
- gRPC Akka service outbound: this is an outbound adapter, as the main role of this component is to send responses from the domain logic to the external clients. This is made using an outbound port to manage responses: this port is an interface that is implemented by the outbound adapter and used inside the domain logic using a sort of inversion of control and dependency injection
- Adapter Data access: this outbound adapter connects the domain logic with a repository, that is essentially the way it is organized the persistent layer, in fact it can have multiple sources of data. More precisely, the persistent layer in this case is represented by the local database, and the adapter is the component that implements all the specific queries needed to interrogate and update the db. The

Access port is the interface implemented by the repository adapters, used by the business logics to access the persistent layer, defining the abstract APIs to access it and implemented by all the repository adapters and specific queries

A more detailed vision of the architecture proposed can be seen with the image 6, with a Components&Connectors diagram, where all the interfaces used and required are drawn.
The business logic is instead composed by the domain logic, that implements the use

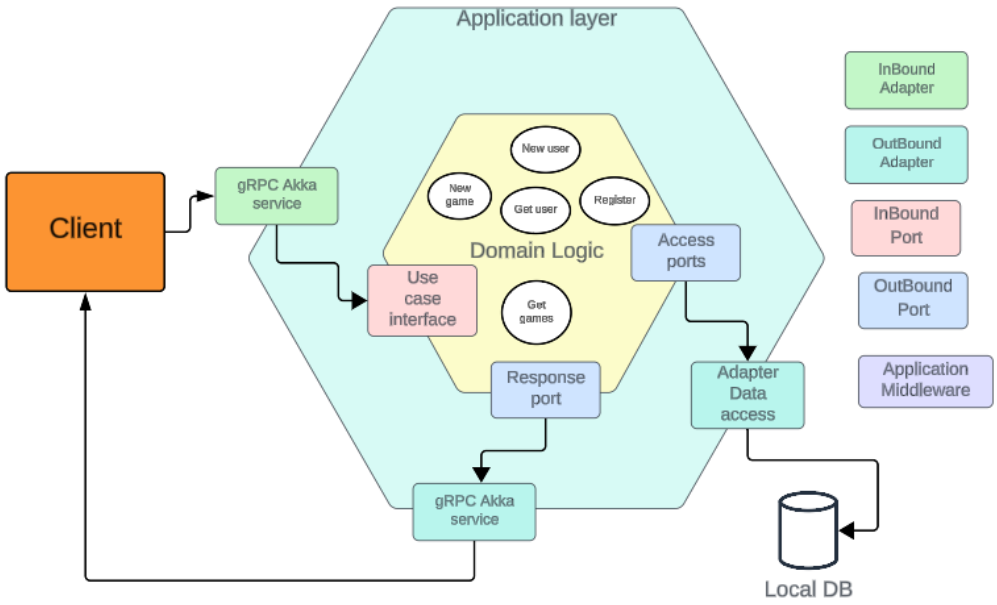

Figure 5: Clean architecture schema

cases with a domain level abstraction (login, registration, update users and so on) and the domain entities, that are essentially:

- The game: it models a match, composed by the address and username of the player that just created the game
- The user: it models the user registered into the system, that is composed by the username, a password and the total points earned by that user by playing

The package structure starts with the root grpcService package (7), that contains two macro-packages:

- client: this package contains all the required client's software components, such as the client implementation that connects to the server, the user interfaces and all the actors stuff (definitions, behaviors, messages)
- server: this package contains all the design entities and business logic of the server, including the server stub, services and data access adapters/ports

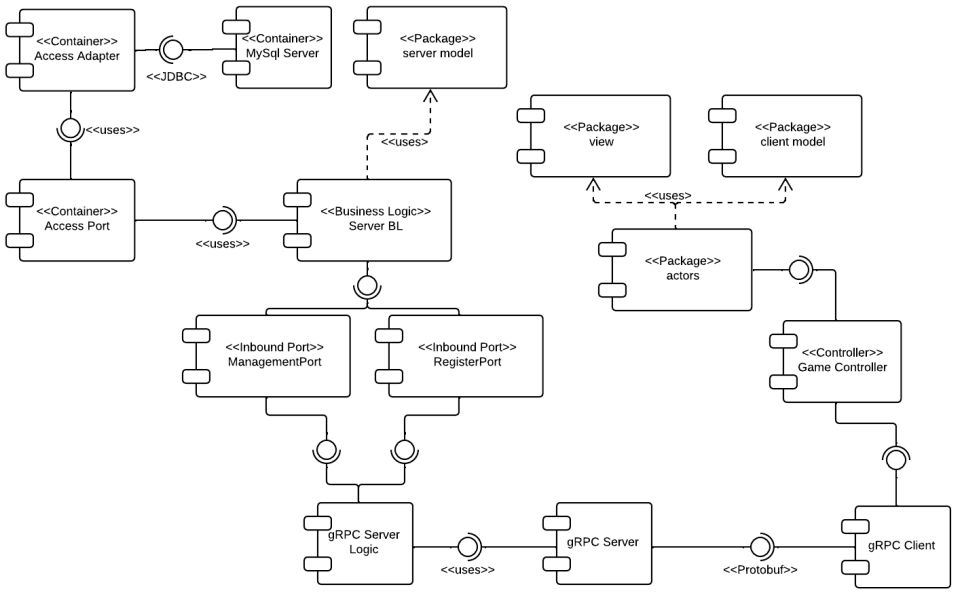

Figure 6: UML Components diagram of client-server

Inside the *server/adapters* package we find the service stub implementation to use within the gRPC server component, and a data type adapter to convert Proto generated object (according to the proto specification) to the domain model objects. Looking into the *server/applicationService* package we find the server service configuration, which specify for example the service implementation to use, the address/port binding and all the elements required by the business domain logic.

Let's now move the focus on the repository package, that contains all the components designed to access the persistent layer. Firstly, we have an AccessAdapter class, that is responsible of connecting to a MySql Server using a JDBC connection and managing all the queries execution, like selection or update, with appropriate policy checking before all operation. Then, we have three different builder classes: they are InsertBuilder, QueryBuilder and UpdateBuilder, that are responsible of simplifying all the different types of queries creation with a more DSL style. Using this approach, further controls are defined in order to avoid SQL injection or other type of errors. Finally we have an AccessPort, that is the access point for the domain logic to the repository: this model the data access from a more abstract view, in terms of use cases. It is then linked to an AccessAdapter, that manages the "physical" connection to a specific persistent layer technology. Then we have the *server/domain* package, that contains the business logic with use cases modeled, the domain entities/value objects and the interface to access the repository, that is implemented by the access port to be used (the already mentioned AccessPort). The client architecture is modeled with a simple MVC pattern:

- The view uses the controller to operate on model and the service adapter

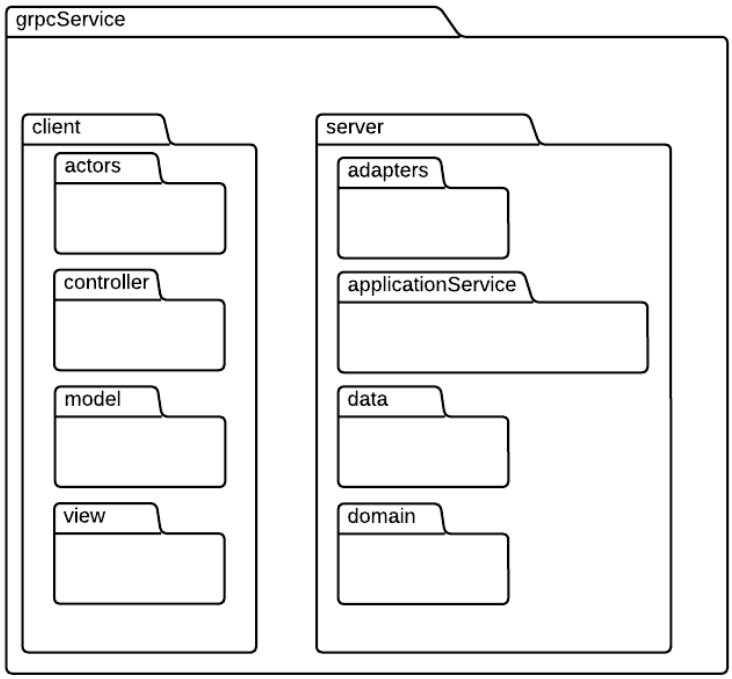

Figure 7: UML Packages diagram of client-server parts

- The model contains the representation of the model classes and extensions
- The controller, represented by the GameController class, is responsible of intermediating between the view and the client

The *client/controller* package, instead, contains the GameController used by the gRPC Client implementation. The GameController is also responsible of requesting login, registering actions and other use cases from the gRPC client. The view, instead, contains the main page view and the game view, both developed using Java Swing. The UML class diagram of everything that we just talked about is represented in the image 8.

4.1.2 Interaction

From a more simple interaction view, the system can be represented as the image 9 depicts. Essentially, we have the server component connected to the client components through gRPC protocol, implemented by the Akka gRPC Framework. Then, the server is connected with the dockerized MySQL local database, through the exposed port and the JDBC connection. The server and the dockerized database and DBMS are deployed in the same component node. The gRPC protocol with the Protobuf protocol defines the interactions and all the messages interchanged between the clients and the server, that essentially are: login, registering, create a new game and retrieve all the open games. The core advantage of this approach is that we can generate the server stub and

