Dubito Online

- Alberto Arduini
- Andrea Bianchi

AI Disclaimer (if needed)

```
"During the preparation of this work, the author(s) used Claude.Ai to generate
small
pieces of code to develop the Game Board for Dubito Online. This was used to
help speeding up the development
of methods such as how to properly rotate game card buttons in the GUI and how
to deactivate certain
buttons in the code based on the developed logic.
After generating the initial prompt, the author(s) then reviewed and edited
the
content as needed and take(s) full responsibility for the content of the
final report/artifact."
```

Abstract

This project's objective is to recreate a version of the famous card game called "Dubito", following the rules and ideas of the videogame "Liar's Bar" and "Master Bluff".

The game rules are the following:

- At the start of each round each player receives 5 cards and a specific type of card is declared (Kings, Queens, Aces);
- Players during their turn can do one of the following actions:
  - Discard up to 3 cards from their hand facedown, those being of the same type that was declared at the start of the round or not.
  - Call the previous player "liar", if the player is not the first one playing in the round.
- When a player calls for a lie, the cards that were discarded are shown and 2 possible scenarios may occur:
  - If that player did lie, the lying player loses 1 life.
  - If that player did not lie, the one who called for the bluff loses 1 life.
- Each player starts with 2 lives, and the last remaining living player is declared the winner.

Concept

Artifacts

The project will result in a *group* of applications that will let the users host, join and play games of dubito with other users.

The complete work is comprised of two applications:

- The LobbyServerHost, a CLI application that will host a server where users can connect to create and join lobbies;
- The GameApp, a Swing graphical application that the users can use to connect to a LobbyServerHost where they can play the game.

Interactions and use cases

All users will be able to:

- Use the application to create or find a game to play with other users;
- Participate in a game by doing the actions expressed in the rules.

Additionally, users who create games will become those lobbies' *owners*, which will give them additional functionality to manage the game(s) they create.

Requirements

The following sections group the system requirements by type and by domain functionality. Each requirement will be followed by its own acceptance criteria(s).

1. Functional

1.1 Lobby Management

- 1.1.1: The app must let a user see available lobbies created by other users.
  - The user must be able to see a list of all lobbies with their name, an indicator if the lobby is password protected and the number of users in the lobby.
- 1.1.2: The app must let a user join an existing lobby, even if password-protected, while it is not in a lobby.
  - The user must be able to join a lobby from the list (see Req. 1.1.1) if the lobby has not reached the maximum number of participants.
  - If the lobby is password protected the app must prompt the user for the password.

    - When a user successfully joins a lobby, all other users should be notified of the updated participant count for the lobby.
    - The functionality must be available only when the user is not in a lobby already.
    - The user must not be able to join multiple lobbies concurrently.
- 1.1.3: The app must let a user create a new lobby, while the user is not in a lobby.
    - The user must be able to create a new lobby that other users will be able to see and join.
    - The functionality must be available only when the user is not in a lobby already.
    - When a user creates a lobby, it becomes also it's owner.
    - The user can't be the owner of multiple lobbies concurrently.
- 1.1.4: The app must let a lobby owner set the lobby name and password.
    - The lobby owner must be able to insert or change the lobby name and the password.
    - The name must not be empty, while the password *may* be empty.
    - When the changes are applied, all other users (both in the lobby and outside) must receive the updated information.
- 1.1.5: The app must let a user that is in a lobby leave it.
    - The app must let a user exit from the lobby it is in, notifying all other users of the updated user count for the lobby.
    - The functionality must be available only when the user is in a lobby.
- 1.1.6: The app must let the lobby owner delete its lobby.
    - The lobby owner must be able to delete its lobby, notifying all users that the lobby doesn't exist anymore and kicking out the users who where in the lobby.
- 1.1.7: The app must let the lobby owner start the game with all the lobby users as participants.
    - The app must let the lobby owner start a game match where the users that are in the lobby become players in the game.
    - Also, no more players must be able to join the lobby.
- 1.1.8: The app must show, when a user is in a lobby, the other participants.
    - The user must be able to see a list of all the participants in the lobby it is currently in, with their name.
    - The app *may* emphasize who is the local user and who is the lobby owner.

1.2 Gameplay

- 1.2.1: The app must let every player know which cards they have in their hand and what's the current round card value.
    - The user must be able to see *only* their hand and the cards that comprise it, without having the ability to see other players' hands.

    - The app must clearly show the current round card value so that players may know which of their cards can be discarded safely.
- 1.2.2: The app must let each player discard a variable amount of cards from their hand.
    - During their turn, a player must be capable of selecting up to 3 cards from their hand that can then be discarded with the press of a button or a key.
    - The app must notify every other users how many cards the turn player has discarded.
- 1.2.3: The app must let each player "call a lie" if they believe the previous player has not played cards that are of the same type as the declared one for the current round
    - During their turn, if a previous player has discarded some cards, the player may press a button or key to call the previous player a liar.
    - The app must then notify every player if the player has made a bluff (lied) or not, removing 1 life from the lying player or the one who called the bluff if he was wrong.
    - The app must start a new round after this event, giving a new random hand to each player and declaring a new card type for the new round.
- 1.2.4: The game must start a new round if no one calls for a lie.
    - If every player decides to only discard cards without anyone calling for a possible lie, the app must then start a new round.
- 1.2.4: The game can only be considered completed if only one player remains.
    - When a player loses all their lives, they are out of the game and will not receive new cards or be part of the turn order for the next rounds.
    - The player must be capable of leaving the current lobby once they are considered dead.
    - When every player except one dies, the last alive player is considered the game's winner, notifying each player of their success.
- 1.2.5: The game must continue working correctly if a player leaves the game early.
    - The app must continue the turn order if one player (the current one or not) disconnects from the lobby, letting the game continue as normal.
    - If only one player remains and all the others disconnect, the remaining player must be declared the winner of the game.

1.3 Miscellaneous

- 1.3.1 The app must let a user set its display name.
    - The user must be able to set a name that will be shown to other users.
    - No restriction should be applied if the name is already used, so two users can have the same name.

2. Non-functional

- 2.1: The lobby management should not have network partition protection systems in place.
  - If the lobby management system has network problems or is not available, then the users can not access it.
  - By consequence, if a single user loses connection to the lobby management system, it's the user responsibility to connect back to it.
- 2.2: The lobby management must provide always consistency and correctness of data.
  - The information the users receive from the lobby management system must always be correct and up-to-date, taking into account network delays.
- 2.3: The game management should not have network partition protection systems in place.
  - If a users loses connection to the game, he won't be able to connect back.
- 2.4: The game management should notify each player of other users' actions in a consistent manner, while still keeping a certain level of availability.
  - When a player discards a certain amount of cards, even players that are not immediately after them in the turn order should be notified of the game change that occurred.
  - When a new round starts, each player must have their UI updated (they must have the most recent and updated view of the game).

3. Implementation

- 3.1: The applications will be developed using the Java language and the Swing framework, since they are what the development team are most familiar with.
- 3.2: The network management will be implemented from scratch, so that the behavior of the applications can be controlled more precisely.

Design

This chapter explains the strategies used to meet the requirements identified in the analysis, describing its architecture, infrastructure and important aspects that will later be expanded in the implementation.

Architecture

The project follows the Mvc architecture to develop its main logic and program.

Focusing on the distributed part, the project is divided into 2 main different parts, each focusing on one specific aspect of the previously established requirements:

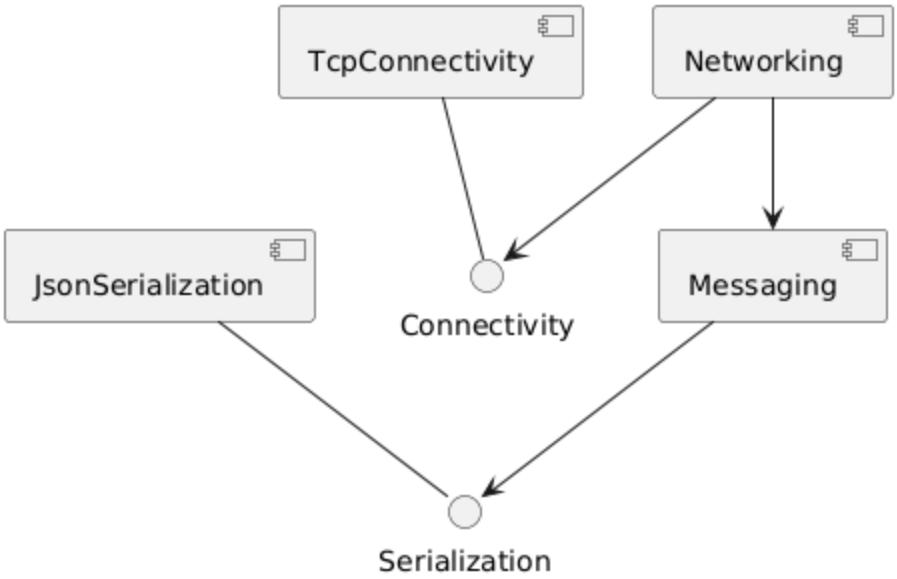
- The lobby system was developed following the **client-server** architecture, one of the most typical distributed application structure. This allowed us to partition tasks/workloads between the providers of a resource or service (the server lobby) and the service requesters

(the user clients). Clients can request resources or operations to the server, and the server will respond with the required data. The server can also send unsolicited data to the clients, to notify when a change in the system has happened;
- The main game is, instead, build using a **Peer-To-Peer (P2P)** architecture. A peer-to-peer network is designed around the notion of *equal peer nodes* simultaneously functioning as both "clients" and "servers" to the other nodes on the network. Each in-game user, along with remaining connected to the lobby server, also becomes a peer.
  Each player can then send resources or operation results based on their in-game actions (discarding cards, calling liar) to the other peers, without requiring the usage of a central coordination system when updating the game state after each action.

