Cabo Online

Michele Bachetti
michele.bachetti@studio.unibo.it

Angelo Tinti
angelo.tinti@studio.unibo.it

January 2025

Contents

1 Concept

The goal of this project is to develop a desktop application that allows users to play the online version of *"Cabo"*, a turn-based card game. Details about the game itself and how it works can be found in section 1.2.

The application runs on PC and provides a GUI to allow user interaction in every step of the system, from creating or joining a game, to the single actions performed during a turn.

Games can be private or public: in the first case, other players can join only by using a unique code; in the second case, the game is shared through a lobby server and becomes visible to all players in the system. The server does not work as a dispatcher of messages between players; it is only used to obtain a reference to the host to start the communication.

Therefore, the application is structured as a peer-to-peer (P2P) system, meaning that the players participating in a game are connected directly to each other.

Without a centralized server, it is necessary that the information and data shared between players are always consistent and up to date. In particular, during a match, the game state (which includes the players' hands and the table situation) must always be the same for every player at every turn, to ensure that the actions chosen by a player are coherent and correct with respect to the current situation.

1.1 Roles

Player The player will handle game creation and joining, and will send commands to play the game. The possible actions are shown in fig. 1 and fig. 2.

Lobby server The lobby server is used to share games that are not set as private, so that other players can find them. It handles requests such as publishing a game, getting information about shared games, and removing a previously shared game.

Since this is a P2P system, players do not communicate with each other through the server; instead, it only serves as a means to retrieve their contact information.

1.2 Rules of the game

1.2.1 Cards

Cabo can be played with specific or generic cards. In this project, French cards will be used.

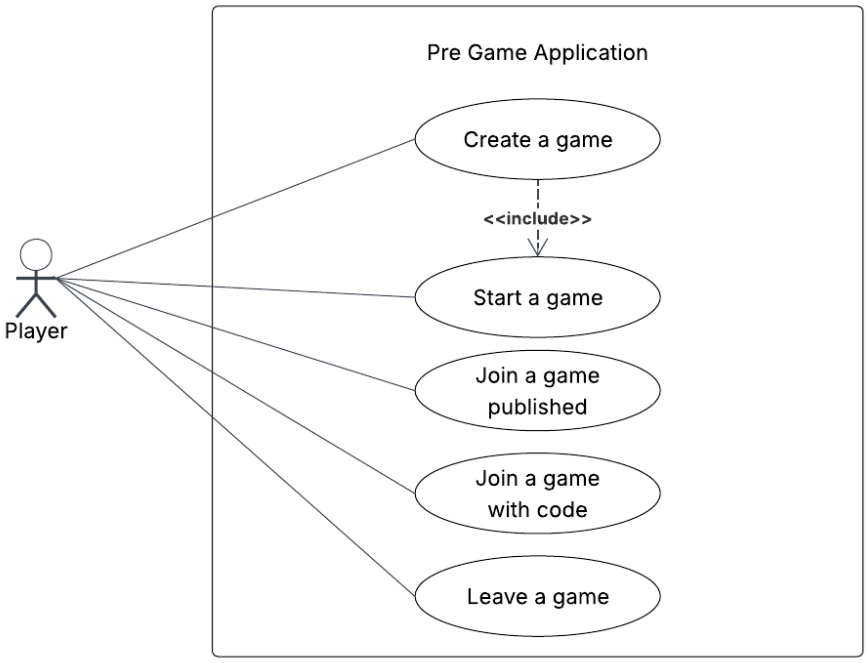


Figure 1: Uses case diagram of player during creation/joining.

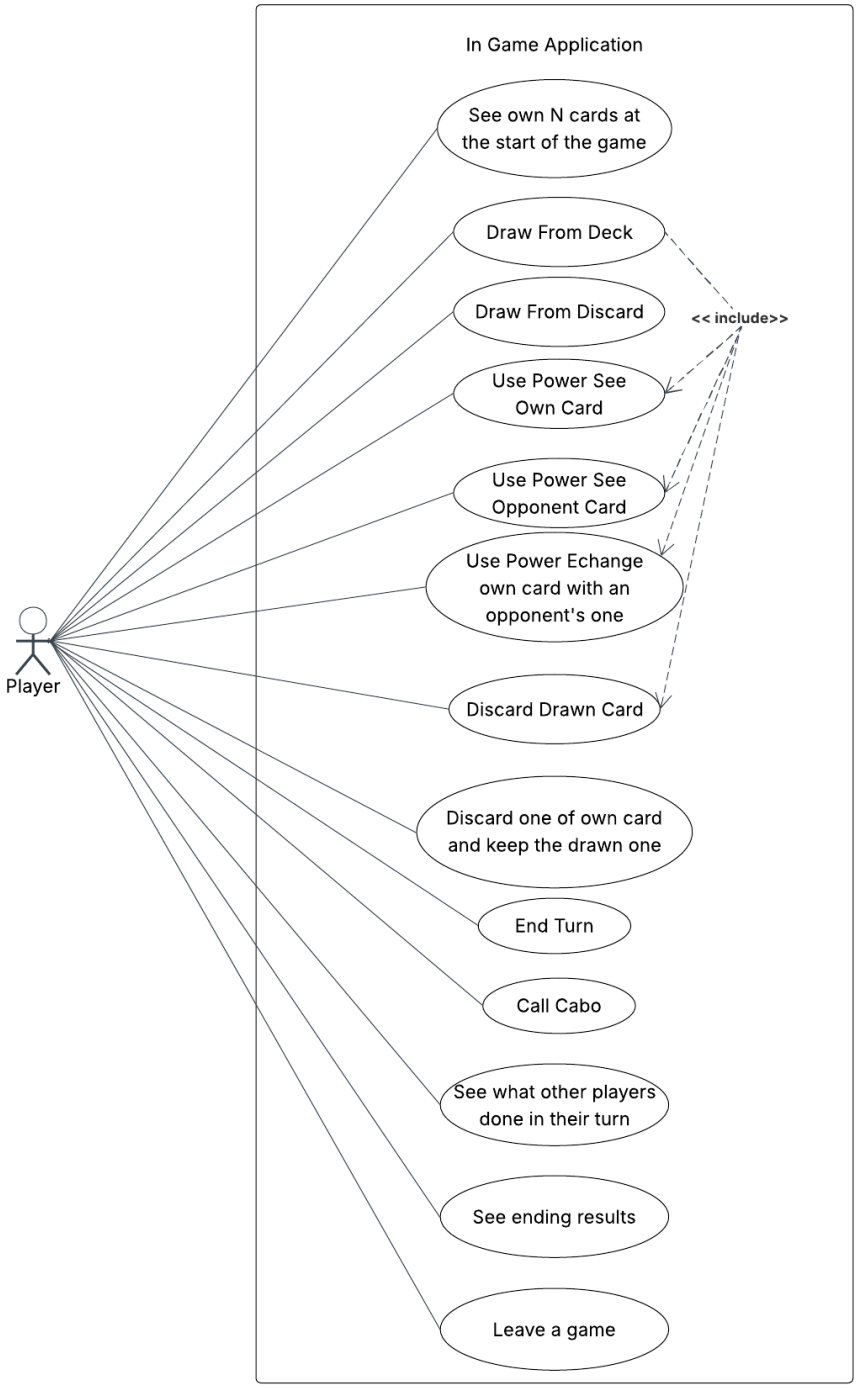


Figure 2: Uses case diagram of player during game phase.

Card Rank	Score	Special Power
Ace	1	No Power
Two	2	No Power
Three	3	No Power
Four	4	No Power
Five	5	No Power
Six	6	No Power
Seven	7	No Power
Eight	8	No Power
Nine	9	No Power
Ten	0	No Power
Jack	10	See one of your cards
Queen	10	See one card of one opponent's cards
King	10	Exchange one of your cards with one of an opponent

Table 1: Cards scores and special powers. The power to exchange cards does not allow to see them.

1.2.2 How it works

Cabo is a turn-based card game for two to five players.

The game begins by dealing four face-down cards to each player. A single card is placed face-up to start the **discard pile**, while the remaining cards form the **deck** (draw pile), placed face-down in the center. Before the first turn begins, each player is allowed to peek at two of their own cards.

The goal of the game is to achieve the lowest total score among the four cards in hand.

Once all players have viewed their two cards, the first player begins their turn. Each turn consists of the following steps:

1. **Draw a card:** Draw the top card from either the deck or the discard pile. If drawn from the deck, the card is revealed to all players.
2. **Use card powers:** If a card with a special power is drawn from the deck, the player may use its ability:
   - **See one of your cards:** reveal one of your cards only to you.
   - **See one card of an opponent:** choose an opponent and one of their cards to reveal only to you.
   - **Exchange one of your cards with one of an opponent:** choose one of your cards and one card of an opponent; the selected cards are exchanged, but nobody will know their value.
3. **Discard or Swap:** If the card was drawn from the deck, the player decides whether to keep it (replacing one of their current cards) or discard it. If the

card was drawn from the discard pile, it must be kept and cannot be discarded immediately. The discarded card is placed face-up on the discard pile.

4. **End turn or Call Cabo:** The player decides to end their turn or "Call Cabo."

Once *"Cabo"* is called, a final round begins. The game ends when the turn reaches the player who called Cabo again. Alternatively, the game ends immediately if the deck is exhausted.

2 Requirements

2.1 Functional

- Create a game and share it using the lobby or a personal game code.
- Join a game using the unique game code.
- Join a game from the lobby list.
- Fault tolerance: if a player is disconnected, their turn is skipped while the game continues.
- Play an entire game to its end.

2.2 Non-functional

- Low latency during each game.
- A nice graphical interface.

2.3 Implementation

- The system will be implemented using the Akka toolkit. We believe it is a very useful framework for distributed contexts, and its adoption allowed us to advance and deepen our technical knowledge in the fields of concurrent and distributed computing.

3 Design

3.1 Architecture

The project follows the MVC pattern and the actor model.

Moreover, the game system is based on a peer-to-peer (P2P) architecture, meaning that each active entity communicates directly with the others without the need for an intermediary.

However, the lobby component of the system follows a client-server architecture, meaning that the lobby is a "single" entity contacted by the player entities to receive information about the games currently active.

This solution was chosen to provide a simple way to connect players to each other and create an indefinite number of games without worrying about maintaining a central server.

3.2 Infrastructure

The two main components of the system are the *Client* and the *Lobby Server*. In particular, each client represents a human player, while the server is an entity that is used by clients to find each other.

Within the system, it is possible to have multiple instances of the server entity to increase fault tolerance and avoid situations where a player cannot share their game. Moreover, multiple instances can help with load-balancing issues, in case a large number of players are active at the same time and a single server cannot manage all their requests.

When a player creates a new game, their client becomes the host for that game. At this point, if the game is set to be public, the host contacts the lobby server to share its information and allow other players to join. In any case, any player can access this match using a unique code that the host receives when creating the game and that they can later share through other channels, such as instant messaging.