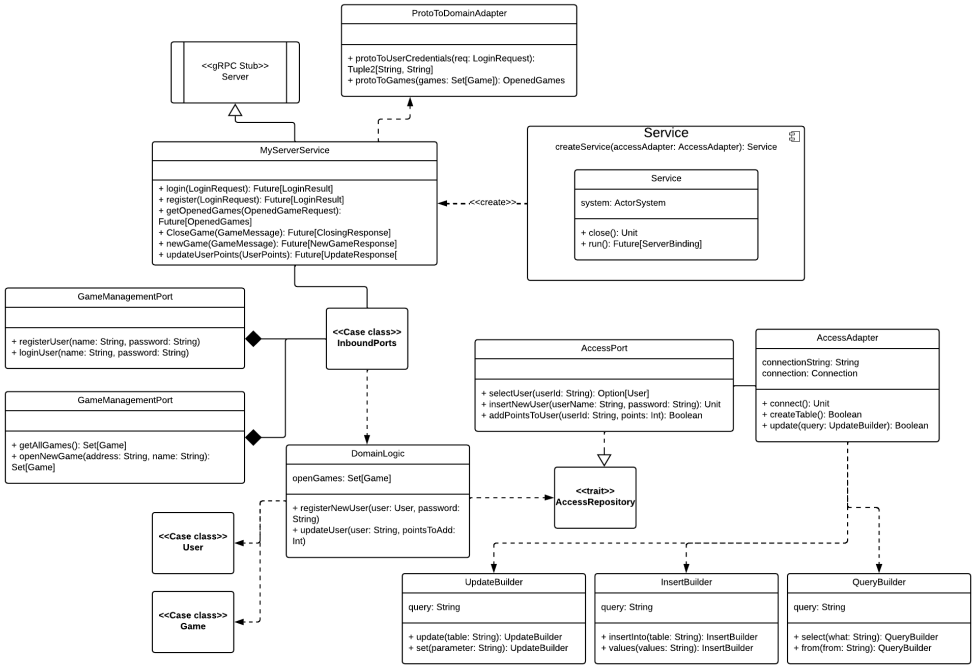

Figure 8: UML Classes diagram of client-server parts

messages classes concentrating all the efforts on the communications, better performance than REST APIs, language agnostic and with its own Interface Definition Language to define in a clear way all the messages and service interface. Other advantages are a rich error handling and a flexible integration with other technologies. Let's now dig into a more detailed view of interactions between client and server, with the following UML Sequence diagram 10. Essentially, the client starts the communication sending a register request, which implies the insertion of a new user (with the right checks before) into the repository database, that is realized as a requests chain through ports and adapters. The same thing is performed with a login request, with the only difference that there is a selection query performed on the database, to retrieve the user information. Finally, the client is able to update its personal points after a game termination: in this case, the server updates the user points in the database, always passing through all the designed components. Referring to the image 11, we can see the way a client joins a game or creates a new one. For the first action, the client requests all the currently open games waiting for players, than he will autonomously (with an appropriate user interaction) select the address of the match he want to participate, and join it. Otherwise, if the user wants to create a new game instead of joining an existing one, he will request the server to create a new open game sending its own address and username and, in case of positive response, will create a new cluster of actor systems.

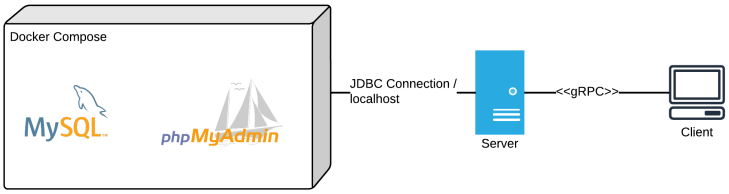

Figure 9: Macro deployment diagram of client-server

4.2 Actor Systems and Cluster Design

It's time to talk about the architecture and design of the actual game, where all players interactions reside and come out. All the advantages, described in the previous sections, leaded to choose a design based on actors and behaviors, they are the core concepts of this design architecture:

- Game entities modeled as actors: these actors are independent from each other and encapsulate their state and behavior, in fact they are stateful actors
- Actor lifecycle: it is fundamental, in a game, to represent a complex and dynamic state change into the cluster, simplifying the modeling of complex interactions. Lifecycle means the internal actor state into its cluster, that could be *joining*, *leaving*, *up* and so on for example
- Behavior: it is essentially the way to represent the personal state for an actor, defining the set of messages admissible and the behavioral operations to trigger when receiving such messages

4.2.1 Structure / Domain Entities

The structure of the actors environment is based on the concept of Actor System (AS), that is the way the Akka Framework manage the execution of actors. First, what is an Actor System?

- Physical point of view: an actor system is a heavyweight structure multi-threaded, so it will allocate N threads to manage the multiplicity of actors created
- Logical point of view: an actor system can be seen as a hierarchical group of actors, each with its own role, like a group of people

An actor is an object that encapsulate state and behavior, and are able to communicate exchanging messages: each actor receive messages and places them into a personal mail-box, in order to receive them in a sequential way. Finally, a Cluster in Akka Framework can be seen as multiple Actor Systems connected to each other. Practically, it is defined as a fault-tolerant decentralized peer-to-peer membership service between a group of

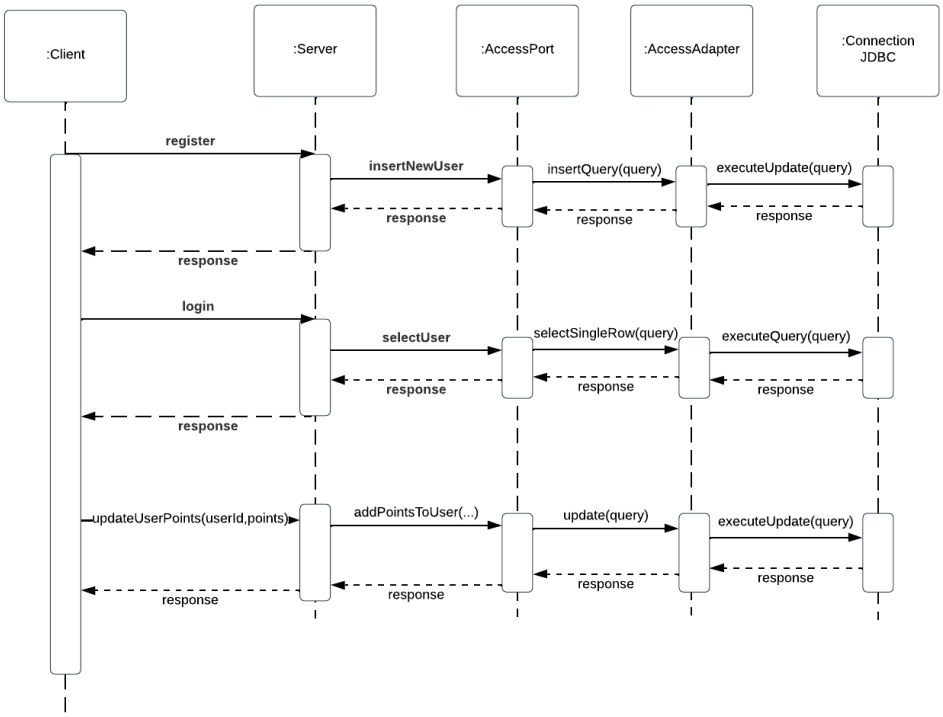

Figure 10: UML Sequence diagram of login

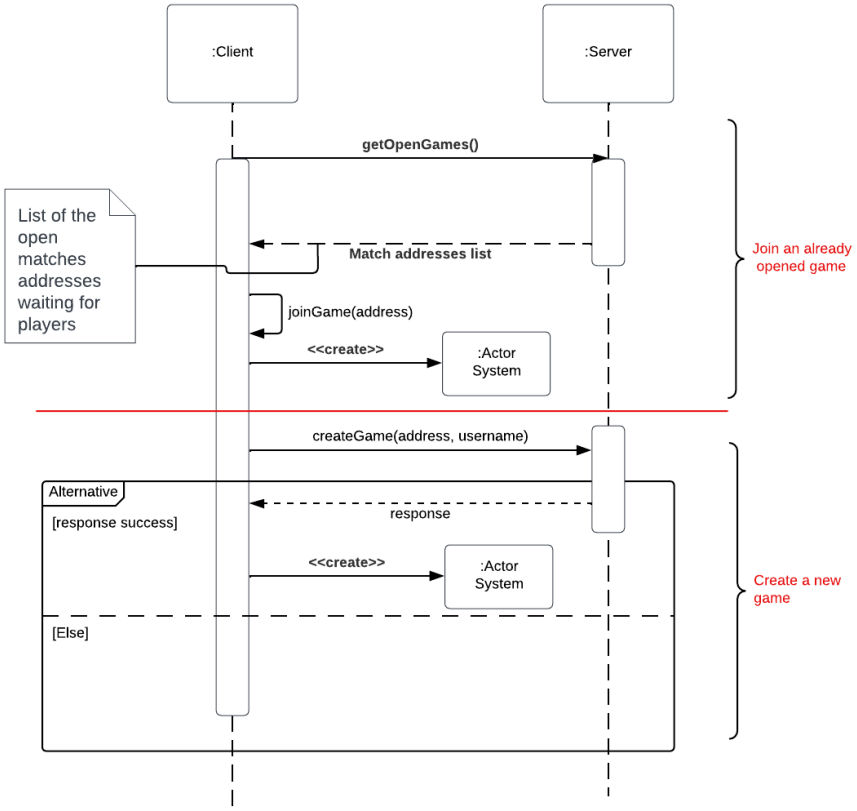

Figure 11: UML Sequence diagram of match creation/join

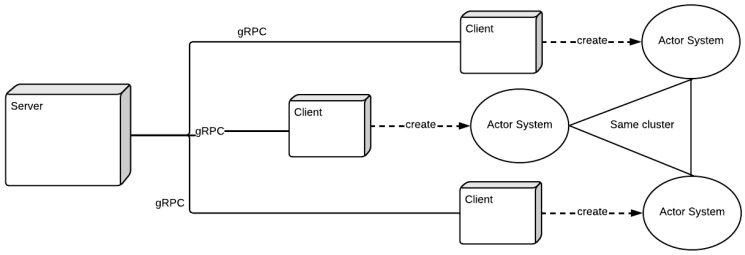

Figure 12: Client-server components and AS

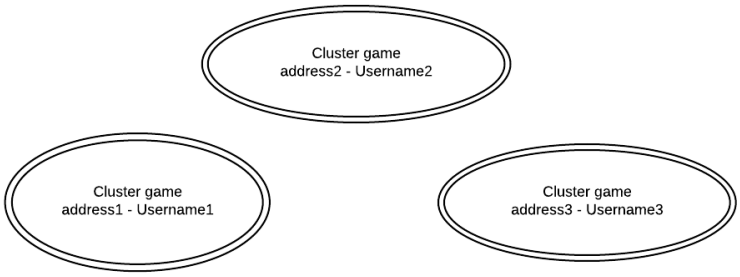

Figure 13: Example of three games opened

actor systems. More specifically, its name is Cluster Membership Service, with many characteristics. The most important to understand are the following:

- It keeps track of the nodes participating to the cluster
- It uses a Gossip protocol, where the current state of the cluster is gossiped randomly through the cluster. It helps to define the membership lifecycle
- Membership Lifecycle: it is the already mentioned actor lifecyle, so each node, from entering to leaving a cluster, changes states describing if it's joining or exiting the cluster or is unreachable or removed from the cluster, for example. A node leader (one per cluster) is responsible to set these states.
- Members Events: events to intercept (listening) to track the lifecycle of all members

All these functionalities are important to manage the sudden exit of a player and consequently its unreachability. Let's now dig into a more detailed vision of the design of these actor systems, starting from explaining, at a high level of abstraction, how these actor systems are linked to the client-server architecture previously described (12): the client who wants to join or create a new game, will create a new actor system to participate the game, modeled as a cluster.

A cluster is designed to represent a single match between its players, so basically, a cluster is created for each new game created, independent to each others (13).

