Final Report: Distributed Hanabi

Final Report for the Distributed Systems Course 2025–2026

Blagoja Savevski
blagoja.savevski@studio.unibo.it

Gabriele Santi
gabriele.santi@studio.unibo.it

Pengyue Xu
pengyue.xu@studio.unibo.it

June 8, 2026

Hanabi is a cooperative multiplayer card game for 2–4 players where players score by guessing their own hidden cards based on hints given by other players. This project aims to implement a working version of the game that manages the shared state consistently and reliably, while being able to tolerate disconnects from a client or the server.

AI Disclaimer
AI tools were used during the development of this project to support coding, debugging, architectural reasoning, and language polishing for the report. The final implementation, design choices, tests, and written content were reviewed, adapted, and validated by the authors.

Contents

1 Concept

The software implements the game Hanabi, a 2–4 player cooperative board game where players attempt to build the greatest "pyrotechnic show" by ordering each other's cards. Each player starts with 5 cards from a 50-card deck. The goal is to order the cards by number (1–5) and by color (red, yellow, green, blue, white). A successful game ends with 5 different colored decks in increasing order. Information for your own cards must only be acquired through the strict communication methods the action systems permits. Each turn, one user does one of the allowed actions, then the updated game state is shared to all the users. Each player can:

- *Give a hint* to another player and spend one of 8 hint tokens in total. The hint can give information about either the color or the number of a subset of the other player's cards.
- *Play a card* and attempt to place it in order in one of the colored decks. If the card cannot be legally placed on the corresponding deck, the move is unsuccessful and the number of misfires increases by one. Then the player draws a card to replace the played one.
- *Discard* one card from their hand and return one of the already spent tokens to the pool. The player must end each turn with the same number of cards, so a new card is drawn.

1.1 Use Case description

The aim of this project is to design and implement a distributed version of the game Hanabi. It consists of a web-based application where players can connect to a server and play together directly from their browsers. The system needs to store basic player information such as full name, username and password, in order to conduct orderly match creation and orchestration. Once the client is connected to the server, authentication is required and if a player is not registered yet, the system should allow the user to sign up. Then a player can choose whether to create a lobby to play with other players or join existing ones. The server is responsible for creating a new game and assigning a unique game ID, enabling persistent storage of its state and potential recovery in case of system restart. Players visualize the shared game state on their clients and, on each turn, send a command to the server. Each command is validated by the server according to Hanabi rules, then updates the game state accordingly and notifies all players of the new state.

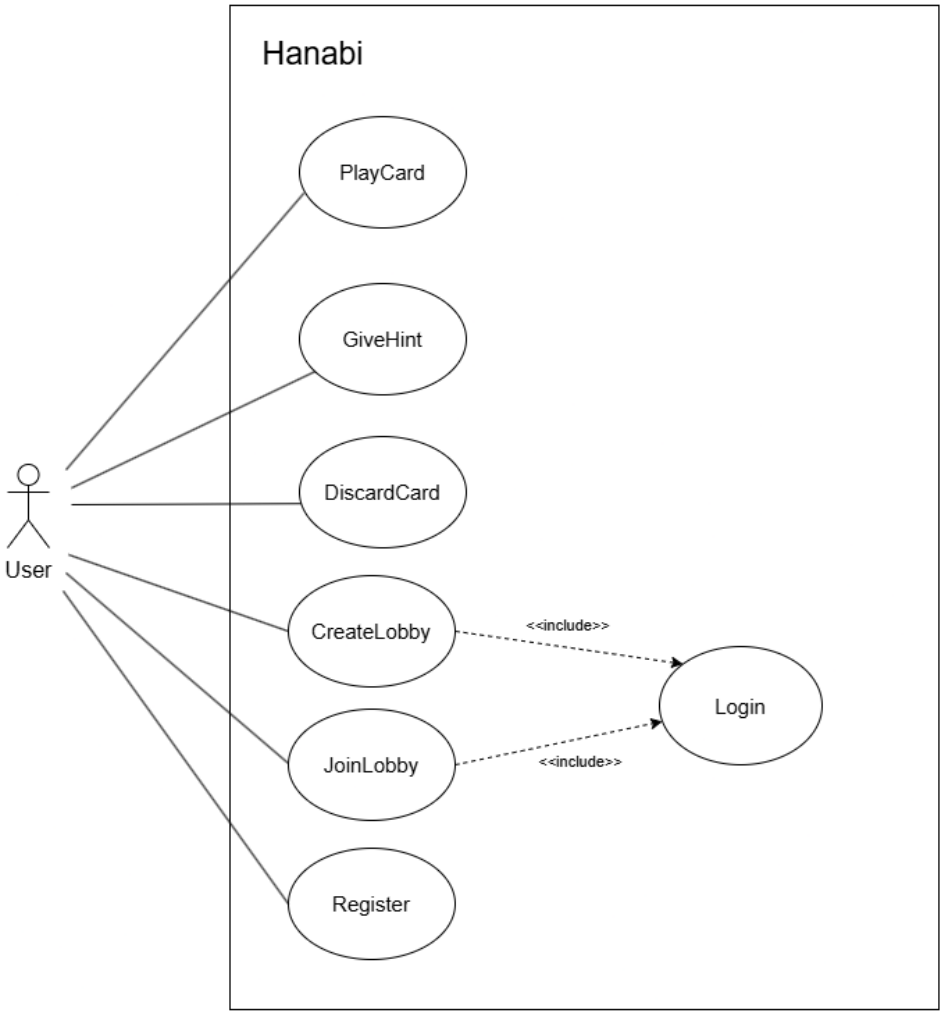

Figure 1: Use case diagram of the system

1.2 Reasons for distribution

The distributed nature of the system is inherent to the problem domain. The game is designed to be played by multiple users connecting from different machines, potentially located in geographically separated environments. Therefore, a centralized single-machine solution would not satisfy the project requirements. From a systems perspective, distribution is required for several reasons:

- **Geographically distributed environment:** Players connect from different devices over a network that may be unreliable, lossy, or subject to delays. The system must therefore tolerate message latency, disconnections, and partial failures.
- **Resource sharing:** The game state represents a shared resource accessed by multiple participants. Although gameplay is turn-based, the system must guarantee consistency of the replicated state and ensure that all clients observe the same authoritative version of the game.
- **Fault tolerance:** To avoid interrupting active game sessions, the server component can be replicated. Redis Sentinel monitors the Redis master and replica. If the master becomes unavailable, Sentinel can promote the replica to become the new master, allowing the backend to continue using Redis after failover.
- **Coordination and consistency:** Even though only one player performs an action per turn, the distributed environment introduces challenges such as message ordering, duplicated requests, and possible race conditions during failures. The system must therefore enforce a well-defined consistency model.

2 Requirements Elicitation and Analysis

This section defines the requirements for transforming the Hanabi implementation into a distributed systems project with explicit correctness, resilience, and evaluation goals. The focus is not only game functionality, but also distributed coordination, consistency, and failure handling.

2.1 Functional Requirements

- **FR1: Lobby and Session Management**
  The system shall allow multiple users to connect, enter a lobby, and start a game session once the configured player threshold is reached.
  - **Acceptance Criterion:** Users can connect to the server, see a lobby with waiting players, and start a game when enough players have joined.
- **FR2: Player Actions**
  The system shall allow players to perform valid game actions (e.g. give hints, play cards, discard cards) and update the game state accordingly.
  - **Acceptance Criterion:** Players can perform actions based on the game rules and their teammates' cards, and the game state updates correctly based on those actions
- **FR3: Player Identification**
  The system shall assign a unique identifier to each player to manage game sessions and actions.
  - **Acceptance Criterion:** Each player receives a unique ID upon connecting, and the system uses this ID to track their actions and game state.
- **FR4: Game Conclusion**
  The system shall determine when a game session has concluded based on the game rules and notify all players of the outcome.
  - **Acceptance Criterion:** When the game ends (e.g. there are no cards left or the number of misfires reaches 3), all clients are notified with the final game outcome and overall score.
- **FR5: Player Disconnection Handling**
  The system shall handle player disconnections gracefully, allowing the game to continue if possible.
  - **Acceptance Criterion:** If a player disconnects, the system allows the remaining players to continue the game, and the disconnected player can reconnect without losing their progress.
- **FR6: Security**
  The system shall implement basic security measures to protect player data and prevent unauthorized access.

- **Acceptance Criterion:** Player data is stored securely, and the system prevents unauthorized access to game sessions and player information through authentication.

- **FR7: User Interface**
  The system shall provide a user-friendly interface for players to interact with the game.
  - **Acceptance Criterion:** Players can easily understand how to perform actions and view the game state through the interface, with clear visual cues for game elements and actions.

2.2 Non-Functional Requirements

- **NFR1: Performance**
  The system shall respond to player actions and update the game state within an acceptable time frame to ensure a smooth gaming experience.
  - **Acceptance Criterion:** The system processes player actions and updates the game state within 500ms, ensuring that players experience minimal lag during gameplay.

- **NFR2: Fault Tolerance**
  The system shall be able to recover from server failures without losing game state or significantly disrupting the player experience.
  - **Acceptance Criterion:** In the event of a server failure, the backup server can take over within a reasonable time frame (e.g., under 5 seconds) and restore the game state, allowing players to continue without losing progress.

- **NFR3: Consistency**
  The system shall ensure that all clients have a consistent view of the game state at all times.
  - **Acceptance Criterion:** After any player action, all clients receive the updated game state, and it reflects the same information across all clients without discrepancies.

- **NFR4: Horizontal Scalability**
  The system must support multiple server instances to handle an increasing number of concurrent game sessions and players.
  - **Acceptance Criterion:** The system can run multiple server instances that can manage separate game sessions, and the load is distributed effectively without performance degradation as the number of players increases.

