

Liar's Poker

Francesco Buda

`francesco.buda3@studio.unibo.it`

Emanuele Sanchi

`emanuele.sanchi@studio.unibo.it`

Tommaso Severi

`tommaso.severi2@studio.unibo.it`

June 4, 2025

This report presents the design and development of a distributed game application for playing Liar's Poker. It outlines all phases of the project, from the initial concept through to the final implementation, highlighting the evolution of the system as it was shaped to meet the specific demands of a distributed environment. The game logic adheres to the rules described in [1], with adaptations made to support distributed execution.

The system architecture follows a client-server model and relies on the MQTT protocol to facilitate communication between components. The report provides an in-depth discussion of the design decisions taken throughout the project, along with detailed insights into the implementation process. Testing strategies are also described, focusing on both functionality and system robustness.

Particular attention is given to fault tolerance, demonstrating how the application manages client disconnections and server failures. The application is developed in Python, leveraging the Paho MQTT library to implement reliable and efficient messaging and the NiceGUI library to create a user-friendly web interface.

Contents

1	Concept	4
1.1	Game Rules	4
1.2	Questions and Answers	4
1.3	Usage Scenarios	5
2	Requirements	7
2.1	Functional Requirements	7
2.2	Non-functional Requirements	7
2.3	Glossary of cards combinations	8
3	Design	9
3.1	Architecture	9
3.2	Infrastructure	9
3.3	Modelling	10
3.4	Interaction	12
3.5	Behaviour	15
3.6	Data and Consistency Issues	18
3.7	Fault-Tolerance	18
3.8	Availability	19
3.9	Security	19
4	Implementation	20
4.1	Network Protocols	20
4.2	Data Formats and representations	21
4.3	Technological details	21
5	Validation	23
5.1	Automatic Testing	23
5.2	Acceptance test	24
6	Release	25
7	Deployment	27
8	User Guide	28
8.1	Requirements	28
8.2	Installation	28
8.3	Running the software	28
8.4	Screenshots	29
9	Self-evaluation	33
9.1	Francesco Buda	33
9.2	Emanuele Sanchi	33

9.3	Tommaso Severi	34
-----	--------------------------	----

1 Concept

This project aims to develop a web-based game application for playing Liar's Poker, a fun and engaging card game centred around deception and bluffing. Inspired by traditional poker, Liar's Poker is played with a standard deck of cards and is suitable for two or more players. Our application will provide web interface where users can enjoy the game with others in real-time.

1.1 Game Rules

Liar's Poker is played with a standard deck of cards without any joker. Each player starts with one card. A round starts with someone naming a poker combination. The turn rotate clockwise and each next player has a choice:

- **Rise the stakes**

Name a higher poker combination than the one named by the previous player that they believe is present in the combined cards of all players.

- **Call bullsh*t**

Call bullsh*t on the previous player's combination if the player thinks the previous player is bluffing or mistaken.

When someone calls bullsh*t, all players reveal their cards and the presence of the contested poker hand is verified. If the hand is present, the player who called bullsh*t loses the round; else, the player who named this poker hand loses the round.

The loser starts all subsequent rounds with one extra card, and names the first combination in the next round. If someone loses when they have 5 cards, they are kicked from the game. The last person remaining in the game is the winner.

1.2 Questions and Answers

- **Question:** *How do users interact with the system?*

Answer: Users can access the application through a web browser.

- **Question:** *What happens if one player disconnect during the game?*

Answer: The game will pause until the player reconnects. If the player does not reconnect within a certain time frame, they will be kicked from the game.

- **Question:** *What happens if the server dies during the game?*

Answer: The application goes on, but the game is lost. The players will be redirected to the lobby screen and can choose to play another game or leave the lobby.

- **Question:** *Does the system need to store player's data?*

Answer: The system doesn't need to store data, as the player does not need to create an account.

1.3 Usage Scenarios

This section describes the typical usage scenario of a group of friends playing Liar's Poker on our platform.

1. **Name choice**

Each player chooses a nickname to use during the game.

2. **Lobby creation**

One user creates a new lobby and shares the access code with friends.

3. **Joining the lobby**

Other users join the lobby by entering the code.

4. **Gameplay**

Each player receive one card and then the game starts. The game proceeds as described in the rules. When the player wants to rise the stakes, he can choose from the possible combinations (non-valid combinations will be disabled), if the combination needs some specific rank or suit to be declared, the player will be provided with a list of possible ones to choose from.

5. **Elimination**

When a player loses, they are eliminated from the game.

6. **End of the game**

When the game ends the winner is declared and players will be redirected to the lobby page where they can choose to play another game or leave the lobby.

The figure 1 shows the use case diagrams for the application.