Having said that, let's analyze the types of actors that take part to the game, their role and the hierarchy. Referring to Akka Framework, the actor model is based on the Actor Model defined by Hewitt, Bishop and Steiger, treating an actor as the basic building block of concurrent or distributed computation. As already described, an actor can send a finite number of messages to other actors he knows, create a finite number of new actors and define the behavior to be applied to the next message to receive. An actor is essentially composed by different things:

- Behavior: it is like a mapping between a subset of messages that can be processed by the actor and the actions to be taken in reaction to receiving such messages
- State: the state of an actor is defined by its values and current behavior, like a finite state machine
- The mailbox: it's the element that connects a sender actor to a receiver one, and works like a classical real mailbox, queuing messages time ordered. The default used is a FIFO mailbox, without any message priority. Custom mailboxes can be created to allow a well defined priority between different types of messages
- Child actors: each actor is able to spawn other new actors, creating a parent-child relationship between actors
- Supervision strategy: describes a set of strategies to handle failures
- ActorRef: it is the actor reference, that contains all the information about an actor to send messages to; the address, or Actor Path of a reference, is composed by the cluster name, host and port associated and finally the parent relationship chained like a path
- Roles: each node is marked with a role, that is of type *player* or *foreman*, this is useful to group actors that do the same things under a role, for example to discriminate the exit of a player or the foreman of the game
- User Guardian: is the parent of all user-created actors in an Actor System, the actor named "user". Each actor system has a user guardian actor, whose address is always recognizable as *akka://sys@host:port/user*, and all the children created in the AS will have the same path concatenated with the name of the actor spawned

All these concepts are important to understand how it works under the hood. Several actors have been developed to accomplish the final goal and to implement all the interactions needed, as we can see in the UML classes/modules diagram.

The actors developed divide into two categories, those who have an active role in the game and those who are specific event listener:

- **InteractionGuiImpl**: this actor is responsible of handling the user interactions
- **PlayerClusterListener**: this actor is responsible of listening and intercept all the cluster events concerning the player, like the exit of the foreman player

- **PlayerBehavior**: this is the root actor used to create the actor system of a player joining a game, it is responsible of spawning the InteractionGuiImpl actor, the effective PlayerBehaviorImpl actor, that is responsible of all the game interactions and gui actor coordination, and finally the PlayerClusterListener actor
- **ForemanBehavior**: this actor is responsible of managing the coordination between players, the game turns and all the preparations pre and post match, such as distributing cards and giving points
- **ClusterListener**: like the PlayerClusterListener, this actor is responsible of listening and intercept events that are interesting for the foreman, like the exit of a player for example
- **PlayersManagerBehavior**: this actor is responsible of managing players for the joining phase, accepting or not players that want to join

The parent-child relationship can be seen in the image 14. The spawn means a creation relationship between a parent and a child actor, while the watch relation is a way to listen child's events, as we will see. Finally, we can analyze how these actor systems are deployed in different remote nodes (15), communicating with *Akka Remoting Artery*: a single player joining a game executes only the player actor system, composed by the effective Player actor, the Interaction actor and the event listener actor. The player who wants to create a new game, will deploy a node with the player actor system (that operate as the other players) and the foreman actor system, composed by the effective foreman actor that manages the game's turns, the player manager that allow players to join the match, and the cluster events listener. The UML Class diagrams are shown in the following images. The first, 16, concerns about the client implementation class and the use dependencies between the gRPC client, the controller and the view, that is a sort of MVC pattern: all the user interactions on the user interface are redirected to the controller, that cooperates with the client sending requests. The second diagram instead, 17, concerns the actors behavior classes: they use the generic trait (the same of the java interface) to define more specific messages that can be received at any specific behavior. Note that the interaction behavior uses the main GUI of the game, that contains all the graphical elements to play: all the user events are handled by the interaction actor. The PlayerSelection case class define a card selection by a specific user, storing the card id and the actor reference of the player that chosen that card. Finally, the PumpMyList module is an extension method property that extends the List type object with new useful methods, like a FilterMap.

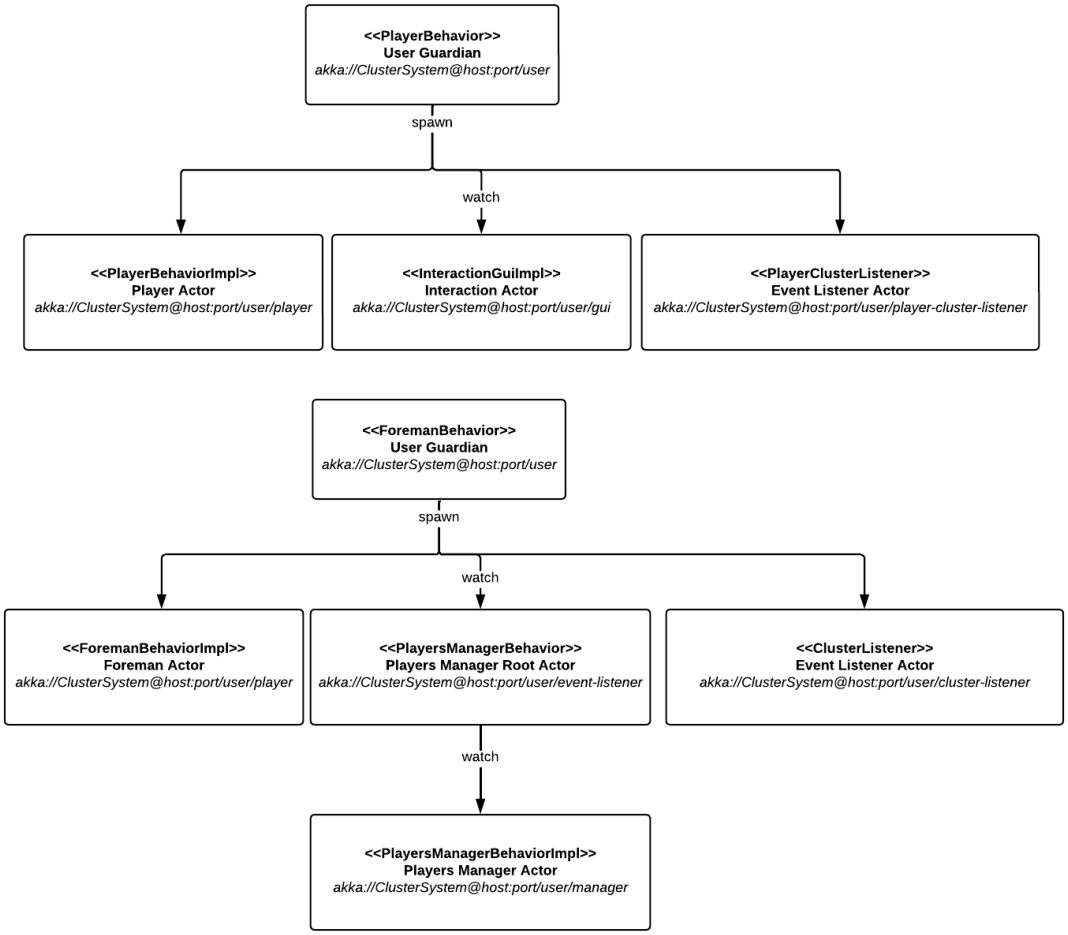

Figure 14: Relations between actors

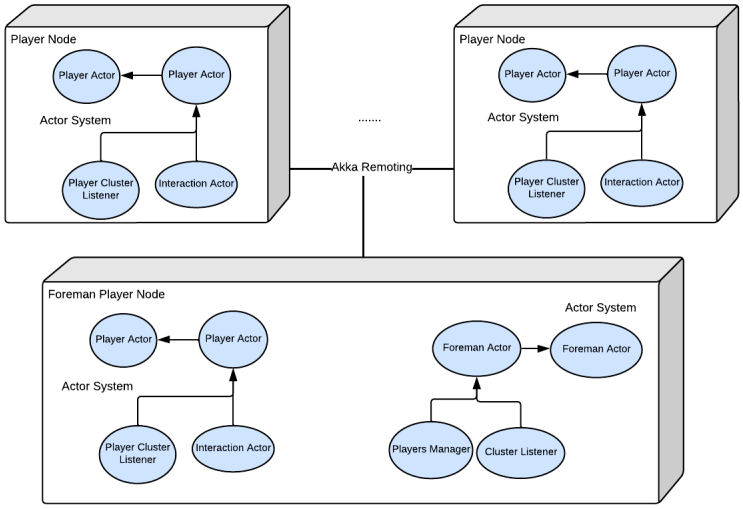

Figure 15: Macro deployment diagram of actor systems

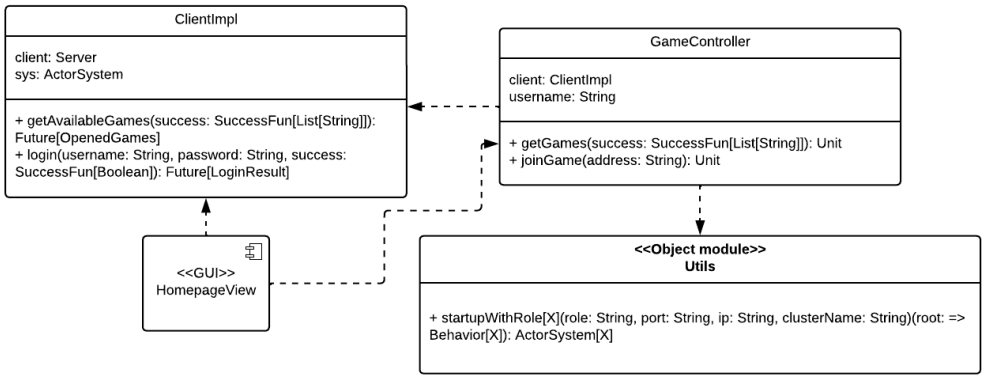

Figure 16: UML Classes diagram of the client

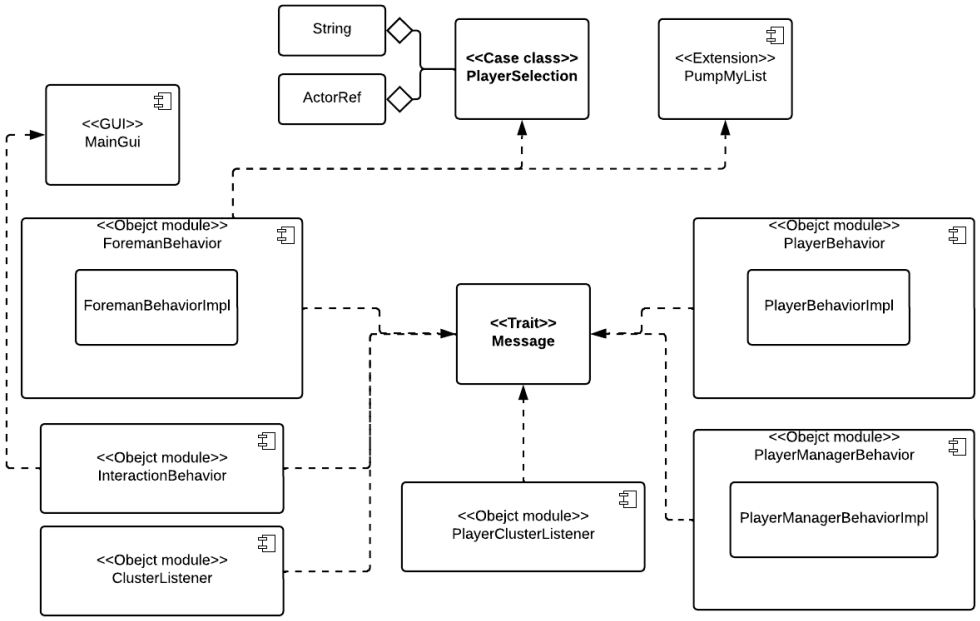

Figure 17: UML Classes diagram of actors

4.2.2 Behaviour

Let's now focus on how these actors behave in order to coordinate a match. Each actor change dynamically its behavior, based on the current state of the game and the requests received. We will see the fundamental and most significant actors, leaving out event-listener actors.