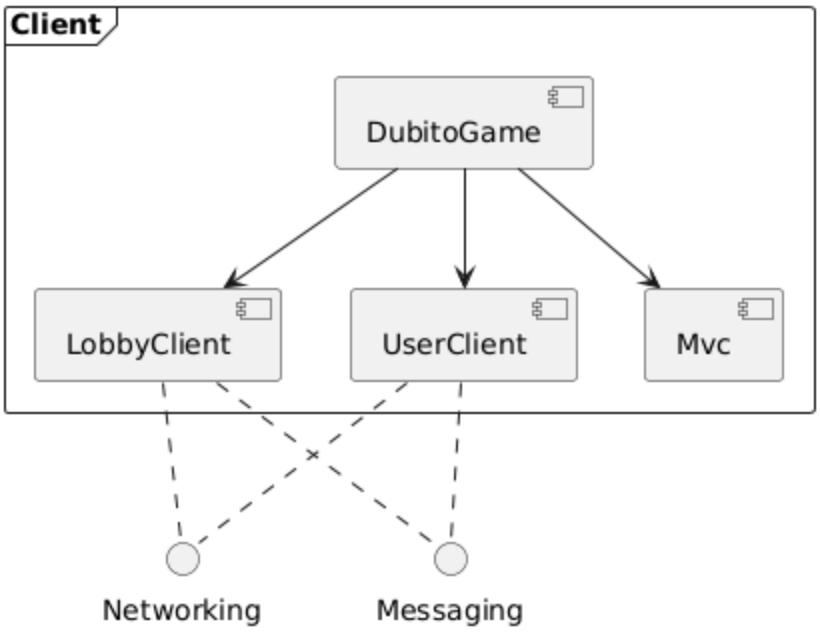
Infrastructure

The project's infrastructure was developed and composed as such:

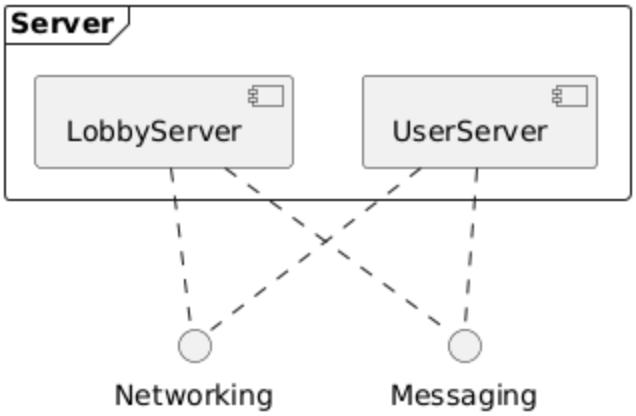
- The distributed configuration is, usually, the following:
  - $[1, N]$ machines host the lobby server(s)
  - $[0, N]$ machines act as clients
- Each client can connect to a lobby server with its IP address and port;
- Each player is capable of creating a lobby (either password-protected or not), containing up to a max of 4 players, where other players may join;
- When starting the game, the lobby owner player creates a new P2P network, while the lobby server sends to all other players in the lobby the owner's IP and port, so that they can connect to it;



Infrastructure of the backend libraries



Client-side infrastructure



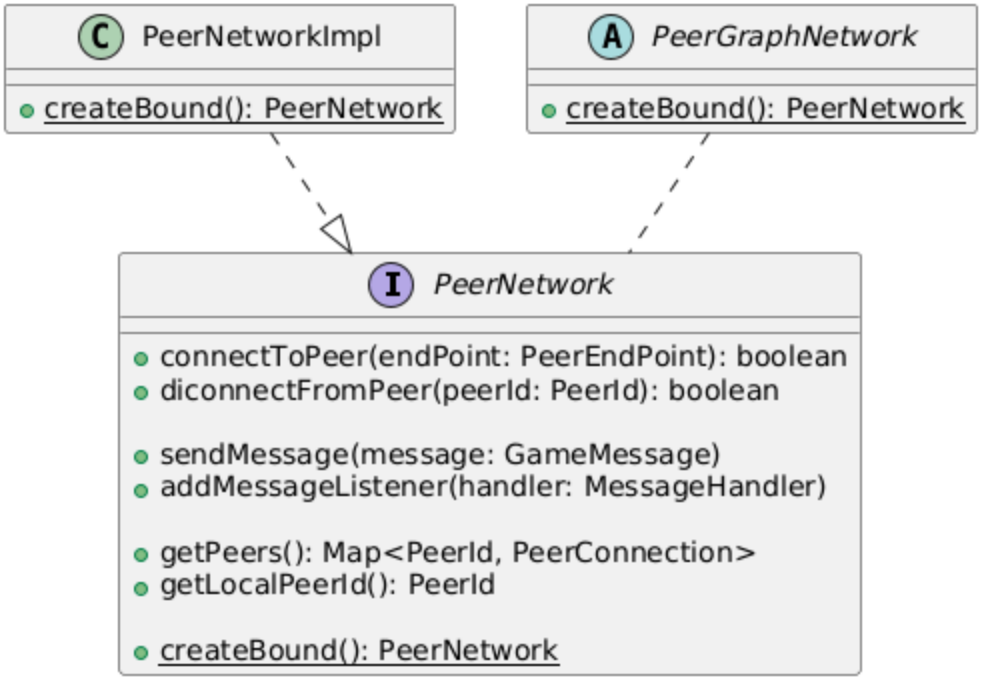
Server-side infrastructure

Modeling

The distributed system is based on the `PeerNetwork` interface. It declares a contract for interacting with the network in terms of Peers and messages, abstracting away how the network is managed.

This interface is then implemented in `PeerNetworkImpl`, which acts as a simple server for other clients, with the ability to connect to other networks.

Then, on top of those, the factory `PeerGraphNetwork` creates a P2P `PeerNetwork` where all clients are connected to one another automatically, by implementing an auto-discovery system for clients.



The system is supported by other classes that create the infrastructure needed to create the distributed system, which are:

- `PeerConnection`: An interface for a client connected to a server;
  - Implemented in `TcpPeerConnection`;
- `PeerConnectionReceiver`: An interface for a server listening to `PeerConnection`;
  - Implemented in `TcpPeerConnectionReceivr`;
- `PeerExchanger`: Exchanges the initial information between two clients after they connect;
- `MessageHandler`: A functional interface for receiving messages incoming from the network;
- `ObjectSerializer`: An interface for an object that can serialize/deserialize another object, so that it can be sent over the network.
  - Implemented in `JsonObjectSerializer`.
- `PeerEndPoint`: A serializable data class wrapping an IP/port pair.
- `PeerId`: A serializable data class wrapping an opaque identifier for a peer.

Entities

- LobbyServer: the networked entity that provides the list of lobbies and users;
- LobbyClient: a client that connects to the LobbyServer and consumes the lobby and user lists;
- GamePeer: a peer connected to other peers during an active game.

Events

Domain events change depending on whether the user is in game session or not:

Events can be divided in groups:

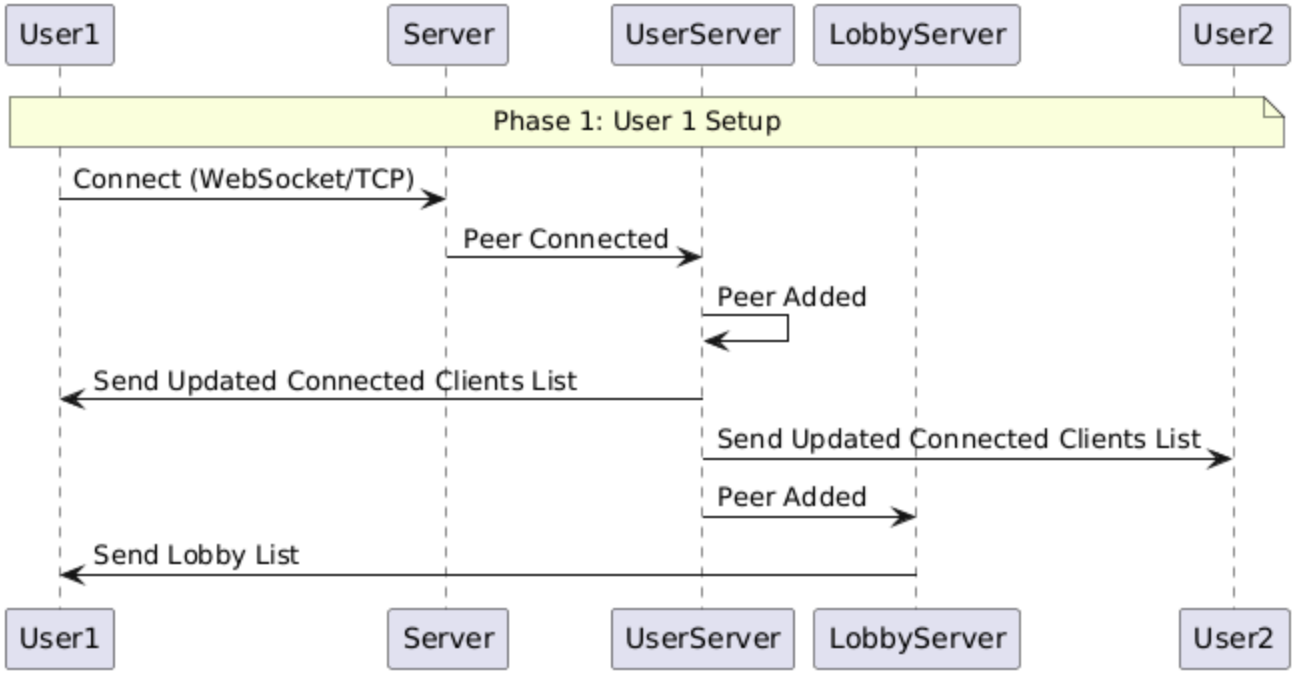
- Lobby management events:
  - A user requested to create a new lobby;
  - A user requested to change the information of a lobby;
  - A user requested to join a lobby;
  - A user requested to leave a lobby;
  - A user requested to start a game;
  - The lobby list is updated;
  - The current client's lobby information is updated;
- User management events:
  - A new user has connected;
  - A user has disconnected;
  - A user requested to change its username;
  - The user list is updated;
- Game events:
  - The current round card is changed
  - A user hand is changed
  - A user plays cards from his hand;
  - A user calls the previous user a liar.