2.3 Implementation Requirements

- **IR1: Technology Stack**
  The system shall be implemented using Python for the server-side logic and React for the client-side interface.
  - **Acceptance Criterion:** The server is implemented in Python, the client interface is built with React.
- **IR2: Communication Protocol**
  The system shall use WebSockets for real-time communication between the server and clients.
  - **Acceptance Criterion:** The server and clients communicate using WebSockets, allowing for real-time updates of the game state and player actions.
- **IR3: Data Storage**
  The system shall use a Redis database to store game state and player information.
  - **Acceptance Criterion:** Game state and player information are stored in Redis, and the system can retrieve and update this information as needed during gameplay.
- **IR4: Containerization**
  The system shall be containerized using Docker to facilitate deployment and scalability.
  - **Acceptance Criterion:** The application can be built and run using Docker, with a provided Dockerfile and docker-compose configuration for easy deployment.

2.4 Relevant Distributed System Features

The project is designed as a distributed system and therefore takes into account the main characteristics and trade-offs typical of distributed architectures.

- **Fault Tolerance and Failover**:
  The system reduces single points of failure at both the application and storage layers. At the application layer, the servers are stateless, meaning that no persistent game information is tied to a specific server instance. This means that in case of a container problem reported to the matchmaking service, the faulty container could be replaced seamlessly. At the storage layer, game state is persisted in Redis, which is configured with master-replica replication and monitored by Redis Sentinel. If the primary Redis node fails, Sentinel can automatically promote a replica and redirect the system to the new master. During infrastructure-level failures, the system prioritizes preserving a consistent game state over maintaining uninterrupted availability.

- **Scalability**:
  The system supports a limited form of horizontal scalability at the game-server level. When a lobby becomes full, the main backend creates a dedicated `hanabi-game-xxx` container for that game. Therefore, multiple games can run in separate containers instead of sharing a single game process. This design helps isolate game sessions and allows the workload of active games to be distributed across several container instances. However, the matchmaking service and Redis remain centralized components, so the scalability of the current implementation is still bounded by the resources of the main backend, Redis, and the host machine.

- **Security and Trust**:
  The architecture includes user authentication mechanisms. Authentication ensures that a user is correctly associated with their identity and prevents impersonation. Secure password storage (hashing and salting) can be incorporated if persistent user accounts are introduced.

- **Resource Sharing and Consistency**:
  The shared game state is centrally stored and accessed by all server instances, ensuring a single authoritative source of truth. Each game action is processed atomically and results in an updated state that is broadcast to all players in the session. Since the game is turn-based, events have a natural total ordering, simplifying consistency management. Persisting the game state enables recovery after server failures and guarantees that all clients observe a consistent view of the system.

- **Performance and Concurrency**:
  The system can handle multiple game sessions concurrently across different server instances. Within each game session, operations are inherently serialized due to the turn-based nature of the application, reducing concurrency conflicts. WebSocket connections allow efficient bidirectional communication without repeated connection overhead. Game state messages are transmitted in compact JSON format; therefore, bandwidth usage is modest under normal operating conditions. No strict real-time latency constraints are imposed, but the architecture aims to maintain responsive interaction for players under moderate load.

3 Design

3.1 Architecture

The distributed system follows a three-tier, layered architecture. An intuitive representation is shown in Figure 2. This approach ensures a clear separation of concerns and responsibilities.

- The presentation tier is composed of the client user interface. The player has a consistent view of the current game state and can interact with the web UI to send commands to the game server.
- The logic tier is composed of the game server. It is responsible for executing commands sent from the client in a precise format.
- The data tier is composed of the database. Its role is to store important data such as user and game information.

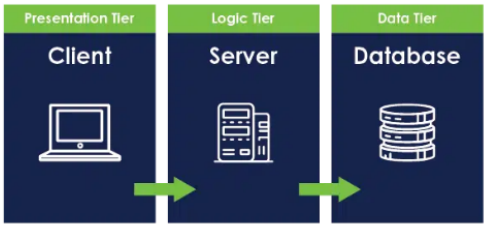

Figure 2: Overall architecture of the distributed system

In order to manage the state consistently, the server process singularly takes up the requests of the client processes and executes each update of the state. After each update, the new state is broadcasted to all the players.

3.2 Infrastructure

As anticipated in subsection 3.1, the distributed system is composed by three components:

- Client process that runs on the player's machine. It handles initial communication with the lobby, then continues communicating with the spawned game server process.
- Server process that is dynamically spawned from the lobby. It manages communication with the client by receiving commands and sending back events.
- Database that needs to store game information hashed per a unique identifier. Access to the database must be managed carefully as writes can still happen concurrently, on different data. This component is divided in:
  - Master as the single source of truth.
  - Replica contains a replicated copy of the data. This ensures fault tolerance in case of crashes on the master.

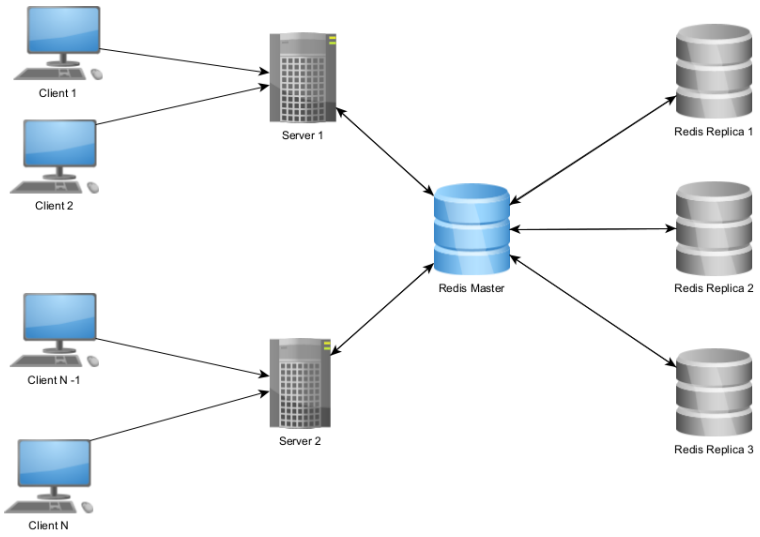

Figure 3: Overall architecture of the distributed system

The servers and the database are on the same Docker network and communicate internally using Docker service names or hostnames, which Docker resolves to container IP addresses. The clients, on the other hand, can be geographically distributed since they only need to be able to reach the lobby through the Internet, so the geographic scale of the system can be much larger than the infrastructure's.

3.3 Modelling

The game server uses a layered internal architecture, where each layer has a different responsibility.

- Domain layer contains all the game logic and is responsible for applying the game rules and modifying the game state accordingly.
- Application layer executes commands coming from the client and addresses the operation to the correct use case.
- Presentation layer directly communicates with the client and manages communications.

Game actions are exposed through an RPC-like command interface. For example, a client sends a command such as `game.play_card`, and the backend dispatches it to the corresponding application method such as `playCard(username, cardIndex)`. The following subsections contain a UML class diagram for each architectural level. Not all classes, methods, and properties have been represented, in order to focus more on the overall behaviour of the system. The full code source base can be found on the repository `https://github.com/ajdonker/Hanabi_DS`.

3.3.1 Domain

The core class is Game, which maintains all the information of an ongoing game. Its responsibility is to execute a player move through applying internal game rules that update its internal state. Before applying the move the player has requested (playing a card, giving a hint, discarding a card), it is first checked whether it is the player's turn. After applying the move, unless a game-over situation occurs, the turn is changed to another player.
Board class is a representation of the board state: contains all the information of cards played on the table, misfires and tokens. It has been separated from Game class to better address roles and responsibilities and facilitate better testing and debugging.
The domain contains two record-like classes: Player and User.
Player is a user that joined or created a lobby and is part of a currently running game. User just contains personal information that is used for authenticating the connected client to the system.

