Final Report

Kentpayeva Madina
madina.kentpayeva@studio.unibo.it

June 2025

1 Concept

This project consists of developing a **distributed multiplayer card game**, inspired by the classic *Rubamazzo*, playable via a command-line interface and accessible through a RESTful server. The system has been designed with fault tolerance in mind, handling of temporary disconnections and timeouts, and consistent recovery of game state.

- **Product Type:** A distributed application based on a client–server architecture, with a CLI for clients. The backend is implemented in Scala using the Akka HTTP framework. The system exposes a REST API for all operations (game creation, join, moves, game state).
- **Primary Use Cases:**
  - A player can create a game and share its Game ID.
  - Other players can join by providing the Game ID and enter the same session.
  - Players can make moves, capture cards, steal an opponent's deck, and view the updated state.
  - The system supports temporary disconnections, reconnections within a timeout, and winner calculation.
- **Interaction & Devices:** Users interact via a CLI terminal, sending HTTP requests to the server. The server maintains a consistent game state and notifies of any changes via active polling.
- **Data Management:** Game state (cards on table, turn, deck, player hands, captured cards) is held in-memory on the backend, and each interaction updates the games map consistently.
- **Roles:**
  - **Host Player**: Creates and starts the game.
  - **Participant Player**: Joins the game and plays turns.

2 Requirements

For the correct functioning of the system, functional and non-functional requirements have been identified; the main requirements are listed below, accompanied by examples, useful for validation.

Functional Requirements

- **R1 – Game Creation**

  Description: The system must allow creation of a new game and return a unique Game ID. **Acceptance Criteria:** A successful `POST /game/createGame` returns a Game ID in UUID format (e.g. `"123e4567-e89b-12d3-a456-426614174000"`) and the game is stored in the internal map. **Example:** A client sending this request receives JSON with the `"gameId"` field containing the unique value.

- **R2 – Joining a Game**

  Description: The system must allow new players to join an existing game using its Game ID. **Acceptance Criteria:** If the provided Game ID is valid and the player is not already registered, the server returns a confirmation message (e.g. `"Joined successfully"`) and updates the turn order. **Example:** For game ID `"abc-123"`, `GET /game/join/abc-123/Anna` adds player "Anna" and confirms the join.

- **R3 – Game Start**

  Description: Only the game creator may start the game once the minimum number of players has joined. **Acceptance Criteria:** On receiving `POST /game/start/{gameId}`, the server deals 3 cards to each player, places 4 cards on the table, and sets the initial turn. **Example:** In a 4-player game, `dealCards()` distributes 12 cards to players' hands and 4 to the table, updates the remaining deck, and confirms game start.

- **R4 – Play and Capture**
  Description: A player may perform moves that capture table cards or attempt to steal an opponent's captured deck.
  Acceptance Criteria:
  - The played card must be in the player's hand.
  - If the card matches the "steal" pattern (e.g. equal in value to the top card of an opponent's pile), the theft is executed.
  - If the move allows capture (either direct match or numeric combination), the system updates state by removing captured cards and passing the turn.

Example: If a player plays "7 of Spades" matching the required value to steal an opponent's deck, the system correctly registers the theft and updates game state.

- **R5 – Game State**

  Description: The system must provide each player with the current game state.
  Acceptance Criteria: The JSON response from
  `GET /game/gameState/{gameId}/{playerName}` must include:
  - Cards currently on the table.
  - The player's current hand.
  - Whose turn it is.
  - Captured cards or accumulated score.

  Example: Possible JSON structure:

```
{
  "tableCards": ["3 of Cups", "7 of Coins", ...],
  "playerHand": ["King of Spades", "5 of Clubs", ...],
  "currentTurn": "Luca",
  "capturedCards": ["Jack of Cups", ...]
}
```

- **R6 – Timeout and Disconnections**
  Description: The system must manage temporary disconnections and timeouts to ensure continuity of the game.
  Acceptance Criteria:
  - If a player disconnects, the system retains their state for a predefined period (e.g. 1 minute).
  - If the player fails to reconnect within that time, they are removed and play proceeds with the next player's turn.