3.3 Modelling

3.3.1 Game Phases

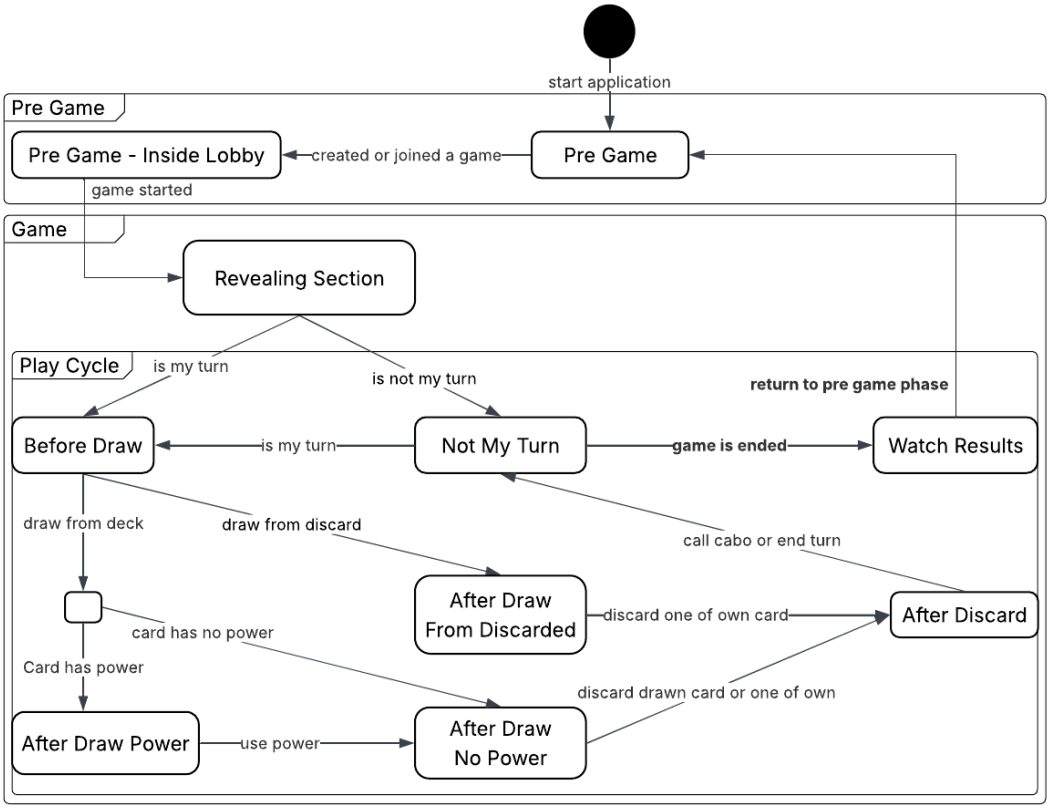


Figure 3: Different phases/states of the game and how they change.

The application can be split into two main macro states:

- Pre Game: the phase in which it is possible to create or join a game;
- Game: the actual game, divided into two phases according to the game logic:
  - Revealing Section: the phase in which players can check two of their cards at the start of the game.
  - Play Cycle: the loop in which turns are played. It ends when the player checks the game's results.

The fig. 3 shows a canonical playthrough from the start to the end, describing the application-game logic while abstracting away the connection/distributed level.

3.3.2 Domain entities

The main domain entities are:

- Game
- Game Parameters
- Player (Opponents)
- Cards
- Hands
- Deck
- Discard pile

This entities are represented using the classes shown in fig. 4

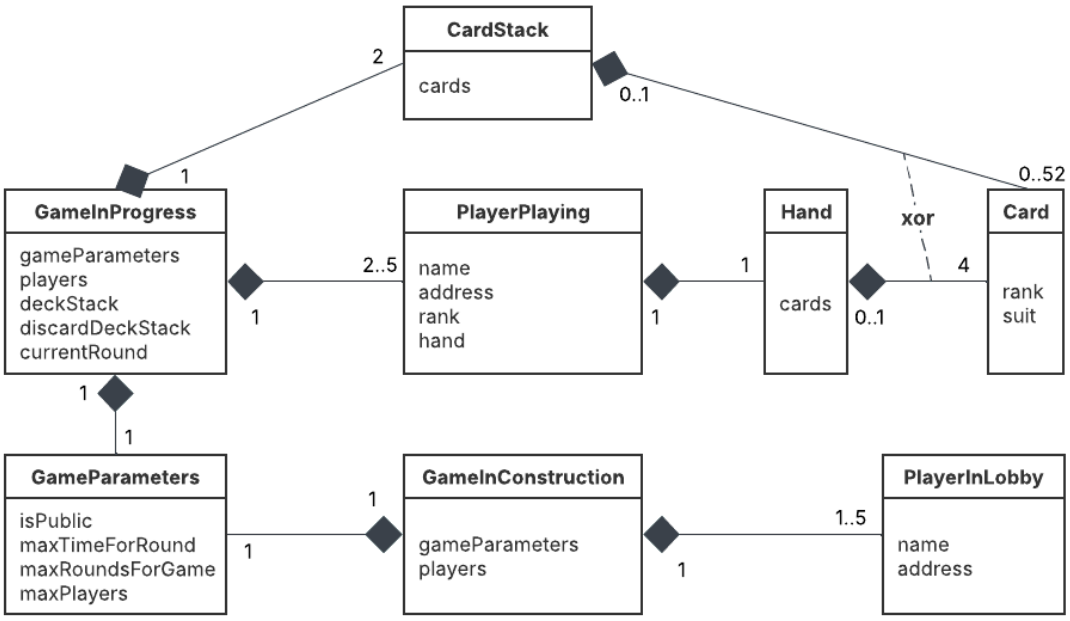


Figure 4: Classes relation of game's entities.

Game The game goes through two phases, as explained before: pre-game and game. The first one has less data than the second one, so two different entities are used:

- **GameInConstruction**: it contains the game parameters and the list of players who are joining.

- **GameInProgress**: once the game begins, it also needs to track the main deck and discard pile, players' hands, and turn information.

These data structures are passed between players during the play cycle to keep their game state updated.

Game Parameters A data structure that represents the game parameters:

- public/private game
- max number of players
- max number of rounds
- time limit for each turn

Player The player entity contains all the information about a player and, similarly to the game, it has two versions depending on the phase of the match: `PlayerInLobby` and `PlayerPlaying`.

The first one represents a player waiting for the game to start, while the second one is used when the match actually begins. In the first case, the player needs a name and a reference to be contacted, while in the latter they also have their hand of cards and a rank for the playing order.

Turn Log The game is turn-based, and a player sees what another one has done during their turn at the end of it. Therefore, a turn log has been designed to represent this sequence of information. Every turn generates a turn log, which is then shared with all players.

This data structure needs the following information:

- which player played the turn;
- the round number;
- which card was drawn and from where (deck or discard pile);
- which special power was used (if any) and its parameters;
- which card was discarded;
- whether the player called Cabo;
- whether the player skipped the round due to time exceeding or connection problems.

3.3.3 Events

Player Only the creator of the game can start it; if the creator becomes unreachable, the game is cancelled and the other players are brought back to the initial state. A game can be public or private, so the lobby server may be involved when someone intends to join, by sharing the list of available games.

Player's action/event:

- Create a game
- Request games
- Join a game
- Leave a game
- Draw a card
  - from the deck
  - from the discard pile
- Use the power of a special card
  - view one of your own cards
  - view a card of another player
  - exchange a card with another player
- Discard a card
- End Turn
- Call Cabo

3.3.4 Messages

Lobby Server Messages

- RegisterGame: send game information to be registered in the lobby server;
- GameRegistered: positive response to the previous message;
- FailedToRegisterGame: negative response to the previous message;
- StartGame: notify the start of a previously registered game, so it can be deleted from the server;
- AbortGame: request the deletion of a previously registered game;
- GetGames: request the list of joinable games;
- GamesList: response to the previous message;

- UpdateGame: send an already-registered game with updated information (e.g., some player joined or left);
- FailedToUpdate: negative response to the previous message.

Client Messages - Messages sent to the client by the other entities to control the flow of the application

- ChangePlayerName
- CreateNewGame: message sent by the view when the player decides to create a game, passing its parameters
- JoinAGame: message sent by the view when the player decides to join a game
- JoinGame: message sent by the view to ask the client to join the game chosen among those received from the server
- JoinWithGameCode: message sent by the view to ask the client to join the game identified by the code
- StartTheGame: start the created game; accessible only by the host
- LeaveTheGame
- GetPlayerInfo & PlayerInfo: request and response to share player information (i.e., name and userId)
- GameEnded: message sent to indicate that the match is over

Game Command Messages - Messages sent to the game logic to notify user actions:

- DrawCardFromDeck;
- DrawCardFromDiscardStack;
- DiscardCardDrawn;
- DiscardOwnNthCard, passing the index of the card to discard;
- ShowOwnNthCard;
- ShowAdversaryNthCard, requiring both the opponent and the card index;
- ReplaceOwnNthCardWithAdversaryNthOne, requiring the opponent, the index of their card, and the index of mine;
- EndTurn;
- CallCabo.

Game Synchronization Messages - Messages sent to the game logic to notify/request a new game-flow situation:

- StartRevealingSection;
- StartPlayCycle;
- LastTurnPlayed, containing the current game state and the last turn played;
- GetEmptyTurn, sent to the game logic from the connection layer when a player is unreachable;
- TurnTimeEnded;
- WhoIsPlaying, a request to know which player is currently playing.

Pre Game View Messages - Messages sent to the view during the pre-game phase from the Client (ViewsProxy):

- WhoToSendResponse;
- FailedToPublishToServer: when the game is not published to the server;
- GameCreated: send the newly created game-in-construction;
- GameList: a list of games in construction;
- GameJoined: send a game-in-construction joined by the player;
- GameJoinedFailed;
- GameInfoUpdate: send an updated game state;
- GameStarted;
- GameAborted.

Game View Messages - Messages sent to the view during the game phase (user input is treated as a message):

- StartGame: notify the view that the game is starting, and communicate the game data and the coordinator reference for future communication;
- RevealingCardsPhaseAdversaryLog: notify the revealing-section log of another player;
- WaitAfterRevealingSection: notify the actor that the revealing section has ended and it is time to wait for others;
- LastTurnPlayed: notify the new game state and the log of the last turn played;

- StartTurnPlayer: notify who will play the next turn;
- CardDrawn: notify the drawn card;
- NewTopCardDiscardStack;
- EmptyDiscardStack: notify that the discard stack is empty (e.g., if it is drawn from on the first turn);
- CardSeen: notify the seen card (due to a power or other logic);
- ChangeCardWithAdversaryAck: notify the view that the card exchange performed with a power is done;
- GameEndedBy(Cabo/TurnsLimit/EmptyDeck): notify that the game ended and why;
- EndTurnByTimeEnded: notify the view that its turn ended because there is no more time;
- OpponentImpossibleToReach;
- OpponentDisconnected;
- AllOpponentsDisconnected;
- GameDeleted.