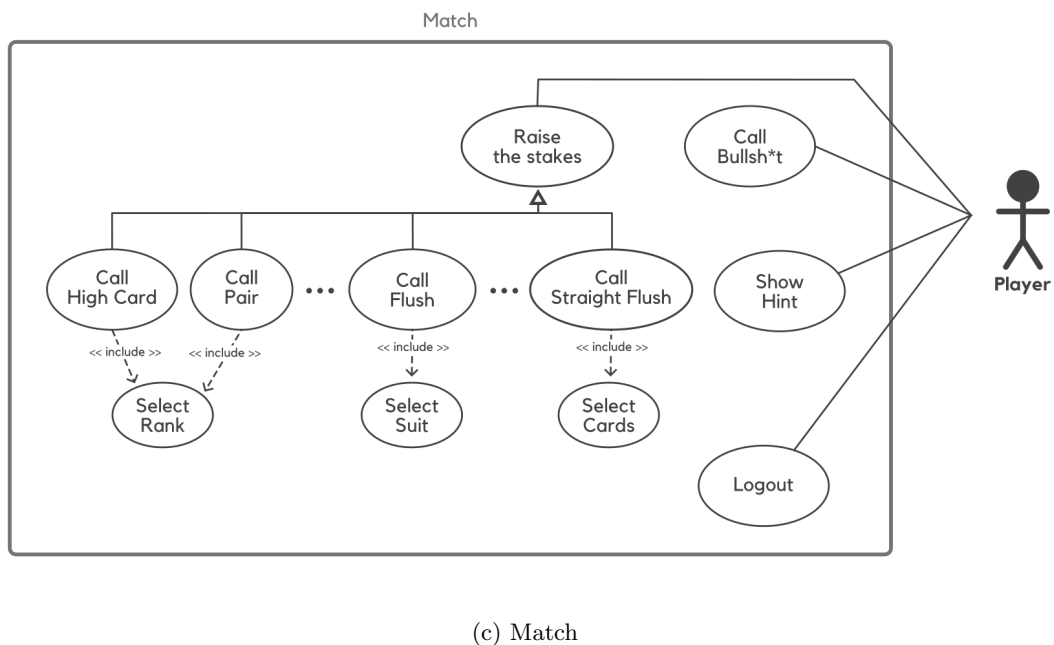
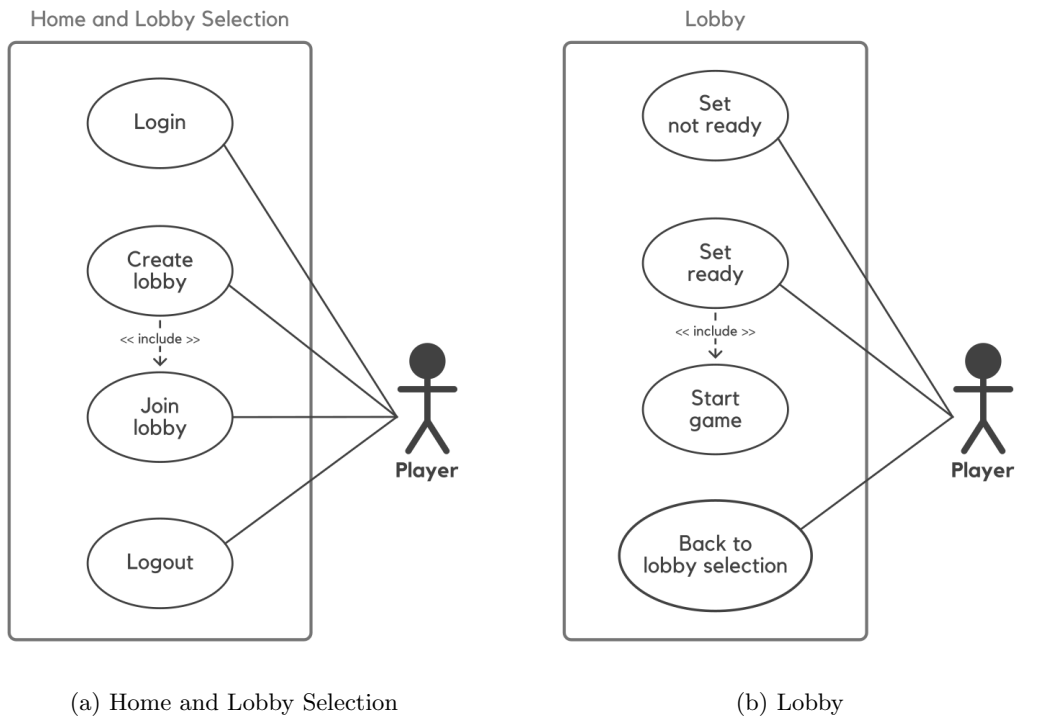


Figure 1: Use Case Diagrams

2 Requirements

Starting from the concept, we can identify the following requirements for the application:

2.1 Functional Requirements

1. Lobby Creation

The application must allow users to create a new lobby and share the access code with friends.

2. Lobby Joining

The application must allow users to join a lobby by entering the access code.

3. Gameplay

The application must allow users to play Liar's Poker following the game rules.

2.2 Non-functional Requirements

1. Scalability

The application must be able to handle multiple lobbies and games simultaneously.

- **Acceptance Criteria:** The system must be able to handle at least 5 concurrent games.

2. Fault Tolerance

The application must be able to handle user disconnections without losing a running game and should recover from server failures.

- **Acceptance Criteria:** The system must be able to recover from server failures within 1 minute.

3. Consistency

The application must ensure that all players see the same game state at all times.

- **Acceptance Criteria:** The system must ensure that all players see the same game state within 2 second from the bullsh*t call.

4. Performance

The application must be able to have good response times.

- **Acceptance Criteria:** A new game must be created within 5 seconds.

2.3 Glossary of cards combinations

In this section we list all the possible cards combinations, that can be declared during the game, ordered by increasing value:

1. **High Card**

In the combined cards of all players there is a card with the rank declared by the player.

2. **Pair**

In the combined cards of all players there are two cards with the same rank declared by the player.

3. **Two Pair**

In the combined cards of all players there are two pairs of cards with the same ranks declared by the player.

4. **Three of a Kind**

In the combined cards of all players there are three cards with the same rank declared by the player.

5. **Straight**

In the combined cards of all players there are five cards with consecutive values declared by the player.

6. **Flush**

In the combined cards of all players there are five cards with the same suit declared by the player.

7. **Full House**

In the combined cards of all players there is a *pair* and a *three of a kind* with the respective ranks declared by the player.

8. **Four of a Kind**

In the combined cards of all players there are four cards with the same value declared by the player.

9. **Straight Flush**

In the combined cards of all players there are five cards with consecutive values and the same suit declared by the player.

10. **Royal Flush**

In the combined cards of all players there are five cards with the same suit declared by the player and the ranks *10*, *J*, *Q*, *K* and *Ace*.

3 Design

3.1 Architecture

The architecture of the system follows the standard MVC principle:

- Model are the classes that represent players and cards
- View is UI realized with Web components
- Controller is the class that manages the game logic

For the communication architecture and the distributed part, we opted for a client-server architecture using a publish-subscribe pattern.

The idea is that the broker manages the communication between the clients and the server, and the server manages the game logic.

When requested by a player, the server creates a new lobby where other players can join. Every time a new player joins the lobby, it automatically subscribes to the game topics.

All the clients are subscribed to the same topic, so they can receive the messages from the server (e.g. moves, game status, etc.).

We choose this architecture because it is simple and suitable for our project. In fact, using a publish-subscribe architecture, clients don't have to request the server for the game status, but they receive it automatically when the server publishes it (and so on for the other messages).

3.2 Infrastructure

- Both players and server are components of the pub-sub model.
- There is one server that manages the game logic.
- There is a broker that manages the communication between the clients and the server.
- Data aren't store permanently, so there is no need for a database.
- There aren't authentication mechanisms so every one who knows the broker address can join the game.

Components are distributed over all the network, so they can be on different machines and different networks. For simplicity, we used the same network for all the components and the same machine both for server and broker.

To find each other, the components use the broker address and the topics to which they have to subscribe or publish; so clients are only required to know the broker address to join the game.

Components are named using a nickname that everybody choose on game launch.

The pub-sub behaviour is shown in fig. 2. Dotted lines are for subscribe messages, while solid lines are for publish messages.

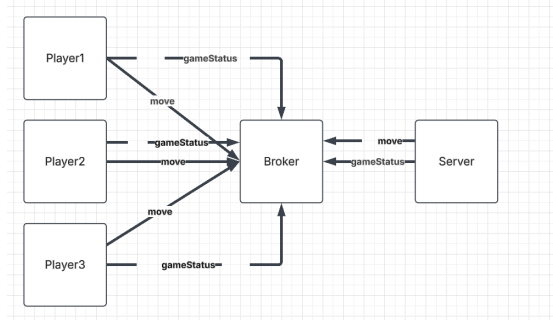


Figure 2: Architecture diagram

3.3 Modelling

The system will be fundamentally composed by the classes shown in fig. 3.

Each component models the real world entities that are needed to play the game in a non-distributed way.