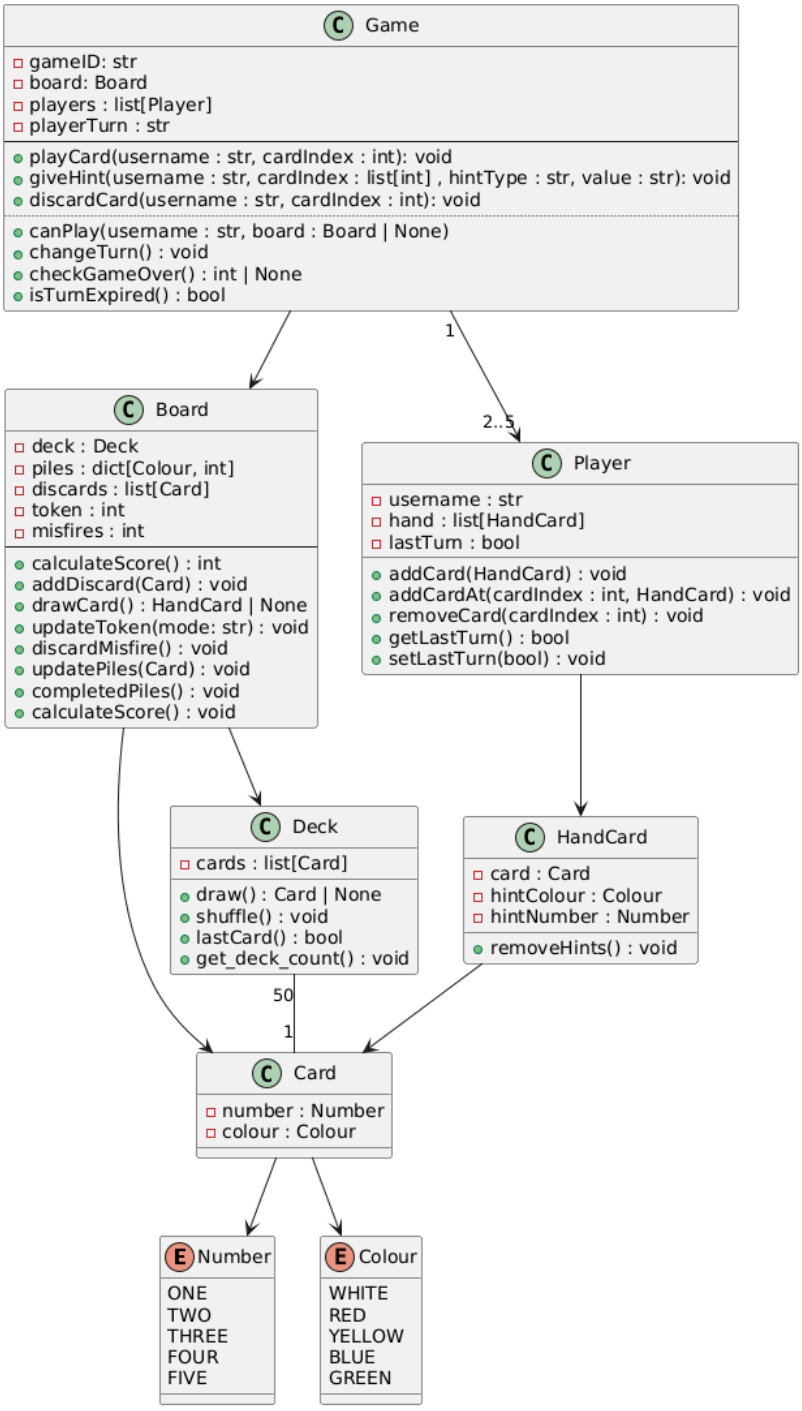

Figure 4: UML domain class

3.3.2 Application

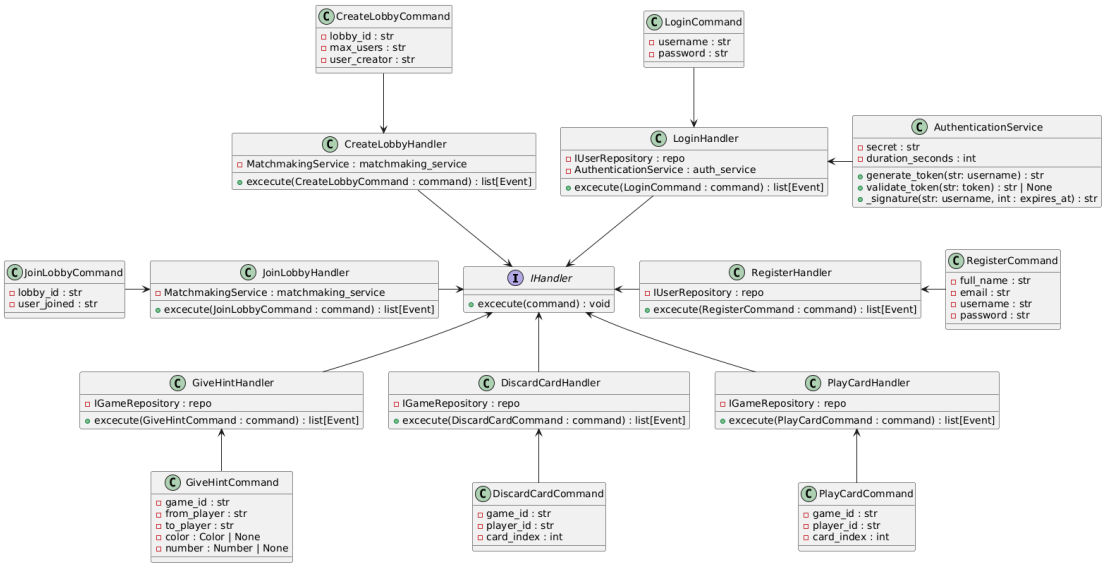

Figure 5: UML application class

The class diagram above shows how commands are managed within the system. Each command is a use case of the system, so an action the user can do. This structure promotes decoupling between the definition of the action and its execution logic, facilitating maintainability and extensibility. The IHandler interface defines the standard contract for all business logic handlers. Every handler class implements an `execute()` method that receives a command and returns a list of events generated by the operation. Commands are pure data transport objects (DTOs) that encapsulate all the parameters required to execute a specific action (e.g. `PlayCardCommand`, `RegisterCommand`, `JoinLobbyCommand`).

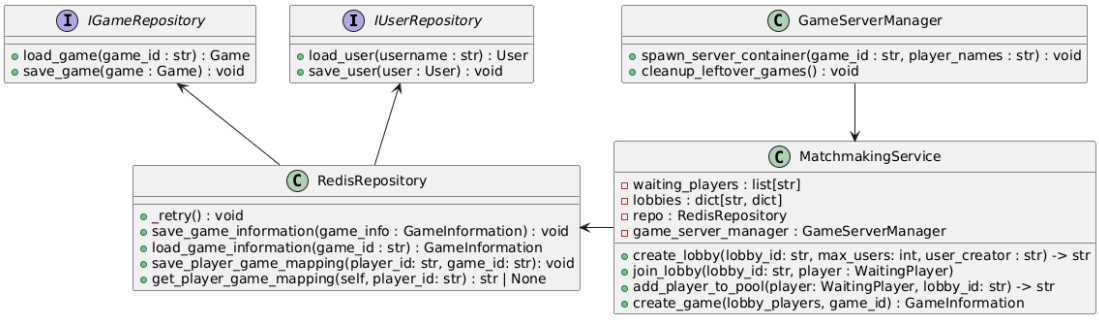

Figure 6: UML application class

Each handler class is responsible for a single unit of application logic and interact with a respective repositories (such as `IGameRepository` or `IUserRepository`) or external services (`MatchmakingService`) to persist state changes or query the database. `IGameRepository` interface is used just for reading and persisting game state, while `IUserRepository` role is reading and persisting user information. `RedisRepository` implements both interfaces but also provides methods for loading and storing information about the container spawned and player-game mapping, which is useful for the reconnection protocol. `MatchmakingService` aims at managing a matchmaking system. Its role is to create a lobby and dispatch a list of waiting player in their corresponding lobby. Once a lobby is full, the game is automatically created and a `GameServerManager` class takes care of spawning a corresponding container.

3.3.3 Presentation

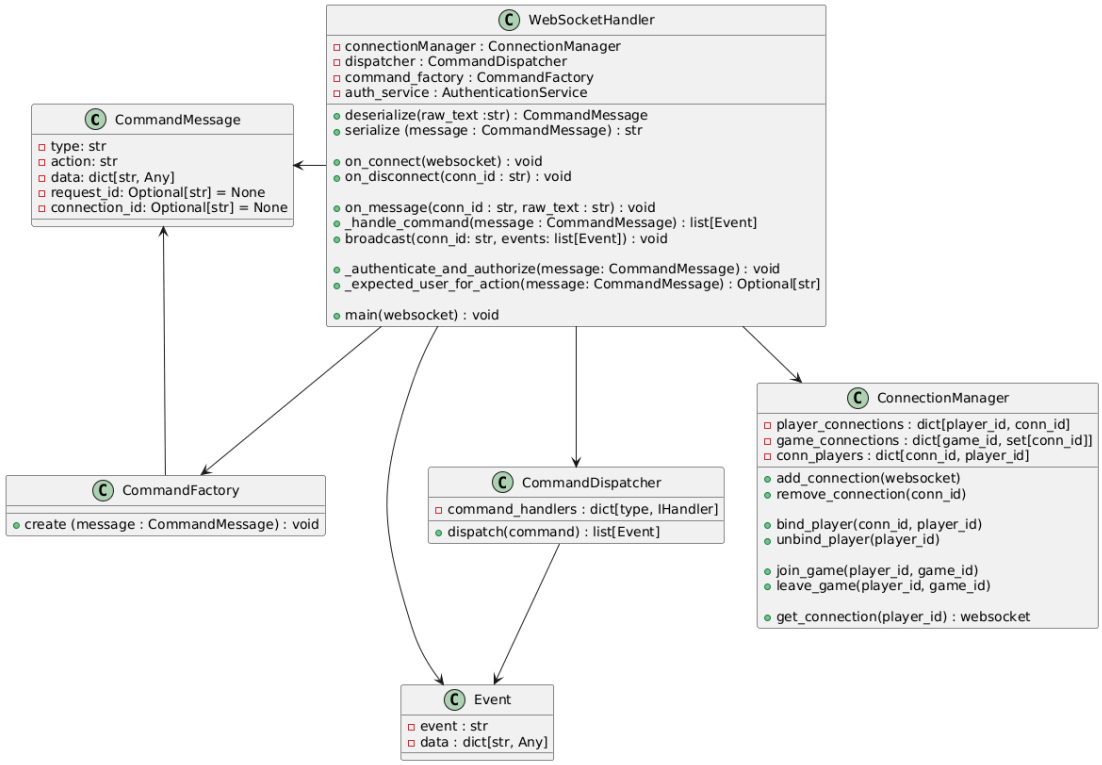

Figure 7: UML presentation class

Presentation layer is responsible for handling real-time communication and the translation of network messages into executable application logic. The design follows the Factory and Dispatcher patterns to ensure a clean separation between the transport layer (WebSockets) and the business logic. Core components are CommandMessage and Event dataclasses, which will be better explained in the next section.