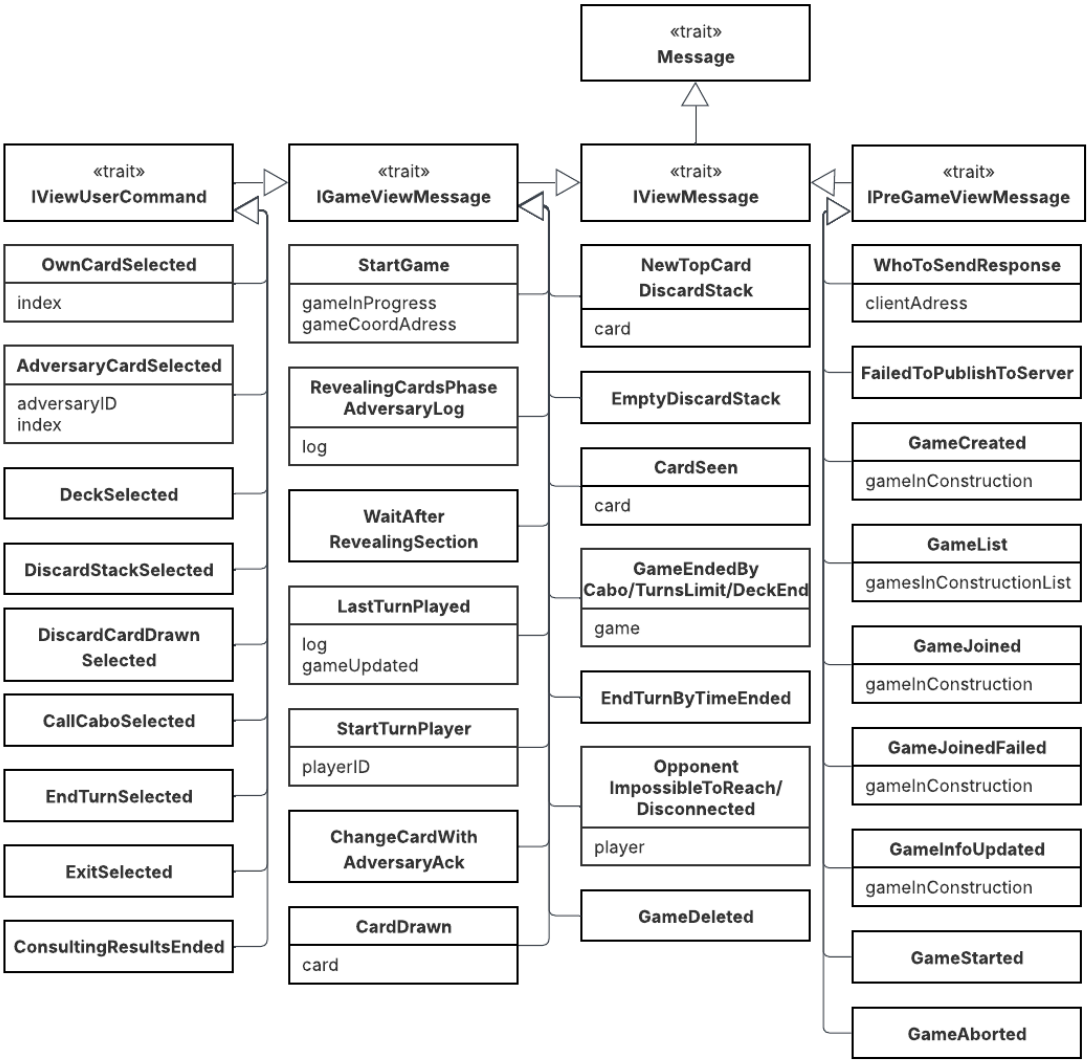


Figure 5: Pre Game and Game view messages structure.

3.4 Behaviour

In following schemas, along transitions will be used the notation "R: `[message1]` / S: `[message2]` to `[actor]` " to indicate: reception(R) of message1 triggers the transition so message2 is sent to actor.

3.4.1 Lobby Server

The lobby entity behaviour is linear and simple: all it has to do is register the games created by players and send them to others on request. Being modelled according to the

actor principle, it reacts to messages sent by clients by adding and/or removing games that it can share.

In the system, it is possible to have multiple instances of the lobby server, so every change requested to one of them must be propagated to all the others in a reasonable amount of time.

3.4.2 Client

The client is one of the entities with the most complex behaviour, since it is responsible for the distributed aspects of the system while also managing some match actions and events.

This entity has been designed with the idea of making it as general-purpose as possible, and not directly related to any specific game. In other words, this client should be usable to manage the connection between players in any turn-based card game, with only one active person per turn.

To do so, this entity does not contain any rules specific to the game developed for this project (CABO), but only general mechanisms and actions that are necessary for the correct functioning of the distributed system.

As for the rest of the project, this entity is designed following the actor principle, which means that it is passive by itself and all its actions are decided and triggered by exchanging messages with the other components.

Every player in the system will have an instance of this entity that will be used to manage the connections with the others, allowing them to communicate and exchange information about themselves and the match they are playing.

In general terms, this entity can follow two slightly different behaviours: that of the host, or that of the joinee. The first one is adopted when the player associated with the entity is the one who creates the match, while the other is followed when the player joins an already existing match. Furthermore, these two behaviours can be divided into two temporal phases: before and during the match, as shown in fig. 6.

Before the match begins The differences beetwen host and joinee are mostly found here, when the host has some more responsibilities involving the synchronization between the players to ensure that all share the same view about the state of the game, which include the informations about the players and the parameters of the game. It's important to notice that all the information collected in this state are not specific to the game that will be played, in order to keep this entity general purpose.

When creating a match, the player has to decide the values of some parameters, such as the maximum number of players, the maximum number of rounds, and whether the game has to be shared with the lobby server to make it public. All this information will be sent to the client through a message sent by the view components.

Once all the operations are completed, the host enters a state very similar to a lobby, where it waits for join requests from other clients (i.e., other players).

Figure 6

Any other joinee can decide to enter the match by sending a join-request message to the host address. To find it, any client can interact with the lobby server component of the system, or use the unique code generated during match creation, which needs to be shared outside the system.

If the joining request succeeds (i.e., the two clients discover each other's addresses and the game is not already full), the host will send a confirmation message to the joinee and register their information (name, personal ID, and address), updating the game state so it will be able to contact them in the future.

From the moment another player joins in, the host will periodically check if they are still online using a heartbeat mechanism. If, for any reason, a player is no longer reachable by the host, they will be removed from the game, and every other player will be updated about the change.

Until the start of the game, only the host can modify the state shared between the clients, since the only events that can happen in this phase are players joining or leaving the game, and they are all managed by the host. Every time the state is updated, all the other clients are informed.

After at least one other player has successfully joined the game, the host player can decide to start the game. First, it sends a message to all the other players to communicate this decision and let them start the operations required to actually begin.

Upon receipt, all the clients will start the actor components responsible for the game logic and ask for a view component update accordingly.

During the match In some games, before the first turn can be actually played, there are some actions that the players need to perform. In CABO, for example, all the players must verify the value of two of their cards. However, not in every card game these actions are required, so, in order to keep this entity general-purpose, a phase prior to the main turns loop has been designed.

Here, all the clients will wait for a message from the game-logic component containing the results of these operations, in the form of a log. Then, they will transmit this message to the host, which will collect all this information. Once the host has received the messages from everyone, it will send them all back, expecting an *ack*. This way, it is ensured that everyone receives the actions of the other players, maintaining a shared view of the situation.

This mechanism is required because all the clients operate at the same time; therefore, to ensure that everyone receives the logs from everyone else, the host is used as a means of synchronization between the clients.

If the game logic does not require such actions before the actual begin of the match, the logs shared in this phase will be empty.

Once this pre-phase is terminated, the client actually enters the "during the match" phase.

Here, the client assumes a more passive role, since the players have already found each other and all that remains is to ensure that all messages are correctly delivered.

In particular, when a player finishes their turn, it is the client duty to share the new state of the game with the other players and ensure that each of them eventually receives it. This is important to maintain consistency during the match.

Meanwhile, it keeps checking (through the heartbeat mechanism) that every player is online and reachable, as in the previous phase.

This time, however, if someone does not respond to the messages, it is no longer removed from the match, but just registered as unreachable. This way its information are not removed from the game state, like the card in the player's hand, since they are necessary for the other players to keep playing. Moreover, to avoid blocking the match, whenever an unreachable player should play their turn, the client of the last player who played a turn will issue a skip command and the next player in list will be able to continue.

If a player disconnects while playing their turn, it will be the host to send the skip command so that the match can continue.

These actions can be applied to every player that becomes unreachable until only one remains. In this case, the match no longer has meaning and it is cancelled.

A special case has to be specified in the management of a disconnection: the host

player. Since the host needs to perform some special actions to ensure the correct flow of the match, if it disconnects from the other players, those who remain have to elect a new one. To do so, the bully algorithm[1] is used as a leader-election mechanism. The ordering criterion chosen for the algorithm is the order followed by the players to play their turns.

This method was chosen because it is simple to implement and is very suitable when working with a small number of nodes.

3.4.3 Game Coordinator

The Game Coordinator actor (GC-actor) manages the game logic, abstracting away the connection layer and the view. It is the entity that creates and handles the evolution of the game data[1]. It is divided into different states, each enabling specific behaviours based on the current phase of the game, as shown in fig. 7. It can be created using general data about the players in the game, or with an already-created game. After that, it waits for a message to switch to the revealing section.

Revealing Section In this section, the GC-actor accepts requests to see the player's own cards; the number of cards is defined in the Game Parameters. After a player has viewed their cards, it sends to the client a log containing the indices of the cards seen and waits for the other players to do the same. Start play cycle is then sent to the GC to indicate that it can enter its turn phase or wait for its turn.

Play Cycle During its own turn, the GC-actor goes through different steps. It starts by waiting for a message indicating from which card stack to draw (deck or discard pile). The drawn card is sent to the view; if the card is drawn from the discard pile, the view is also notified about the new top card of the discard stack. This choice creates two possible paths, because drawing from the discard pile does not allow using a card power and forces the player to replace one of their own cards.

If a "see your own card" or "see an opponent's card" power is drawn, the actor uses a shared state to inform the view about the seen card; if an "exchange cards" power is drawn instead, a message confirming the exchange is sent.

Then, the "After Draw No Power" state is reached and it is possible to handle messages to discard the drawn card or one of the player's own cards; this triggers a message to the view to inform it about the new top card of the discard stack.

Now the GC-actor accepts End Turn and Call Cabo messages to end the round and notify the view and the client about it. The client's message encapsulates the current game data and a turn log containing the choices made during the turn. During the round-ending procedure, the current turn is updated to compute whether the game has ended and who the next player to play is. Then, the actor changes state to "Not My Turn" if the game has not ended.

[1]Only the host creates the game data from scratch; the other players receive it

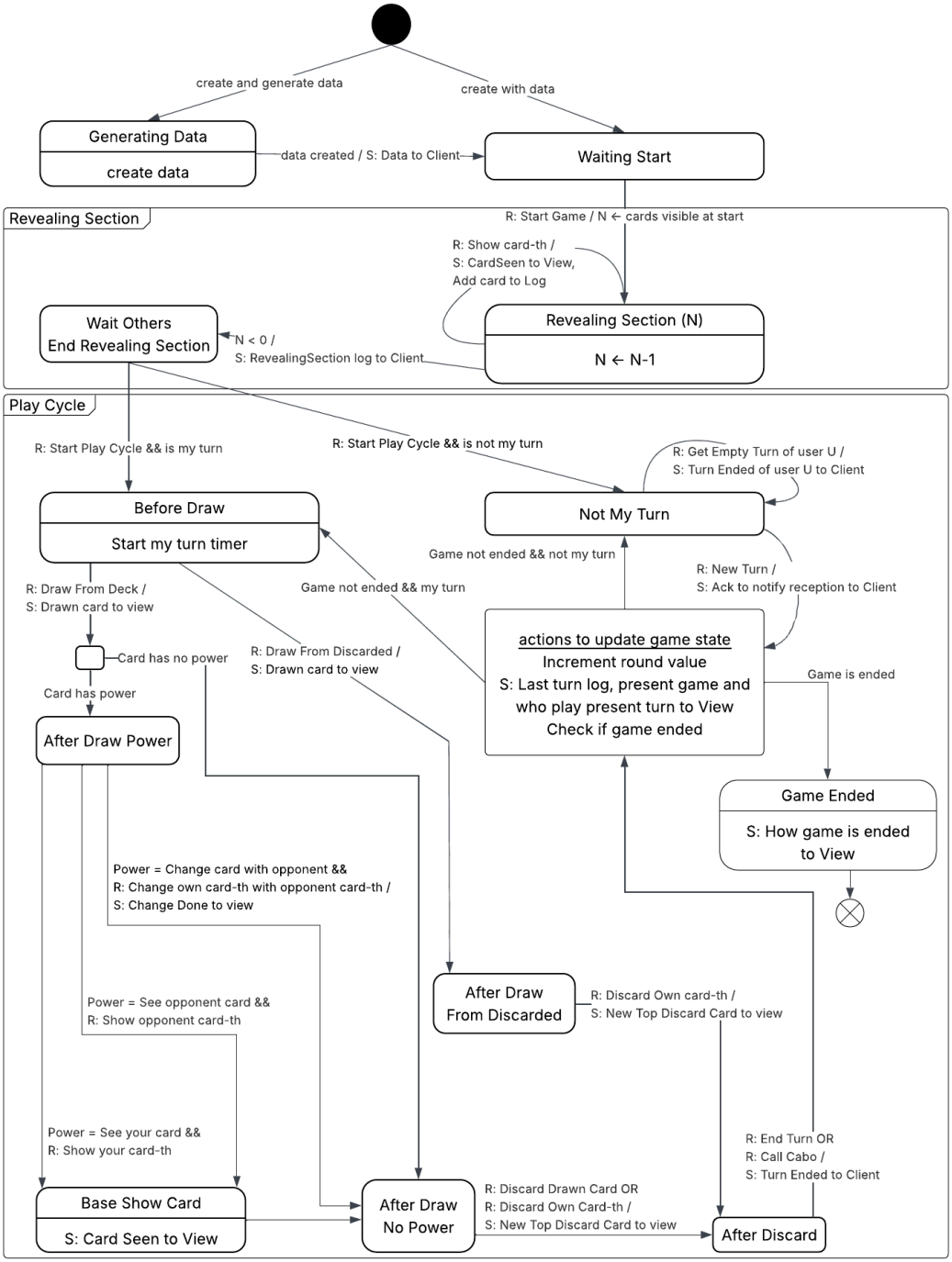


Figure 7: State diagram of game coordinator.