Foreman Actor

The effective foreman actor uses the ForemanBehaviorImpl class, that contains the set of states represented by the 18 image. When this actor starts, it means that the game has just been created, so it will set to a *Startup* state waiting for players to join. It is responsibility of the players manager actor to let the players join, counting them and sending the foreman the start message. So, basically, when the foreman receives the **Start** message, he will send a finite set of random cards to each player and goes to the manage turn state: this defines the state where to start a new turn, selecting the next narrator and waiting for the card to guess. When the card to guess is received by the narrator, he will tell the title to each player and go to the *wait for selections* state, waiting for all players to select their most significant card to that title. When all players sent its selection, the foreman communicates all the selections to each player and goes to the *wait for guessing phase*, where each player sends its card guess. It is important to note that actors exchange the id of the card they want to send. After this last phase, the points are assigned and the game proceeds with the next turn. The actor stops if a **Stop** message is received at any state, and the current turn is restarted if a **PlayerRejoined**

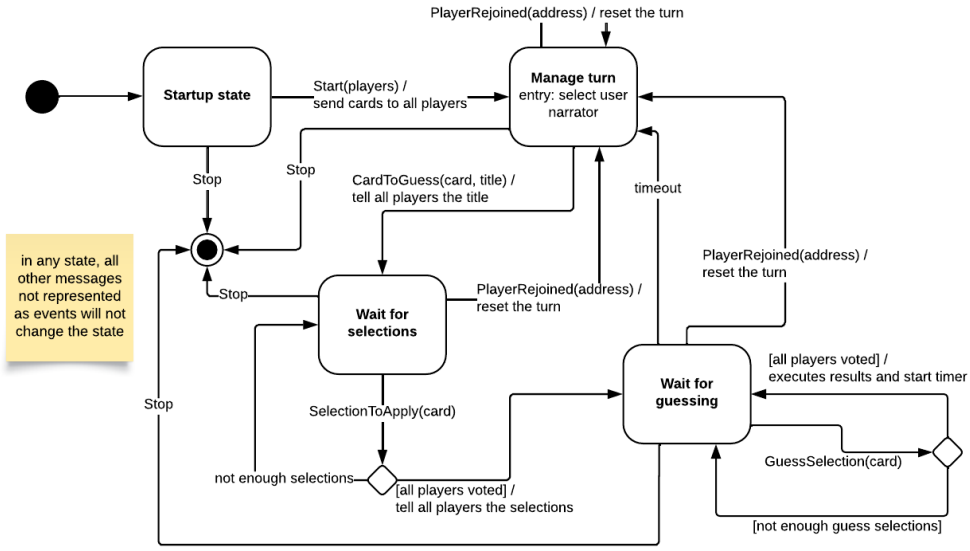

Figure 18: State diagram for the Foreman Actor

message is received at any state.

Player Actor

The player actor changes state depending on the foreman requests (19). When this actor starts, it goes to the *Startup* state, where it will wait for the OK message to join the game, or it will stop if receives a KO message. After joining the game, it goes to the *Wait for cards* state, waiting the foreman to give him cards to play. After card assignment, it goes to the central state, the *Game on*, where it can receive a new card or the role to play for the current turn. If the turn assigned is the narrator, it goes to the *Choose* state, where the actor choose a card (contacting the appropriate interaction actor) and sending the card and the title chosen by the user. If the player is assigned the guesser role instead, he goes to the *Guess* state, waiting for the narrator's title and choosing the most coherent card to submit. After that, he goes to the *Choose* state for the final guessing phase, receiving the players selections and selecting what the player think is the right card chosen by the narrator. All these states requires the interaction with the interaction actor, that simply catch all the user interactions with the GUI and send appropriate messages to the player actor.

Player Manager Actor

This actor behavior is much more simple than the player and foreman actors (20). There are two main states, after the startup phase:

- *Wait for players*: this define a state that means there are not enough players who have joined the game. A player who wants to join, sends to this actor a request

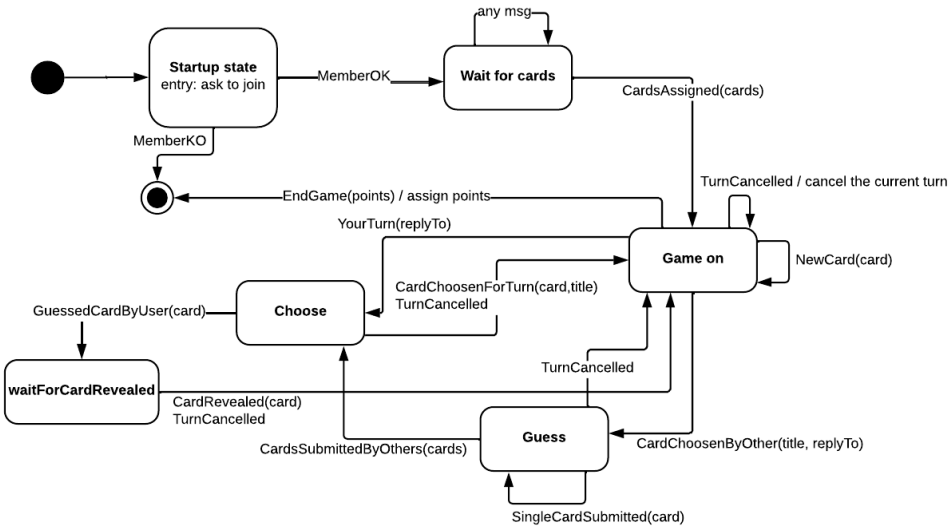

Figure 19: State diagram for the Player Actor

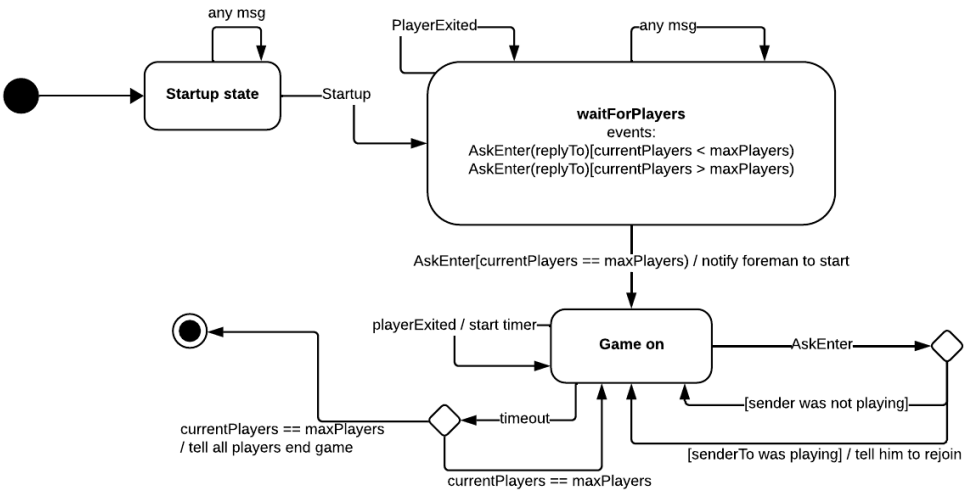

Figure 20: State diagram for the Player Manager Actor

to enter, so that it can count all the players ready to play. Otherwise, if a player exits, it sends a PlayerExited message.

- *Game on*: this state represents the game started, so that if a player exits it will starts a timer to wait the player/s to rejoin, otherwise a Stop message will be sent to the the foreman actor system

4.2.3 Interaction

All the actors interacts with each other sending messages and changing state based on what a specific actor wants to receive. All the messages extend the basic trait *Message*, adding specific information based on the meaning of the message object. Let's now show a subset of the messages exchanged by the different actors. Note that in the following lists are explained the different types of messages exchanged without the sender reference, that is the way an actor responds to a received message, and all the messages details. A set of the messages that a player actor is able to handle are:

- AskToEnter: message sent in order to get the confirmation to enter the game
- MemberOK and MemberKO: responses sent following a AskToEnter request message. MemberOK means that the player can participate in the match; MemberKO means that the player can not participate, due to a already full game
- GameInfo: message with information to be displayed to the user
- YourTurn: this message means that in that moment the receiver player is playing as a narrator choosing the card and the title
- CardsAssigned: this message contains the set of cards assigned to that player before the start of the game
- CardChosenByOther: this represent the title chosen by the narrator
- CardsSubmittedByOthers: list of cards submitted by all the other players
- and many others...

A set of the messages that a foreman actor of a match is able to handle are:

- Start: message sent by the *PlayersManager* actor, meaning that the number of players is reached and the game can start
- CardToGuess: it is the card and title chosen by the narrator actor
- SelectionToApply: this is the card selected by each actor based on the title written by the narrator actor
- GuessSelection: this is the card selected in the guess phase by each actor from all the cards applied mixed with the narrator's one

- NewTurn: message that means the beginning of a new turn
- PlayerRejoined: a player rejoined after exiting the game
- RestartTurn: the turn must be restarted
- and many others...

The Sequence Diagram that shows off how many interactions take part in a turn, is visible in the 21 image.

Message Delivery Reliability

It is essential to understand the message delivery guarantee level and mechanisms provided by the Akka framework. First of all, by default, Akka provides a simple *at-most-once* delivery semantics, it means that for each message sent, that message is delivered once or not at all: it can be lost for any reason. It's important to remark that this framework uses an Asynchronous Message Passing model, so that actors send messages and do not block waiting for the response, but go on working. Last, but not least, is the message ordering: messages sent from one actor to another are guaranteed to be delivered in the order they were sent.