Interaction

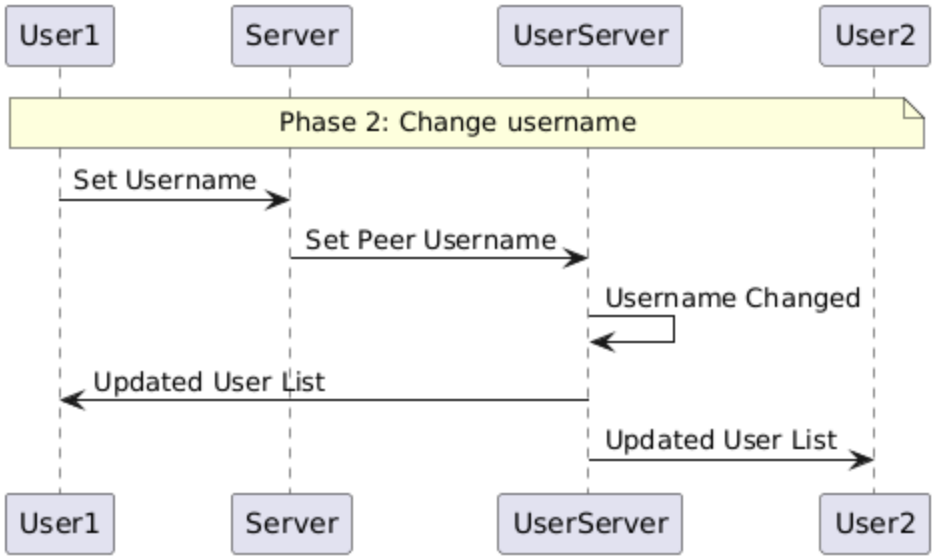
At a high level, interactions between components can be grouped by:

- When the user is not in a game:
  1. A client sends a message on the network to the server;
  2. The server receives message and performs a specific action accordingly, replying to the client and/or notifying all clients;
- When the user is in a game:
  1. A client send a message to all other clients, so that the state remains synchronized between all clients.

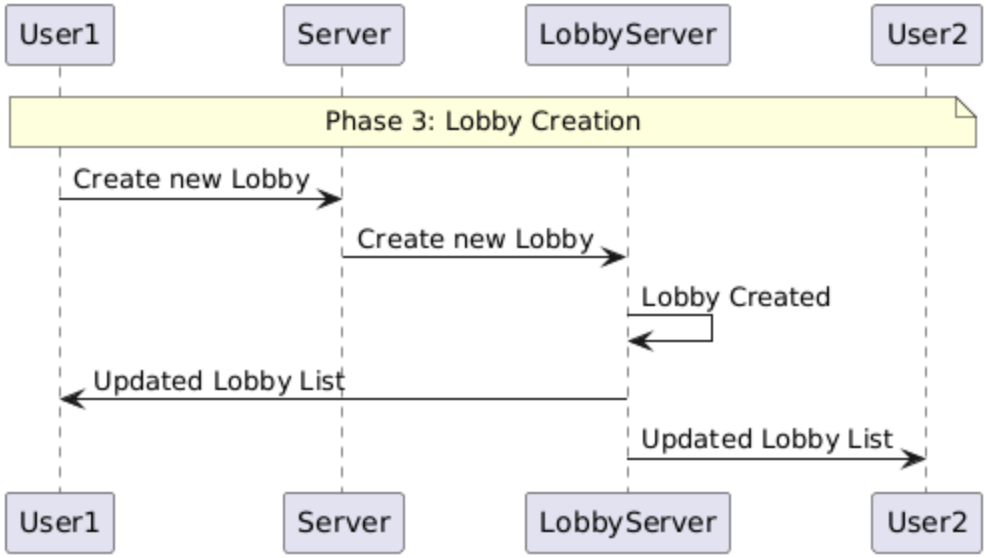
More in detail, the interactions for specific actions are the following:



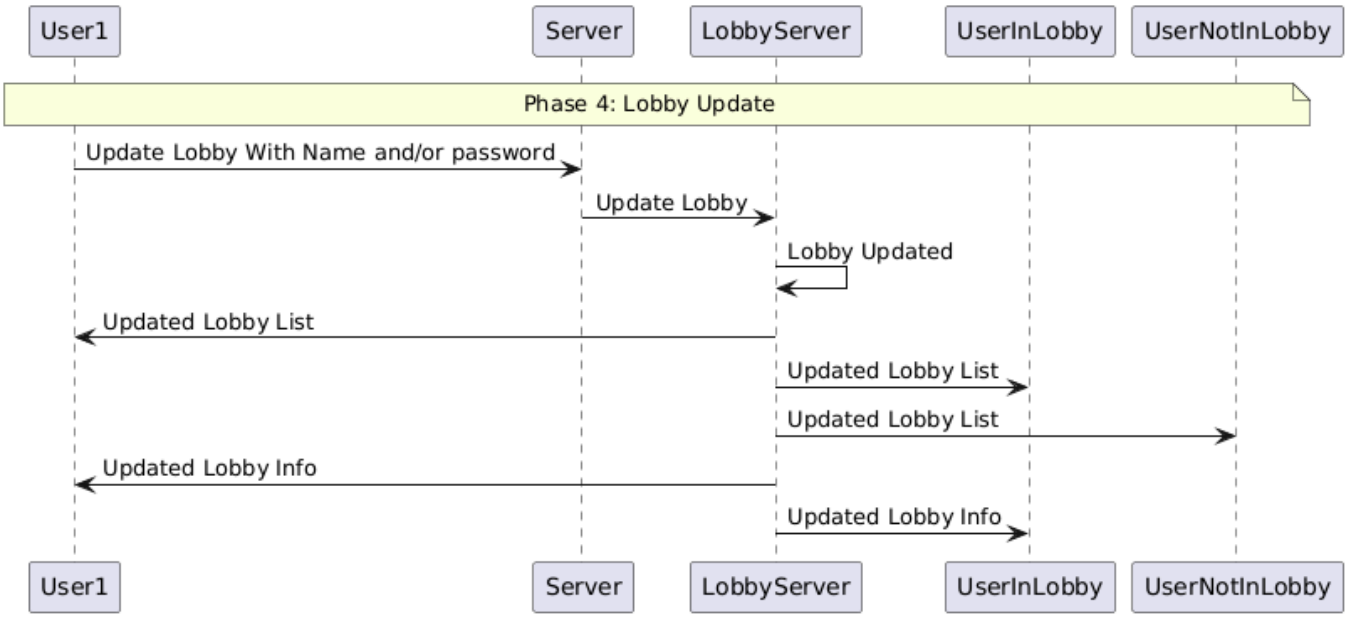
Application Startup and User Setup sequence diagram