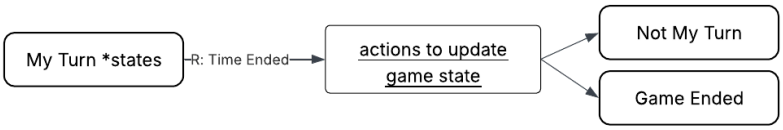


Figure 8: During own turn a "time ended" message is sent to notify the ending of turn, that triggers ending turn protocol.

When in "Not My Turn", the actor accepts messages about the new turn played, explaining what happened in the last turn using the Turn Log. This information is propagated to the view and an acknowledge message is sent to the Client.

The host's GC-actor can receive a message to produce an empty turn for a specified user (Get Empty Turn). This allows handling problems for unreachable players. Therefore, the actor does not consider connection problems directly, but it knows that a player cannot play at the moment.

During the turn update (after receiving a new turn or at the end of its own turn), the actor checks whether the game has ended, considering three possibilities: the next player to act is the one who previously called Cabo, the deck is empty, or the round limit has been reached. A message is sent to the view to indicate which condition occurred. Then, the actor transitions to the final state and stops receiving game messages.

As shown in the diagram, a timer is started at the beginning of each turn. If it expires during the actor's own turn, it blocks turn requests and starts the end-turn protocol, notifying the Client and the View (fig. 8). The end-turn-by-time policy restores the game conditions to those at the start of the turn, incrementing only the number of turns played.

3.4.4 Pre Game View

The actor handles the pre-game view during the pre-game phase: it is the first view shown to the player when starting the application. During initialization, it waits for the reference of the actor it will communicate with; once received, it creates and shows the pre-game view.

The initial state is idle from the moment the graphical view exists.

There are three ways to enter a game: creating a new game, joining an existing game provided by the central server, or joining a game using its code (without the server). After creating or joining a game, it creates the lobby UI and changes to the corresponding state. During this phase, the actor can receive different messages that trigger different behaviours:

- game update: it updates the lobby view;
- game publishing error: it shows an error message and the user can decide whether to keep the game (still joinable via game code) or close it;
- game aborted: it shows an error message and, when the user decides to close it, the actor restarts the view;

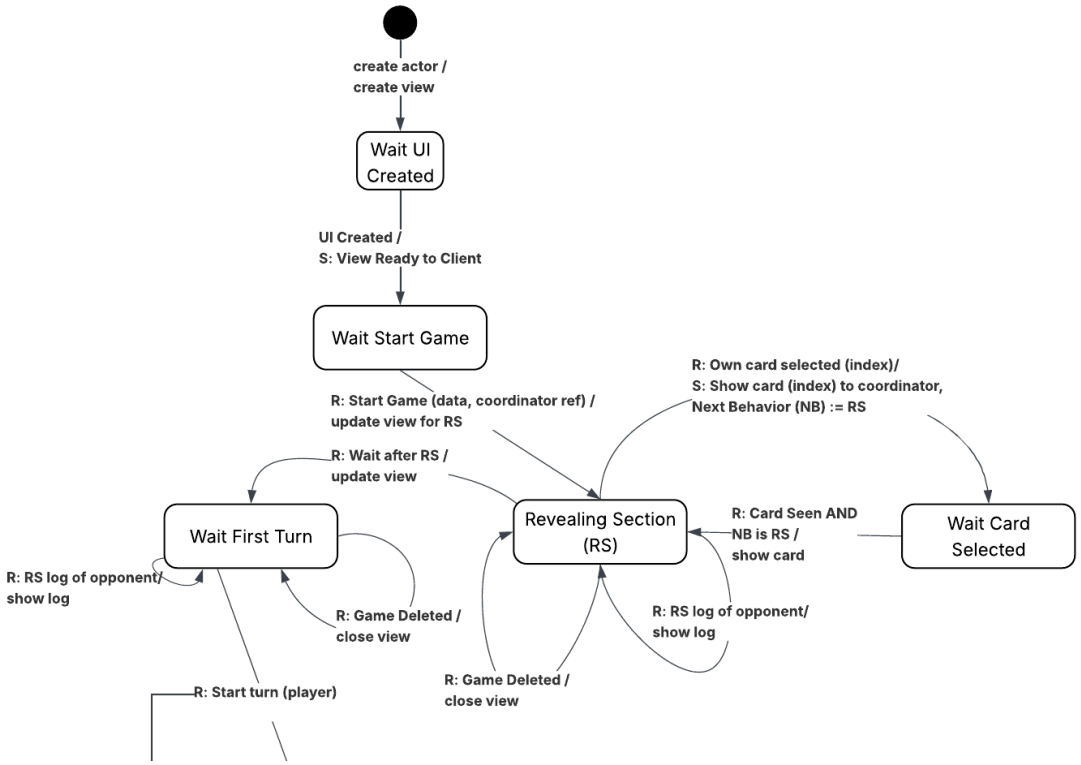


Figure 9: First phase of game view actor: creation and revealing section.

- game start: it closes the pre-game view.

3.4.5 Game View

The actor controls the view during the game: transition actions (update, show, etc.) indicate changes in the rendered view.

After creation, it notifies the client that the UI is ready and waits for the start message, which includes the coordinator reference. The states follow a structure similar to the coordinator: a revealing section and a play cycle (divided into the steps of the actor's own turn). Messages with "selected" in their name are sent by the view through an internal listener. This approach was chosen so the actor can update the view and prepare its internal state before replying to the coordinator. Some transitions are not shown in the diagram, including:

- exit selected: sent to the actor when the user clicks exit to leave the game;
- opponent disconnected / opponent impossible to reach: these update the view with connection-status information;

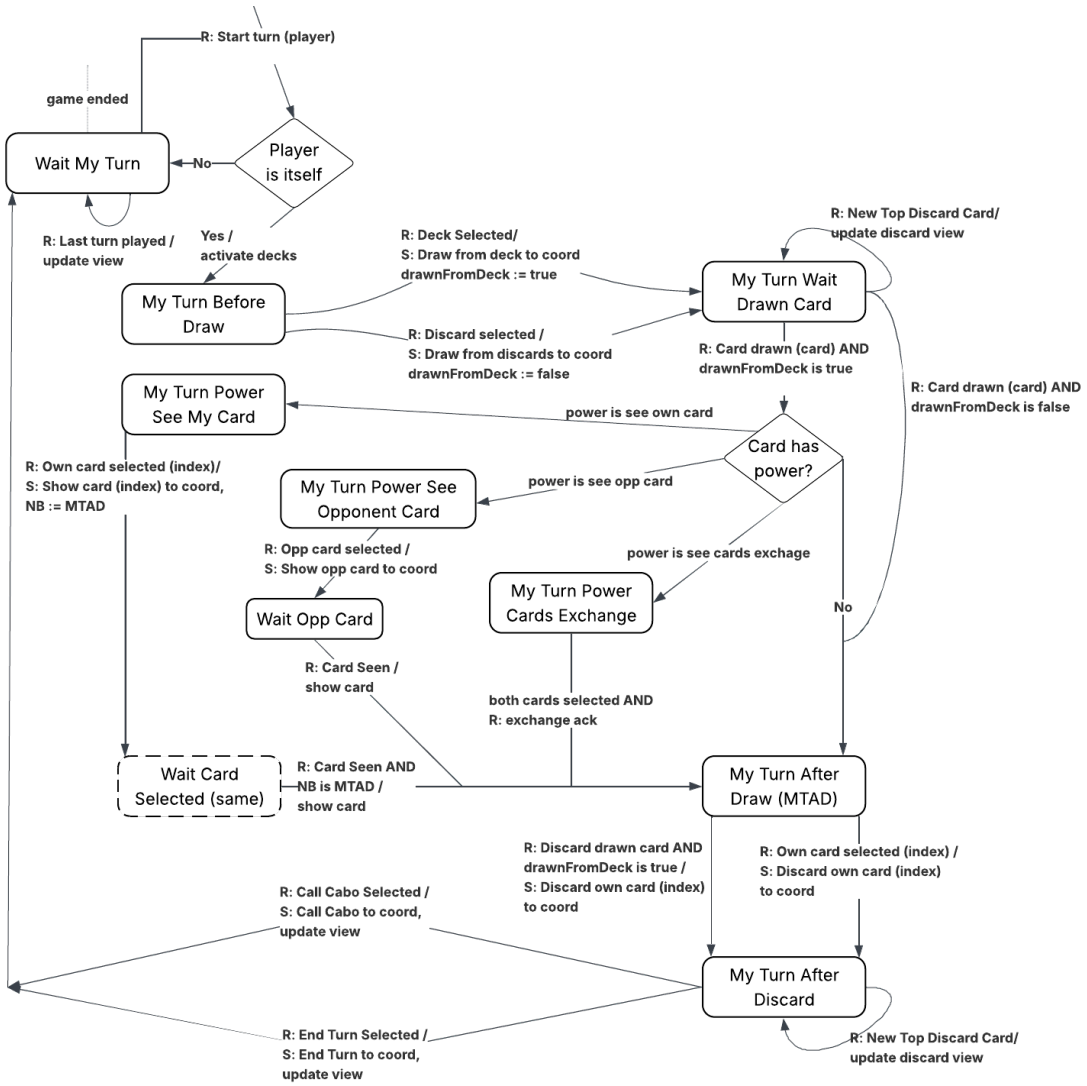


Figure 10: Play Cycle phase. Game Ended is more detailed in following diagram. Same states are present multiple times to make more readable the diagram.

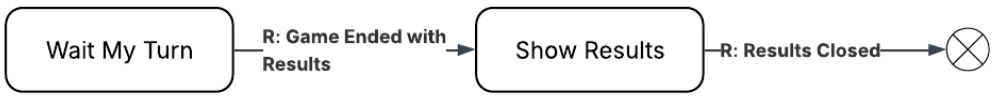


Figure 11: Ending of playcycle.

- end turn by time: indicates that the "My Turn" phase is over and the actor returns to Wait My Turn.

The actor stores one or more data structures to keep references (e.g., the coordinator reference and the view) and any other information needed for correct behaviour.