- `CommandMessage` acts as a generic Data Transfer Object (DTO) that captures incoming raw data, including the action type and its payload.
- `Event` represents the system's reactive output, encapsulating the state changes that need to be communicated back to the clients.

WebSocketHandler is the central entry point for the real-time communication layer: it orchestrates the lifecycle of a connection (connect, message, disconnect) and it is responsible for:

- Serialization/Deserialization: Converting raw socket strings into structured CommandMessage objects.

- Orchestration: Coordinating with the Factory to create commands and the Dispatcher to execute them.
- Broadcasting: Sending generated Events back to the appropriate clients.

CommandFactory acts as a creational layer that parses the CommandMessage and instantiates the specific Command object required by the application, while CommandDispatcher maintains a registry of handlers. It receives the instantiated command and routes it to the correct IHandler, returning a list of resulting Events. Finally, ConnectionManager is a specialized component that manages the state of active network connections. It maintains bidirectional mappings between players, connections, and games.

3.4 Events and commands

At architectural level, the system distinguishes between two types of messages.

- **CommandMessage**: represents a request sent from the client to the server to execute an action.
- **Event**: represents a notification generated by the backend to inform the clients of an incoming change to the state of the game.

In the backend, both structures are implemented using Python `dataclass`, so as to define the content of the messages in a clear and typed manner.

3.4.1 CommandMessage

As shown in the UML class diagram in figure subsubsection 3.3.3, a command contains:

- the message type;
- the requested action;
- a payload containing the data required for the operation;
- token for authentication and authorization purposes;
- any additional information such as the request or connection identifier.

Commands are used to represent actions requested by players, such as entering the lobby, starting a game or executing a move. The `type` field follows the convention:

`entity.action`

where the first part identifies the domain or entity involved, while the second part specifies the action to be performed. This naming convention provides a clear and extensible way to categorize commands and events, simplifying both message routing and system maintenance.

Examples of commands

- `player.login`
  A user tries to log in to the system.
- `player.register`
  A user tries to register in the system.
- `lobby.create`
  A user creates a new lobby.
- `lobby.join`
  A user joins an existing lobby.
- `game.play_card`
  A player plays a card from his hand.
- `game.discard_card`
  A player discards a card from his hand.
- `game.give_hint`
  A player gives a hint to another player.

3.4.2 Event

Events, on the other hand, represent changes produced by the system following the processing of commands. Each event contains:

- the name of the event generated;
- a payload containing the associated information.

Events are sent from the backend to clients to synchronise the game state and update the user interface in real time.

Examples of events

- `login_success`
  A user logs in successfully.
- `player_reconnected`
  A disconnected user reconnects to an ongoing match.
- `registration_success`
  A user registers successfully.
- `lobby_created`
  A new lobby is created.

- `match_found`
  A lobby reaches the maximum number of users, so the game starts.
- `turn_change`
  The turn changes.
- `card_correct`
  The player plays a correct card.
- `card_wrong`
  The player plays a wrong card.
- `card_discarded`
  The player discards one card from his hand.
- `card_drawn`
  The player draws a new card from the deck.
- `hint_given`
  The player gives a hint to another player.
- `hint_failed`
  The player fails to give a hint to another player (e.g., wrong turn).
- `misfire`
  A player mistake raises one misfire.
- `game_over`
  The game ends.
- `error`
  A generic error occurs (e.g., unknown command).

3.5 Interaction

The client and the server interact with a request-response interaction protocol. The client always has the initiative in the communication and sends well-formatted commands to the server. The server is responsible for validating the command, applying the game rules and updating the current game state. After that, one or multiple events are sent back to the client. The only case in which the server sends events to the client unprompted is when the current player's turn is expired. In this case the game forces a change of turn to the next player. For each command a sequence diagram representation is shown. In the case of a game command, all the three moves are in the same diagram for simplicity.

- Registration sequence diagram is shown at Figure 8
- Login sequence diagram is shown at Figure 9
- Game sequence diagram is shown at Figure 10

- Lobby creation and joining sequence diagram is shown at Figure 11

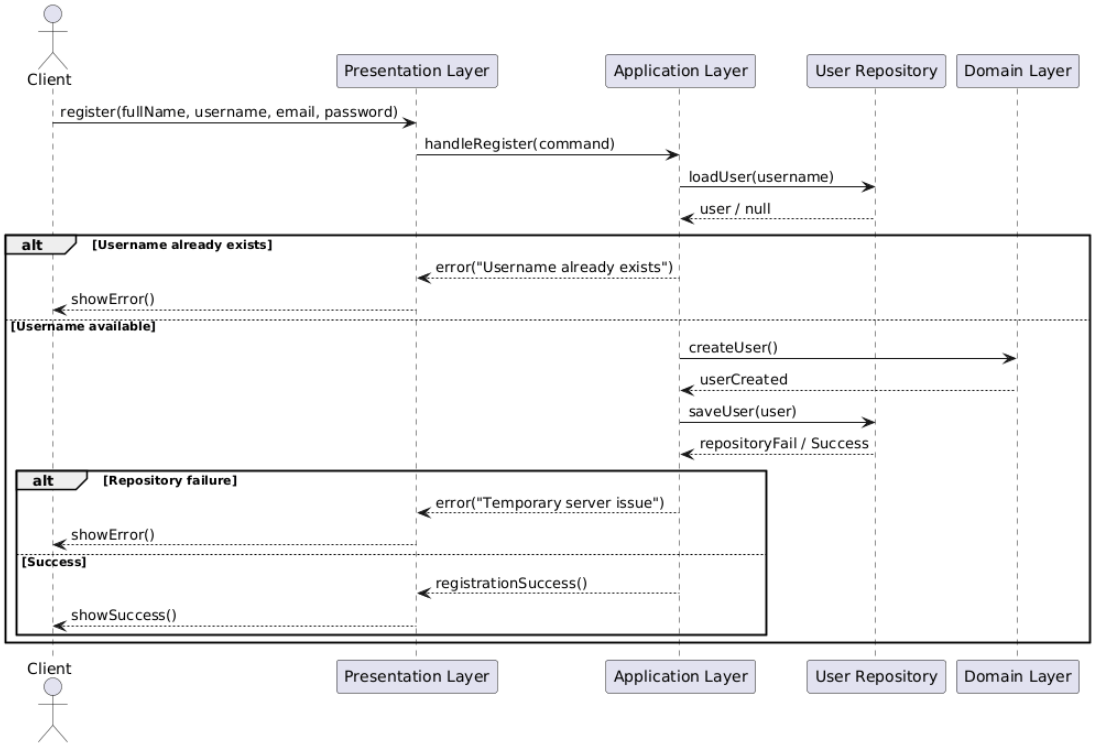

Figure 8: User registration sequence diagram

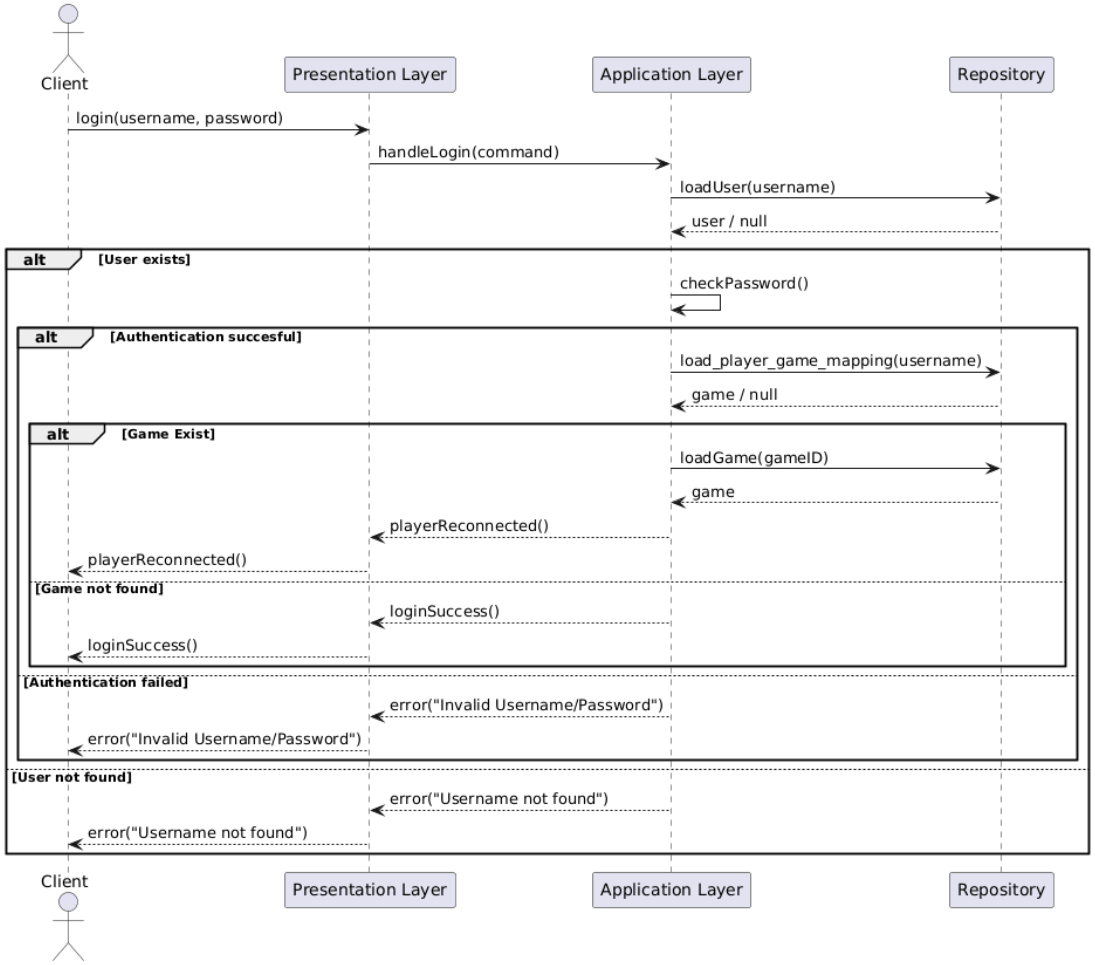

Figure 9: User login sequence diagram

In the diagram above it's interesting to notice the reconnection mechanism: once a user successfully logs in, the system check wheter a game is mapped to the username. In case it exists, the game is loaded and the player is automatically reconnected to the ongoing match. Otherwise, it's simply redirected to the lobby view.