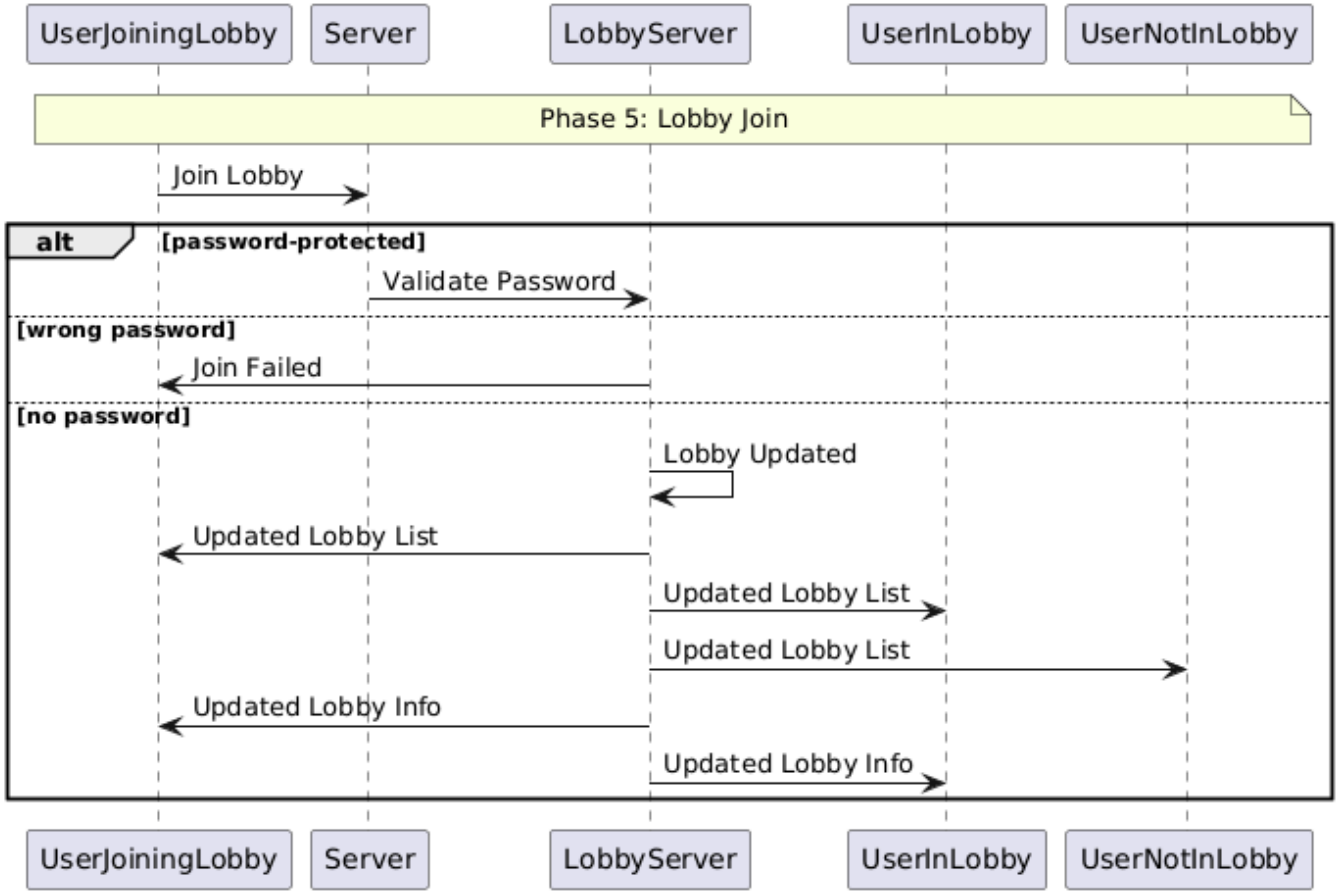
Change of username sequence diagram



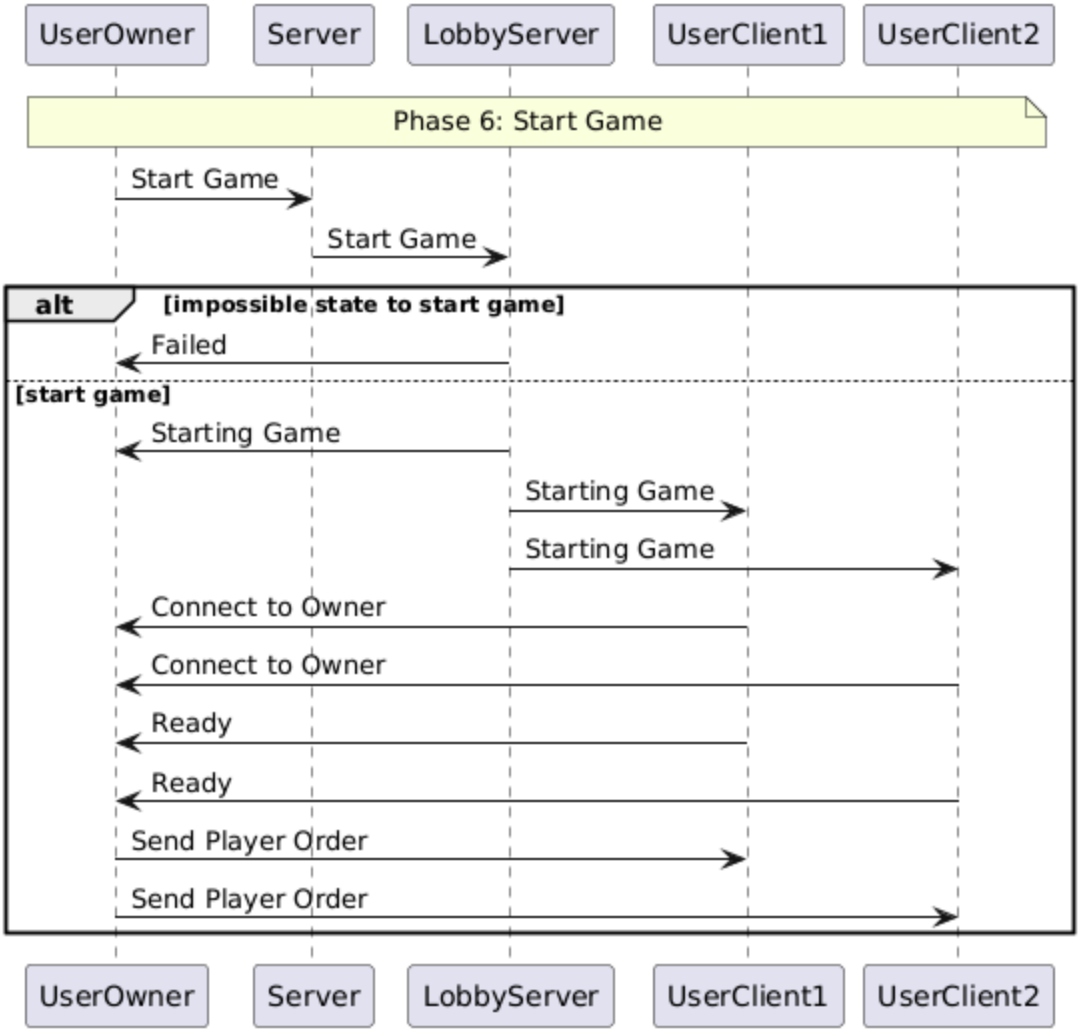
Creation of new lobby sequence diagram



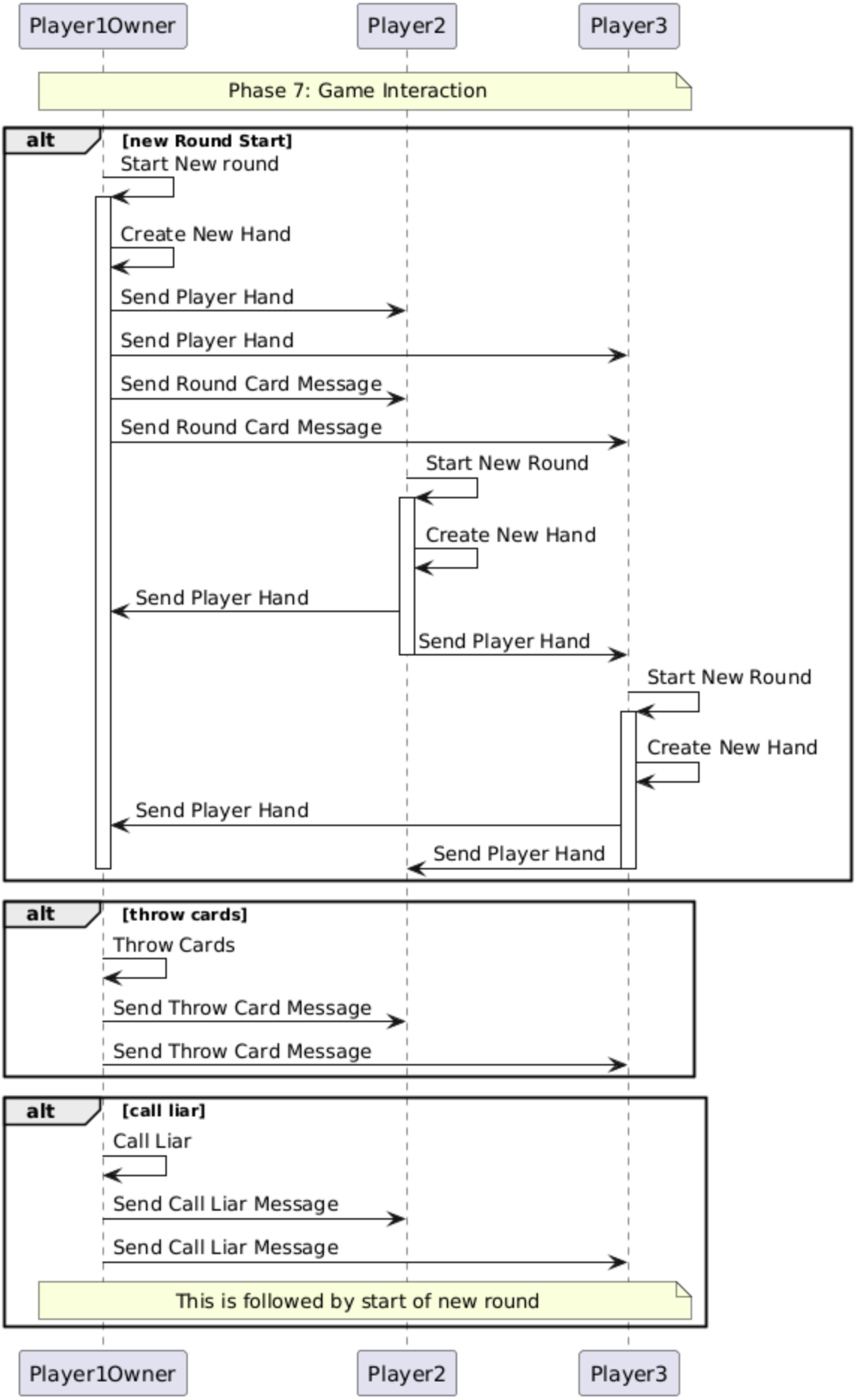
Update of lobby (either setting username or password) sequence diagram



Lobby Join sequence diagram



Start of game sequence diagram


Player1Owner
Player2
Player3
Phase 7: Game Interaction
alt
[new Round Start]
Start New round
Create New Hand
Send Player Hand
Send Player Hand
Send Round Card Message
Send Round Card Message
Start New Round
Create New Hand
Send Player Hand
Send Player Hand
Start New Round
Create New Hand
Send Player Hand
Send Player Hand
alt
[throw cards]
Throw Cards
Send Throw Card Message
Send Throw Card Message
alt
[call liar]
Call Liar
Send Call Liar Message
Send Call Liar Message
This is followed by start of new round
Player1Owner
Player2
Player3