  Example: Player "Luca" disconnects; if he doesn't return within 1 minute, he is removed and the game state updates for all.

- **R7 – Game End and Winner**
  Description: At game end, the system must determine the winner based on the number of captured cards.
  Acceptance Criteria:
  - The game ends when there are no cards left in the deck, players' hands and on the table or only one player remains.

– The server outputs the final score and the winner's name.

Example: On completion, the server returns:

```
{
  "winner": "Anna",
  "finalScore": {"Anna": 20, "Luca": 15, ...}
}
```

Non-Functional Requirements

- **Availability (RNF1):** The server must remain continuously up during games.
- **Consistency (RNF2):** Game state must be consistently updated and shared among all clients.
- **Fault Tolerance (RNF3):** The system must handle momentary disconnections without interrupting gameplay.
- **Performance (RNF4):** Each operation (create, move, state update) shall receive a response within 2 seconds. *Note:* Manual tests and empirical measurements show average response times under 500 ms.

Implementation Requirements

- **RI1:** The project shall be developed in *Scala* using the *Akka HTTP* framework.
- **RI2:** Client–server communication must occur via a REST API.
- **RI3:** Game state shall be kept in memory (RAM) without persistent database usage.

Glossary

- **Turn**: Indicates which player's turn is current.
- **TurnCompleted**: Boolean flag per player denoting if their turn has finished.
- **Steal**: Special action where a player plays a card to steal another's captured deck.
- **Deck**: Remaining cards not yet dealt.
- **CapturedDeck**: The pile of cards a player has captured during the game.

- **TableCards**: Cards currently on the table.
- **Timeout**: The period (in milliseconds) after which an inactive player is considered disconnected.
- **Disconnect/Reconnect**: Mechanism allowing temporary exit and re-entry without losing state.
- **Game Over**: Final state reached when no cards remain to be played.

3 Design

This chapter explains the strategies used to satisfy the requirements, illustrating architectural choices and diagrams that clarify interaction among main components.

3.1 Architecture

The system follows a **distributed client–server** architecture. Multiple CLI clients interact with a central REST server. Centralization simplifies management of shared state and ensures consistency across participants.

- Clients are terminals issuing HTTP requests (join, move, get state).
- The server exposes REST endpoints and contains full game logic, including turn cycle management, disconnection handling (via PlayerManager), and winner determination.
- The system supports concurrent games, each identified by a Game ID and maintained in memory.

Centralization eases state maintenance and coordination of asynchronous events (e.g. timeouts).

3.2 Infrastructure

The system is built in **Scala** using **Akka HTTP**. Key components:

- **Scala**: chosen for its functional and object-oriented strengths.
- **Akka HTTP & Akka Actor**: manage REST server, timeouts, and concurrency.
- **Spray JSON**: for JSON serialization/deserialization.
- **CLI**: text-based interface for user interaction.
- **In-Memory State**: active games stored in a `Map[String, Game]`.

The Figure 1 illustrates the main components of the system and their communication flow.

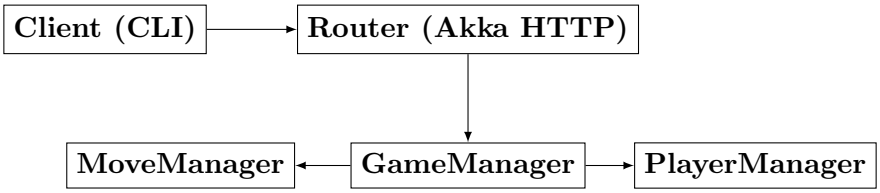

Figure 1: Component Diagram: interaction between Client, Router, and Managers

3.3 Modelling

The primary domain entity is the immutable **Game** object, capturing full game state (ID, players, hands, deck, turn, etc.). Main managers:

- **GameManager**: handles game lifecycle (create, join, start, end), card dealing, and turn updates.
- **MoveManager**: applies functional logic for captures, steals, and redistributions.
- **PlayerManager**: coordinates disconnections, reconnections, and timeouts.
- **Utils**: helper functions for rendering and transformations.

