Liar's Poker

Francesco Buda
francesco.buda3@studio.unibo.it

Emanuele Sanchi
emanuele.sanchi@studio.unibo.it

Tommaso Severi
tommaso.severi2@studio.unibo.it

Febraury 2025

Up to ∼2000 characters briefly describing the project.

1 Concept

This project aims to develop a web-based game application for playing Liar's Poker, a fun and engaging card game centered 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 an online platform where users can enjoy the game with others in real-time, no matter where they are.

1.1 Game Rules

Each player starts with one card. A round starts with someone naming a poker combination. With the turn rotating clockwise, each next player has a choice:

- **Rise the stakes**

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

- **Call bullshit**

  the player calls bullshit on the previous player's combination if the player think the previous player is bluffing or mistaken.

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

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 game goes on

- **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. **Lobby creation**

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

2. **Joining the lobby**

   Other users join the lobby by entering the code. While joining, the players will be asked to choose a unique nickname to use during the game.

3. **Gameplay**

   Each player receive one card and than the game starts (The first player is chosen randomically). The game proceeds as described in the rules. when the player want to rise the stakes, he can choose from the possible combinations (non valid combinations will be disabled), if the combination needs some specific card to be declared, the player will be provided with a list of possible cards to choose from.

4. **Elimination**

   When a player loses, they are eliminated from the game and they can chose to spectate the game or leave the lobby.

5. **End of the game**

   When the game ends the winner is declared and players who are still in the lobby will be asked wether they want to play another game or leave the lobby.

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. **Joining the Lobby**

   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 100 concurrent games.

2. **Availability and Fault Tollerance**

   The application must be able to handle disconnections and server failures without losing game data.

   - **Acceptance Criteria:** The system must be able to recover from server failures within 5 seconds.

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 1 second from the bullshit 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**

   the player has the highest card in the game

2. **Pair**

   in the combined cards of all players there are two cards with the same value

3. **Two Pair**

   in the combined cards of all players there are two pairs of cards with the same value

4. **Three of a Kind**

   in the combined cards of all players there are three cards with the same value

5. **Flush**

   in the combined cards of all players there are five cards with the same suit

6. **Straight**

   in the combined cards of all players there are five cards with consecutive values

7. **Full House**

   in the combined cards of all players there are three cards with the same value and two cards with the same value

8. **Four of a Kind**

   in the combined cards of all players there are four cards with the same value and one card with a different value

9. **Poker**

   in the combined cards of all players there are four cards with the same value

10. **Straight Flush**

    in the combined cards of all players there are five cards with consecutive values and the same suit

3 Design

This chapter explains the strategies used to meet the requirements identified in the analysis. Ideally, the design should be the same, regardless of the technological choices made during the implementation phase.

> You can re-order the sections as you prefer, but all the sections must be present in the end

3.1 Architecture

The architecture of the system is a classic MVC where:

- Model are the classes that represents 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 also 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.
The server is the one that creates the game; then the clients can join the game publishing their nickname to the broker; every time a new player joins the game, it automatically subscribe to the game topics.
All the clients are subscribed to the same topic, so they can receive the messages from the server (eg. moves, game status, etc.).

We choose this architecture because it is simple and it is suitable for our project. In fact, using a publish-subscribe architecture, clients have not 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 so is the only one who can publish messages on some specific topics
- 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 semplicity, 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 (unique) that every body choose when they join the game.

Dotted lines are for subscribe messages, while solid lines are for publish messages.

3.3 Modelling

- which **domain entities** are there?
  - e.g. *users*, *products*, *orders*, *etc.*
- how do *domain entities* **map to** *infrastructural components*?
  - e.g. state of a video game on central server, while inputs/representations on clients
  - e.g. where to store messages in an IM app? for how long?
- which **domain events** are there?
  - e.g. *user registered*, *product added to cart*, *order placed*, *etc.*
- which sorts of **messages** are exchanged?
  - e.g. *commands*, *events*, *queries*, *etc.*
- what information does the **state** of the system comprehend
  - e.g. *users' data*, *products' data*, *orders' data*, *etc.*

  Class diagram are welcome here

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 is going to send 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; on the other hand, 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 subsections will explain the interaction between the components in more detail.

3.4.1 Connection

3.4.2 Game Move

3.4.3 End Game

3.5 Behaviour

- how does *each* component **behave** individually (e.g. in *response* to *events* or messages)?
  - some components may be *stateful*, others *stateless*
- which components are in charge of updating the **state** of the system? *when*? *how*?

  State diagrams are welcome here

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 datas are shared between components, for example the game status, the players' nicknames, and other crucial information to perform the game.

3.7 Fault-Tolerance

- Is there any form of data **replication** / federation / sharing?
  - *why*? *how* does it work?
- Is there any **heart-beating**, **timeout**, **retry mechanism**?
  - *why*? *among* which components? *how* does it work?
- Is there any form of **error handling**?
  - *what* happens when a component fails? *why*? *how*?

3.8 Availability

- Is there any **caching** mechanism?
  - *where*? *why*?
- Is there any form of **load balancing**?
  - *where*? *why*?
- In case of **network partitioning**, how does the system behave?
  - *why*? *how*?

3.9 Security

There aren't any security mechanisms in our project.