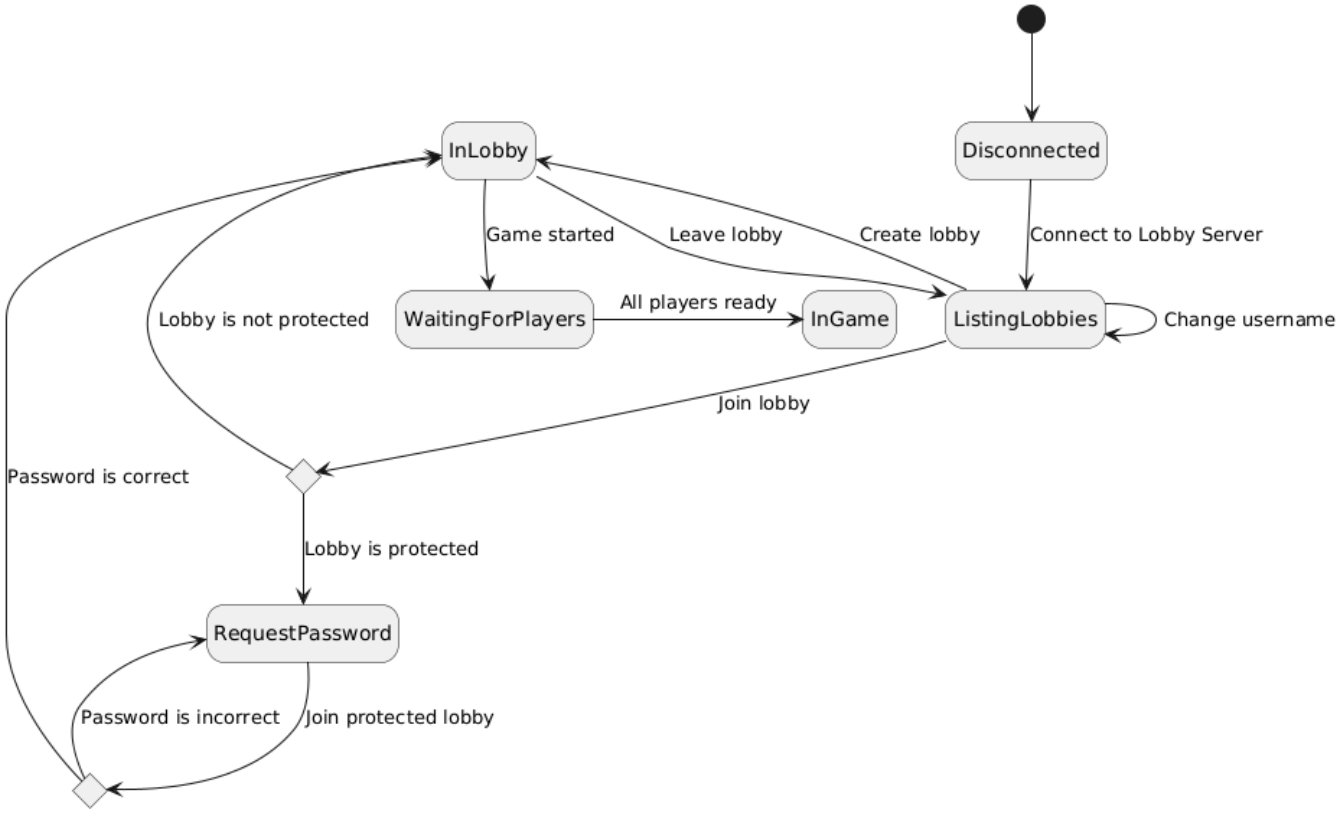
In-game interaction sequence diagram

Behavior

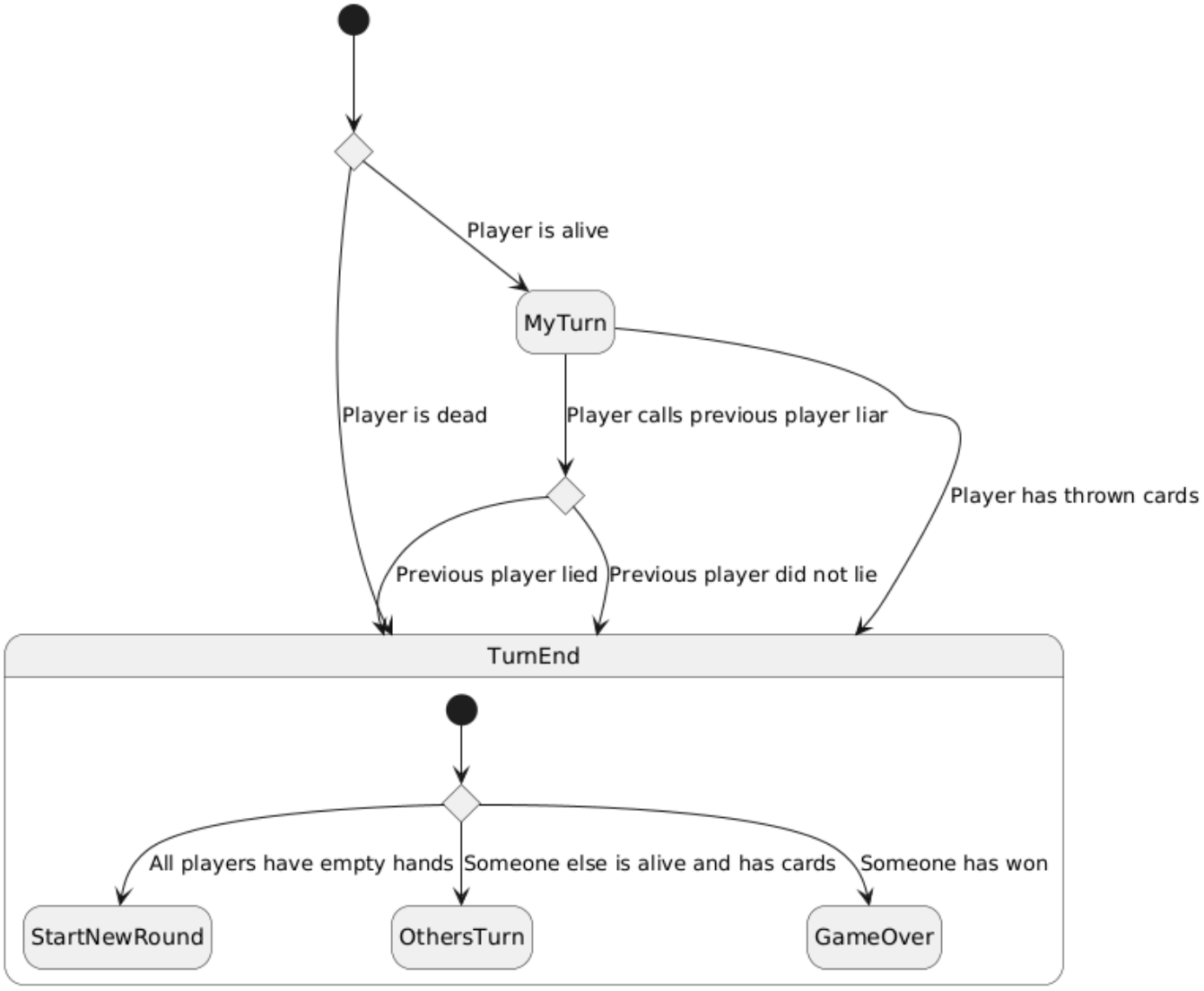
The distributed components always react to messages by updating their local state, optionally notifying other components of their new state.

Specifically:

- The lobby server contains and owns the active lobbies and users, while keeping track of which users are in which lobbies and who owns every lobby. Its state is then replicated on the clients;
- The lobby clients contain a view of the active lobbies, users and, optionally, the current lobby in which they are in. It does *not* own this state, and is always being updated based on server's state. Also, every modification/operation is always first sent to the server for approval before being applied.
- The game clients contain the active game state (which includes the state of all players in a game). It is owned locally by each player, and it is updated based on events received by the game owner and by other clients.



Application State Diagram



State Diagram for a player turn

Data and Consistency Issues

All data in the system is *volatile*, meaning it does not need to be stored on disk or in a database.
For this reason all information is kept in memory by the lobby servers, and it is lost when they are closed.
A representation of this data is sent to the clients, based on their status:

- Clients not in a lobby receive the lobby list, with only a subset of their information available;
- Clients in a lobby receive the full lobby information;
- All clients connected to the server receive the full clients list after every update.

Fault-Tolerance

The system does not implement any fault-tolerance at the application layer, instead it offloads it to the underlying network protocol.
This was chosen since the requirements do not specify any kind of auto-reconnect or retry mechanisms.

A partial fault-tolerance strategy may be to host multiple lobby servers on different machines. This gives the user the choice of connecting to a different server in case the one they wanted is not available.
However, no data is shared between servers, so each server has an independent lobby list.
An improvement would be using a shared, distributed database of lobbies, so that multiple servers can use the same lobby list.

Availability

In case of network partitioning, as specified by the requirements, the system prioritizes consistency over availability.
This means that:

- In case a client goes offline while it is in a lobby, the lobby server removes them form the lobby, and the other lobby users are notified of the disconnection.
- If a non-owner client disconnects during a game session, the owner removes them from the game.
- If the owner client disconnects during a game session, the other clients perform a *host-migration*, where the next valid player becomes the game owner.