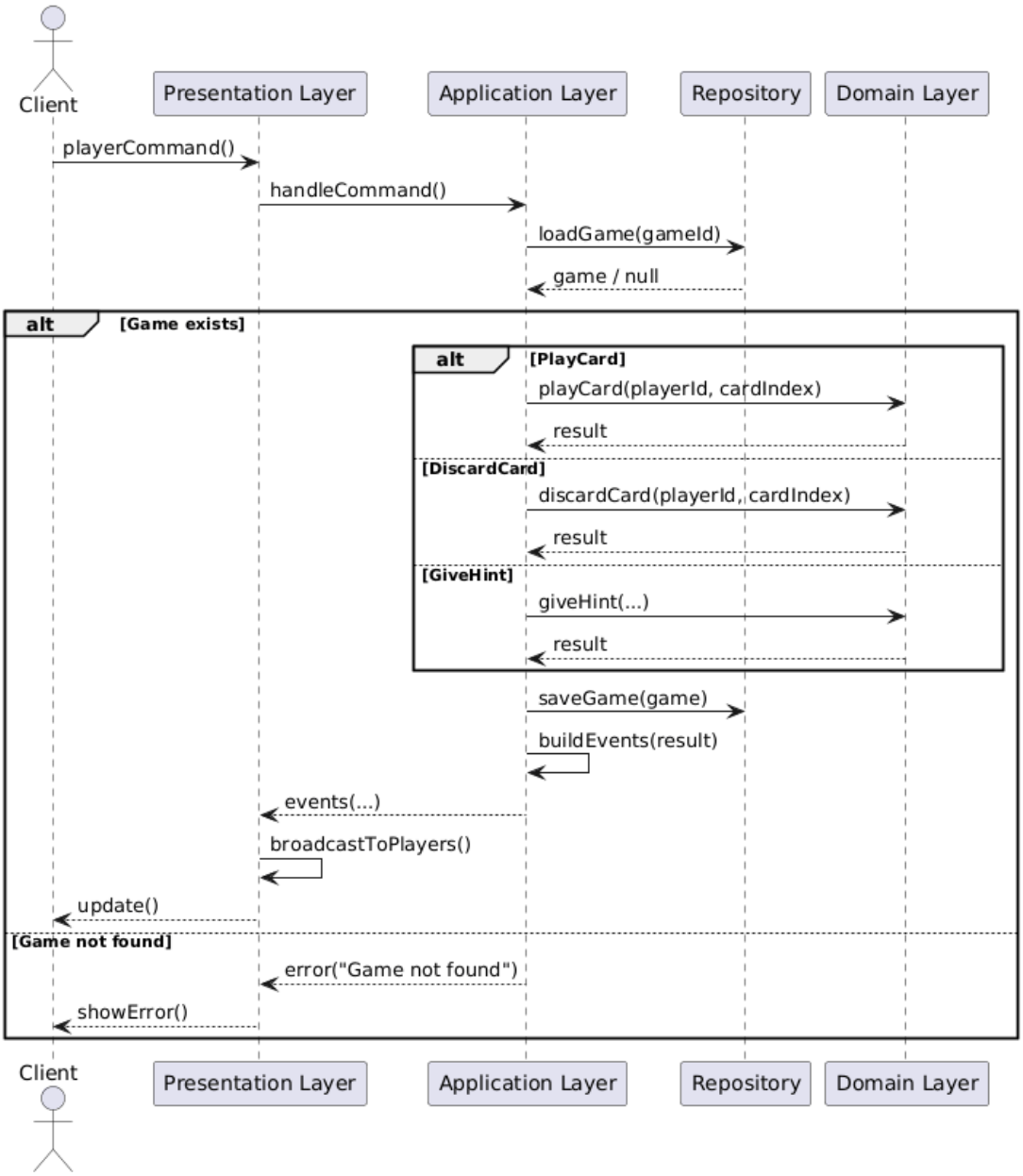

Figure 10: User game move sequence diagram

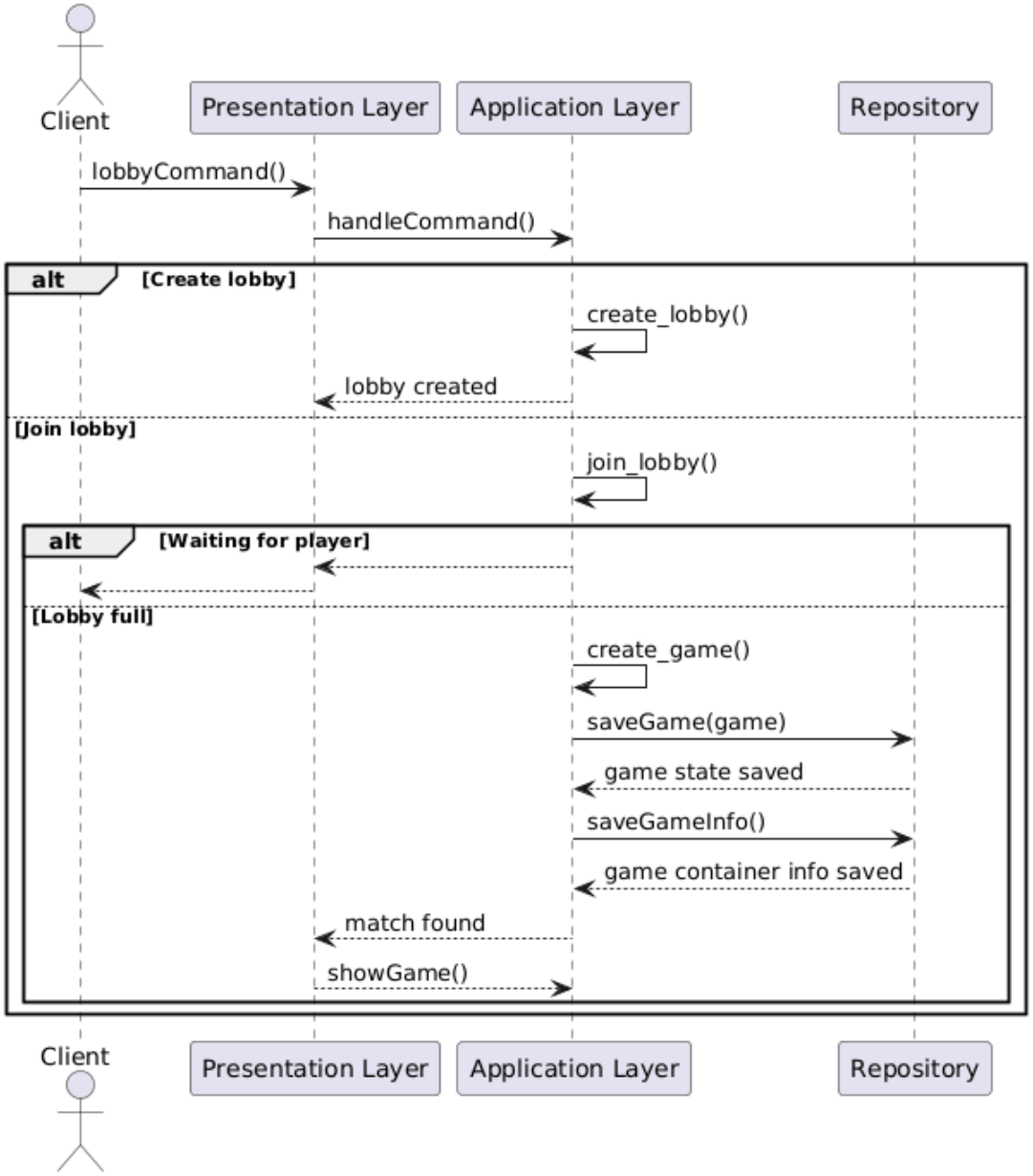

Figure 11: User create/join lobby sequence diagram

3.6 Behaviour

The game server is meant to be a stateless component. The information about each game is stored at the data tier. That choice is due to the fact that if Game server crashes through a valid game, information isn't lost in case of reconnection. Once the game is created, the game server is in charge of waiting for commands from the client

side and validating the move. If a move is valid, it is applied and the game state changes accordingly. After the game state update, it is always checked if a gameover situation occurred to possibly end the match. The overall situation is shown in Figure 12.