Message Ordering

We said that Akka performs a message ordering between actors, based on the send order: but what if we need to put priority into specific messages? There is the possibility of using prioritized mailboxes, that use specific priorities for particular messages, in order to manage a certain type of messages before others in case of simultaneous receptions. This is used, for example, for prioritize the Stop message over the others when an error occurs when a player exited does not reconnect.

Supervisioning and event listening for failures

Supervisioning is a key technique in order to define relationships between actors that want to respond in case of failures of other actors. More specifically, when an actor needs to respond if another actor fails and stops: I used the *watch* mechanism, that is not a proper supervision method in Akka Framework, but it can be seen as the same. In fact, this mechanism is used to establish a monitoring relationship between two actors, to monitor the lifecycle of the watched actor. In this case it is used by all the User Guardian actors to react in case of a failure for its children, creating a chain of responses that will stop the system. Referring to the actor hierarchies (14), the watch is defined between the user guardian of the player actor system and the interaction actor: if the interaction actor stops, it means that the user closed the GUI, so the actor system of the player must be shut down. Referring to the Foreman actor system instead, the watch relationship, for example, is chained up to the effective player manager actor: if it stops, it means that an internal error has occurred, so that the foreman must quit the

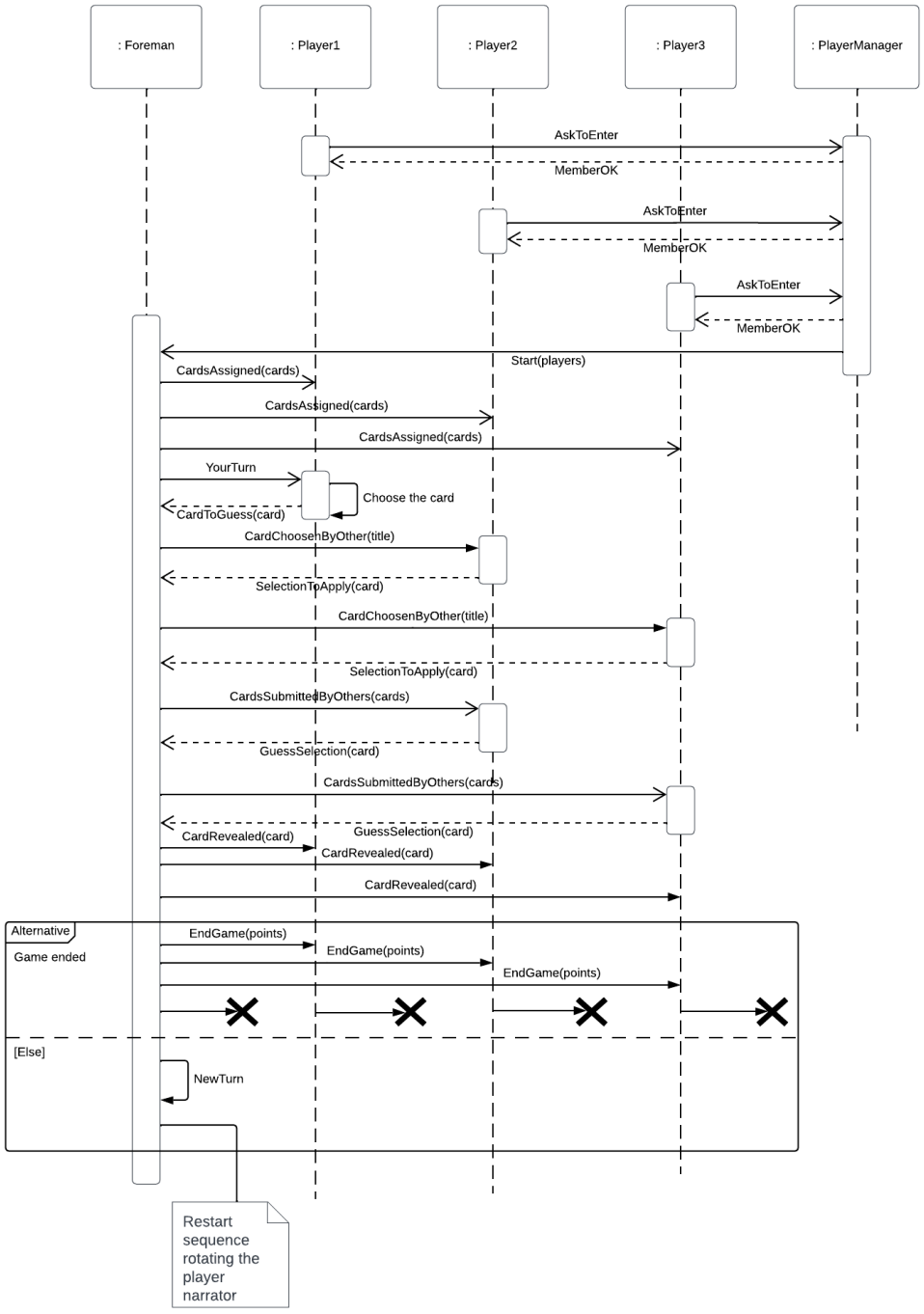

Figure 21: UML Sequence diagram of a game turn

game. This is a method to catch children's errors and react in a proper way. In order to implement a fault tolerant system, there are the so called *event-listeners* actors for each actor system: they are responsible of intercepting the cluster events and react in a proper way. In particular, they intercept the *MemberUp* and *MemberRemoved* events, in a classic message form. This is used to implement a mechanism that react to a player exited event, triggering the *Event Listener Actor* of the Foreman actor system and send a notification to the *Player Manager Actor*, that will start a timer to wait the player exited to rejoin and notifying the foreman in case of rejoin, in order to restart the current turn. The same event listening is used to implement a mechanism to intercept when a specific actor becomes unreachable (and reachable again). The way it works can be seen from the *Player Manager Actor* state diagram (20): when a PlayerExited message is received from the *Event Listener Actor*, a timer is started, and after timeout it is checked that the number of players is again equals to the max number of players. It works because the behavior remains still the same: while the timer is running out, the actor is still reactive to the messages received, so that if a player rejoin before timeout, the situation is restored, and after timeout the game proceeds. If more than one player exit, multiple timers will start, but only the first timeout will trigger the number of players check: if it is not satisfied, the game will be stopped, forcing the *Player Manager Actor* to stop and starting a stopping chain leading to the stop of the entire Foreman actor system. The stop of the Foreman actor system can be treated simply as an event: the event listener of the Player actor system will be notified that an actor with role "foreman" has exited, that is the root actor of the Foreman actor system, triggering the player to notify the interaction actor and notify the user via GUI, and to stop the entire Player actor system. This is essentially how the exiting of a player is managed, but this is also how the foreman exit is treated 22.

Discovery and Receptionist service

Using the Receptionist is a way to discover dynamically actors and get their reference, by registering to the receptionist registry with a specific key, called *ServiceKey*. All the actors registered with a specific service key will be discoverable under that key, by a simple request to the receptionist: in fact, the API of the receptionist is also based on messages, like all the other actors. The registry is also dynamic: actors can be registered and de-registered (manually or automatically caused by a stop of the actor) during the execution. Actors that want to discover actors simply send to the receptionist a Subscribe request specifying the service key. Obviously, this is supported also with a cluster environment: the state of the receptionist is propagated to all nodes, so that each node eventually reach the same set of actors for each ServiceKey. This mechanism is used in the system to register the *Players Manager Actor* as discoverable, so that each player joining the game will firstly contact that actor to request to join: it is the entry point actor to communicate first, by an actor joining the match. This is the simplest way: each node joining the game does not have any other actors references, and this dynamic method removes the need to store the entry point actor server side somehow 23.