In any case, the system does not try to reconnect in case of a disconnection.

Security

Clients are not authenticated by the system, since requirements do not specify it.

Authorization is optional and it is performed by the lobby server when a client tries to join a password protected lobby.
Whenever a user sets a new lobby, it can decide to set a lobby password.
Whenever another user tries to join a protected lobby, it is required to send a password along with the join request.
If the two passwords match, the user is added to the lobby (if it's not full). Otherwise, the server sends an error back to the client.
These passwords are not encrypted and are handled by the systems as simple strings.

Implementation

The project uses **Transmission Control Protocol (TCP)** as its network protocol, which was chosen instead of UDP since it fits best with our requirements.
TCP provides reliable, ordered, and error-checked delivery of a data stream between applications, and since our use cases focus on *consistency* and *reliability* instead of fast delivery, this was the obvious choice.

The application data transmitted over the network is encoded using **Json**.
This was chosen both because of its maturity as a standard, which implies robust support from languages and libraries, and because of its ability to represent complex data without becoming too verbose.
We used the library **jackson** for serialization, because it had better support out of the box for handling inheritance of classes, which is used for the messages' representation.

We have developed our ad-hoc **messaging protocol** in order to exchange data between users. It is based on the `GameMessage` interface, which defines the message *sender* (the peer who sent the message on the network) and the *receipients* (an optional set of peers that will receive the message. If not provided, the message is considered a broadcast).

```
public interface GameMessage {
    PeerId getSender();
    Set<PeerId> getReceipients();
}

public abstract class GameMessageBase implements GameMessage {
    private final PeerId sender;
    private final Set<PeerId> receipients;

    public GameMessageBase(final PeerId sender, final Set<PeerId> receipients)
{
        this.sender = sender;
        this.receipients = receipients;
    }

    public PeerId getSender() {
        return this.sender;
    }

    public Set<PeerId> getReceipients() {
        return this.receipients == null ? null : Set.copyOf(this.receipients);
    }

    @Override
    public int hashCode() {
```

```
        // Omitted for brevity
    }

    @Override
    public boolean equals(Object obj) {
        // Omitted for brevity
    }
}
```

Abstract Class that implements `GameMessage`

The `GameMessageBase` shown here is then extended by multiple different types of `GameMessage`, each one for a specific purpose. These can be divided into these groups:

- *Connection Messages*, used to handle connections between users (used to create the P2P network used during game sessions);
- *Lobby Messages*, created to manage lobbies (creation, join, disconnect, failures, start of game);
- *Game Session Messages*, developed to treat game interactions (player order, new hand for players, new round card, call liar and throw actions);
- *User Management Messages*, just a couple of messages to update user list to each connected user (new user connected, username updated);

Technological details

We used the following technologies and libraries:

- Java 21 as our programming language;
- Swing as GUI framework;
- JUnit 5 as a testing environment;
- Mockito for building mocks when testing;
- Jackson for JSON serialization
- JGoodies Binding as support for model-view bindings;
- Gradle as build system;
- Git as source control;
- GitHub for hosting the repository;
- GitHub issues and pull requests for assigning and tracking tasks;

Validation

Automatic Testing

Components and important aspects of the project were unit-tested by creating simplified environment versions.

Tests are mainly divided into these sections:

- **Game Tests**
  - Tests made to check the correct execution of the main game loop and interaction between game entities and *offline* players;
- **Utilities Tests**
  - Since many utilities were created for the project, a series of tests were made to check if they would correctly work in their required scenarios;
- **Networking Tests**
  - Being a core and complex part of the project, many tests were created to ensure all the networking components of the project were working correctly;
  - This includes testing basic connection between two peers, the exchange of basic peer information and the management of a network of peers, along with many other test cases;
- **Lobby Tests**
  - Tests that ensure the correctness of communications and interactions between clients and server while creating and updating lobbies;
  - Lobby connections are tested by ensuring the server has the correct amount of active players in the same lobby;
  - Lobby disconnection is tested by disconnecting one client and making sure the number of lobby players decreased;

Acceptance test

Manual testing was performed in order to:

- Analyze triggering of event messages due to player actions: since tons of actions are related to player interaction, it was deemed to be more intuitive to perform these tests manually;
- Check proper integration of lobby and game application. While not impossible to be automatically tested, we decided to perform manual testing for this section to see if the whole project would be able to properly create peer networks and functioning game sessions in a reasonable amount of time.

These manual tests where perform both on single and multiple machines, using Windows 10 and 11.
The original game was also tested on Linux.

Release

The solution was divided into three projects:

- A `lib` project, that builds a `.jar` with the core libraries that the apps use;
- A `lobbyServerHost` project, that builds the executable `.jar` for launching a lobby server;
- A `gameApp` project, that builds the executable `.jar` for the user application;

The latter twos executables are the final output artifacts.

Deployment

Project deployment is done following these instructions:

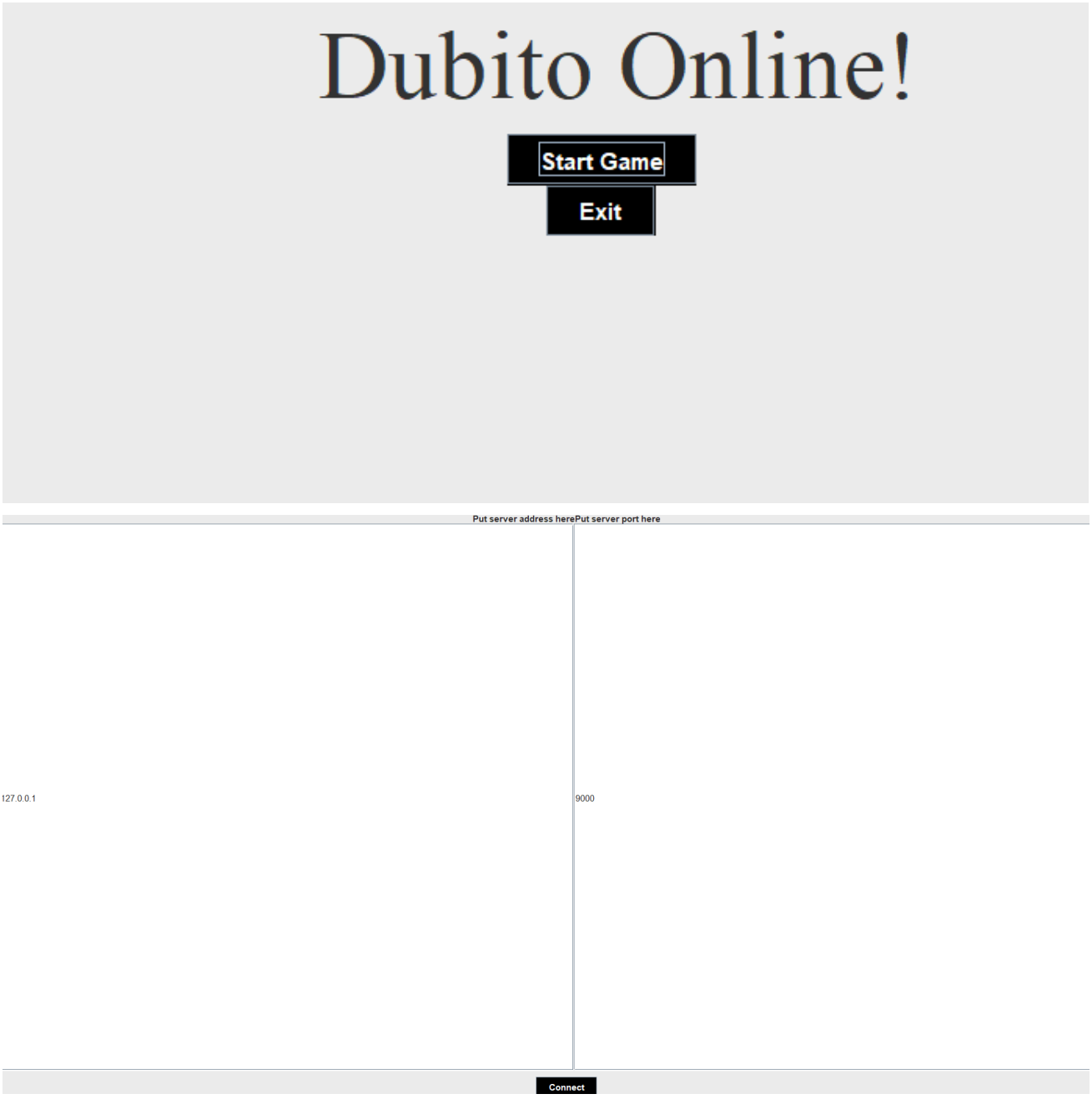
1. Build: Create the `lobbyServerHost.jar` and `gameApp.jar` executables;
2. Start Server: Launch the `lobbyServerHost.jar` executable

The users can now launch the `gameApp.jar` executable and connect to the server.