The game is the most important entity of the whole model. It has a deck of cards, that can be shuffled, and a set of players, each one with: a nickname, its given cards and its lobby ready state. To handle the game logic it keeps in memory the current player and the latest raise.

It can be asked to:

- Add or remove a player from the game
- Get a fresh set of shuffled cards for the players
- Raise the stakes and advance the turn with the new minimum stake
- Check if the latest stake was a lie
- Get the currently playing user
- Get the player that raised the stakes

At the core of the entire system are messages, these, given the simple nature of the game, are quite simple too: containing a single object that represents the move the player wants to make or the response of the server.

The real difference and context of the message is given by the topic to which it is published.

All the topics used by the system can be viewed in it's associated enum but are not directly used when sending a message since most of them are different depending on the context in which they are used. For example the "remove player" topic is used as it is when logging out a player that is not in a game, but when used in a game it will be linked together with the lobby ID. Most notably these are the topics used when playing:

- **Start Turn**

The game publishes the current player and the minimum stake for him.

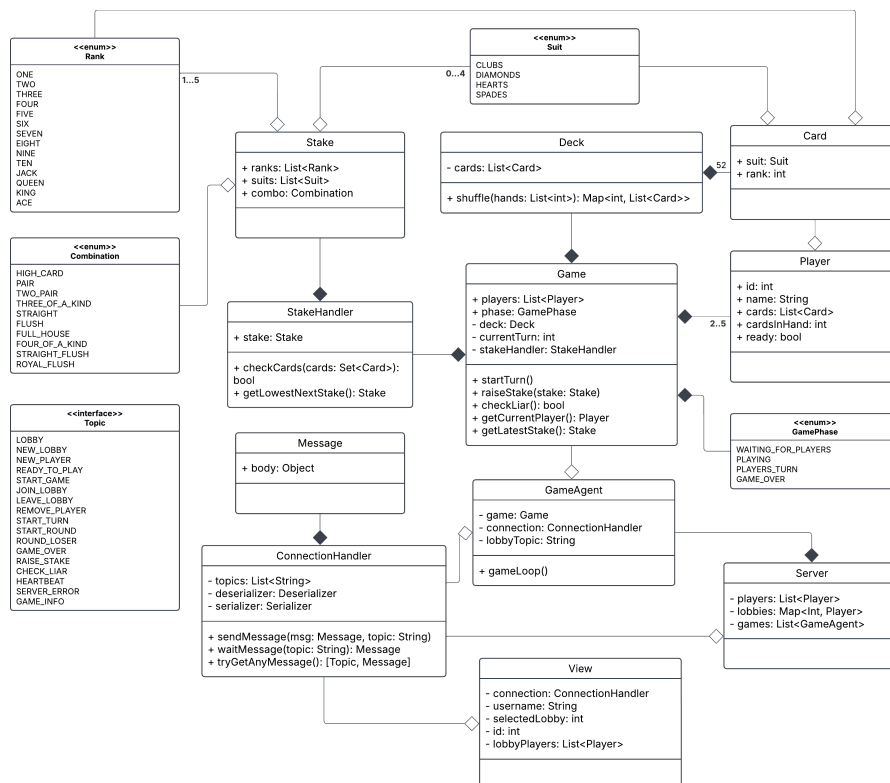


Figure 3: Class diagram of the system

- **Start Round**

The game publishes the list of all players and their cards at the start of a round.

- **Round Loser**

The game publishes the player that lost the round and all the cards in game, also indicating if the player was eliminated.

- **Raise Stake**

The player informs the game of his new stake.

- **Check Liar**

The player doubts the previous stake and asks the game to check if it was a lie.

- **Game Over**

The game publishes the winner signalling the end of the game.

- **Remove Player**

The player announces that he will log out of the game.

- **Game Info**

If a player tries to reconnect to a game, the game will publish the current game state

3.4 Interaction

Components communicate over the network using the MQTT protocol so using a Message Broker pattern. Using this pattern, publishers can send messages without knowing who is going to receive them, and subscribers can receive messages without knowing who sent them. The broker is the one who manages the communication between the clients and the server, so the clients don't have to know the server address, but only the broker address; moreover, the server has to know the broker address to publish messages, but it doesn't have to know the clients' addresses.

This kind of architecture is highly decoupled, so the components can be easily replaced or modified without affecting the others.

The following list will explain the interaction between the components in more detail.

- **Connection**

The connection behaviour is shown and explained in fig. 4.

- **Game Move**

The game move behaviour is shown and explained in fig. 5.

- **End Game**

The end game behaviour is shown and explained in fig. 6.