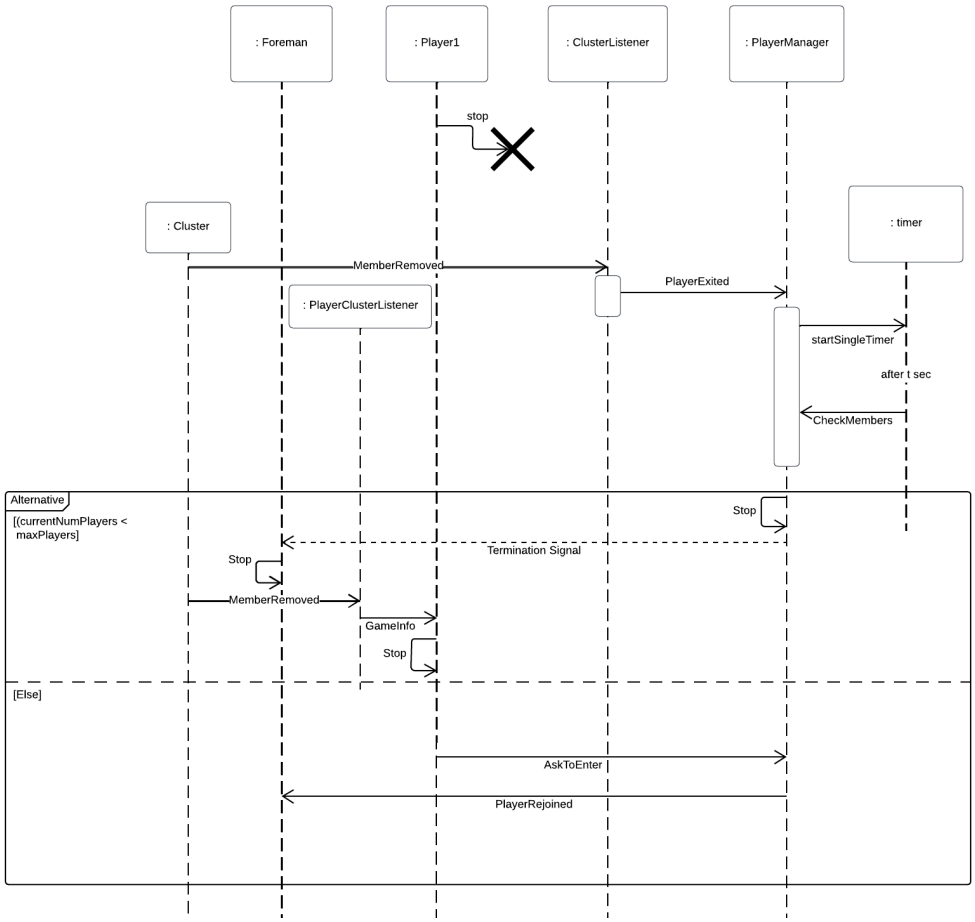

Figure 22: UML Sequence diagram of a stopping client

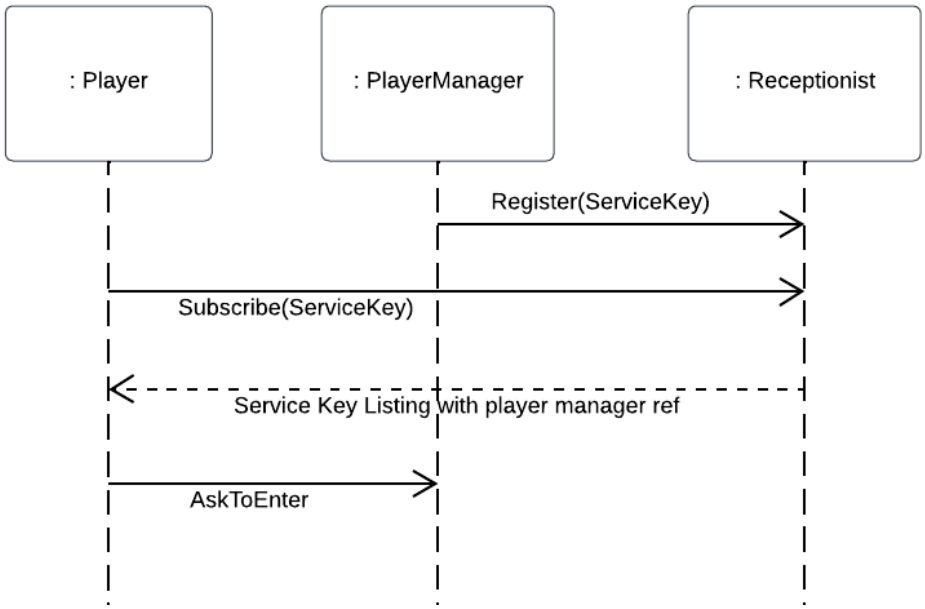

Figure 23: UML Sequence diagram of registration/subscribe into Receptionist

Interaction patterns used

The unique messaging pattern used in the interactions between actors is the classic *Fire and Forget*: this is the classic *tell* operation between Akka actors, that is an asynchronous way to send a message to an actor through its actor reference. There is no guarantee that the message has been processed by the destination actor. This is also used to schedule messages to self, that is an actor sending messages to itself, useful to schedule a message to be managed next. Last but not least, the *Adapted Response* pattern, that is specified by Akka documentation as a way to adapt the type of messages of an actor with the specific types of the receiving actor: an example is when the sending actor does not support receiving the response messages of another actor, so that is possible to adapt the response message to a type that the sending actor can handle. This is used for example to handle cluster events messages after subscribing for them.

5 Implementation Details

In this section we analyze the system with a more technical point of view, starting from the used technologies and little examples of the most interesting implementation details.

5.1 MySQL, PhpMyAdmin and Docker

In order to develop the persistent layer of the server architecture, the choice fell on *MySql*, the world's most popular open source database and relational database manage-

ment system (RDBMS), coupled with SQL language to execute queries on the relational database, creating, updating and querying the database content. The tools chosen as graphical interface to manage the MySql Server and databases created is *phpMyAdmin*, a web-based user interface that allows the admin manager to operate and administrate the databases and the persistent infrastructure. MySql Server and phpMyAdmin services are deployed together exploiting a Docker environment, creating a virtual container executing the MySql server and the graphical tool, accessible from the localhost of the server machine 1.

Listing 1: Docker compose configuration file

```
services:
  db:
    image: mysql:5.7
    container_name: db
    command: --default-authentication-plugin=mysql_native_password -h 127.0.0.1
    environment:
      MYSQL_ROOT_PASSWORD: root_password
      MYSQL_DATABASE: DIXIT
      MYSQL_USER: user_name
      MYSQL_PASSWORD: root_password
    ports:
      - "6033:3306"
    volumes:
      - dbdata:/var/lib/mysql
  phpmyadmin:
    image: phpmyadmin/phpmyadmin
    container_name: pma
    links:
      - db
    environment:
      PMA_HOST: db
      PMA_PORT: 3306
      PMA_ARBITRARY: 1
    restart: always
    ports:
      - 8081:80
volumes:
  dbdata:
```

The unique database used for this project is composed by a single table User, the only information to be stored into the system, created with the following SQL query 2.

Listing 2: Create table SQL query

```
CREATE TABLE `User` (
    `UserId` int NOT NULL AUTO_INCREMENT,
    `Name` varchar(200) NOT NULL UNIQUE,
    `Password` varchar(512) NOT NULL,
    `Points` int NOT NULL,
    PRIMARY KEY (UserId)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

The connection with the MySql Server is managed by the *AccessAdapter* module, the low-level component that access the persistent layer executing the first creation of the

table and manage it with the update and select queries, through a *JDBC Connection*, that simplify the connection and statement execution, including results management and error handling. An example of an update query can be seen at 3, where the query passed with its Builder object is executed. The connection is established with standard config values if not specified an external connection string.

Listing 3: Database access example code

```
def update(query: UpdateBuilder): Boolean = try {
    if(connectionStringExt == "")
        connect()
      else
        connection = DriverManager.getConnection(connectionStringExt, username,
            password)
    val statement: Statement = connection.createStatement()
    statement.executeUpdate(query.query)
    return true
  } catch {
    case e: Exception => e.printStackTrace
      return false
  } finally {
    if (connection != null) try connection.close
    catch {
      case e: SQLException => /* Ignored */
    }
  }
```

5.2 Akka gRPC framework

As already said, the gRPC framework is used to implement the server stub, the server service and the client component, as well as the communications messages. First, gRPC is a transport mechanism for request/response communications over an HTTP/2 connection, supporting streaming messages as well. Akka provides support for building gRPC servers and clients on top of its technologies, starting from a *proto* description file of the service APIs and the messages structure to be exchanged. As already said, the advantages are many, such as favoring decoupled service interfaces, thanks to its schema-first design and language-independent, as well as the well standardized and known efficiency of the Protobuf protocol. The protobuf service descriptor can be seen at the 4 portion of code (only the API description and some messages as example).

Listing 4: Service and messages proto description

```
...
service Server {

  rpc Hello(HelloMessage) returns (HelloMessage) { }
  rpc Login(LoginRequest) returns (LoginResult) { }
  rpc Register(LoginRequest) returns (LoginResult) { }
  rpc GetOpenedGames(OpenedGamesRequest) returns (OpenedGames) { }
  rpc CloseGame(GameMessage) returns (ClosingResponse) { }
  rpc NewGame(GameMessage) returns (NewGameResponse) { }
  rpc UpdateUserPoints(UserPoints) returns (UpdateResponse) { }
}
```

```
12
13 message UserPoints {
14   repeated string userName = 1;
15   repeated int32 points = 2;
16 }
17 ...
```

After the stubs and interfaces generation (of the server and messages objects, using the corresponding sbt task) in Scala language, it is possible to instantiate the server simply extending the stub and overriding all the methods referred to the APIs specified into the proto specification file, that are all unary calls, that are to say single requests that return a Future with a single response. The portion of code reporting the server service running instance and the server service implementation is represented in the 5 listing.

Listing 5: Server service implementation

```
1  class Service(system: ActorSystem, repository: AccessAdapter) {
2    .....
3    def run(): Future[Http.ServerBinding] = {
4      // Akka boot up code
5      given sys: ActorSystem = system
6      given ec: ExecutionContext = sys.dispatcher
7      given inboundPorts: InboundPorts = InboundPorts(registerPort, managementPort)
8
9      // Create service handlers
10     val service: HttpRequest => Future[HttpResponse] =
11       ServerHandler(new MyServerService())
12
13     // Bind service handler servers to localhost:8080/8081
14     val binding = Http().newServerAt("127.0.0.1", 8083).bind(service)
15
16     // report successful binding
17     binding.foreach { binding => println(s"gRPC server bound to: ${binding.
           localAddress}") }
18
19     binding
20   }
21 }
22
23 class MyServerService(using mat: Materializer, inboundPorts: InboundPorts) extends
       Server{
24
25   ...
26
27   override def getOpenedGames(in: OpenedGamesRequest): Future[OpenedGames] =
28     Future.successful(ProtoToDomainAdapter.protoToGames(inboundPorts.
           gamesManagementPort.getAllGames()))
29
30   ...
31
32 }
```

5.3 Akka Actor framework

The Akka Actor framework is the core of the system: as we already seen, it is the base for creating clusters of actor systems, that simply model different running matches. This framework allows creating systems composed by multiple actors organized with a

hierarchical structure and many useful mechanisms offered to simplify the clustering of actor systems and its management, events handling and messages exchange, by defining behaviors and stateful actors. An example is the Players Manager actor system creation at 6, where we can see the actor startup configuration and behavior definition:

- The code represents the root actor of the *Players Manager Actor System*, as already described
- There is one children actor spawned: the effective players manager actor with the *PlayersManagerBehaviorImpl* behavior
- The *receiveMessage* defines the final behavior to use by the root actor, and the messages to handle
- The *case classes* and *case objects* are the messages that all the actor's behaviors can accept, extending Command that is a sealed case class that extends the common Message trait: this is a simple method that improve the developing phase, to encapsulate all the messages accepted by a behavior with the definition of the behavior itself
- The *watch* relationship is set between the root actor and the effective manager actor: if it stops, it will send a notification intercepted by the root actor in the *receiveSignal* method, stopping the entire actor system
- The last thing we can analyze is the service key used to register that actor into the Receptionist registry, in order to be discoverable from other actors subscribing to that public service key through the Receptionist
- Right before the definition of the *receiveMessage* property we use the *setup* to pre-configure the user guardian actor and its AS

Listing 6: Player Manager behavior code

```
object PlayersManagerBehavior {
  val Service = ServiceKey[Command]("GameManager")
  sealed trait Command extends Message
  case class AskEnter(replyTo: ActorRef[PlayerBehavior.Command]) extends Command
  case object Startup extends Command
  case object PlayerExited extends Command
  case object CheckMembers extends Command
  case object Stop extends Command

  def apply(maxPlayers: Int, foreman: ActorRef[ForemanBehavior.Command]): Behavior
      [Command] =
    Behaviors.setup[Command] { ctx =>
      ctx.system.receptionist ! Receptionist.Register(Service, ctx.self)
      val manager = ctx.spawn(Behaviors.setup[PlayersManagerBehavior.Command](ctx
          => PlayersManagerBehaviorImpl(ctx, maxPlayers, foreman)), "manager")
      manager ! Startup
      ctx.watch(manager)
      Behaviors.receiveMessage[Command] {
        case msg: Command =>
          manager ! msg
```

```
            Behaviors.same
        }.receiveSignal { case (ctx, t @ Terminated(_)) =>
          ctx.log.info("[MANAGER] System terminated. Shutting down")
          Behaviors.stopped
        }
    }
}
```

The actor systems are created following a configuration procedure: an AS is made out of several actors having each one a proper IP address and a port. In a local environment, the addresses are the same for each actor, that is the *localhost* address (127.0.0.1), while the port is the way we differentiates them from each other.

Listing 7: Cluster configuration file