State updates occur by replacing the old `Game` instance with a new one via the `copy()` method, ensuring immutability.

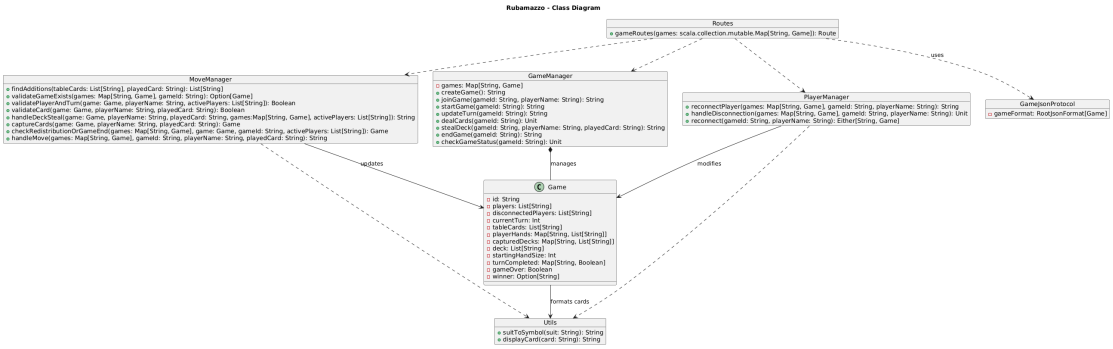

Figure 2: Class Diagram of the Game model and managers

3.4 Interaction

To design the interactions between the components, an Outside-In approach was adopted, starting from the functionalities requested by the user. The CLI provides structured commands, creating an orderly interactive experience even in the absence of a GUI.

Typical game flow:

1. **Create Game:** A user creates a new game by making a `POST /game/createGame` request; the system returns a unique Game ID.
2. **Join Game:** Other players join with a `GET /game/join/{gameId}/{playerName}` request, entering the lobby.
3. **Start Game:** Once everyone is connected, the host sends a `POST /game/start/{gameId}` request, the server deals the cards and starts the turn rotation.
4. **Gameplay:** Players may:
   - Play a move via `POST /game/makeMove`.
   - Fetch state via `GET /game/gameState`.
   - Disconnect via `POST /game/disconnectPlayer`.
   - Reconnect via `POST /game/reconnectPlayer`.

   The system automatically handles the disconnection, maintaining the state for a limited time.
5. **End Game:** Automatically triggers when no cards remain or one player left, returning final scores and winner.

All interactions are synchronous, stateless HTTP calls carrying JSON (via Spray JSON) returned in plain text.

3.5 Adopted Interaction Patterns

- **Synchronous Request/Reply**
  Each game action (create, join, start, move, disconnect/reconnect) is processed synchronously; the client waits for an immediate response.
- **Shared-State + Copy-Update**
  Game state lives in a mutable map in `GameManager`. On each action, `MoveManager` reads a snapshot, applies transformations, and reinserts the updated snapshot, preserving immutability.
- **Functional Pipeline**
  The `handleMove` method implements a chain of responsibility. After validation (turns and card availability), the logic proceeds: implements a chain of responsibility:
  - If the move matches the "steal" pattern (card played = starting value of an opponent's stack), a steal of the deck is attempted.
  - If the move allows capturing cards (direct correspondence or numerical combination), the capture is performed.
  - If no conditions are met, the card is added to the table.

At the end of the chain, the system first evaluates whether all players have run out of cards and, if so, starts the *Refill* phase; if it is not possible to redistribute further cards, it declares the end of the game; in all other cases, it simply continues by passing the turn to the next player.