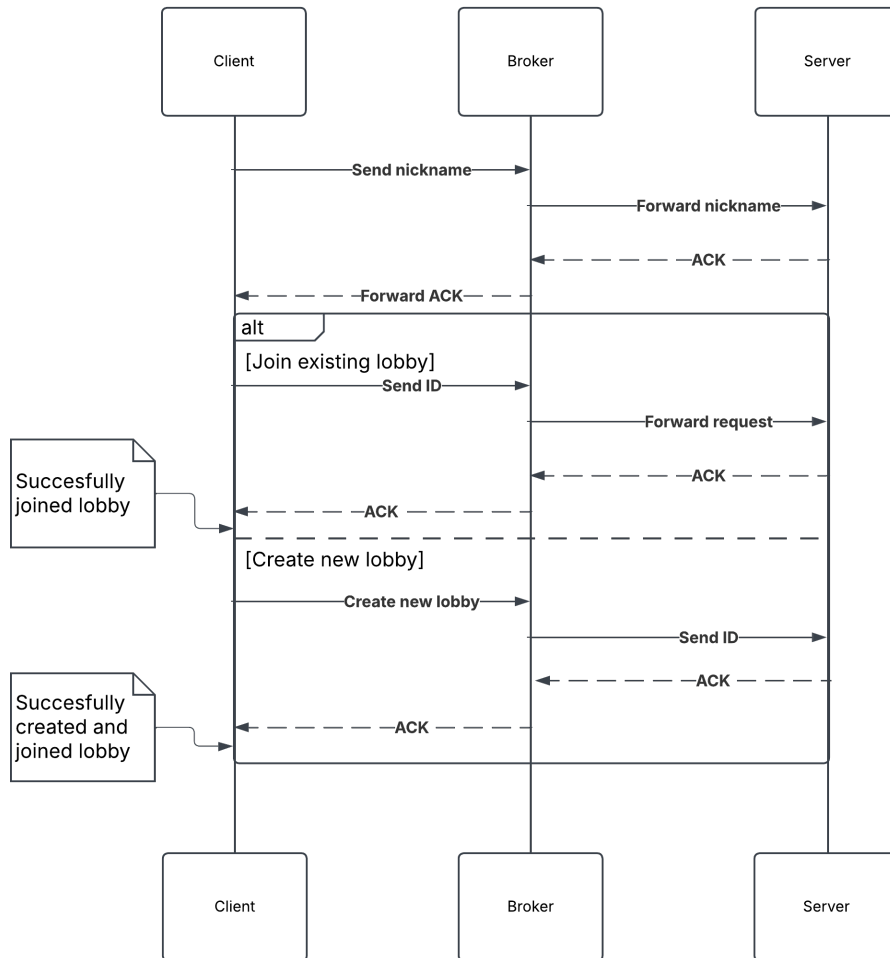


Figure 4: Sequence diagram of the connection phase

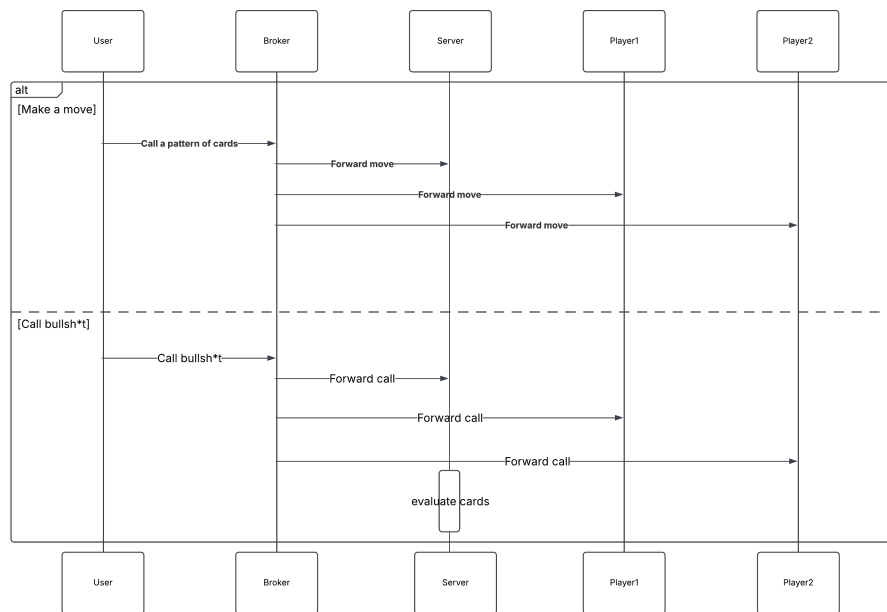


Figure 5: Sequence diagram of a generic game move

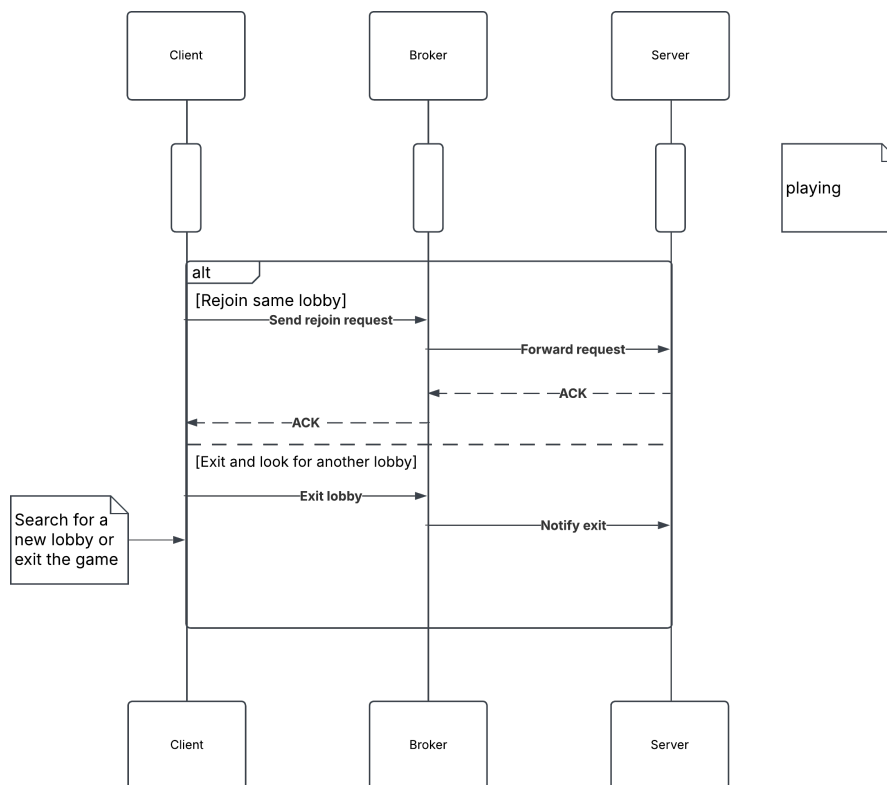


Figure 6: Sequence diagram of end game phase

3.5 Behaviour

Many actors are needed to keep the game going and to manage the communication between them.

The following are all the components actually needed and their behaviours:

- **The client** fig. 7

The client starts by establishing a connection to the MQTT broker, since it's fundamental for each part of the system.

Right after it asks the player for their name and waits for the server approval. If the request is successful, the client can now choose to join or create a lobby.

A player can set its status to ready when waiting for a game to start. Only when everyone is ready the server will send a message to start the game.

When the game starts, the client waits for the server to publish the game status to show the interface to the player. The client can only send messages to the server when it is his turn, either rising the stakes or calling bullsh*t. If a player loses the match then he can only wait for it to finish. The player leaves the lobby only when a winner is found. When the game ends the player is put back into the same lobby and can either wait for a new game to start or look for a new one.

At any point during the process the player can leave or be disconnected. If the player attempts to reconnect during a game the system will try to resume the status of the match.

- **The server** fig. 8

The server is mainly in charge of connecting new players and creating lobbies, delegating the game logic to different agents.

At any point in time a new player can ask to connect or disconnect himself, causing the need to check whether the player was alone in a lobby, deleting it if left empty. The players connected can also ask to create or join a new lobby and, when everyone is ready, it will launch a new game instance.

- **The broker** fig. 9

The broker is the central core to all communications between all the components. Always active using the MQTT standard.

- **The game agent** fig. 10

The game agent is a part of the server that takes on the role of managing the single matches. It is in charge of managing the game state, the players' moves and the game logic.

Every time a player rises the stake, it calculates and sends the next minimum stake that the next player can make. It only waits for a minute for each player to make a move, if the player does not make a move in time, the game agent will kick him from the game and the game will continue with the remaining players.