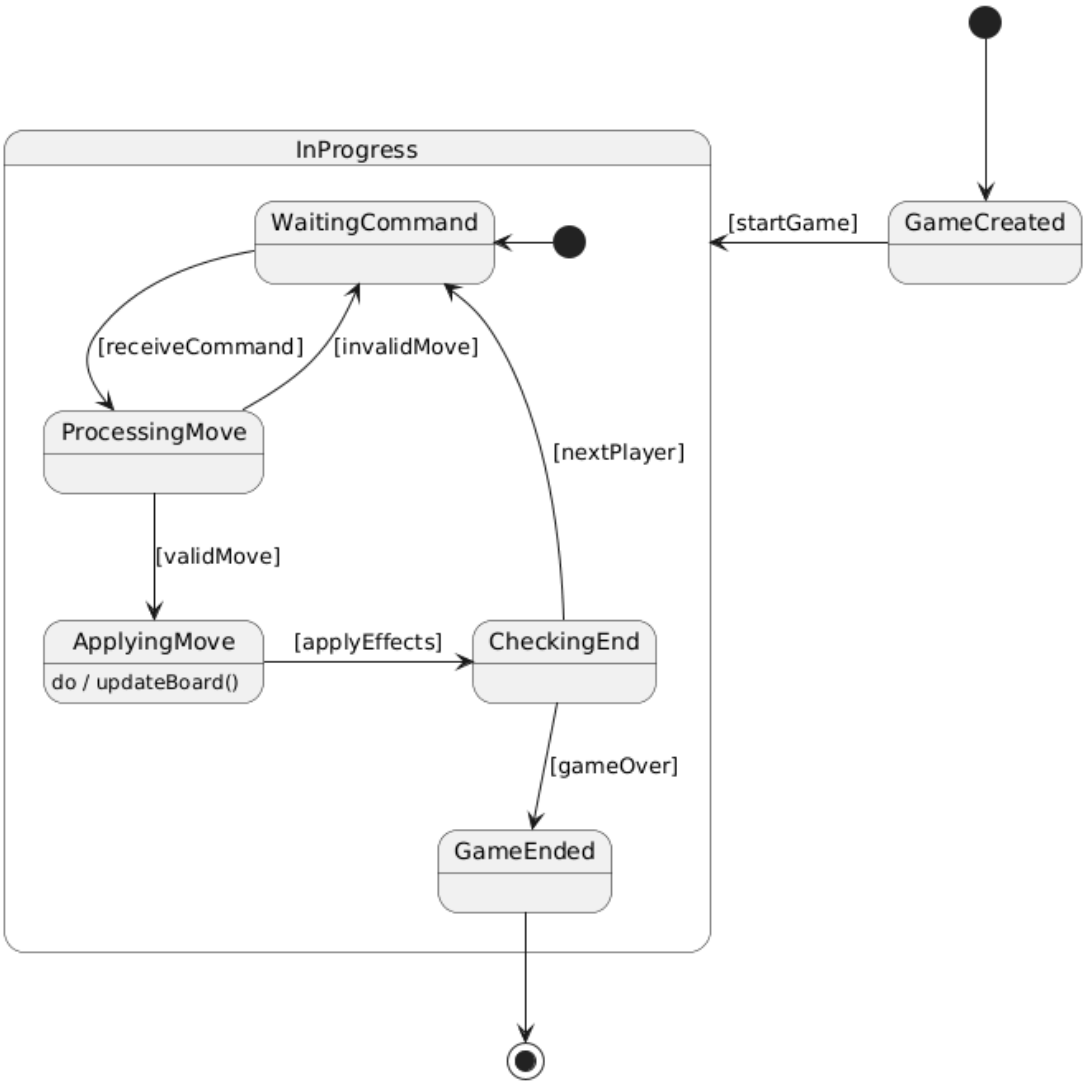

Figure 12: Game lifecycle state diagram

3.7 Data and Consistency Issues

The database layer is divided into two repositories: one responsible for user data and one responsible for game data. Both repositories rely on Redis key-value storage, where data is accessed through hash-table-based structures. This allows common operations such as insertion, lookup, and deletion to be performed in constant average time.

3.8 Fault-Tolerance

The current architecture uses WebSockets for client-server communication, so each client maintains a persistent bidirectional connection with the server. This allows the server to immediately propagate game events to the affected players whenever the game state changes.
Connection failures are handled by the WebSocket connection manager. If a client disconnects, its connection is removed from the active connection registry, so the server no longer attempts to send game updates through an invalid connection.

A *timeout* mechanism is implemented at the game logic level: its purpose is to prevent the game from remaining blocked when the player whose turn it is becomes inactive, does not perform an action in time or crashes.

A *retry* mechanism is implemented between the application server and Redis. Since Redis is managed using Sentinel, a failure of the current master may require the application to rediscover the newly promoted master. In that case, the repository retries the database operation after master rediscovery, until the configured retry limit is reached. The system does not rely on client-side polling or repeated command retries during normal gameplay. Instead, WebSockets are used as the main event propagation mechanism, while Redis retry logic is reserved for infrastructure-level failures in the storage layer.

The client and server structure the messages so that all communication proceeds in a predictable way and messages are always well-formed. If a client disconnects during an active game, it can reconnect to the corresponding game server using the game information stored by the matchmaker. After reconnection, the server sends the latest game state to the client, allowing the user interface to recover from the temporary connection loss.

At the storage level, Redis Sentinel monitors the Redis master and replicas. If the master fails, Sentinel coordinates the failover process and promotes one of the available replicas to become the new master. If the old master later rejoins the system, it is reconfigured as a replica of the newly promoted master. During this process, the application may temporarily lose access to the storage layer, but the retry mechanism allows it to rediscover the current master and repeat the failed operation.

3.9 Availability

Availability in the system is supported through a combination of stateless application servers, Redis replication, Sentinel-based failover, and controlled recovery mechanisms. The system is designed to remain operational when individual components fail, although it prioritizes consistency of the game state over continuous availability during severe infrastructure failures.

3.9.1 Caching

The client keeps a local copy of the most recently received game state. This copy is used only as a temporary fallback for the user interface, for example when a short disconnection occurs or when the client is waiting for an updated state from the server.

The cached state is not treated as authoritative and cannot be used to continue the game independently from the server. Its purpose is therefore to preserve UI consistency, not to replace the persistent game state stored in Redis.

3.9.2 Load balancing

The system provides load distribution at the game-server level. Instead of running all matches inside a single backend process, the matchmaker creates separate game server instances for active games. This distributes gameplay execution across multiple server processes and prevents one game from blocking the execution of another.

Since the application servers are stateless and the persistent state is stored in Redis, requests do not depend on a specific backend instance. This makes the architecture compatible with load balancing in front of the application layer, for example through a reverse proxy such as NGINX. In that case, incoming client connections can be distributed across available backend instances, while Redis remains the shared storage layer.

3.9.3 Network Partition

In case of a network partition, the system prioritizes consistency over availability. Since Hanabi is a turn-based game, allowing different partitions to independently update the game state could lead to conflicting moves, duplicated turns, or an invalid game history. Therefore, the system should avoid accepting updates unless the authoritative Redis master can be reached.

Redis Sentinel monitors the Redis master and replicas. If the current master becomes unreachable and Sentinel can establish the required quorum, a replica may be promoted to become the new master. The application then rediscovers the current master and retries failed Redis operations. During the failover interval, some operations may temporarily fail or be delayed.

If a game server becomes separated from Redis or from the clients, it cannot safely continue the game using only local information. Connected clients may temporarily keep displaying the last received state, but new game actions should only be accepted once the server can again access the authoritative state in Redis.

3.10 Security

- Authentication:
  Users authenticate with a username and password. Registration stores the user information in Redis, while login verifies the submitted credentials on the server side. Passwords are stored as SHA-256 hashes rather than as plaintext. When login succeeds, the backend generates a token and returns it in the login event. The token is a Base64-encoded JSON object containing the username, an expiration timestamp, and a SHA-256 signature generated with a secret configured through an environment variable. The frontend stores this token and sends it with every WebSocket command.

On the backend, login and registration are public commands. All other WebSocket commands pass through the authentication check in the WebSocket handler. The handler decodes the token, verifies that it is not expired, recomputes the signature, and rejects the command if the token is missing or invalid. This allows the server to authenticate normal game requests without loading the user from Redis each time.

- Authorization:
  The system does not implement role-based authorization, since all users have the same role of player. Instead, authorization is enforced by comparing the authenticated username with the player name contained in the command. For example, a user can only send a play or discard command for their own player name, and a hint command must come from the authenticated user. Additional checks are still performed by the application and domain logic: the backend verifies whether a player is already in an unfinished game, whether a lobby exists, whether the player belongs to the game, and whether it is actually that player's turn before applying an action.

4 Implementation

4.1 Network protocols

The main communication protocol between the browser and the backend is WebSocket. WebSocket was chosen because the game requires bidirectional communication: the browser sends commands to the server, while the server must also push events to all players in the same game. This is used for actions such as playing a card, discarding a card, giving a hint, notifying turn changes, and announcing the end of the game.

The system also uses HTTP for auxiliary operations that do not require a persistent bidirectional channel. For example, when a game container fails and the browser loses its WebSocket connection, the frontend can send an HTTP request to the main backend to resolve the current address of the game server. The backend can then recreate the failed game container and return the new host and port to the client.

Communication between backend components, Redis, Redis Sentinel, and dynamically spawned game containers happens through the Docker network defined in the Compose configuration. This allows containers to reach each other by service name, for example when the backend connects to Redis Sentinel.

4.2 Data format and representation

In-transit data is represented using JSON dictionaries. The client sends command messages containing an action name, a request identifier, and a data payload. The backend replies with batches of events, also represented as JSON objects. This structure makes the protocol simple to inspect and easy to handle in both Python and TypeScript.

On the backend, incoming WebSocket messages are decoded and transformed into command objects by the presentation layer. The command dispatcher then forwards each command to the appropriate handler. After the command is executed, the server returns events such as successful card plays, discarded cards, hints, turn changes, errors, or game-over notifications.

Game state is also serialized as JSON before being stored in Redis or sent to the frontend. The serialization logic converts domain objects such as games, players, cards, piles, discard piles, tokens, and misfires into dictionaries. When a game server loads a game from Redis, the inverse conversion reconstructs the corresponding domain objects.