- **Timeout Management**
  On disconnection, if the player fails to reconnect within the timeout, the system removes them, returns their cards to the deck, and passes the turn. This ensures the continuity of the game in the presence of connection anomalies.

The primary flows are illustrated by the following sequence diagrams:

- Game creation and join (Figure 3);
- Game start (Figure 4);
- Move execution (Figure 5);
- Disconnection and timeout handling (Figure 6).

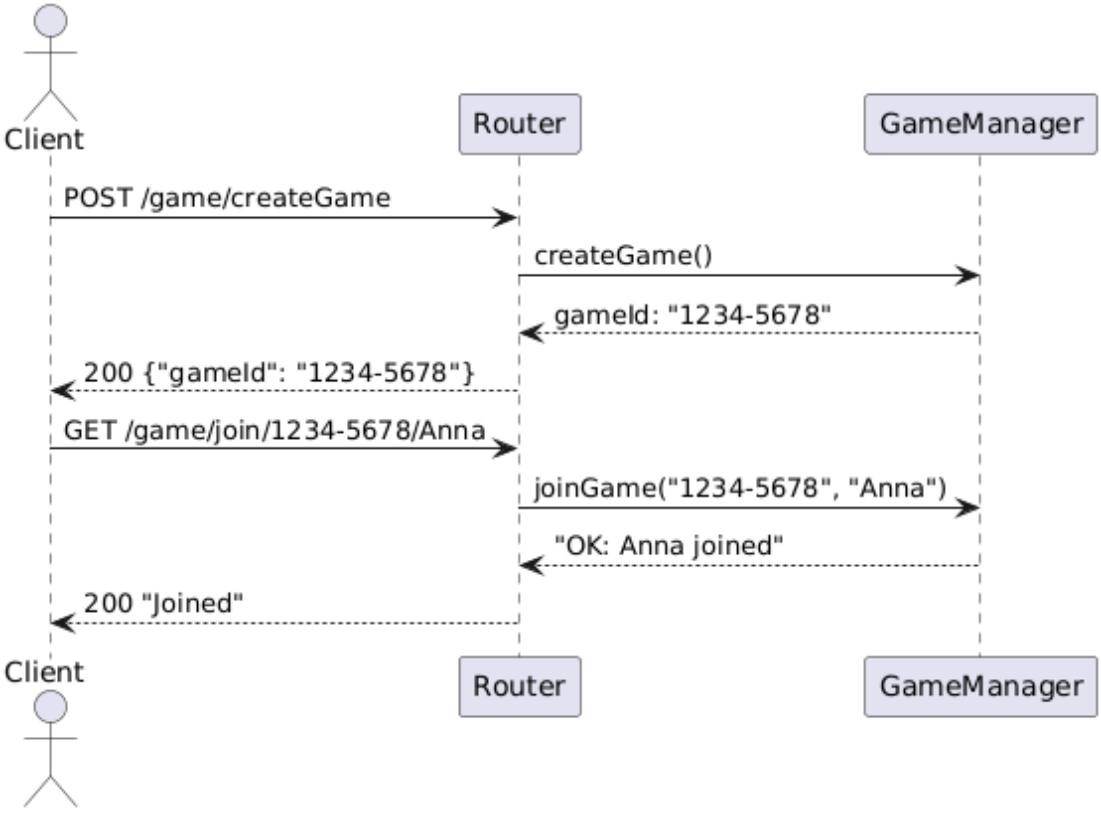

Figure 3: Game Creation and Join Sequence

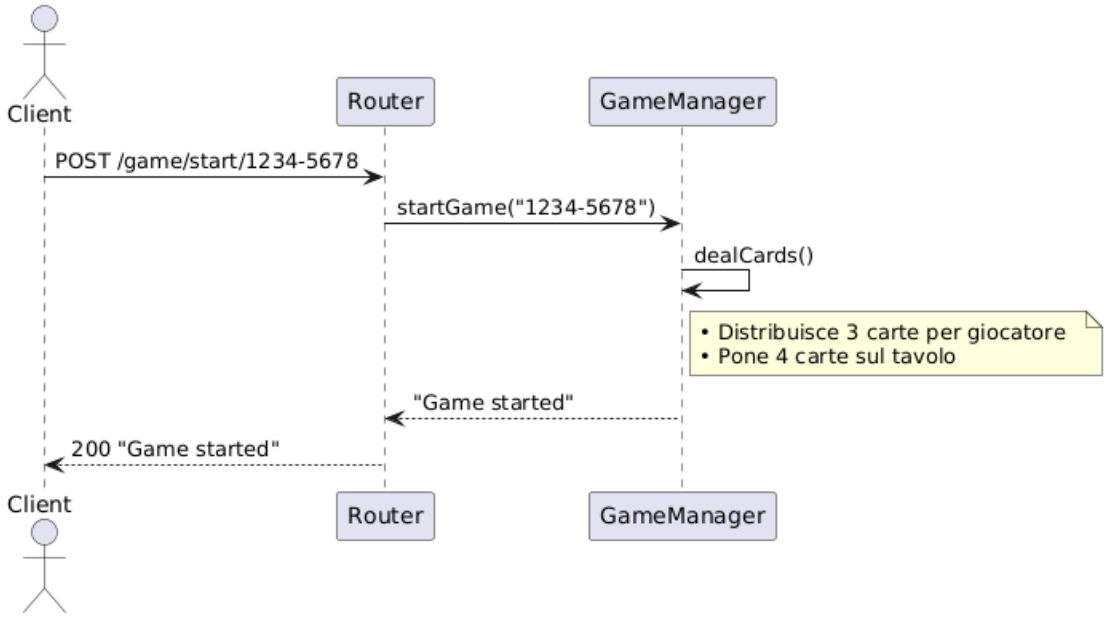

Figure 4: Game Start Sequence

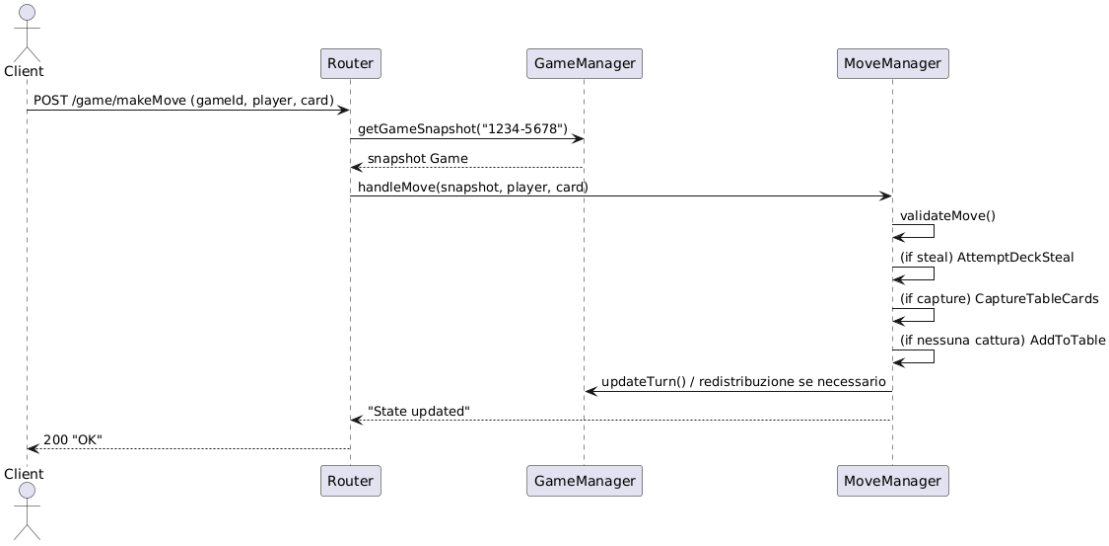

Figure 5: Move Execution Sequence

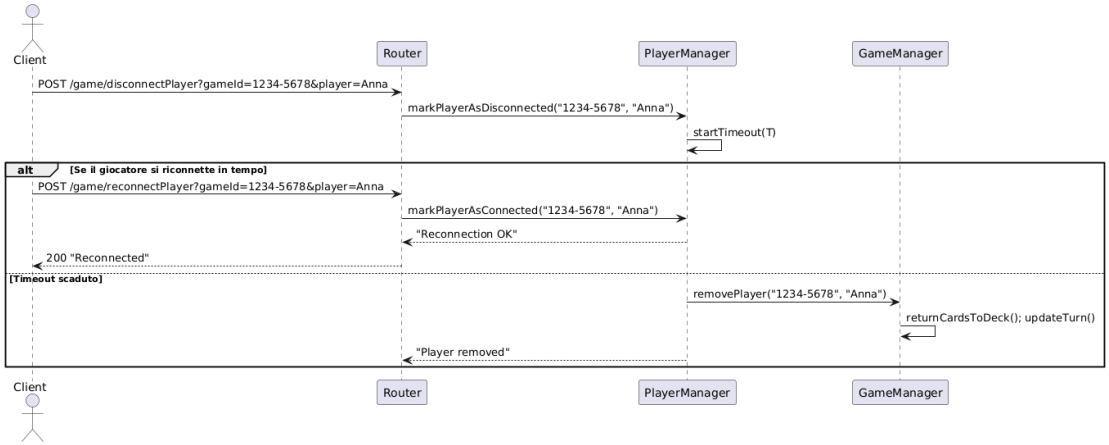

Figure 6: Disconnection and Timeout Handling Sequence

3.6 Behaviour