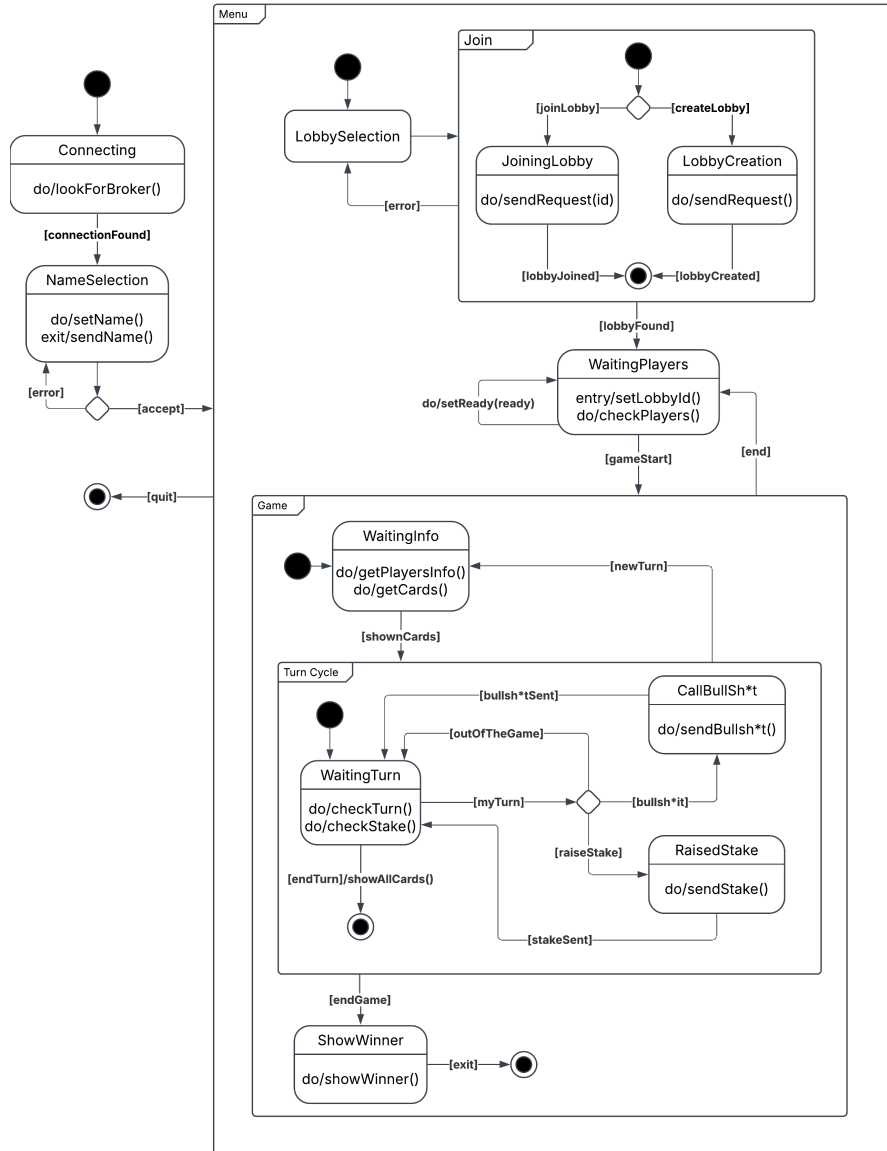


Figure 7: State diagram of the client

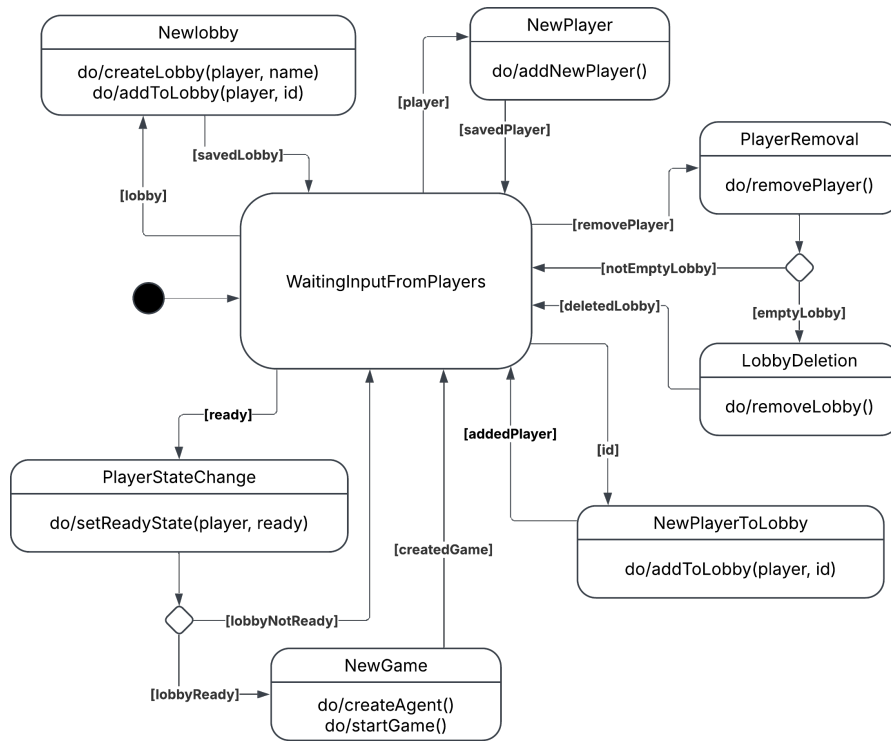


Figure 8: State diagram of the server

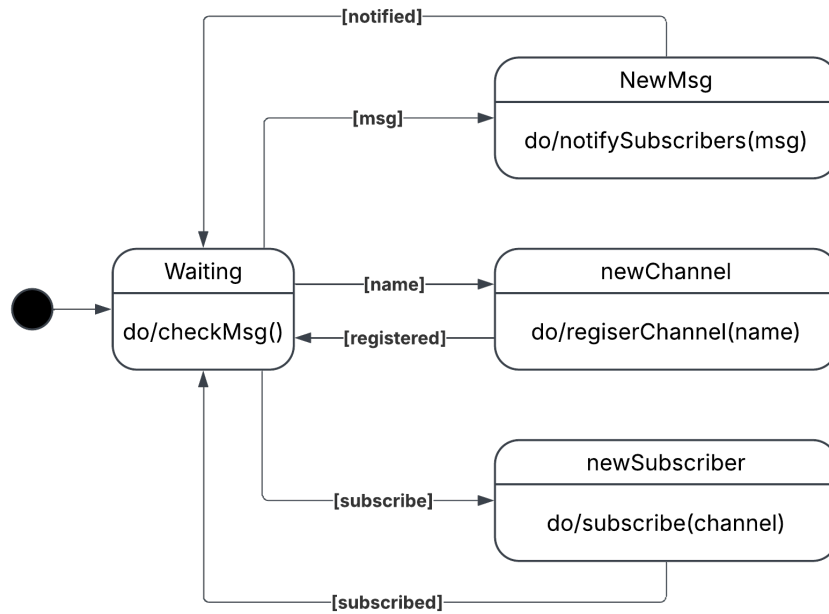


Figure 9: State diagram of the broker

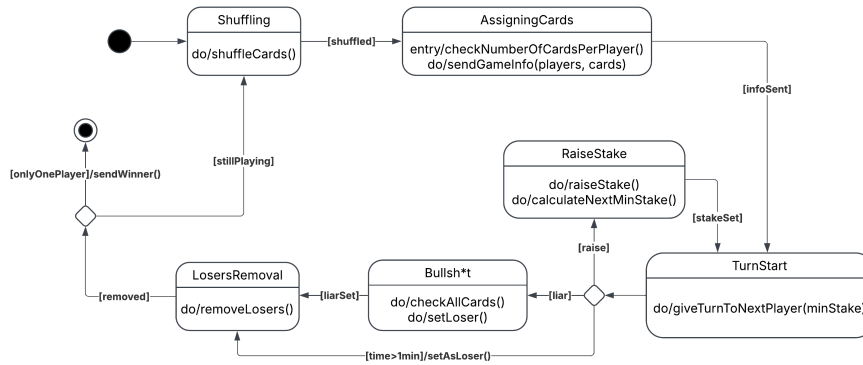


Figure 10: State diagram of the game agent

3.6 Data and Consistency Issues

Since we don't need to store data permanently because this is a card game, we don't deal with queries and databases.

Some data is shared between components, for example the game status, the players' nicknames, and other crucial information to perform the game.

3.7 Fault-Tolerance

Different types of fault tolerance are implemented in the systems, everything is set up to keep the game running even if some components fail.

The mechanics differ in their scope and purpose:

- **Server-Side**

The server and the broker are at the core of the game, if they fail the game is lost. To prevent this from happening, they are constantly monitored.

These share the same mechanism of fault-tolerance, redundancy and backup, but differ in implementation:

- **Backup Server**

The server is backed up by a second server that runs as a server but is not active; so it only listens to the messages sent by the clients and the broker and keeps a copy of the game state.

If the main server fails, the backup server can take its place and resume the communication between the clients and the broker. This mechanism has a limitation since doesn't keep a record of ongoing games, simply placing players back in lobbies. This happens because the game loop runs as a separate agent launched by the main server, so if the main server fails during a game, the game loop is lost too; in that case, all the players receive a message that the game is over before being redirected to the lobby screen.

All of this will use a heartbeat mechanism, where the server sends a heartbeat message to the backup every 2 seconds. If the backup does not receive the heartbeat message within the fixed time, it assumes that the main server has failed and takes his place elevating itself to the role of main server.

– **Broker Backup**

The broker is backed up by a second broker that keeps a copy of the messages sent by the clients and the server. If the main broker fails, the backup broker can take its place and resume the communication between the clients and the server. This will be using the mosquitto configuration file, where the main server is configured to send a copy of every message to the backup broker with a bridge connection.

The backup broker will be configured to listen to the same topics as the main broker, so it can receive the messages sent by the clients and the server.

When the main broker fails, the `ConnectionHandler` class both used by the client and server should try to connect to the backup broker, and if it succeeds, it can resume all communications where they left off.

• **Client-Side**

To make sure that no accidental disconnection can ruin the game, the game agent gives to each player a minute to make a move, if the player does not respond in time, it will be kicked from the game, continuing with the remaining players.

Nevertheless, if anything goes wrong the client always has the opportunity to try to reconnect, picking where it left off. While playing the web browser will save the user information in the local storage so that the client can retrieve it when trying to reconnect. If successful, the client will send a message to the server asking for the current game state, and the game agent will publish a message with the current game state.

3.8 Availability

We don't have any availability mechanism in our project. We only use caching for the broker that keeps the last message sent in a topic and the backup server that keeps a copy of the game state in case of failure.

3.9 Security

There aren't any security mechanisms in our project.

4 Implementation

4.1 Network Protocols

The main network protocol used is MQTT for the communication between the components. To implement the MQTT protocol we used the Paho library for Python, which is a client library for the MQTT protocol.

Thanks to MQTT protocol and its path-like structure, we can easily create topics for each lobby and each player, allowing them to communicate with the server and with each other.