```
akka {
  actor {
    provider = cluster
    serialization-bindings {
      "grpcService.client.actors.utils.Message" = jackson-cbor
    }
  }
  remote.artery {
    transport = tcp
    canonical { }
  }

  prio-dispatcher {
    mailbox-type =
      "grpcService.client.actors
           .prioritizedQueue.MyPriorityActorMailbox"
  }

  cluster {
    seed-nodes = [ ]
  }
}
```

As we can see in the configuration file reported in the 7, there are some interesting notions:

- Actor: it defines the common specifications for the actors we will create, like the serialization methodology and message trait to be used; to enable the creation of clusters we set the provider as cluster type
- Remote: is the mechanism by which Actors on different nodes talk to each other internally; we specify the *tcp* protocol to be used and leaving canonical IP and port empty, to be defined later in the cluster access (dynamically)
- Cluster: here we define the seed nodes, an array of addresses, that is empty by default
- Custom mailboxes: this is the point where we define all the custom mailboxes created and used

Two are the port that must not be used: the 2551 and the 2552. These are the ports that indicate into the project the so called *seed nodes*, that are (as the Akka docs tells) *"the initial contact points for joining a cluster"*. They are basically normal actors, that participate in the cluster in exactly the same way as other nodes, but they also take part to the joining phase for each other new actor started: it firstly sends a message to all the configured seed nodes of the cluster game, and then it sends a join command to the one that responds first.

Listing 8: Actors Systems startup

```
def startupWithRole[X](role: String, port: String,
            ip: String,
            clusterName: String = "ClusterSystem")(root: => Behavior[X]):
                ActorSystem[X] =
    val config = ConfigFactory
      .parseString(
        s"""
        akka.remote.artery.canonical.hostname=$ip
        akka.remote.artery.canonical.port=$port
        akka.cluster.roles = [$role]
        akka.cluster.seed-nodes = [
          "akka://$clusterName@127.0.0.1:2551",
          "akka://$clusterName@127.0.0.1:2552"
        ]
        """)
      .withFallback(ConfigFactory.load("base-cluster"))

    // Create an Akka system
    ActorSystem(root, clusterName, config)
```

In the portion of code showed in the 8, we can see the *startup* of the Actor Systems, specifying the base config file and all the configurations missing, like the seed nodes of the cluster to which refer in order to join, the canonical ip and port for that AS (and more specifically its user guardian actor) and its role on the *Cluster*.

5.4 Scala sbt build automation

The sbt build automation system is an open-source tool used to automatize and simplify compilation, test phases, manage dependencies and creating custom tasks. The sbt config file configures also the coverage tool used and the conventional commit plugin.

5.5 Github Actions

GitHub Actions is the Github's tool for CI/CD (continuous integration and continuous delivery) of the software, a platform that allows to automate the build, test, and deployment pipeline: it is used for example to deploy new releases with *semantic-release* and manage the GitHub Page deployment, that contains the web-form of the coverage report.

6 Self-assessment / Validation

In order to verify the compliance to the requirements and the effective global functionalities offered by the application, several tests have been written and executed during the development, with the purpose of testing all the components. Tests divide into two categories: automatic and manual tests. The latter are the manual tests done by the direct usage of the application by a supervisor, annotating all the possible errors and anomalies. The automatic tests instead, are all the Unit Tests written in order to assert the expected results from an automatic running of the system. In particular, the automatic testing divided into three steps: the testing of all the domain concerns and persistent layer access, the testing of all the actors in different configured environments and the final testing of the client-server communications using gRPC. The testing of the domain entities and logic is made using the *ScalaTest* library, using the assert statements to verify what is expected against what is received.

Listing 9: Test of a model class

```
class PumpMyListTest extends AnyFunSpec with Matchers:
    describe("A list") {
        describe("extended with new methods") {
            it("should use them correctly") {
                var l = List(1,2,3,4,5,6)
                var newL = l.filterMap {
                    i => i % 2 == 0
                } {
                    i => i * 2
                }
                newL = newL :+ 7
                newL = newL - 8
                assert(newL == List(1, 4, 3, 5, 12, 7))
                val l1 = List(1,2,3,4)
                val l2= List(3,4,5,6)
                assert(l1.removeIntersection(l2) == List(1,2))
                assert(!l1.equals(l2))
            }
        }
    }
```

In the 9 example code, we can see the usage of *AnyFunSpec*, that allow a more BDD-style testing of the component under test, specifying for example the context using the *describe* and *it* constructs. For what regards the actor systems testing, it has been used the *Asynchronous testing* framework offered by Akka, that permits to create real ActorSystems that allows to test all the Actors in a more realistic environment, simulating messages exchange and using *probe actors* to assert the expected messages to be received.

Listing 10: Actor behaviors testing

```
describe("in a game with 2 players") {
  describe("when the game starts") {
    it("a player should send the choosen card for its turn") {
      val testKit = ActorTestKit()
      val probe = testKit.createTestProbe[ForemanBehavior.Command]()
      val interaction = testKit.spawn(interactionTestActor("1"), "interaction")
```

```
      val foreman = testKit.spawn(ForemanBehavior(Option(probe.ref), 2), "foreman")
      val player = testKit.spawn(PlayerBehavior(interactionExt = Option(interaction)), "player")
      val player1 = testKit.spawn(PlayerBehavior(interactionExt = Option(interaction)), "player1")
      probe.expectMessage(20 seconds, ForemanBehavior.Start(Set(player, player1)))
      val message: ForemanBehavior.Command = probe.receiveMessage()
      message match {
        case ForemanBehavior.CardToGuess("1", "myTitle", _) => succeed
        case m => fail("Unexpected message: " + m)
      }

      testKit.shutdownTestKit()
    }
  }
}
```

The 10 shows the usage of an *ActorTestKit* to create a real actor system, spawing all the actors under test and simulating interactions in order to verify that the actual messages received are equals to the expected ones. Last but not least, the testing of the client-server communication, included the access repository testing, is made using *Futures* handling to manage server responses, sending direct messages between a local client and server (11), and using the *TestContainer* framework to create a testable MySql server using the docker container environment.

Listing 11: Server service testing

```
describe("Assuming the gRPC server service started") {
    describe("when the client opens") {
        it("should send correctly an hello message to the server and receive correctly the response") {
            val value = "value"
            val client = ClientImpl()
            val future: Future[HelloMessage] = client.helloMessage(value)
            assert(future.isReadyWithin(2000 millis))
            whenReady(future) { s =>
                s shouldBe HelloMessage("Ciao " + value + "!!!")
            }
        }
    }
}
```

The testing of the persistent layer is made by testing the *AccessAdapter*, creating a docker container environment with the MySql server image, more specifically the *mysql:5.7.34*, creating the container before each test and stopping it after the end of testing. An example code is reported in 12.

Listing 12: Docker containers testing

```
var mysql = new MySqlContainerWrapper().getContainer()
mysql.start()
val url = mysql.getJdbcUrl() + "?autoReconnect=true&useSSL=false&enabledTLSProtocols=TLSv1.2"
val user = mysql.getUsername()
val pass = mysql.getPassword()
val accessAdapter = AccessAdapter(username = user, password = pass, connectionStringExt = url)
```

```
7 val server = Service.createService(accessAdapter)
```

The final coverage level reached is quite high, except for the gRPC generated classes (under *grpcService* package), with only the server stub tested and *utils* and *behaviors* packages, where the tests did not cover the interaction actor, that is made of GUI interactions (24).

service	100.00%
ports	90.32%
actors	86.00%
model	82.35%
adapters	81.79%
prioritizedQueueExample	75.00%
client	68.00%
applicationService	63.16%
behaviors	61.45%
grpcService	34.39%
utils	12.07%

Figure 24: Coverage level and packages

7 Deployment Instructions

The pre-requisites necessary to run the project are the following:

- Git tool to clone the repository
- A Java $\geq$ 11 distribution (like Eclipse Adoptium Java 11.0.21)
- The sbt automation/build tool to compile the Scala code and execute it
- Docker Desktop installed

7.1 Pre-start

First thing to do, in order to execute all the components, is executing the persistent layer of the system, so we have to start the Docker service by opening the Docker Desktop application and, via terminal, launching the following command inside the root of the project repository:

```
$ docker-compose up
or
$ docker compose up
```

After all the tests, the containers should be shutdown using the following command:

```
$ docker-compose down
or
$ docker compose down
```

The containers running state can be verified using the Docker Desktop application or trying to access the *PhpMyAdmin* page at the *localhost:8081* and access the db using the credentials reported into the README file.

7.2 Execute the server

Once the MySql server is running, we can proceed executing the server with the following command, always from the root of the repository folder:

```
$ sbt runServer
```

7.3 Execute the clients

Finally, we can open three fresh new terminals, where we are going to execute the three local clients that will simulate a game. The only thing to remember is not to use the 2551 and the 2552 ports, because they are used as *seed nodes* of the cluster game.