The system follows a well-defined game life cycle: it starts with creation, when a player sends a `POST /game/createGame` request and receives a unique Game ID; it continues with the participation phase, during which other players join
via `GET /game/join/{gameId}/{playerName}`; then the creator starts the game with `POST /game/start/{gameId}`, dealing three cards to each player, placing four on the table and setting the first round. During the game, only the player whose turn it is can act: if the move is valid, the turn passes to the next player, otherwise it remains unchanged. The game ends automatically when there are no more cards neither in the hands, nor in the deck, nor on the table or only one player remains in play, at which point the system calculates the winner. Finally, if a player temporarily disconnects, his state is retained for a fixed period: a reconnection within the timeout restores his position, otherwise the player is removed and his cards re-dealt. The diagram in Figure 7 illustrates this complete flow, including the handling of anomalies such as disconnections.

3.7 Data and Consistency Issues

The system manages the game state through an immutable map, for example Map[String, Game]. Since all the changes to the state are performed by a single thread (the responsible actor), there are no critical concurrency conditions. The use of the copy() function is only used to work with immutable data structures and to guarantee a clear and functional management of updates, without resorting to explicit locking mechanisms.

3.8 Fault Tolerance

The system withstands partial failures:

- **Temporary Disconnect:** Player state is retained.