---

4 Implementation

- which **network protocols** to use?
  - e.g. UDP, TCP, HTTP, WebSockets, gRPC, XMPP, AMQP, MQTT, etc.
- how should *in-transit data* be **represented**?
  - e.g. JSON, XML, YAML, Protocol Buffers, etc.
- how should *databases* be **queried**?
  - e.g. SQL, NoSQL, etc.
- how should components be *authenticated*?
  - e.g. OAuth, JWT, etc.
- how should components be *authorized*?
  - e.g. RBAC, ABAC, etc.

4.1 Technological details

- any particular *framework* / *technology* being exploited goes here

5 Validation

5.1 Automatic Testing

- how were individual components ***unit*-test**ed?
- how was communication, interaction, and/or integration among components tested?
- how to ***end-to-end*-test** the system?
  - e.g. production vs. test environment
- for each test specify:
  - rationale of individual tests
  - how were the test automated
  - how to run them

  - which requirement they are testing, if any

  recall that *deployment* **automation** is commonly used to *test* the system in *production-like* environment

  recall to test corner cases (crashes, errors, etc.)

5.2 Acceptance test

- did you perform any *manual* testing?
  - what did you test?
  - why wasn't it automatic?

6 Release

- how where components organized into *inter-dependant modules* or just a single monolith?
  - provide a *dependency graph* if possible
- were modules distributed as a *single archive* or *multiple ones*?
  - why?
- how were archive versioned?
- were archive *released* onto some archive repository (e.g. Maven, PyPI, npm, etc.)?
  - how to *install* them?

7 Deployment

- should one install your software from scratch, how to do it?
  - provide instructions
  - provide expected outcomes

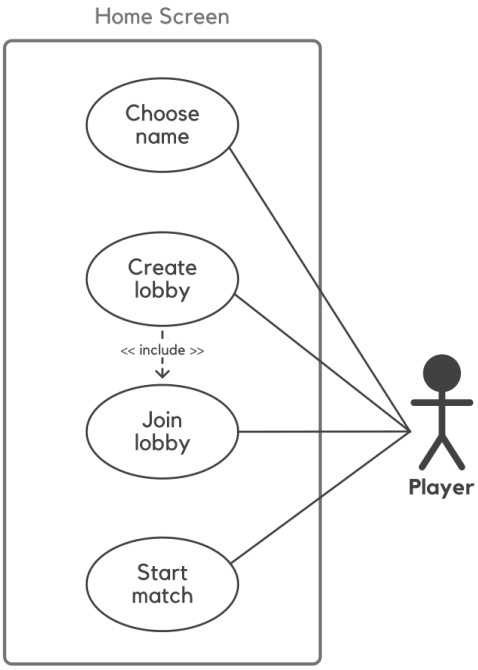
8 User Guide

- how to use your software?
  - provide instructions
  - provide expected outcomes
  - provide screenshots if possible

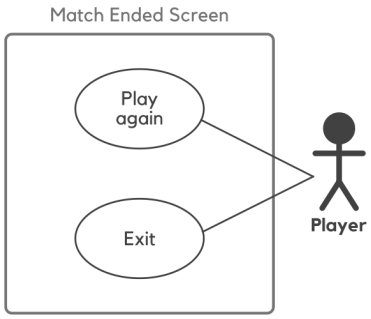
9 Self-evaluation

- An individual section is required for each member of the group
- Each member must self-evaluate their work, listing the strengths and weaknesses of the product
- Each member must describe their role within the group as objectively as possible.

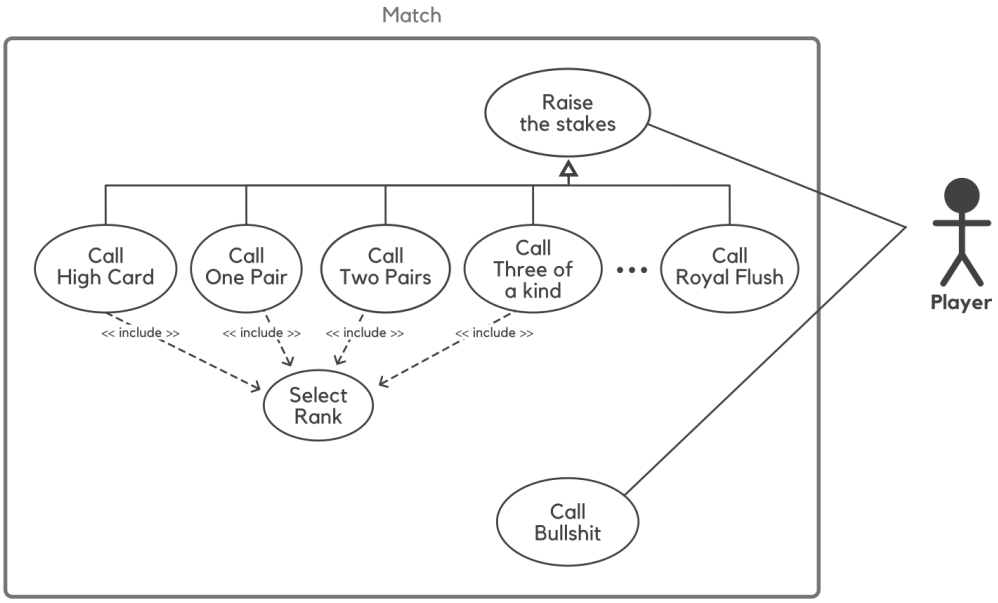
It should be noted that each student is only responsible for their own section [**?**]



(a) Home Screen



(b) Match Ended Screen



(c) Match

Figure 1: Use Case Diagrams

Figure 2: Architecture diagram