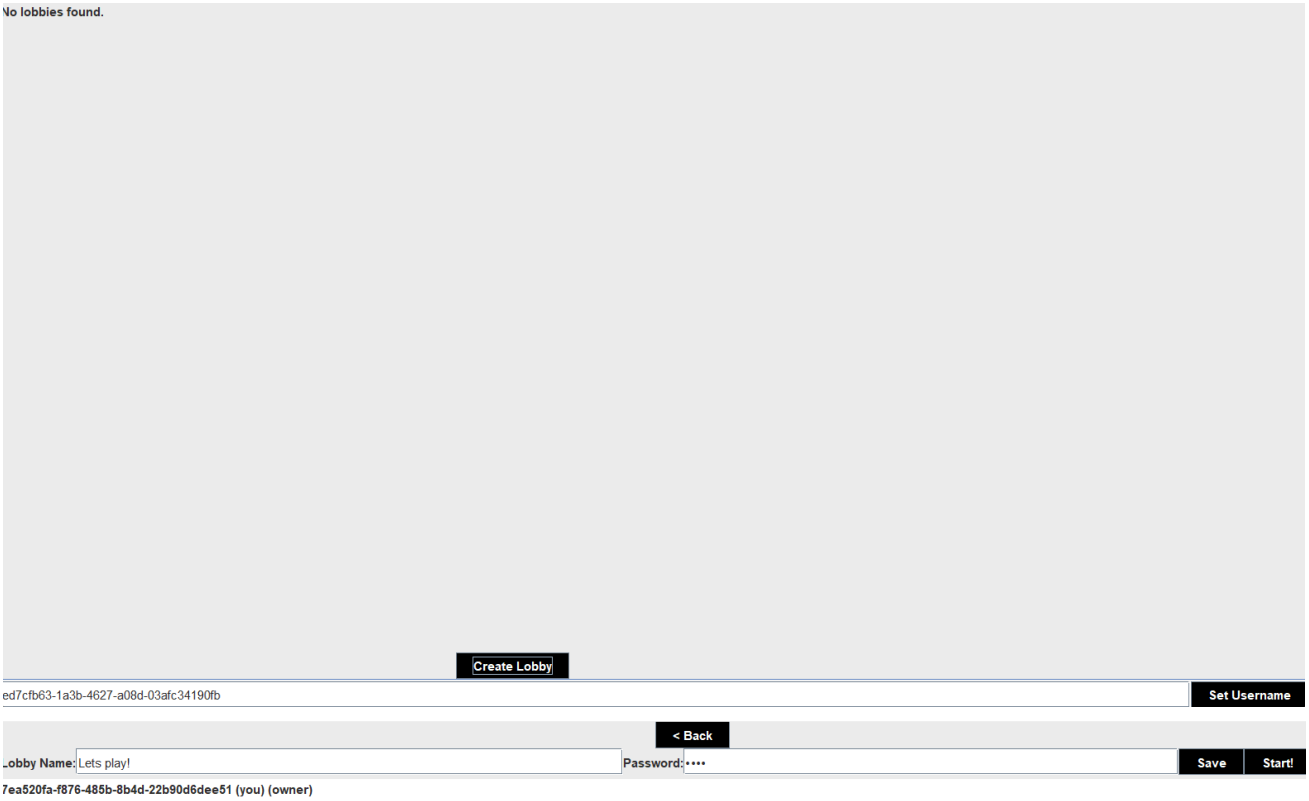
User Guide

Here we have provided all the necessaries steps to play the game:

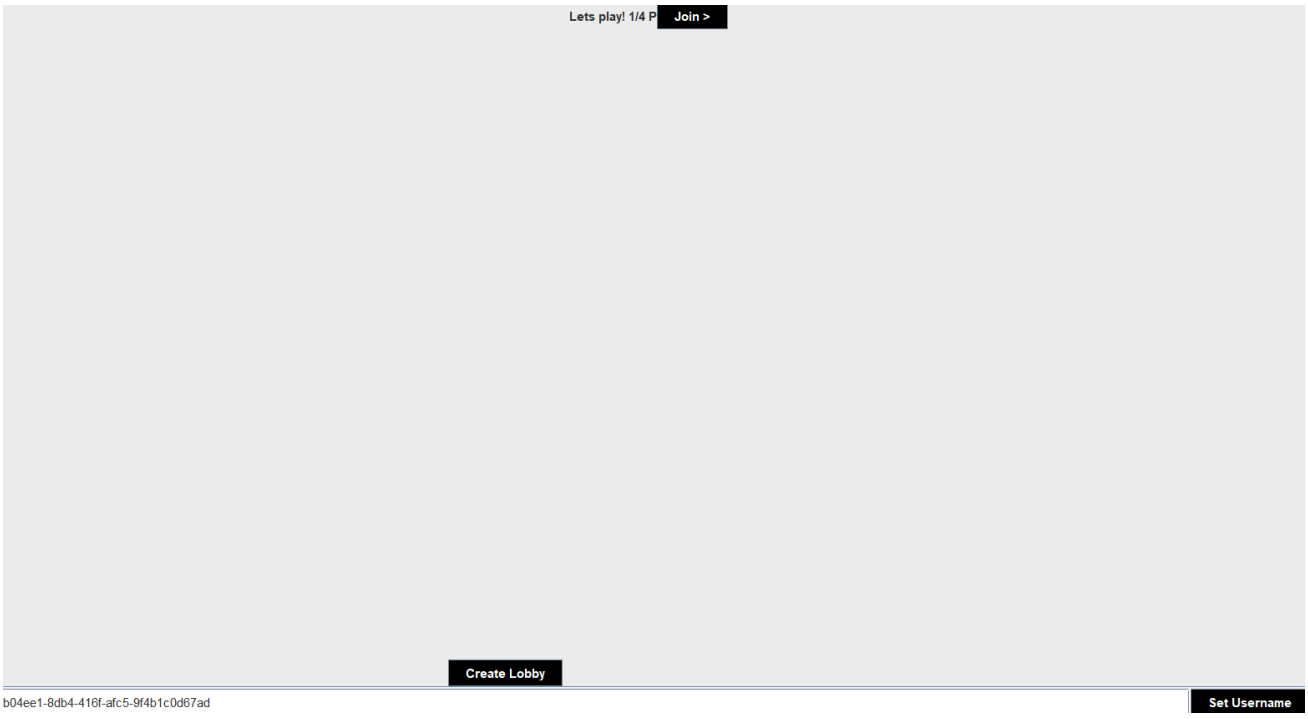
1. Launch at least one `LobbyServer` that users can later join into;
2. Launch the game: after clicking *Start* button, players will input the server IP address and port of the currently active server
   (default one has IP: 127.0.0.1, port: 9000);



3. Once connected to the server, player can either set a new username at the bottom of the view;
4. **Lobby Creation**: user can click the *Create Lobby* button to set up a new lobby. Lobby is created with a new name and a possible password: the lobby will be shown to other players as soon as the owner of said lobby presses the *Save* button;

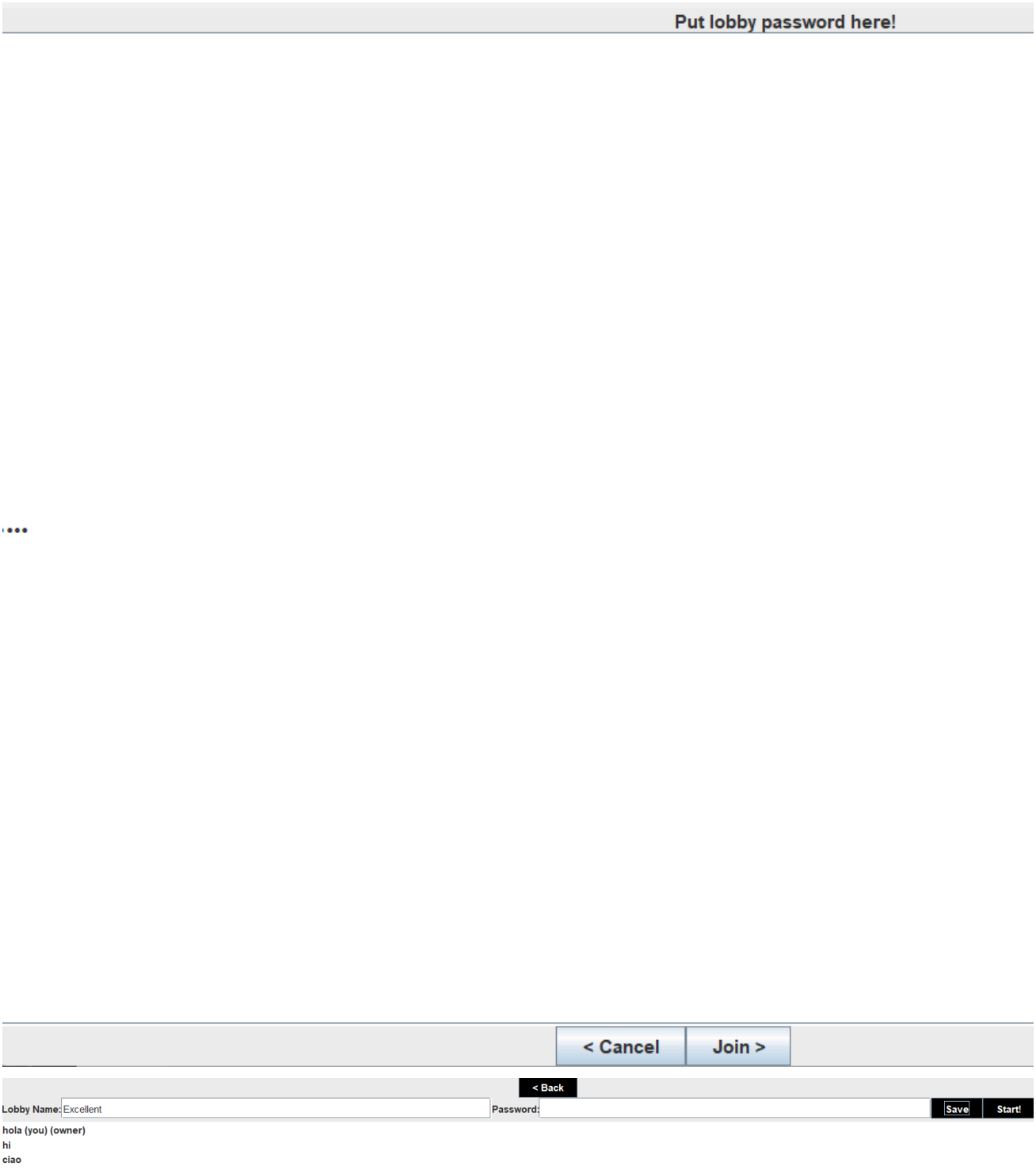


5. **Lobby Join**: users will see at the top of their view all currently available lobbies. Pressing the *Join* button near one of them to enter;
6. (Optional) If the lobby is password protected, the user must first input the correct password in order to enter said lobby;