```
# Terminal 1
$ sbt "run 127.0.0.1 2553"
# Terminal 2
$ sbt "run 127.0.0.1 2554"
# Terminal 3
$ sbt "run 127.0.0.1 2555"
```

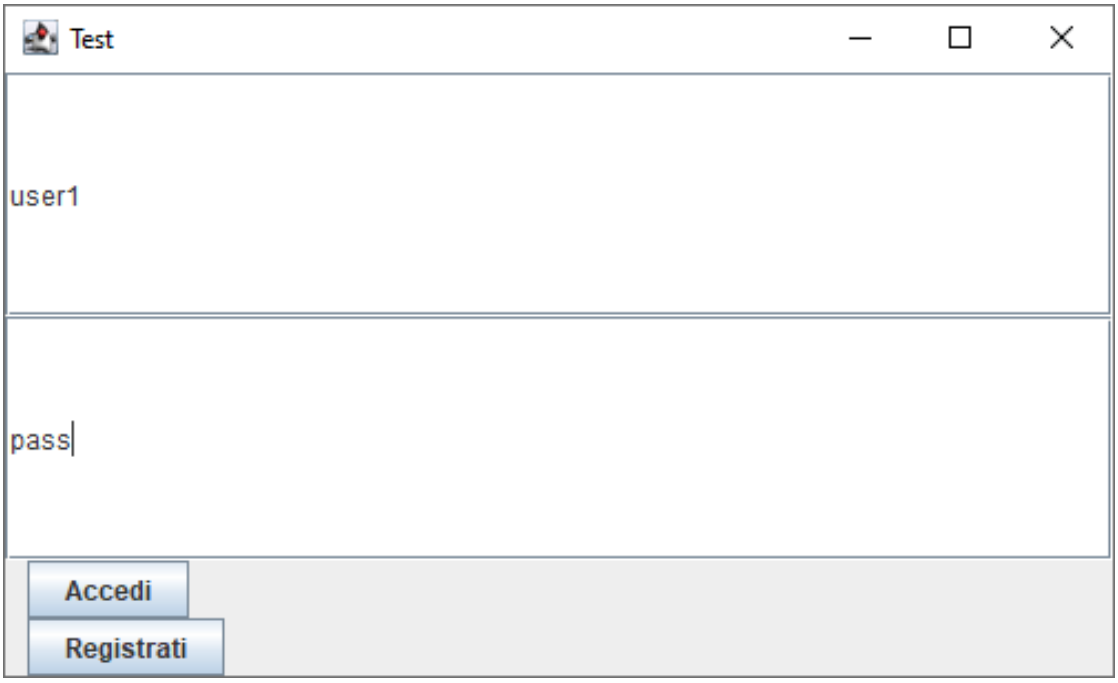

Figure 25: Registration/login page

The sbt build tool will probably ask you about creating a new server to build and execute the project in the different terminals, so you need to respond *yes* in order to continue.

8 Usage Examples

Once the server and all the database's services are running, we can execute 3 clients. There should opens the following view (25), where we are going to register and login with the credentials listed above:

- Client1: *user1* as username and *pass* as password
- Client2: *user2* as username and *pass* as password
- Client3: *user3* as username and *pass* as password

Clicking on register, we'll be directly logged into the system, more specifically into the main page, that shows us the list of the open games (that we can refresh manually) and some controls to create a new game or join an existing one. To create a new game from the *user1*, we can click on the appropriate button (26), that will open a new window representing the game view waiting for other players. All the other players (*user2* and *user3*) can refresh now the list of all the opened games, select the match created by the *user1* and join it with the appropriate button (27). Next, the cards will be distributed and the first narrator will see its cards (28): that means is its turn to choose a card

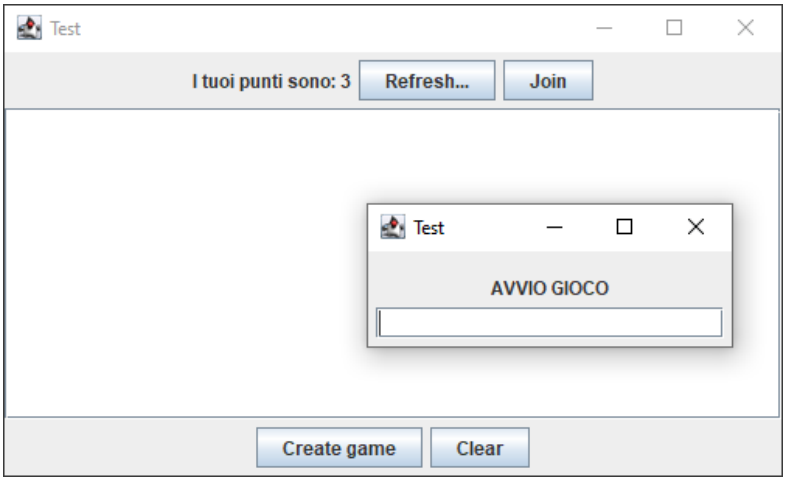

Figure 26: Creation/joining of a new/existing game

Figure 27: Open games list page

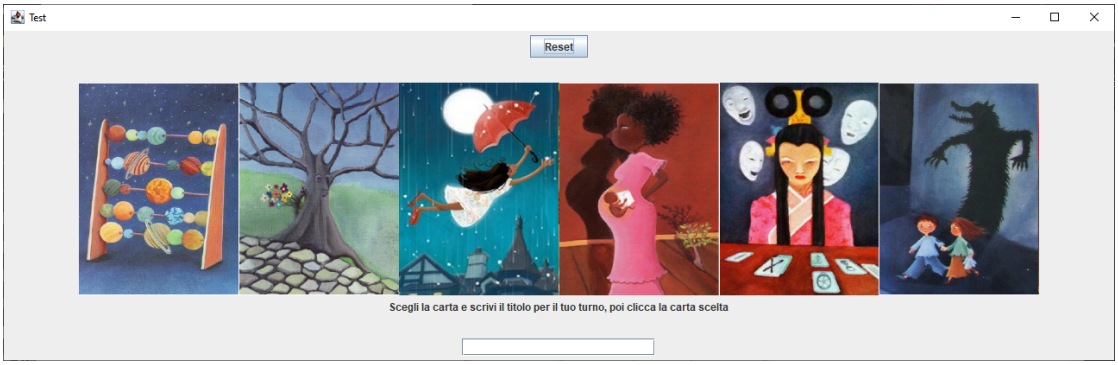

Figure 28: Narrator's page

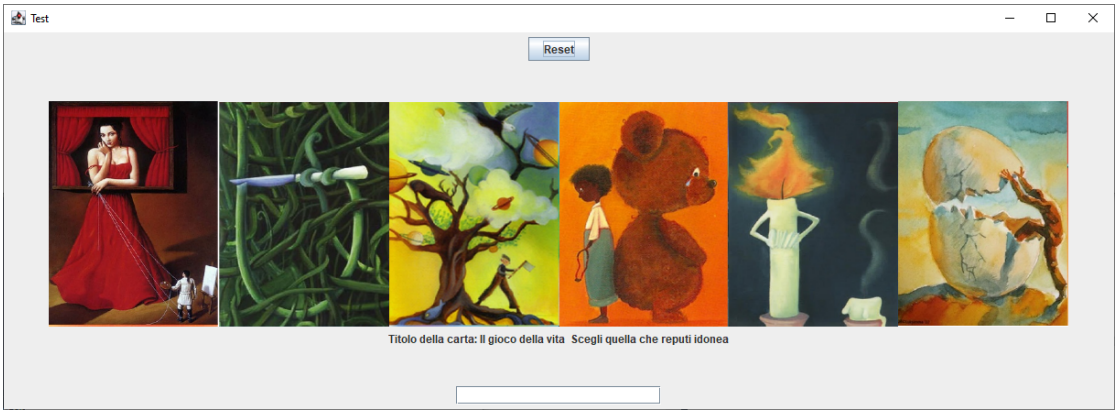

Figure 29: Guesser propose page

and the title, firstly inserting the title and then clicking the card chosen. Then, all the players will be able to see their cards and the title chosen by the narrator for that turn: all the players are going to select the card from theirs that they think is the most similar and significant for the title proposed by the narrator, just by clicking on the card chosen to propose (29). Now, after each guessing player has chosen their card to propose, they will be able to see all the proposed cards (except the one they chosen) mixed with the narrator's card: it is the guessing phase of the turn, where all the guessers will choose the card they think is the narrator one (30). Once all guessing players have chosen their guessed card, the turn is finished and another one will start after 5 seconds, rotating the narrator until the end of the game. The game will end after 3 turns for the preview mode: it is possible to change the number of turns, but for simplifying the user GUI and the demo, there's no view controls to change the number of turns, that is standard set at 3.

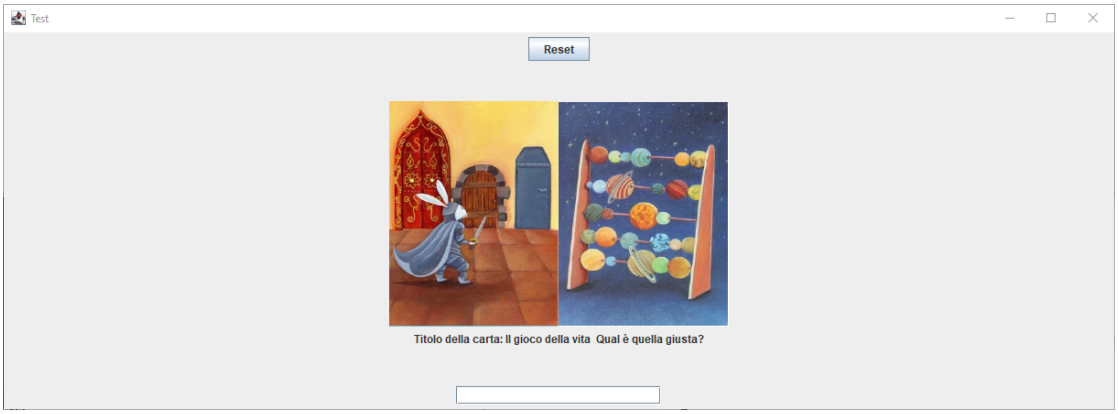

Figure 30: Guesser choose page

9 Conclusions

Concluding, the project goal was to create a turn-based game demo using an actor model based programming paradigm, aiming to create a highly scalable system, easy to maintain and evolve and message based interactions. Personally, I think that all these goals have been reached, because of the intrinsic nature of a distributed actor system and the BDD style that lead the development, that helps a lot creating elements highly evolvable in their dynamics, a more understandable and verifiable code. Thanks to the Akka Framework, as we said, we have also tools to support a strong reliability and robustness of the system.

9.1 Future Works

As future works, it would be interesting adding some new features:

- Manage the unreachability of a node: currently, the match is able to intercept exiting players, that happens when a player close the window during running game or crashes, causing the exit from the cluster. Instead, it would be also interesting the unreachability event, that happens when an actor is still present on the cluster but isolated from it: currently the system only intercepts the unreachability event of a player but does nothing, waiting for him to be reachable again
- A better GUI experience
- Improve the usability in a real distributed environment, with nodes dislocated around the world: it means using public IPs in a easier way instead of launching the game via console specifying IP address and port
- Add a *matches done* history for each player to store victories and losses

- Try to implement Cluster Sharding, useful to distribute actors in the network and be able to interact with them not using the information about their physical location (that may change over time) in the cluster but a simple logical identifier

9.2 What did I learned

This project guided me to learn many things about how it works developing a project involving different technologies, different paradigms and the coordination of active entities using messages exchange, the usefulness of containerization with Docker, how to design a distributed system having in mind different nodes and roles, how to design a server in a extendable way.

References

[1] Asmodee. Regolamento dixit, 2023. https://www.asmodee.it/giochi/dixit/Dixit_regolamento_italiano.pdf [Accessed: (2023)].

[2] AtomicJar. Testcontainer docs, 2023. https://testcontainers.com/ [Accessed: (2023)].

[3] Nicolas Farabegoli. Sbt conventional commits, 2023. https://github.com/nicolasfara/sbt-conventional-commits [Accessed: (2023)].

[4] Google LLC. Protocol buffers docs, 2023. https://protobuf.dev/ [Accessed: (2023)].

[5] Prof. Alessandro Ricci. *ARCHITECTURAL STYLES - PRINCIPLES OVERVIEW*. 2023/2024. Software Architecture and Platforms LM course.

[6] Akka Technologies. Akka actors documentation, 2023. https://doc.akka.io/docs/akka/current/general/index.html [Accessed: (2023)].

[7] Akka Technologies. Akka actors documentation, 2023. https://doc.akka.io/docs/akka-grpc/current/index.html [Accessed: (2023)].