An example of a topic structure is:

- `create_lobby` for the lobby creation messages
- `join_lobby` for players to join lobbies
- `lobby/<LOBBY_ID>/raise_stake` for the raise stake messages
- `lobby/<LOBBY_ID>/check_liar` for the liar detection messages

As we can see, the topics are structured in a way that allows us to easily identify the lobby and the type of message being sent. This is not always true, because all the messages sent before the lobby joining are sent to the general topic, so every player can see them, but after joining the lobby, the messages are sent to the specific lobby topic.

To recognize the messages in the general topic, we use a specific format for the messages, where the body contains the serialized player object. This part will be explained in more detail in the next sections.

The communication is done using a class called `ConnectionHandler`, which is responsible for managing the connection to the broker and the communication with the server. That class provides methods to subscribe to topics, publish messages, and handle incoming messages.

The clients are automatically subscribed to the game topics when they join a lobby, so they can receive the messages from the server. On the other hand, the server is subscribed to all the topics, so it can receive the messages from the clients and publish the game status.

This is the list of all the topics used in the system: (as said before, some topics are used directly in this way while others are used with a specific lobby ID and pattern):

- `lobby`
- `new_lobby`
- `new_player`
- `ready_to_play`
- `start_game`
- `join_lobby`

- `leave_lobby`
- `remove_player`
- `start_turn`
- `start_round`
- `round_loser`
- `game_over`
- `raise_stake`
- `check_liar`
- `heartbeat`
- `server_error`
- `game_info`

4.2 Data Formats and representations

To make the communication between components through the MQTT protocol possible, a way to serialize and deserialize the information in the form of messages is needed. We decided to use JSON as the data format of choice, since it is lightweight and easy-to-use. Through the use of the `json` library in Python, it was possible to create two classes used by the connection handler to, respectively, serialize and deserialize the information:

- **Serializer** is used to serialize the objects composing a Message dictionary into a JSON string.
- **Deserializer** is used to deserialize the JSON string back into a Python object, which can be used by the application.

To implement these classes, we used as a starting point the code provided in the lab lessons with professor Ciatto Giovanni.

In short the message sent start as a Python dictionary of objects, then it gets serialized into a JSON string and sent to the broker, which then publishes it to the subscribed clients. When a client receives a message, it deserializes the JSON string back into a Python dictionary, which can now be used to access the objects.

4.3 Technological details

The project is completely implemented in Python, using the following external libraries:

- `paho-mqtt` is used to implement the MQTT protocol and manage the communication between the components. This library provides a simple and easy-to-use interface to connect to the broker, subscribe to topics, publish messages, and handle incoming messages.

- nicegui is used to implement the user interface of the game, allowing players to interact with the game through a web browser. The strength of this library is the ability to create both the interface and logic of the client as a single Python script, which is then transformed into a single web application.

5 Validation

5.1 Automatic Testing

For the automatic testing we used the `unittest` framework, which is a built-in Python module for testing code. Every component of the system is tested with a set of unit tests that check the functionality of the methods and the correctness of the data. The command to run the tests is:

```
python -m unittest discover -s test -p '*_test.py'
```

- **Game**

The game class is tested with different inputs and expected outputs, to ensure that the game logic is correct. The tests check if the game can be started, if the players can be added and removed, if the game can be played, and if the game can be ended. The tests also check if the game state is correctly updated after each move.