The "exchange with opponent" card power is handled in two steps, so the transition occurs only after both the player's card and the opponent's card have been selected.

3.4.6 Views Proxy

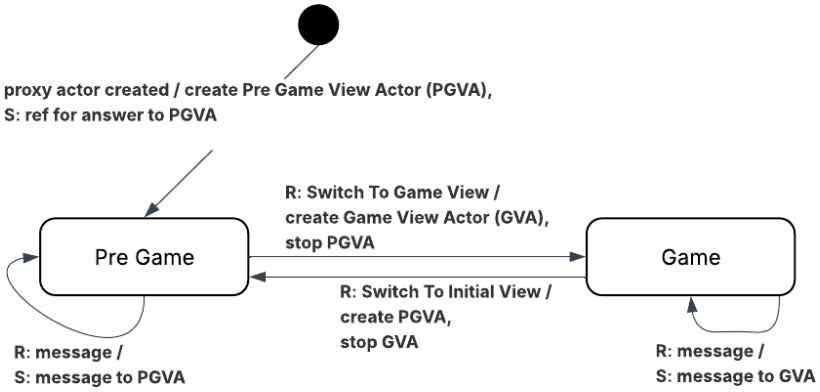


Figure 12: Views proxy.

This actor acts as an intermediary between the view actors and their users. In the application there are two view actors (as shown previously); the ViewsProxy spawns them and forwards messages to the appropriate view actor.

3.5 Interaction

All the components of this project follow the actor model, so they interact by exchanging specific messages.

A common exchange scenario is the one shown in the fig. 13. This represents the message sequence used by the client components to start a match.

In this case the lobby server is involved in the exchange and is used to share the game among the players.

The fig. 14 expands what happens internally inside the host and the joinee when playing a turn.

Most of the messages are exchanged between the *GameCoordinator*, which contains the game logic, and the *View*, which is used by the user to play the game.

As explained in its behaviour, the *Client* plays a secondary role here, taking care only of sharing which actions were taken with the other player.

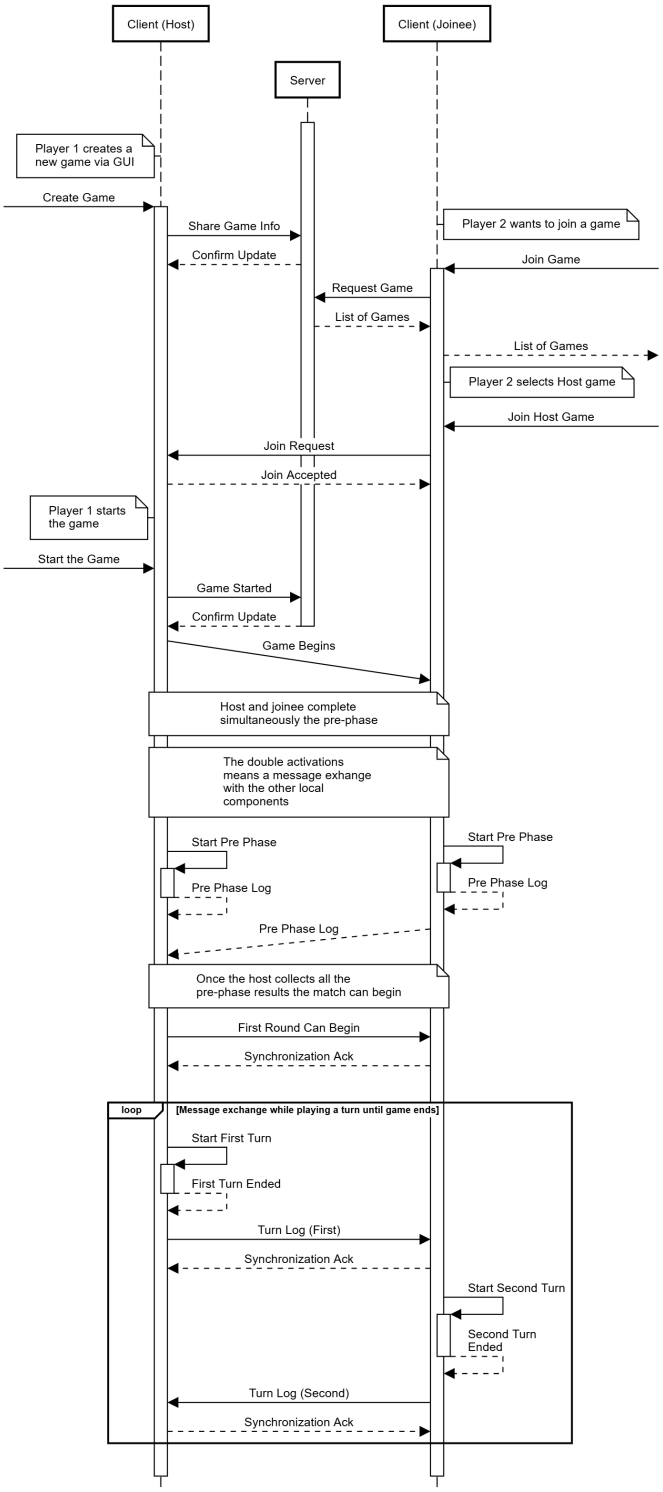


Figure 13: Message exchange between clients to start a match

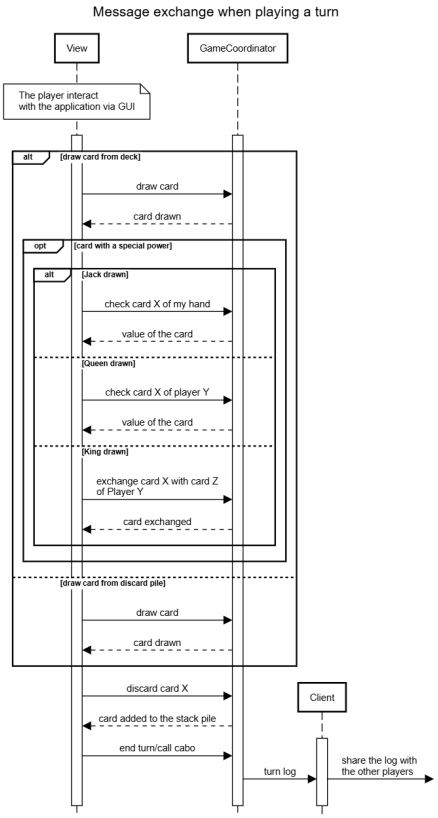


Figure 14: Messages exchange between View and Game Coordinator of same player during their turn.

3.6 Data and Consistency Issues

In this application, there is no data that requires a long-term storage strategy, since there is no authentication mechanism and no local history of past games. For this reason, no database has been used.

The main data shared between the system components is the information related to a game (e.g., its parameters and the list of players), which is collected in the classes described in section 3.3.

All components follow the actor paradigm; therefore, this information is exchanged through messages.

Moreover, multiple instances of the lobby server component can be active at the same time, and they all need to share the same list of games created by players.

The main reason behind the decision of having multiple instances of the server is to create a service that is as available as possible. According to the CAP theorem, achieving this requires sacrificing something else; in our case, we opted for an *eventual consistency* solution. This means that when the list is modified by one instance, it is not shared instantaneously with all the others, but "eventually". With this strategy, in

most cases the server can respond to a player request with low delay, but possibly with an out-of-date list, causing some available games to be missing.

A case in which consistency is fundamental, instead, is the exchange of information between players during a match. In particular, it is vital that every player shares the same view of the game state. This means that every player must always know which turn is being played, who is playing it, and how the cards are distributed among players (both in the deck and in players' hands).

To achieve this, different strategies are adopted depending on the match phase.

Before the actual match begins, the only entity that can modify this data is the host, i.e., the player who created the game. In this situation, the only events that can occur are a player joining or leaving (either by choice or due to external reasons). Both of these events are managed by the host: every change to the game data is performed by it and then reflected to all the other players currently connected to the game.

During the match, the game state is modified at each turn following the decisions and actions made by the player, such as drawing and discarding a card. These changes are made locally by the player who is playing that particular turn and then shared with all the others in the game.

To ensure that everyone receives the new data, a synchronization mechanism is applied. Essentially, to declare a turn as ended, all clients must send an *acknowledgement message* to the current player to indicate that their view of the game data reflects what happened in the last turn. Once everyone shares the same information, the next player can start their turn and the cycle starts anew.

In this phase, if any player disconnects, the host is informed thanks to a *heartbeat mechanism*; if the connection is not re-established within a given time limit, the player is registered as offline in the game data.

Whenever an offline player is supposed to play their turn, to avoid blocking the entire game, the host sends a *skip turn* command, giving the floor to the next player.

During the game, a leader election can be started if the host exits from the game (for any reason); this allows the system to keep an host always active, which is needed to maintain consistency in the case previosly mentioned.

As previously stated, the most important data is the game being played: its information is modified by the GC-Actor and then passed to the Client to distribute it. Inside the coordinator actor, a copy of the game data modified during its turn is also kept, so that if any problem arises during data modification, a consistent version of the game can be used and distributed.

3.7 Fault-Tolerance

Since this is a distributed system, it is important to implement fault-tolerance mechanisms to ensure that the application continues to work as expected even in the presence of failures.

First, to keep the lobby server highly available, multiple instances of the actor that provides the lobby functionality can run at the same time, and they share the same

information. This way, even if one actor fails for any reason, another one can still respond to players' requests.

Moreover, this solution also helps with load balancing: additional server instances can be started when the number of players becomes too large for the currently active ones.

Additional fault-tolerance mechanisms are also used during a match to avoid situations where a player disconnects and the game becomes blocked because their turn cannot be played.

In particular, a *heartbeat mechanism* is used by the host to check whether players are still online. If a player fails to respond, the host flags them as disconnected and informs the others. However, the player is not removed from the game entirely, since it is always possible to re-establish the connection and continue the match. In the meantime, the turns of disconnected players are skipped so that the game can continue for everyone else. To do this, when a player ends their turn, they check whether the next player is online: if they are, the match continues normally; otherwise, the same player sends a message to all the others, emulating an empty turn (i.e., a turn in which the game state does not change, except for the turn number).

The same heartbeat mechanism is also used to detect a host failure and, if it happens, to trigger a leader-election process to select a new host among the players that are still connected.

3.8 Availability

Every player has a vision of the game after last turn played, so if errors occur a consistent version of it could be requested to others.

Also, server entity could be managed as multiple nodes sharing information about joinable games. After one server received request to upload new game, the heartbeat system already cited is exploited to share data about games using a versioning system to keep the most recent. Considering heartbeat not instantaneous, a player could ask to one of servers game list when some games are uploaded but not shared yet with every entitites. We don't consider it a problem because the player has the possibility to refresh games list and, anyway, could enter other games already shared[2].

If a partition occurs we chose to keep alive the biggest one so most players can continue to play, avoiding inconsistent network. This is possible checking the own heartbeat state. During the process each node has a vision of the system, when a partition occurs (partition A, PA, partition B, PB), nodes of PA know how many they are and how many nodes are present at the last time check before partition. A node turns itself down if is in the little partition.

3.9 Security

In this project, we did not plan to adopt specific security mechanisms.

The game is based on a P2P infrastructure, which is inherently less secure than a client-server architecture because it requires direct connections between peers.

[2]If they want enter a specific game they can use code to ask directly to host of designed game.

Moreover, during the match, the actions taken by a player during their turn are sent directly to the others without any validity checks. Therefore, implementing an anti-cheating mechanism would be a complex task and exceeds the scope of this project.

Finally, since we are not working with sensitive information and are simply implementing a card game, we considered the adoption of cryptographic protection mechanisms to be a superfluous investment of resources.