4.3 Database access

Redis is used as a key-value database. Instead of SQL-style queries, the system accesses data through structured keys. The most important stored data includes user accounts, game states, game metadata, and mappings between players and their active games.

Each game server can load and update the game it is responsible for by using the game identifier. Game metadata stores information such as the game container name, players, host, and port. Player-game mappings are used to prevent a player from joining multiple unfinished games and to support reconnection after login.

Redis Sentinel is used by the backend to discover the current Redis master. This avoids depending permanently on a single Redis master address and supports failover if the original master becomes unavailable.

4.4 Technological details

The backend is implemented in Python using FastAPI. FastAPI is used to expose both HTTP endpoints and WebSocket endpoints, while Uvicorn is used as the server. The backend follows a layered structure: the presentation layer handles WebSocket messages and command creation, the application layer dispatches commands to handlers, and the domain layer applies the actual Hanabi game rules.

Redis is used as the persistence layer, while Redis Sentinel provides master discovery and failover support. The backend accesses Redis through a repository class that hides the concrete key-value operations from the rest of the application.

Docker and Docker Compose are used for deployment. Compose starts the main backend, Redis master, Redis replica, and Sentinel. The backend also uses the Python Docker SDK to create separate game server containers dynamically when matchmaking starts a new game. If a game container fails, the main backend can recreate it, and the client can reconnect using the game state stored in Redis.

Testing is performed with pytest. The test suite covers domain logic, command handlers, matchmaking behavior, Redis repository operations, WebSocket message handling, and Docker container management.

The frontend is implemented with React, TypeScript, and Vite. React is used to build the registration page, login page, lobby page, waiting room, and game interface. TypeScript is used to implement the WebSocket client, which manages the connection to the backend, sends commands, and processes incoming events to update the user interface. Vite is used as the build tool to bundle the frontend code and serve it during development.

5 Validation

The system was validated at different levels. First, the application logic was tested separately from the communication protocol, so that the correctness of the game rules and matchmaking logic could be checked independently. Afterwards, integration and end-to-end tests were used to verify that the complete distributed system behaves according to the specification.

5.1 Automatic Testing

5.1.1 Unit Testing

Individual components were unit-tested in isolation. The game domain logic was tested through the `Game` object, focusing on the game actions that can be invoked through the RPC-like command interface. These tests check whether actions such as playing, discarding, drawing cards, and enforcing player turns behave correctly.

The matchmaking service was also unit-tested separately. These tests verify that lobbies are created correctly, players can join lobbies, active games are tracked, and finished or failed game containers are cleaned up to free resources.

The client-matchmaking communication channel was tested with respect to the main communication cases: connecting, sending messages, receiving responses, and handling timeouts. This ensures that the protocol behaves correctly even when clients do not respond immediately or when communication fails.

5.1.2 Integration Testing

Integration testing was used to verify the interaction between components. In particular, the matchmaking service was tested together with the game server management logic to ensure that a game container is created when enough players join a lobby.

The tests also check whether the system keeps the correct account of active games and whether dead or stopped containers are removed from the active game list. This is important because the system dynamically creates game server instances, so incorrect cleanup could lead to stale state or wasted memory.

Multiple server instances were also spawned concurrently in order to simulate a more production-like environment. This was used to check whether the system can support several active games at the same time.

5.1.3 End-to-End Testing

End-to-end testing was performed by running the system through Docker Compose in a production-like environment. This includes the backend server, Redis master, Redis replica, Redis Sentinel, and dynamically created game server containers.

The goal of these tests was to verify the complete flow from the client perspective:

- a client connects to the backend;

- a player creates or joins a lobby;
- the matchmaking service detects when enough players are present;
- a new game server container is created;
- clients connect to the game server;
- players exchange game actions through the WebSocket protocol;
- game state updates are propagated to the connected players.

This type of testing is necessary because some behavior cannot be fully validated with unit tests only. For example, container creation, WebSocket communication, Redis persistence, and Sentinel failover require multiple components to run together.

5.1.4 Running the Tests

The automatic tests can be executed from the backend directory using:

```
pytest
```

or, for a specific test file:

```
pytest tests/<test_file>.py
```

The rationale behind the tests is to check both normal behavior and corner cases, such as wrong turns, failed containers, timeouts, disconnected clients, and Redis failover. These tests are linked to the system requirements: matchmaking correctness, game state consistency, fault tolerance, and automatic deployment.

5.2 Acceptance Test

The manual tests included:

- starting the complete system using Docker Compose;
- connecting multiple clients;
- creating and joining lobbies;
- starting a game after matchmaking succeeds;
- performing game actions from different players;
- checking whether game state updates are received by all connected clients;
- observing the behavior of the system under multiple active connections;

- testing Redis Sentinel failover by stopping the Redis master and checking whether the replica is promoted.

These tests were in general performed manually because they involve real-time behavior that is difficult to fully capture with automated tests, such as perceived response time, WebSocket message propagation, and the user-visible behavior of multiple clients interacting with the same game.

6 Deployment

The system has been deployed using the versioning system Git and source code of the system is available on GitHub at the link:

`https://github.com/ajdonker/Hanabi_DS`

The system is deployed using Docker and Docker Compose. This makes it possible to start the backend, Redis infrastructure, and supporting services in a reproducible environment.

6.1 Requirements

To run the project from scratch, the following software should be installed:

- Docker;
- Docker Compose;
- Node.js 20.0.0
- Python3;
- python dependencies listed in `requirements.txt`.

The Python dependencies can be installed with:

```
pip install -r requirements.txt
```

6.2 Starting the System

6.2.1 Backend

The system can be started using:

```
docker compose up --build
```

Note that the following command has to be launched in the */web/backend* subdirectory. Otherwise, the command can be run with the explicit file path, for example:

```
docker compose -f web/backend/docker-compose.yml up --build
```

This command builds the required images and starts the containers defined in the Docker Compose file.

6.2.2 Frontend

Inside the `/web/frontend` subdirectory, these commands need to be launched:

```
npm install
npm run dev
```

6.3 Expected Outcome

After the deployment command is executed, the following components are expected to be running:

- the main backend server;
- a Redis master instance;
- a Redis replica instance;
- a Redis Sentinel instance;
- dynamically created game server containers when matches are found.

Redis configuration files for the master, replica, and Sentinel are included in the project. These configuration files allow Sentinel to monitor the Redis master and promote the replica if the master becomes unavailable.

6.4 Deployment Design

The deployment was engineered using containerization so that the system can be started in a production-like environment with a single command. Docker Compose defines the required services, their network configuration, and the Redis failover setup.

The backend connects to Redis through Sentinel, instead of depending only on a fixed Redis master. This allows the backend to discover the current Redis master after a failover. The game server manager can then dynamically create separate game containers when the matchmaking service starts a new game.

6.5 Screenshots

Below are some screenshots of the system.