- **Deck**

The deck class is tested to ensure that the deck is correctly shuffled and that the cards are correctly dealt to the players.

- **Stake**

The stake class is tested to ensure that the stake is correctly raised and that the minimum stake is correctly updated. The most complicated combinations of cards are tested to ensure that the stake is correctly checked when raised and the next minimum stake is correctly calculated.

- **Service**

The service module is tested to ensure that it can correctly serialize and deserialize messages.

- **Connection**

The connection module is tested to ensure that the connection to the broker is correctly established and that the messages are correctly sent and received. The tests check if the client can connect to the broker, if the client can subscribe to topics, if the client can publish messages, and if the client can receive messages. Mosquitto is required to run the tests, so the command to run the tests is:

```
mosquitto  
python -m unittest test.services.connection.connection_handler_test
```

5.2 Acceptance test

To test the interaction between the components we used manual tests, for example, we created a lobby and joined it with multiple clients, then we tested the interaction between the clients and the server. To run this type of test use the command:

- Windows

```
.\start_test_game_ui.ps1 [number_of_clients] [d]
```

- Linux

```
./start_test_game_ui.sh [number_of_clients] [d]
```

This will start the game with the specified number of clients, two mosquito instances and two servers. The clients connections are established from 127.0.0.1:8080 incrementing the port number for each client, so you can open multiple clients in different browser tabs. More over, the `d` flag is used to run the clients in debug mode, so you can see the logs in the console.

All the acceptance criteria were tested manually with these scripts and successfully passed.

We decided to not automatize this type of test because it requires a lot of interaction with the UI.

6 Release

The components are organized into a set of modules, each one with its own functionality. As shown in fig. 11, all the modules depends on models classes, but every other module is independent of the others, so they can be used separately; in fact it is possible to run single components importing only the modules described below:

- **Client**
model, services and view
- **Server**
model, controller
- **Broker**
config

Eventually, it is possible to add the utils module to use debug and log features.

7 Deployment

The software is deployed on GitHub, where the source code is available. The repository is public, so anyone can access it. The repository is organized in a way that allows to easily navigate through the code and find the files needed to run the software.

To download the software, you can clone the repository using the following command:

```
git clone https://github.com/FrancescoBuda3/LiarsPoker
```

or download the zip file from the repository page.

8 User Guide

8.1 Requirements

There are some requirements to run the software, they are listed below:

- Mosquitto broker installed (version 1.6.9) You can download it from the official website choosing the right version for your operating system:
 - Windows: <https://mosquitto.org/files/binary/>
 - Linux: <https://mosquitto.org/files/source/>
- Python installed (version 3.12 or higher) You can download it from the official website: <https://www.python.org/downloads/>
- Poetry installed (version 2.1.3) You can download it following the documentation from the official website: <https://python-poetry.org/docs/#installation>

8.2 Installation

Since the project is developed using Python, there are no particular installation steps, but you have to install the dependencies using Poetry.

To install the dependencies, you have to run the following command in the project root directory:

```
poetry install
```

8.3 Running the software

To run the software, you have to run the following command in the project root directory (each command must be run in a different terminal):

- Broker

```
mosquitto -c ./config/mosquitto1.conf  
mosquitto -c ./config/mosquitto2.conf
```

- Server

```
poetry run python src/controller/server/__init__.py primary  
poetry run python src/controller/server/__init__.py secondary
```

- Client

```
poetry run python src/view/__init__.py [port]
```

To run the system in debug mode, you just have to add the `-d` flag at the end of each command.

LIARS POKER!

START

Figure 12: Screen after the startup

8.4 Screenshots

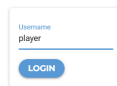
Below are some screenshots of the software in action.

The application starts in the browser with the screen shown in fig. 12 and after pressing start you get redirected to the login screen fig. 13.

After logging in, the fig. 14 shows the lobby selection screen, where the player can choose to create a new lobby or join an existing one. In fig. 15 is shown the lobby screen, where the player can see the players in the lobby and set his status.

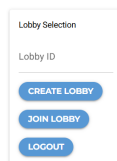
The other screenshots are related to the game:

- fig. 16 shows the in-game screen, where the player can see his cards, the current player, the last stake and two buttons to raise the stake or call bullsh*t.
- fig. 17 shows the dialog to choose the stake to raise.
- fig. 18 shows the dialog with the hint of the possible combinations of stakes
- fig. 19 shows the dialog with the all the cards that were in game when bullsh*t is called.



A screenshot of a login interface. It features a text input field with the placeholder text "Username" and the value "player". Below the input field is a blue button with the text "LOGIN" in white capital letters.

Figure 13: Screen to choose the nickname



A screenshot of a lobby selection interface. It has a title "Lobby Selection" and a text input field labeled "Lobby ID". Below the input field are three blue buttons with white text: "CREATE LOBBY", "JOIN LOBBY", and "LOGOUT".