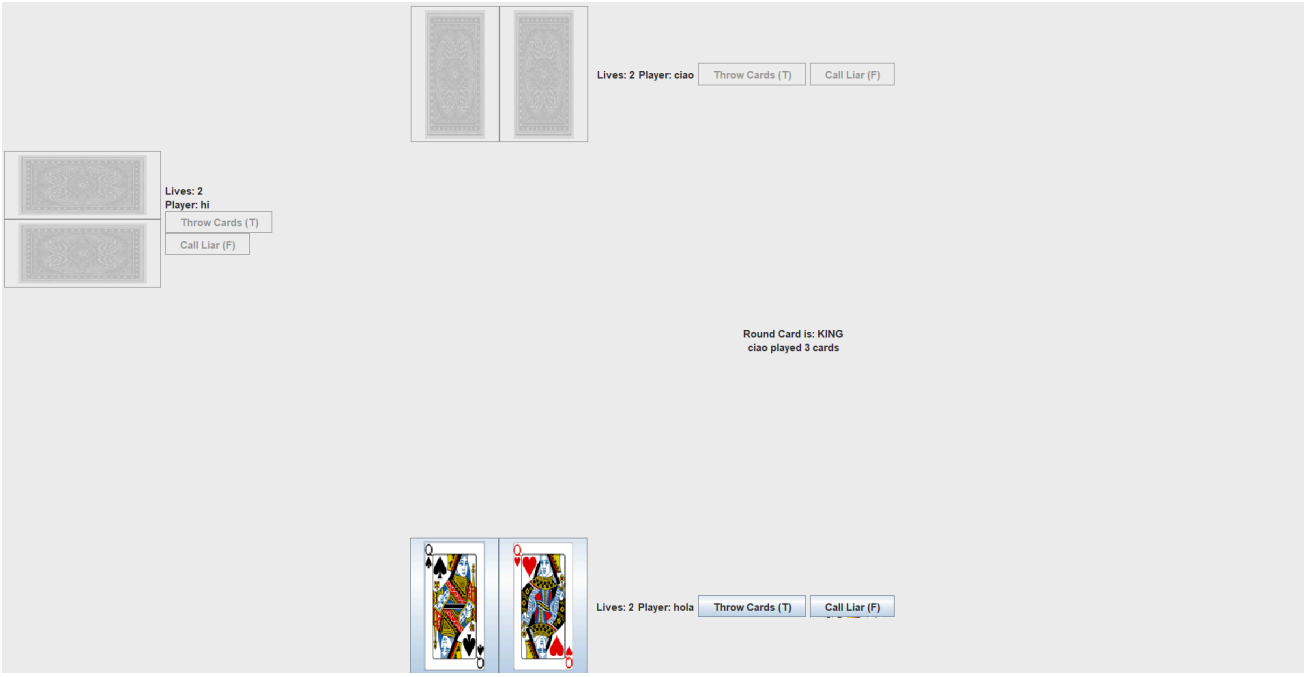


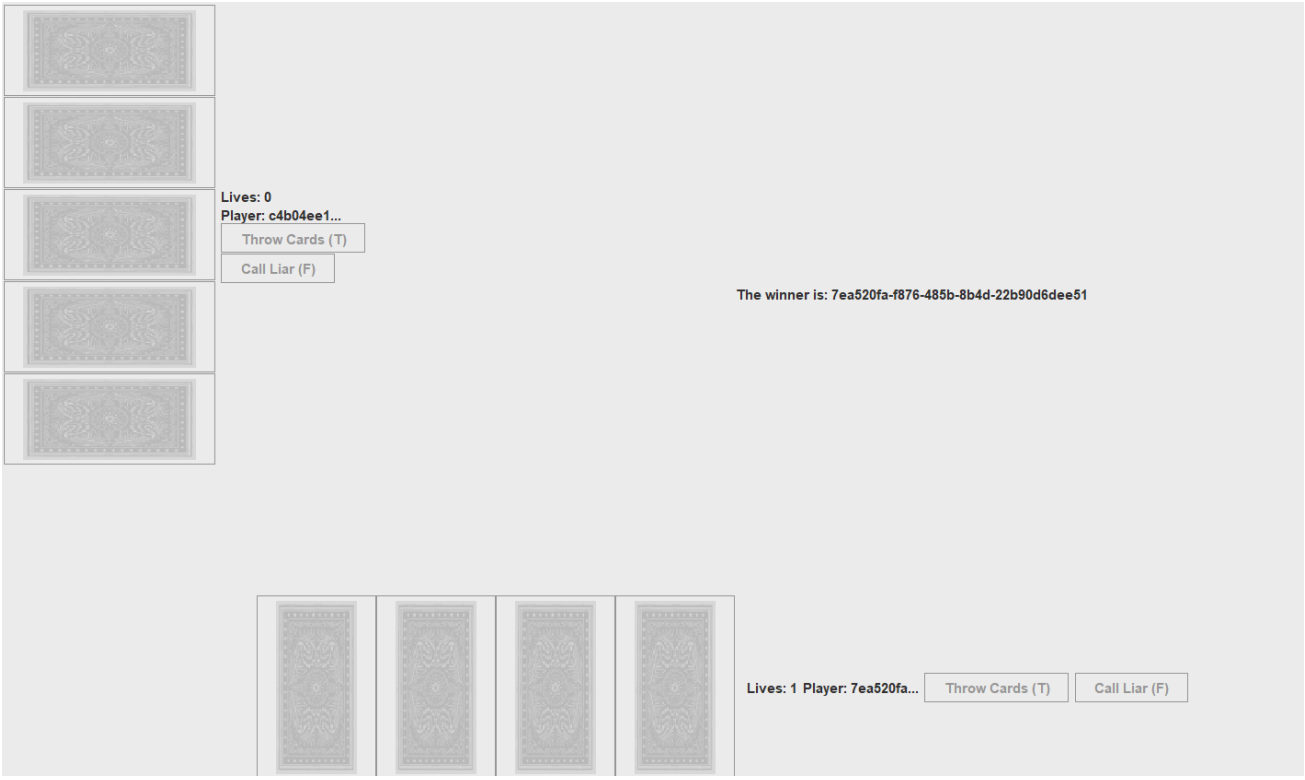
b04ee1-8db4-416f-afc5-9f4b1c0d67ad

Set Username



7. Lobby owner can press *Start* button in their view to start a new game session once there is at least one more user;
8. In game, players (in turn order) can either throw cards or call other players liar;
9. Game continues until only one player remains (in order to leave, they must close their window).





Throw Cards	Call Liar
T	F

Table to showcase game key bindings

Self-evaluation

Andrea Bianchi

I was in charge of developing the main game app for the project. My main focus was developing the offline version (models, views, controllers) and focusing on the messaging system we've created together, generating new types of messages
that would expand the game logic into the online `PeerGraphNetwork` that would be created for the players. We've also helped each other as best as we could
during development, setting many meeting sessions (both online and in-presence) to develop together what we thought was necessary at each step of the way.
While working on this project, I've discovered many interesting and peculiar findings on distributed systems
and how they operate. The work proved to be quite a challenging task, more than what we initially thought.
Both me and my colleague were quite busy with our everyday life, spending what little free time we had to study and continue the project's development.
Despite that, once everything started to "click", we've both felt a great sense of satisfaction and accomplishment.
The main strengths of the project I feel are its quite intricate (yet easy to understand) messaging system, that allowed us to create fluid and consistent
interactions between each player (and between user and server while in lobby).
These strenghts came at quite the cost though: implementing our initial model design into a working system felt at first easy to apply, but we soon found out
many issues and bugs caused by message serialization (especially regarding key bindings) and the initial setup of P2P network for users in a lobby when starting the game.

Overall, I personally feel satisfied and i'm happy to have finally completed this long (but enjoyable) journey.

Alberto Arduini

I mainly focused on developing two systems:

- The lobby and users management system, both on the client and server side;
- The networking system.

For the first point, I feel like the final code is pretty well organized, with good separation and abstraction between application layers.
In particular, the `LobbyClient` and `LobbyServer` services abstractions (and their other counterparts `UserClient` and `UserServer`) over the messaging and networking layer and the data-persistence layer I feel is a good solution to limit dependencies between the application code and the core networking library.

In regards to the networking system, I feel it was a good exercise to understand how a complex network architecture can be built and managed by an application.

I'm especially satisfied with the use of interfaces for enabling the possibility of switching the underlying behavior of the network.
As an example, we could pretty easily change the serialization method from Json to Xml, or we could use UDP or add encryption to the existing TCP connections by implementing a new class or two.
This also extends to the whole network management: we could switch from a client-server architecture to a peer-to-peer one without having to change the application code.

This, however, came at the cost of complexity: correctly implementing the whole system took a lot of time and effort, and ensuring that all edge case were managed was not always easy.
In this regard, unit tests were really helpful to ensure that the code was working correctly, and especially to find any regressions.

I've also helped my colleague as much as he helped me, both for implementing features and for testing the app.

In conclusion, I feel this project was really helpful in understanding how a distributed system works under the hood: manually implementing core features instead of using off-the-shelf solutions gave me a deeper understanding in how these systems work.