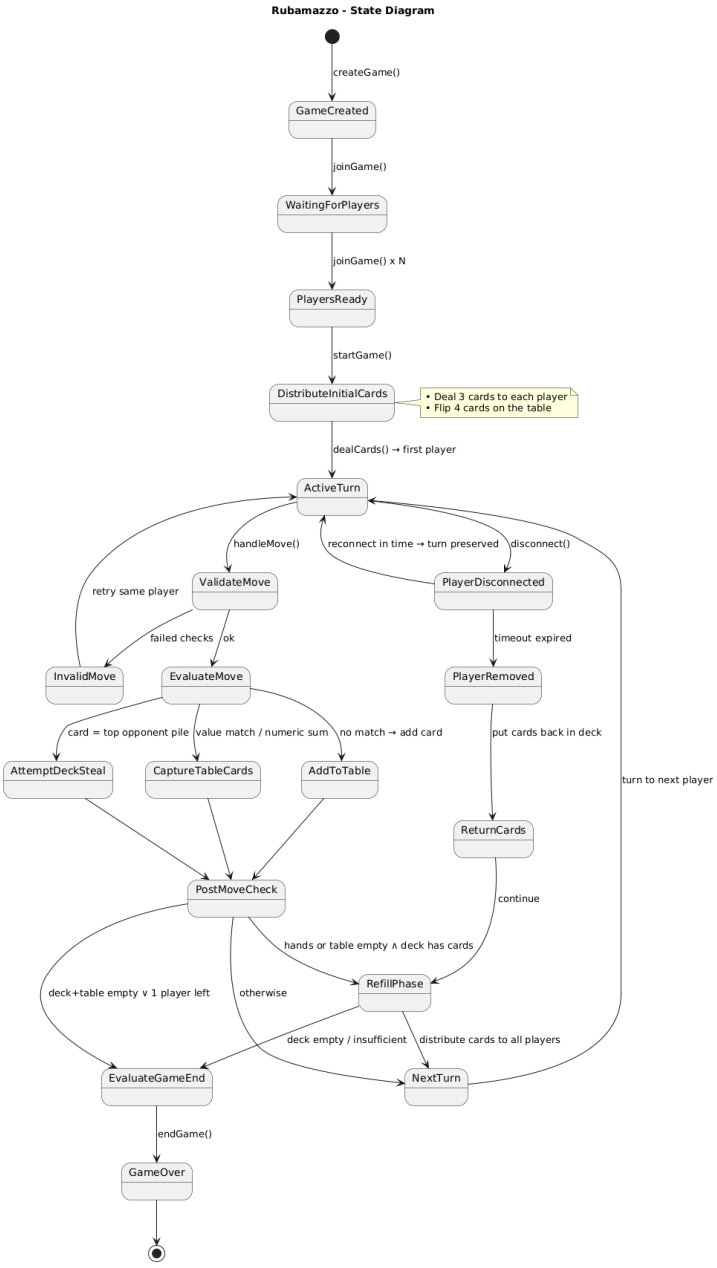

Figure 7: Game State Lifecycle Diagram

- **Reconnection:** If within timeout, state is restored.
- **Timeout Expiry:** Player is removed and cards redistributed.
- **Auto-Cleanup:** Abandoned games are removed from memory.

3.9 Trade-offs and Design Reflections

A centralized Akka HTTP-based design provides:

- Simple, efficient in-memory state management.
- Clear separation between business logic (`MoveManager`) and state handling (`GameManager`).

Thanks to Akka Actors, it remains inherently scalable: clustering and multi-region deployment allow horizontal scaling and maintain a single logical access endpoint even under heavy load.

3.10 Design Conclusions

In summary, the chosen design delivers a consistent, coherent game state while simplifying testing through a clear separation of components. Moreover, its modular structure makes it straightforward to extend the system in the future—for example, by adding graphical or web-based user interfaces. Together with the provided diagrams, these solutions have produced a robust, maintainable platform that fully meets both the functional and non-functional requirements.

4 Implementation

The entire codebase is in Scala, organized into modules:

- **model/** – defines the `Game` case class.
- **server/** – contains `GameManager`, `MoveManager`, `PlayerManager`.
- **routes/** – exposes REST endpoints via Akka HTTP.
- **utils/** – helper functions and JSON protocols.

The `Game` model is immutable, and each update uses `copy()`.

4.1 Technological Details

- **Language:** Scala 2.13.12.
- **Build Tool:** sbt, standard project layout in `src/main/scala`.
- **Libraries:** Akka HTTP, Akka Actors, Spray JSON, Scala Logging.