4 Implementation

4.1 Technological details

4.1.1 Akka Cluster

Akka and Akka Cluster are chosen to realize this project because they offer a gossip system and possibilities to get information using key/code with instruments like Receptionist. Also we wanted improve our knowledge about this framework.

Gossip - Nodes execute the gossip protocol to disseminate cluster membership and state information; it is also possible to subscribe to the corresponding events.

In particular, the client leverages these events as a heartbeat mechanism to detect when another player is no longer reachable.

To do so, it spawns a dedicated actor, called `ConnectionHandler`, to intercept the notifications emitted by Akka Cluster when another node becomes unreachable or leaves the cluster. These notifications include the affected node address; therefore, the actor can match it against the peers participating in the current match and notify the client accordingly.

The primary motivation for introducing a separate actor is modularity. In particular, we initially considered implementing our own heartbeat mechanism, but we opted to reuse the cluster facilities since they already provide similar failure-detection signals. Since we were not entirely sure that relying solely on these events would cover all the failure cases we anticipated, we isolated this functionality so that it can be replaced more easily in the future with one of our creation, if needed.

In the end, these events met our requirements, so we retained this approach to handle node failures.

Receptionist - The receptionist service allows to register an actor reference under a well-known key so that other actors can look it up without knowing its address. Each cluster node runs its own receptionist, which is kept up to date through the gossip protocol, thus creating a distributed data structure.

When the host creates a game, the client actor registers itself to the receptionist using the generated game code. Once this is done, other players can join the game using this code, which can be obtained from the lobby server or shared directly by the host. Therefore, the real address of the actor is decoupled from a more generic identifier used to find it.

Serialization - All application messages are serialized to make them readable by different actors. In the Cluster configuration, it is possible to specify which serializer to use; in our case, we used `jackson-cbor`. `CborSerializable`[3] is a marker trait we created for this purpose: application messages extend it so they can be serialized as specified.

Split Brain Resolver - The split brain resolver is an Akka Cluster mechanism that defines how the cluster should behave when a group of nodes can no longer communicate with another group. There are different strategies; we chose to shut down the smaller partition. During gossip, nodes know the cluster state and, when a partition occurs, each group can estimate both its own size and the size of the other one (at partition time). This choice was made to allow the largest possible number of players to continue playing without compromising the integrity of the system.

Distributed Data - Distributed Data is an Akka tool used to share data between nodes in a cluster. Data is accessed through an actor that provides a key-value store-like API. Keys are unique identifiers that include type information for the associated values. The values are *Conflict Free Replicated Data Types (CRDTs)*.

This data can be shared among all nodes, or only among those with a specific role within the cluster.

It is eventually consistent and designed to provide high read and write availability while maintaining partition tolerance.

We used it to implement the lobby server, specifically as a mechanism to share the list of matches created by players across different server instances.

In the Akka Cluster configuration file, we added the "Server" role as a marker for the nodes where the lobby server actor is active. Distributed Data is configured to replicate data only across nodes with this role, so players cannot directly access this list.

In this case, eventual consistency is not a problem: the worst-case scenario is that a player receives an out-of-date list of available matches, which is not considered a major issue.

4.1.2 Card

The card model is inspired by this model. During the initial phase of the project, we found this repository while designing a possible solution to implement the card features, so we decided to use it as a starting point. We then added some methods and some custom message codification that were required by our solution.

5 Validation

5.1 Automatic Testing

Project components were tested using ScalaTest and ScalaTestWithActorTestKit for actors' behaviour. This allows us to run automated tests and verify that the actors

[3]Module `cabo-commons`, package `messages`

follow the expected behaviour.

Tests were run using the IntelliJ environment with the sbt system. In addition, a GitHub Actions workflow was set up to ensure that newly committed code does not break previously passing tests.

5.1.1 Model's elements

CardSpec

- What it checks: `Card` has scores and powers; `Hand` has a score and allows swapping cards; `Deck` allows drawing and adding a new card.
- How to run:

```
sbt "project commons" "testOnly *CardSpec"
```

RevealingSectionTurnLogSpec

- What it checks: `RevealingSectionTurnLog` only allows adding `SeeSelfCard` events, and the order of events is enforced. In particular, a `RevealingSectionTurnLog` must:
  - have no events when it is created;
  - allow seeing one card when it is created;
  - allow seeing a second card after seeing the first one;
  - throw an exception when trying to add further events after seeing two cards;
  - throw an exception when trying to add events different from `SeeSelfCard`.
- How to run:

```
sbt "project commons" "testOnly *RevealingSectionTurnLogSpec"
```

PlayCycleTurnLogSpec

- What it checks: `PlayCycleTurnLog` only allows adding valid events, and the order of events is enforced. In particular, a `PlayCycleTurnLog` must:
  - have no events when it is created;
  - allow drawing a card from the deck/discard stack and contain only that event after the draw;
  - allow using the `SeeSelfCard` power when it is drawn;
  - allow using the `SeeOpponentCard` power when it is drawn;
  - allow using the `ExchangeCardWithOpponent` power when it is drawn;

  - allow discarding one of your cards after drawing a card from the deck/discard stack;
  - allow discarding or keeping a card with a power after using it;
  - throw an exception when adding an invalid event.

- How to run:

```
sbt "project commons" "testOnly *PlayCycleTurnLogSpec"
```

5.1.2 Game Coordinator Testing

This test was created to check the behavior of the Game Coordinator actor. The Client and View are emulated using probes, so it is possible to verify the messages they receive. A Status probe is also used to evaluate the game state returned via `GameInformation`, a specialised message introduced to retrieve the current status on request.

Helper methods are provided to emulate the turn steps, making the code more readable and making it easier to understand what is happening during the test. Moreover, if the messaging protocol changes in the future, the test will be easier to update.

The test checks:

- data generation and turn start;
- sending a card;
- the revealing-section phase;
- drawing from the deck and from the discard stack;
- updating the top card of the discard stack;
- discarding the drawn card or one of the player's cards;
- ending a turn due to timeout, command, or Cabo call;
- sending the *last turn played* to the view;
- sending the turn-updated acknowledgement to the client;
- creating an empty turn;
- using powers: see own card, see an opponent card, or exchange a card with an opponent;
- signalling that the game ended when the turns limit is reached, the deck is exhausted, or Cabo is called.

How to run:

```
sbt "project client" "testOnly GameCoordinatorActorSpec"
```

5.1.3 View User Command Listener Testing

What it checks: this test verifies that the listener inside the view sends the correct messages when its methods are called.

How to run:

```
sbt "project client" "testOnly ViewUserCommandListenerSpec"
```

5.1.4 Client Testing

As for the other components, a series of unit tests has been developed to ensure that the actor respects its message protocol and follows the correct behaviour. Moreover, since the client is responsible for most of the distributed aspects of the system, a series of tests has been designed to ensure that it works correctly even when different client instances are not on the same machine but are connected through an Akka cluster, as in a real deployment.

Local Machine Environment These tests are designed to verify that the actor switches to the correct behaviour upon receiving specific messages, and that it sends the correct information to the other components that need to interact with it. In particular, the tests are:

- **should be able to change the player's name**
- **should be able to join a game created by another player**
- **should be able to enter a game using the game code**
- **should not be able to join a game if the maximum number of players is already connected**
- **should receive a notification when another player joins the game**
- **should be able to leave a game it joined**
- **should be notified when someone leaves the game**
- **should receive an abort notification if the host leaves the game**
- **should be able to start a game with a stub coordinator**: this test is used to ensure that the client follows the correct message protocol while interacting only with the game coordinator component. To avoid possible failures caused by the coordinator (and not by the client under test), a *"stub"* coordinator is passed to the client as the target of its messages. The only action that the *"stub"* performs is forwarding any message it receives to a *"probe"*, which is used to verify message type and content.

- **should be able to pass the turn around**: this test verifies that all clients joined in a match are updated whenever a player takes a turn. Also in this test, a *"stub"* is used to isolate the client component and verify its actual behaviour.
- **should be able to start a game with a true coordinator**: this test is equivalent to the one with the stub coordinator, but in this case a real instance is instantiated and used inside the client actor. This test is included just to ensure that everything works correctly.
- **should be able to play with a true coordinator**: this test simulates the creation and start of an actual match. Here it is verified that the client follows all the steps needed to go through all the designed phases, from creating the game to executing the first two rounds (one for each client used in this test).
- **should be able to leave while playing if it is the host**: this test is used to verify the client's behavior when one of the players decides to leave the game during the in-game phase. Here, we simulate a player who is also the host leaving the game.
- **should be able to leave while playing if it is a joinee**: same situation as the test above, but in this case it is one of the players who joined the match who decides to leave it.
- **should leave if it remains alone in a game**: this test verifies that if only one player remains active in a game, the client leaves the match and returns to its initial state.

The last five tests listed are designed to verify the message protocol between the Client and the GameCoordinator components. For this reason, they require a working *gameCoordinator* component and are located in a separate file from the other tests.

These tests can be executed from a terminal in an *sbt* environment using the command:

```
sbt "project client" "testOnly ClientTest"
```

and

```
sbt "project client" "testOnly ClientAndGameCoordinatorTest"
```

Distributed environment simulation To simulate a distributed environment, we take advantage of Akka's *multi-jvm* and *multi-node* testing modules. The first one is used to execute Scala code in different JVMs, while the second one allows tests to be configured to run in JVMs that are physically executed on different machines, simulating an Akka cluster formed by distributed nodes.

In the latest versions of the *SbtMultiJvm Plugin* (the main component behind these two modules), it is possible to run multi-node tests on a single machine without any special configuration by running them as standard multi-JVM tests. These tests can then

be run over multiple machines without any changes by using the multi-node additions to the plugin. For this project in particular, all the tests in this section have been written following the *multi-node* specifications, but were only executed locally on a single machine.

All these tests can be found inside the sbt sub-project *cabo-client*, in a folder at the same level as the *source* folder, as requested by the module specifications. Each test is located in a different file so that it can be launched individually, since it is mandatory to follow specific naming rules for the plugin to discover and execute them.

The main focus of these tests is to verify the correct behaviour of the client in the case of a host failure and the subsequent election of a new one. In particular, some specific cases were chosen to analyse the messages sent and received by the clients.

To recreate some of the expected scenarios, it was necessary to force the behaviour of the client by using specific messages that exist only for testing purposes. For example, the message *RemoveCheckPlayerStatus* can be sent to a specific client to prevent it from receiving the host disconnection notification. This is used to control which client reacts to the host failure and to create exactly the scenario we want to test.

All of this is necessary because, in Akka, even in a simulated distributed environment, we cannot control the order in which messages are received, especially when they are sent by the Akka system itself and different nodes are involved.

More specifically, the test cases designed and implemented are:

- **Client Multi Node**: this test is fairly simple and is mainly used to verify that the cluster is built correctly and that the clients can communicate with each other.
- **Disconnection Notify**: this test verifies that a client correctly receives the expected message when another client crashes.
- **Single Election**: this test verifies the scenario where the host disconnects during a match and the joinee that first receives the notification is the one that has to be elected as the new host.
- **Multi Election**: the scenario analysed here is very similar to the previous one, but in this case the joinee that recognises that the host has disconnected is not the one who should become the new host; therefore, even if it starts the election process, it must be interrupted.
- **Eventually Election**: this test covers the general case. Here, no order is forced on the clients. Any node can receive the host-disconnection notification and start the election. It is only verified that the correct one wins.

To run all these tests using different JVMs on the same machine, use:

```
sbt "project client" "multi-jvm:test"
```

To execute only a specific suite, for example the *Multi Election* one, use:

```
sbt "project client" "multi-jvm:testOnly MultiElection"
```

where "MultiElection" is part of the name of the `Scala class` used by the *SbtMultiJvm Plugin* to discover and run the tests. These classes can be found inside each Scala file that represents one of these tests, before the `abstract class` that implements the suite.