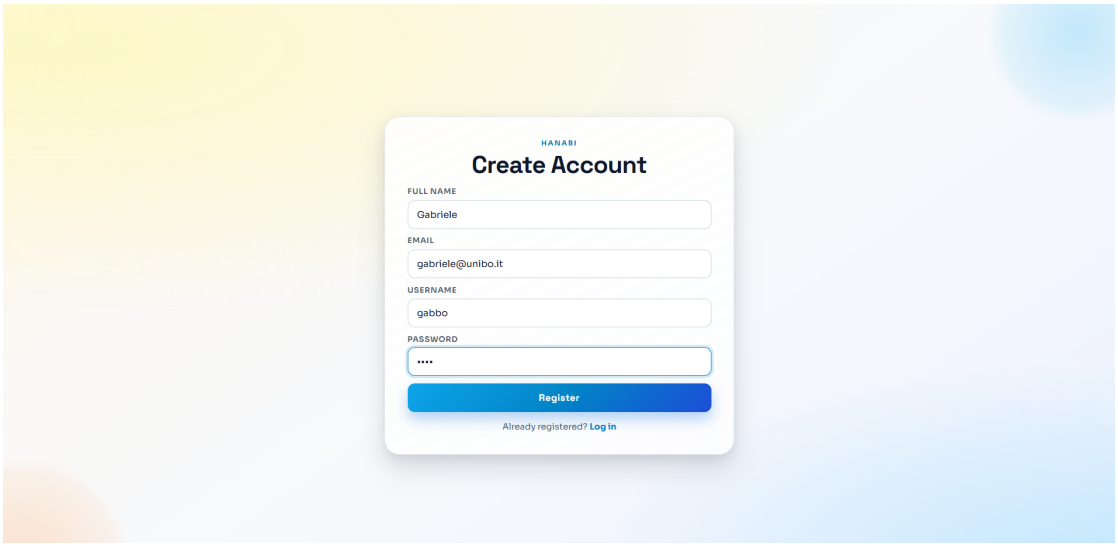

Figure 13: Registration view

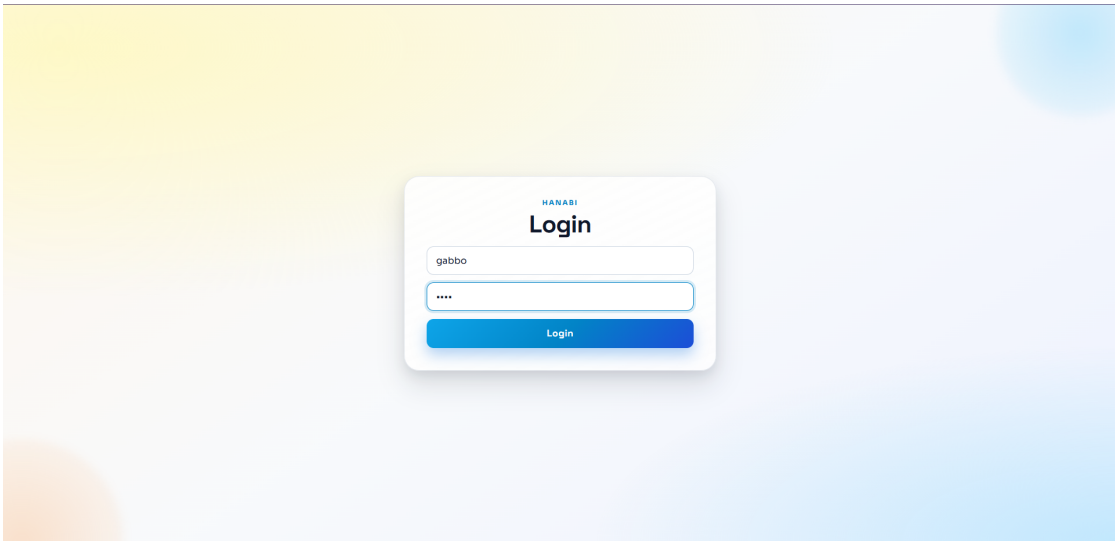

Figure 14: Login view

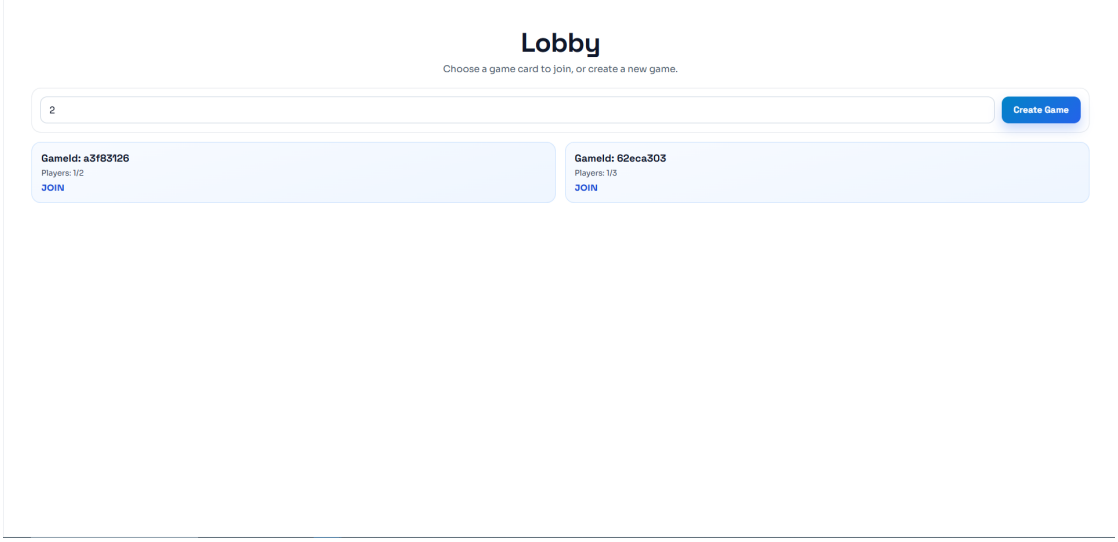

Figure 15: Lobby view

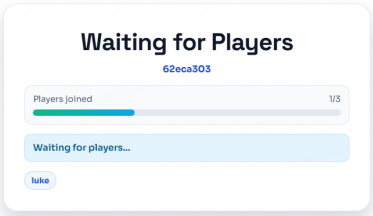

Figure 16: Waiting player view

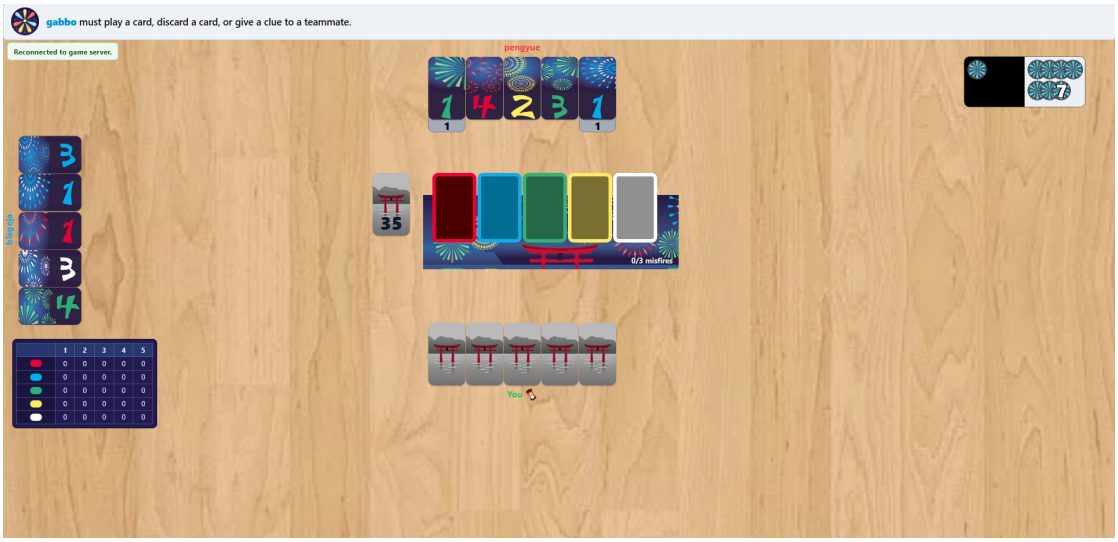

Figure 17: Game view

7 User Guide

The deployed application can be accessed at `http://hanabi.casa/register`, which has different images compared to the local version to avoid copyright issues.

To use the system locally, the user first starts the backend infrastructure using Docker Compose:

```
docker compose up --build
```

Each client connects to the backend server and can perform actions such as logging in, creating a lobby, joining a lobby, and playing a game once matchmaking succeeds. The expected usage flow is:

- open one browser tab for each client (make sure to be in incognito mode);
- type `http://localhost:3000/register` to go to registration page;
- once registration is succesful, the client is redirected to login page, which is reachable at the address `http://localhost:3000/login`
- if login is correct, the user is redirected to the lobby page `http://localhost:3000/lobby`, where they can create a lobby or join an existing one;
- join the lobby with another client;
- wait for the matchmaking service to create a game server;
- connect to the game server;
- play the game by sending game actions.

The expected outcome is that the backend starts successfully, Redis replication and Sentinel are active, and a new game server container is spawned when enough players join the same lobby.

8 Self-evaluation

8.1 Gabriele Santi

My role focused on designing the overall architecture system, including distributed system architecture and internal backend. I provided an implementation of parts in domain layer (game logic) and application layer, in addition to test regarding that parts. I also focused on writing several parts of the report, especially section section 3, including all the images and diagrams.

8.2 Blagoja Savevski

My role focused on implementing the backend, for example the command-handler flow responsible for connecting the different layers of the system. I also designed and ran unit tests for most of the core objects, with a particular focus on the domain layer, to ensure that the implementation conformed to the project specification.

Furthermore, I was responsible for prototyping earlier versions of the system that relied on communication protocols different from those used in the final version. I also worked extensively on the Docker orchestration of multiple containers, which enables service failover and contributes to the overall fault tolerance of the system.

8.3 Pengyue Xu

My role focused mainly on the frontend implementation and its integration with the distributed backend. I designed and implemented the pages for registration, login, lobby, the waiting room, and the game interface.

On the backend side, I implemented the server-side WebSocket flow for game actions and state synchronization. I contributed to lobby management, game-state retrieval, player-game cleanup, bug fixing, and the Docker-based recovery mechanism for failed game containers.

9 Future works

Although the final implementation provides the core functionality of the distributed Hanabi system, there are several possible improvements and extensions that could be added in future version.

One important future improvement would be the introduction of a load balancer, such as Nginx, in front of the backend services. In the current implementation, clients connect directly to the matchmaker or to the game server assigned to them. A load balancer would make this process more transparent and would allow incoming connections to be distributed across multiple backend instances. This would also make the system easier to scale horizontally, because new backend instances could be added without requiring changes on the client side, and could effectively decouple individual game servers from particular game instances.

The fault-tolerance mechanism could also be improved further. Redis Sentinel is used to detect master failures and promote a replica to master, which provides automatic failover for the persistent state layer. In future work, this could be combined with more extensive failure testing, such as deliberately stopping containers, simulating network partitions, or testing delayed messages between services. This would make it possible to evaluate the behavior of the system under more realistic distributed-system failures.

A further improvement would be adding better monitoring and logging. The system could expose metrics about active games, connected players, WebSocket connections, failed messages, Redis failovers, and container health. This would make debugging easier and would also provide a clearer understanding of how the system behaves under load.

Some features were not fully implemented due to time limitations. One such feature is a complete automated game-server lifecycle manager. In the current version, game containers can be created and cleaned up, but this could be extended with more advanced health checks, automatic restart policies, and graceful shutdown logic. This would ensure that inactive or failed game servers are removed safely, while active games are either preserved or migrated.

Now the server container and the game container contain the same codebase, but they execute different roles. In future work, it makes sense to split the server and game logic into separate images. This would allow for more specialized containers, for example with different resource limits or scaling policies. It would also make it possible to deploy them independently.

Finally, the testing strategy could be extended. The current implementation includes unit tests and some end-to-end testing, but future work could include stress testing with many concurrent players, automated failover tests, and integration tests for WebSocket reconnection. These tests would provide stronger evidence that the system remains correct and available under realistic conditions.

Overall, the project provides a functional distributed architecture, but future work could focus on making the system more production-ready by improving scalability, observability, security, automatic recovery, and fault-tolerance testing.