Figure 14: Lobby selection screen

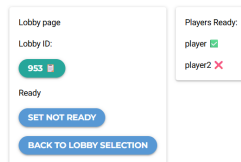


Figure 15: Lobby screen

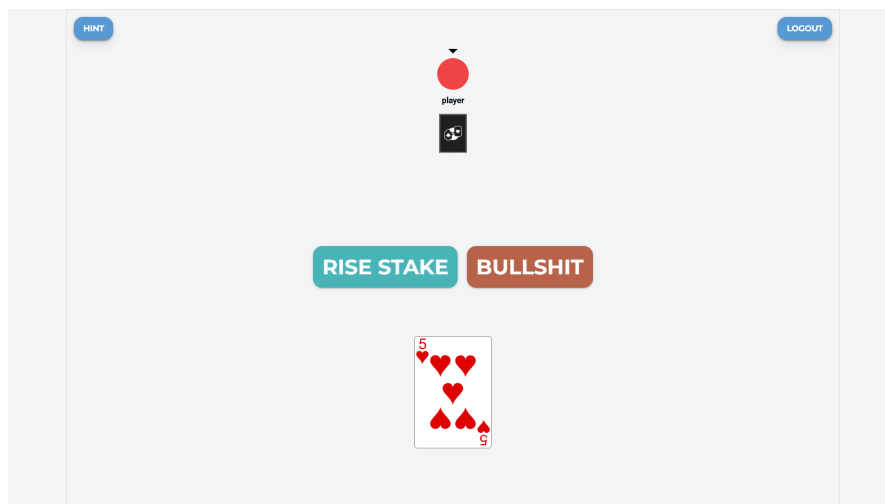


Figure 16: In-game screen

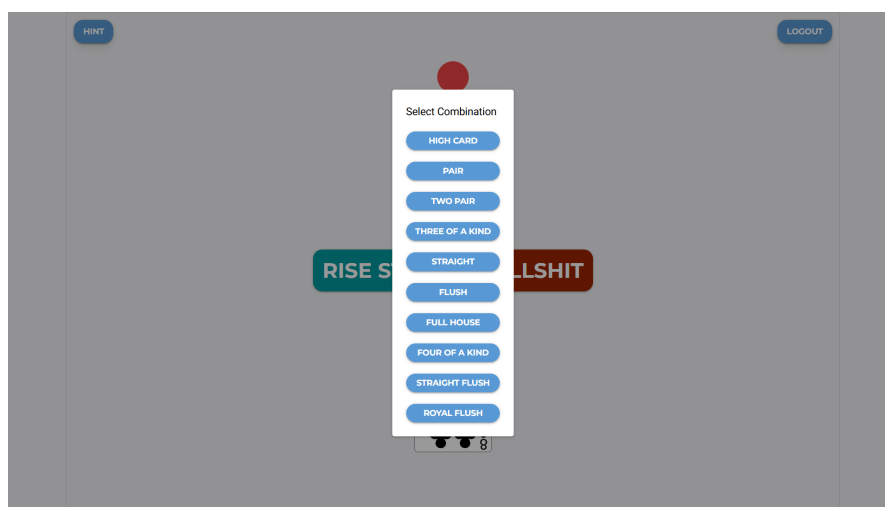


Figure 17: Dialog to choose the stake

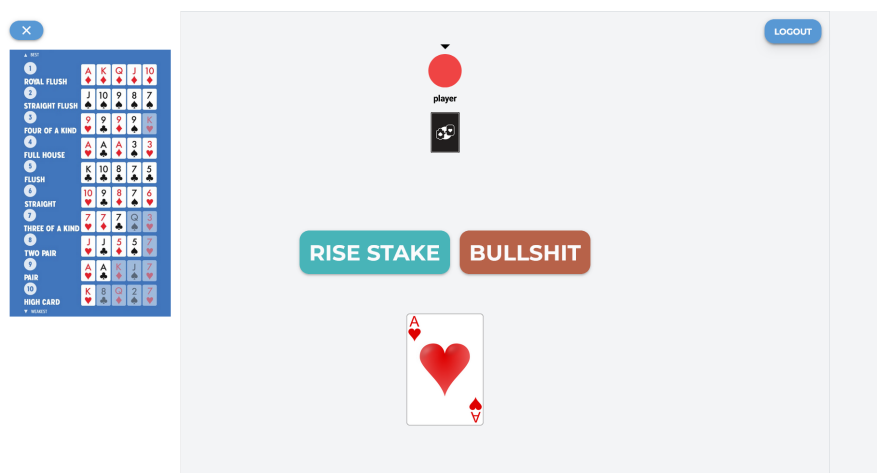


Figure 18: Dialog showing the hint for combinations

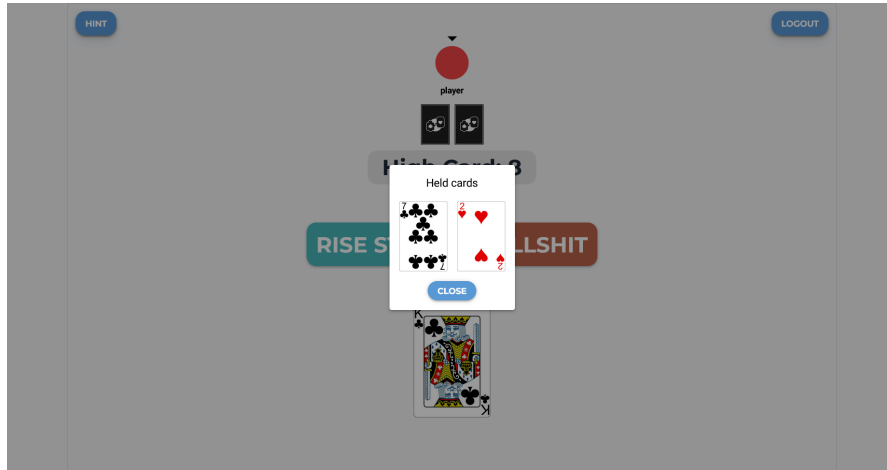


Figure 19: Dialog showing the cards in game after bullsh*t being called

9 Self-evaluation

9.1 Francesco Buda

As a member of the group, my first role was to think about the requirements and use cases of the application, considering also what we wanted the behaviour of the system to be in case of errors or unexpected situations. I implemented the core game logic classes, in charge of managing the game state, and designed the related message structures. I also implemented the deserializer class, the connection handler class, which is responsible for managing the message exchanging between the server and the clients, and the client-side fault tolerance mechanisms.

I think the workflow of the project was well organized, also thanks to Tommaso Severi, who was the one who kept the group together and made sure that everything was working as expected having a precise overall view on the project. On my side, I personally gave my best to be as good as possible in all the situations and I think the final product is a well-made project, with a good level of complexity and good code quality. As the design is, on my opinion, strong and well done, the weaknesses or improvements that could have been made are mainly related to the technical side in order to improve the reliability, security and availability of the system.

9.2 Emanuele Sanchi

My role in the group was to design the architecture of the system and to implement the server-side logic designing also the interactions between the components. I implemented the server-side classes, which are responsible for managing the game state and the players' actions, and I also implemented the fault tolerance mechanisms on the server-side (both for python server and for the MQTT broker). I also implemented much of the MessageFactory class methods; this class is responsible for creating the messages with a

certain structure. In addition, I helped to create a base version of some client-side pages like the game page or the lobby selection page, to test my previous work like interaction with the server and with the other players. The final version of these pages were then implemented by the rest of the team. The weaknesses or improvements that could have been made are mainly related to the user experience, the security of the system which could be improved in future versions and the fault tolerance mechanisms that could be more robust for example by adding a database to store the game state and the players' actions. Overall, I am satisfied with the result and I think that the project is a good example of a distributed system.

9.3 Tommaso Severi

I was in charge of designing the class and state diagrams of the system in the design phase, always keeping an eye on the requirements and the architecture of the system during the whole process. During the implementation I mainly occupied myself in the stake related logic and representation in the system, but also adapted the serializer class. I created the game-agent, implemented the ready logic of lobbies in the server and further implemented the view modules. My role in the group was to be the glue between the components, making sure that everything was working together and that the system was coherent with the design. I think that the final product has a high level of code quality, even with the restrictions of the Python language, and the architecture is well-thought-out, allowing for easy maintenance and future extensions. Despite that it surely could be polished adding more controls for corner cases and improving the user experience, overall I am more than satisfied with the result. I think that it is a great proof of concept, completely working, of a distributed system, with a good level of complexity that shows the capabilities of the technologies used.

References

- [1] Dmitrii Krashennnikov. Liar's poker: a new card game you should try. <https://krashennnikov.github.io/liars-poker>, August 2024.