In the *Multi Election* example, the relevant code is shown in listing 1. The name to use is the one of the class that extends the abstract class that contains the actual tests, without the final "`MultiJvmNode1(2,3)`".

Listing 1: Multi Election naming example

```
1 ---
2 class MultiElectionMultiJvmNode1 extends MultiElection
3 class MultiElectionMultiJvmNode2 extends MultiElection
4 class MultiElectionMultiJvmNode3 extends MultiElection
5
6 abstract class MultiElection extends MultiNodeSpec(MultiNodeConfig) with
      STMultiNodeSpec with ImplicitSender{}
7 ---
```

There are a couple of important notes regarding the execution of these tests.

First, at the end of tests where the crash of one of the clients is simulated, an exception may be raised, namely

```
java.net.SocketException: Connection reset
```

or

```
java.util.concurrent.RejectedExecutionException:
Worker has already been shutdown.
```

The occurrence of these exceptions does not indicate an error, but is an expected event. In fact, the tests pass correctly even if one of them occurs.

These exceptions occur because this scenario simulates a failure, so not everything is terminated gracefully.

In particular, the first exception is raised because `testConductor.exit` is used to crash the client, which kills the JVM where it is executing without closing any sockets, preventing the cleanup of *Akka remoting*. When this happens, *Netty* (used by Akka as a network framework) logs the exception when trying to read from that socket.

To avoid the first exception, we tried using `testConductor.shutdown` for a more graceful shutdown. The difference between these two methods is that the first one uses `System.exit` inside the node that needs to crash, instantly killing the JVM, while the second one executes Akka's shutdown procedure to remove the node from the cluster and close everything correctly (including the socket).

While this effectively resolved the first problem, we encountered another one, namely the second reported exception.

This happens because of a race condition in the teardown procedure of the *testConductor*. In other words, this exception may occur during the shutdown phase of multi-JVM tests because `testConductor.shutdown` performs a cooperative termination of the remote JVM. During this process, Akka shuts down the ActorSystem and

the remoting infrastructure, including Netty I/O workers. Due to concurrent shutdown steps, internal components such as the *TestConductor ClientFSM* (the actor that manages communications between the JVMs) may still attempt to close or handle network channels after the Netty workers have already been stopped, resulting in the `RejectedExecutionException`. This is caused by a shutdown race condition, is intermittent by nature, and does not indicate a functional issue in the test logic.

During the development of the *Multi Election* test case, we ran into another problem: sometimes, even if the node where the host was launched was removed (either by using `testConductor.exit` or `testConductor.shutdown`), the actor remained active and interfered with the test logic.

In particular, the old host is reported as crashed and a new election is initiated; however, since the actor keeps running, it blocks the other joinee's election attempt and tries to become the host again. This is logically correct, because it was the host in the first place, so, following the bully protocol, it is the most suitable candidate. The problem is that, by trying to reclaim the host role, it breaks the expected message sequence and the test fails.

We could not understand why this situation occurs. Looking at the code that simulates the host failure (listing 2), the logic should be correct. In fact, in lines 103–104, `testConductor.shutdown` is used to shut down the second node (where the host is); then `Await` is used to block execution until the future completes. We also used a longer-than-necessary timeout to ensure that the value actually returns and is not interrupted by the system.

The failure occurs at line 111, where `probeClient` (used to check which messages are sent to the client) does not receive the expected message. This means that the future completes correctly, because the await passes.

Listing 2: Multi Election code failure

```
102 ---
103 val exitFuture = testConductor.shutdown(node2)
104 Await.result(exitFuture, 30.seconds)
105
106 enterBarrier("host4-removed")
107
108 val m = probeClient.expectMessageType[ElectionStarted]
109 assert((m.replyTo.toString.toLowerCase contains "client4-2") && m.candidateRank
        == 3)
110
111 probeClient.expectMessageType[ElectionWon](10.seconds)
112 ---
```

As shown in fig. 15, in the first two lines of the logs we see node 2 (which should not be present at this moment): the old host logs that it is interrupting the current election because it believes it has priority.

After that, we see the failed assertion about the received message, as stated above.

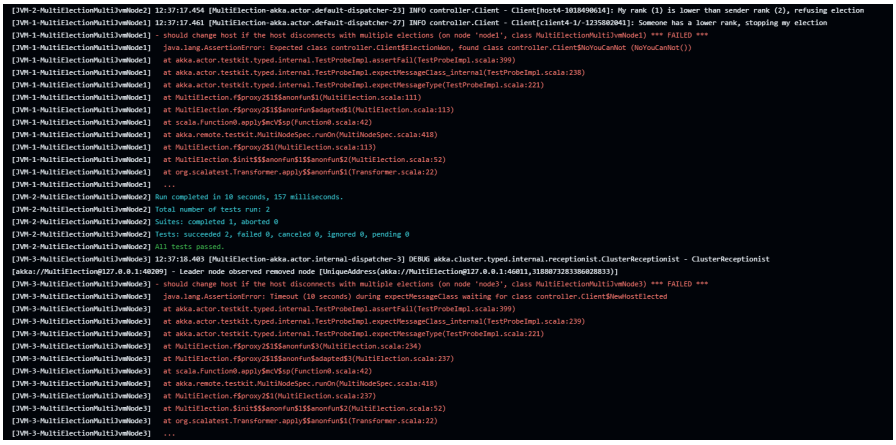


Figure 15: Error log of the Multi Election test suite in case of error

Then, we see a log confirming that node 2 has concluded its part of the test and that, from its point of view, there were no issues. It is not possible to determine whether the second node actually terminates its execution at this moment, because the three nodes work simultaneously but print on the same interface; therefore, it may have happened earlier but was logged only now.

Finally, we see another error log from node 3, which is linked to the problem that occurred on node 1, since the messages are no longer what was expected.

This situation no longer occurred during the final tests, in which we actively used the system to verify that all its components worked correctly, but we did not specifically change anything to prevent it. So, in case it may happens again in the future, we left this explanation here to describe our thoughts about it.

5.2 Acceptance test

The view was also tested manually to verify that frames, panels, and dialogs are rendered correctly. Testing was performed incrementally, starting from the pre-game frame and verifying that all view transitions work as expected.

6 Release

The project is composed of four modules:

- `commons`: the application model; it contains classes such as `Card`, `Game`, `Player`, and other information shared by the client and server, such as `ServerMessages`;
- `client`: the application run by the player;
- `server`: the server used to play public games; it is not mandatory to run it to play Cabo;

- `seed`: runs two seed nodes to enable Akka Cluster functionalities; it is mandatory for them to be running.

All modules are available in a single GitHub repository: cabo-online-distributed-systems-project. In a real-world application, they would likely be separated into different repositories; however, for didactic purposes, it is convenient to keep them together. Moreover, as explained earlier, the seed nodes are mandatory for Akka Cluster to work correctly; therefore, in a real deployment, players should not download them locally but rather expect them to be online and reachable.

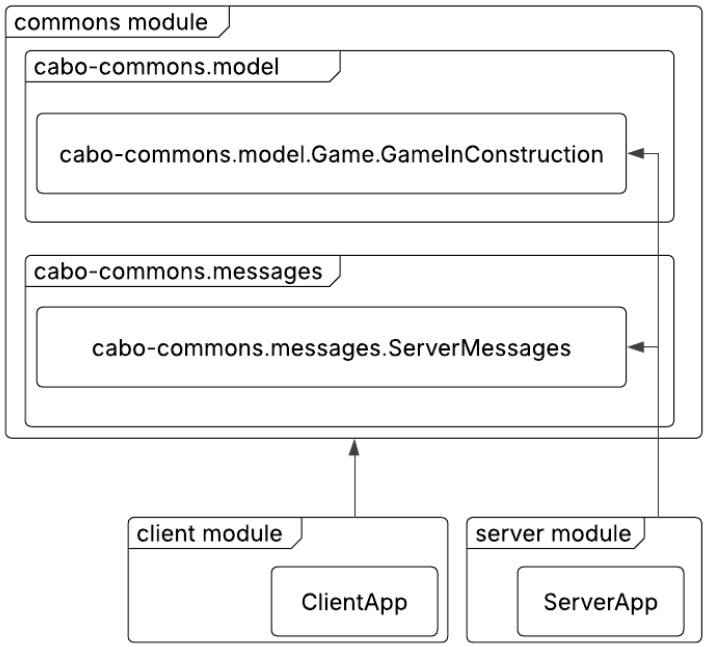


Figure 16: Dependency graph of application. Seed module is not showed because run only seeds with empty behavior.

7 Deployment

To install the software, Java 17+ and the Simple Build Tool (sbt) (`https://www.scala-sbt.org/download`) are needed.
Download the code from the GitHub repository: `git clone https://github.com/angelo-will/cabo-online-distributed-systems-project.git`
Run:

```
sbt install
```

Three `.jar` files will be generated: `seed.jar`, `server.jar`, and `client.jar`.

The seed module must be started first so that Akka Cluster can work:

```
java -jar seed.jar
```

Information about cluster creation will be printed (e.g., nodes joining the cluster and being marked as `UP`).

The server is not required for private games, as the game code can be used to find a match created by a host. However, it can be started with:

```
java -jar server.jar
```

It prints cluster and server information whenever a game is published, updated, or deleted.

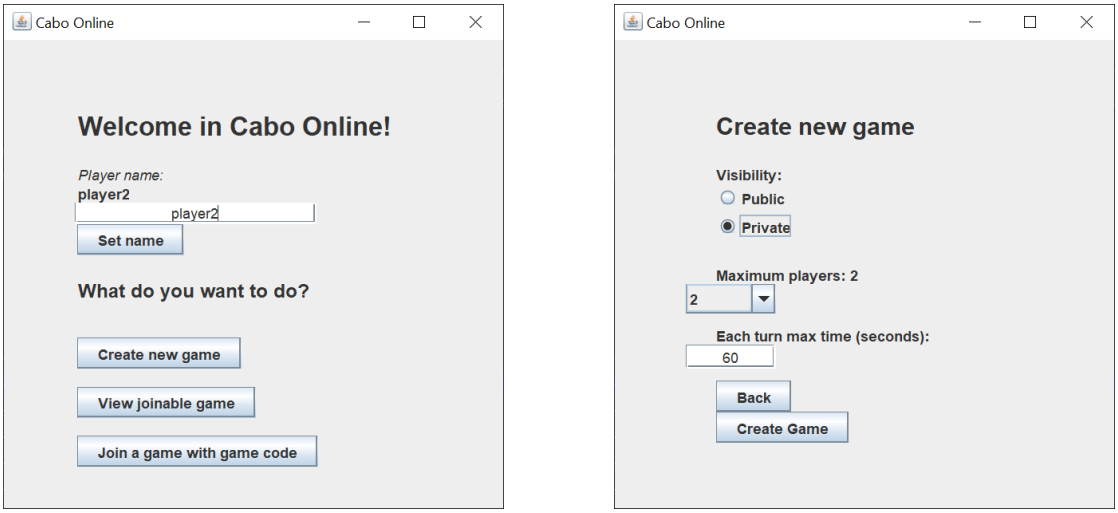
The client is the player application and can be started with:

```
java -jar client.jar
```

For debugging purposes, game-related logs are printed (e.g., opponents' cards are visible, and the deck structure is shown).

Akka now uses a license model that allows 15 minutes of execution, so each node (seed, server, client) is stopped after this period.

8 User Guide



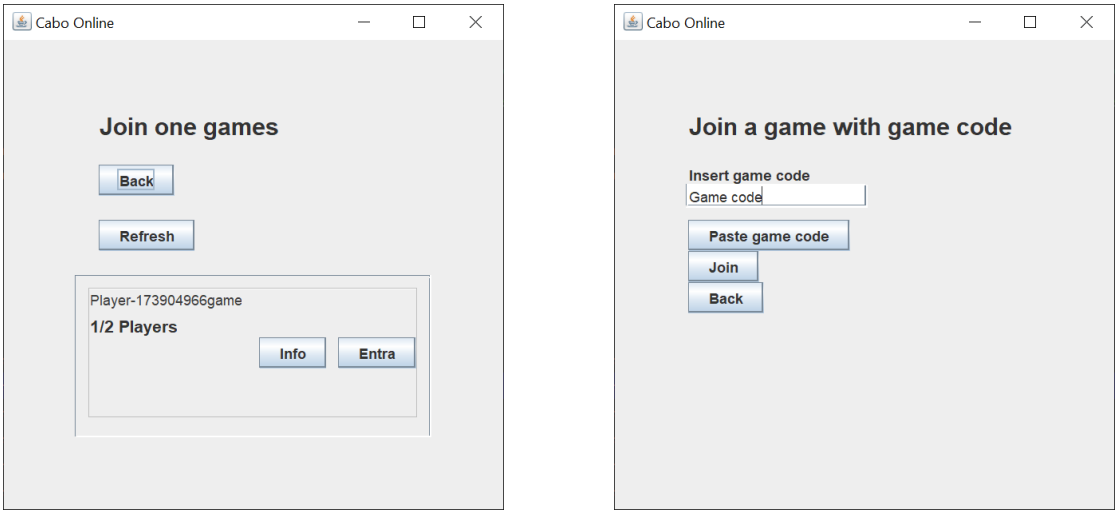
(a) Start screen at application's start.

(b) Create a new game panel.

Figure 17: Welcome and create new game panel.

After running the client application, a welcome panel (fig. 17, fig. 17a) is shown, offering the player three choices: create a game, join a game, or join a game using a game code.

The first option brings the user to fig. 17b, where they can set up a new game by choosing the game parameters. When *public* is selected, the game contacts the server to publish the match (if the server cannot be reached, a warning message is shown, but the lobby is created anyway as a private game). The lobby view is then opened and, once there are at least two players, the game can be started.



(a) List of games joinable

(b) Join a game with game code.

Figure 18: Join game possibilities.

If *Join a game* is selected (fig. 18), the player is brought to a panel where all public games are listed; games information are shown and the player can choose one to join.

If *Join using game code* is selected, the player is brought to a panel where they can enter the game code received from the game creator to join a private game.

Once the game is joined, the lobby view is opened (fig. 19).

Once the game starts, the system opens a new window showing players, cards, and game information: this is the revealing-section phase. Here, it is possible to see two of your own cards. When all players finish this phase, their selections are shown in the right log panel and the play cycle starts.

During your turn (fig. 20), you can draw from the deck or the discard stack by clicking the buttons below the game information. Use the buttons under "YOU" to select one of your cards when requested, and the button on the right to decide how the turn ends. Examples of power using are shown at fig. 21 and fig. 22

At the end of the game, the results are shown (fig. 23).

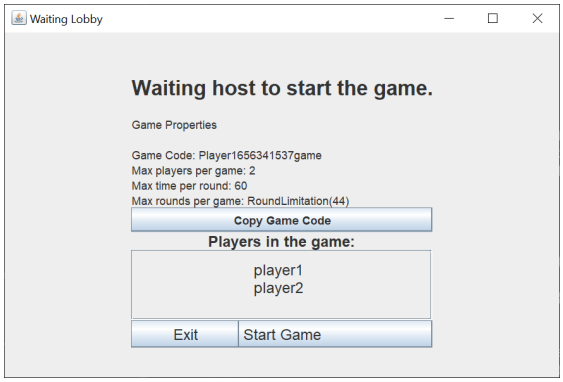


Figure 19: Lobby screen. The start game button is active when there are at least two players (host view).

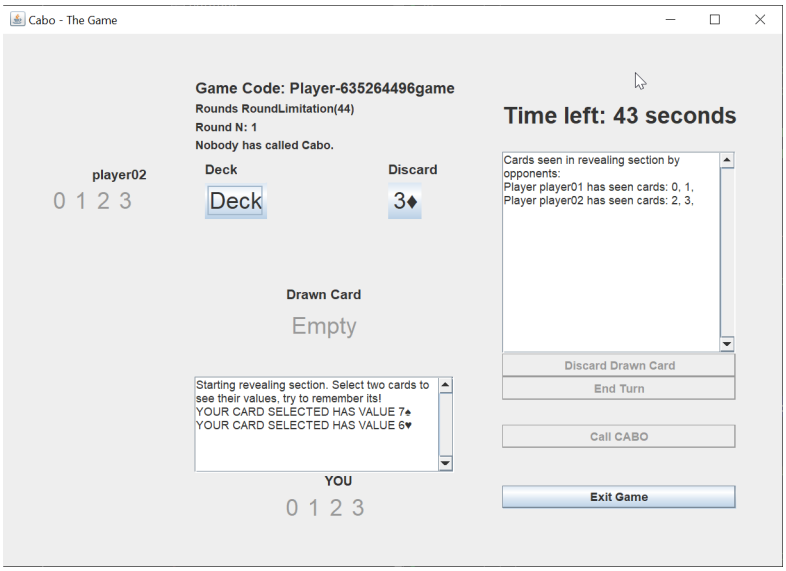


Figure 20: Game view when everybody ends revealing section. Which cards have been seen by players are showed in the right panel. Values of cards selected by you are shown in bottom panel. Also, here your turn is just start so you can see deck and discard stack are active.

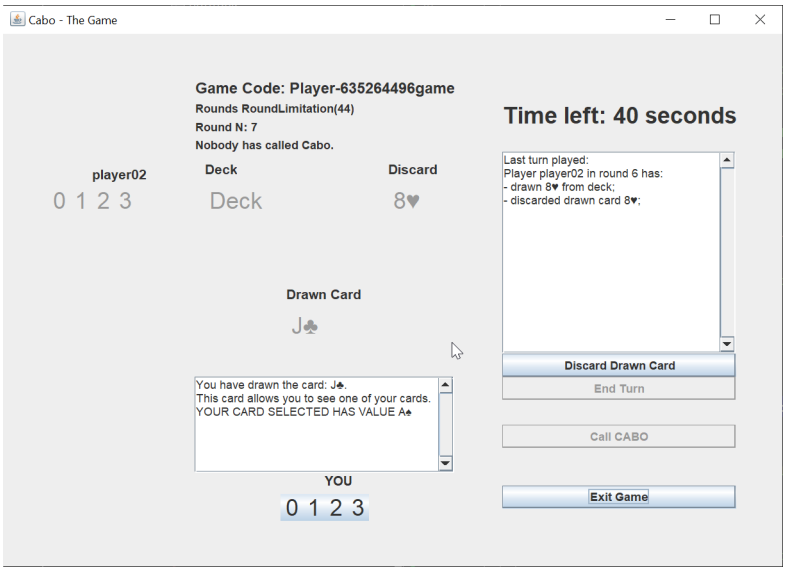


Figure 21: Example of view after draw card with power to see one of own card. The value of card selected is shown in bottom panel. In the right panel the last turn log is visible.

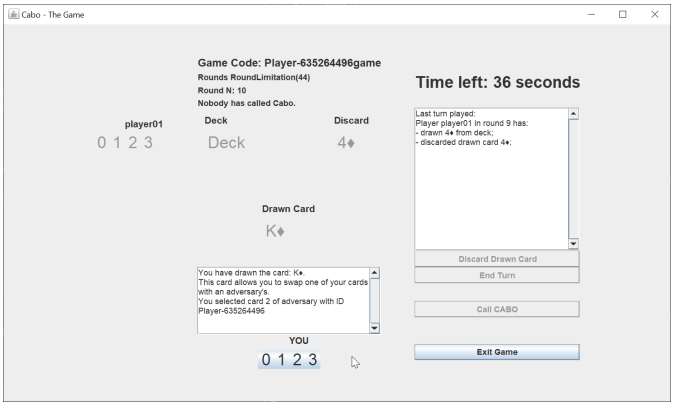


Figure 22: Example of view when draw the power to change one of your cards with opponent. Your cards and opponents are active until you chose wich cards replace.

Figure 23: At the end of the game (for cabo calling, deck ending or max turn reach) results are shown.

9 Self-evaluation

9.1 Angelo Tinti

The design of the actors, game phases, model components, and other core elements was carried out collaboratively. We believed it was essential to maintain a holistic, "all-around" vision of the application from the very beginning. Given the system's heavy reliance on message passing, defining the communication protocol early on significantly streamlined the development process, particularly for the agents that would utilize it. Furthermore, we wanted to encourage continuous discussion and peer review to better evaluate potential solutions.

Regarding the implementation phase, I focused primarily on the Game Coordinator and the View (including their respective messages), while the Server and model elements of common module were developed through a joint effort.

I am satisfied with the final application, especially considering that we initially underestimated the workload required for software of this nature. Utilizing Scala and Akka was both intriguing and educational, though at times it added complexity to the development of the single-player logic. While more corner cases could have been addressed and some solutions further refined, I am overall pleased with the result.

Ultimately, this project proved highly valuable for understanding the complexities involved in maintaining communication and state integrity within a distributed context.

9.2 Michele Bachetti

The design phase of the project was carried out collaboratively with the other members of the group, focusing especially in the architectural and infrastructural decisions concerning the overall system.

It was essential for the team to be fully aligned on these aspects, as we decided to adopt an actor-oriented architecture; for this reason, it was necessary to define the message protocol from the very beginning, since it has a significant impact on how each actor is implemented. Furthermore, we aimed to remain as aligned as possible in order to

share our individual perspectives and support each other during development, given the intrinsic complexity of the system.

The decision to use Akka and Scala also emerged from a shared proposal, as we had previously worked with these technologies and considered them a suitable solution for addressing our problem, with the additional goal of exploring their capabilities and limitations.

With respect to development activities, I was primarily responsible for the implementation of the *Client* component, and consequently also of the management of the connections between players. In addition, I also worked on the development of the *Server Lobby* component, with a specific focus on the use of the *distributed data*. Finally, all group members collaborated during the final testing phase, with particular attention to identifying as many problematic corner cases as possible and scenarios that could lead to unexpected system behaviour.

Overall, I am mostly satisfied with the final outcome, especially considering that we underestimated the overall workload required by the project and, in my case, did not always manage time as effectively as intended. The choice of using Akka proved effective in framing the project from an architectural perspective; I believe that an actor-based solution is suitable for this domain; however, it also introduced additional complexity in certain aspects of development, which might have been avoided by using a specialized framework or a game engine designed for distributed environments. This is likely also due to the fact that, although we had prior experience with Akka, we are not experts in its use; a deeper understanding of its components and mechanisms might have led to different, and potentially better, design decisions.

Undoubtedly, this project allowed me to gain first-hand experience with the "hidden" complexity underlying distributed systems, as well as with the strategies and solutions that enable their correct operation.

10 Future works

In this version of the system, if a player disconnects from a game due to a connection failure, they have no means to rejoin that match. A possible update would be to effectively provide this possibility.

To improve the consistency of the game state, a useful update would be to add a hashing mechanism when sending game updates to other players, allowing them to verify that a message has not been corrupted.

Moreover, it would be possible to implement a simple anti-cheating system in which, upon receiving a turn log from another player, the actions are validated against the previous game state shared by all players, to prevent forgery.

Another nice feature would be to add an authentication mechanism to keep track of players and their past matches. This would allow each player to have a record of their performance and would also enable a ranking system to reward the best players.

References

[1] Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48–59, 1982.