- **Execution:** Start server with `sbt run`; clients run via CLI.
- **Limitations:** State is in-memory only, without persistence.

5 Validation

5.1 Automatic Testing

The system underwent thorough automated testing, including both unit tests and end-to-end tests. Coverage spans normal scenarios and corner cases, ensuring robustness and functional consistency under adverse conditions.

Unit Testing Unit tests were written with ScalaTest for each major component:

- `GameLogicTest.scala` – validates moves, deck-stealing, and state updates (Requirements R3, R4).
- `EdgeCasesTest.scala` – covers full captures, invalid-card plays, and disconnected-player actions (R5, R7).
- `CheckRedistributionTest.scala` – verifies hand/table redistribution and end-of-game (R6).
- `TurnManagementTest.scala` – tests turn advancement, multiple disconnections, skips, and cyclic rotation (R4, R6).
- `ReconnectPlayerTest.scala` – ensures correct reconnection behavior within and beyond timeout, preserving player state.

All tests run via `sbt test`. Each test case has a descriptive name and precise assertions (`assert`, `should`, `contain`, etc.) to guarantee reproducible, verifiable results.

Integration & Fault-Based Testing Component interactions were validated using realistic multi-user scenarios:

- Deck theft and turn passing
- Player removal on timeout and game-state update
- End-of-game with only one active player
- Turn skipping for disconnected users
- Correct cyclic turn rotation after complete sequences

Corner cases covered include:

- Playing a card not in hand

- Move by a non-current-player
- Timeout after late reconnection
- Simultaneous multiple disconnections

End-to-End Testing Several Bash scripts have been developed and used to simulate full games via HTTP:

- Starting games, chaining moves, and turn rotation
- Disconnection and reconnection within and beyond the timeout
- Automatic player removal and fallback to end-of-game
- Dynamic extraction of card values from CLI output and URL sanitization

Technologies used:

- `curl` for HTTP testing
- `awk`, `sed` for parsing card output
- `sleep` to simulate timeouts

All scripts can be reproduced locally by running `./nomeTest.sh` while the REST server is active.

5.2 Acceptance Testing

Manual acceptance testing validated the CLI interface by checking:

- Accurate rendering of card symbols
- Clear presentation of turns, hands, and table state
- Robustness of CLI input handling
- Realistic multi-user usage experience

Manual tests optimized text output for readability and debugging, covering components not easily automated such as ASCII art and CLI intuitiveness.

6 Deployment

To install and run:

1. Install Java 8 (or above), Scala 2.13, and sbt.
2. Clone the repository:

   ```
   git clone https://github.com/madina9229/rubamazzo-game.git
   ```

3. Change directory:

   ```
   cd rubamazzo-game
   ```

4. Start the server:

   ```
   sbt run
   ```

   Then open another terminal and run the CLI client:

   ```
   sbt run
   ```

   The server listens at `http://localhost:8080/game/`.

   Note (Windows users): Before running the application in your terminal, you may want to enable UTF-8 support to display special characters correctly:

   ```
   chcp 65001
   ```

   This ensures proper rendering of accented letters, card symbols, and any non-ASCII characters.

   Follow the CLI prompts to join a game or create a new session.

5. Run the complete test suite:

   ```
   sbt test
   ```

7 User Guide

After launching the server (`sbt run`), open one or more terminals for clients (`sbt run`). Each client displays:

1. **Create** – starts a new game and returns a unique Game ID.
2. **Join** – joins an existing game by entering its Game ID.

Game Flow:
Once the host shares the Game ID, other players select **Join**, enter the ID and their username, and enter the lobby. When all players are connected, the host chooses `[1] Start the game` to deal cards and begin turns.

During play, each user can:

- `[2] Make a move`: play a card from hand.
- `[3] View game state`: see table cards, hand, turn, scores.
- `[4] Disconnect`: simulate temporary disconnect.
- `[5] Reconnect`: re-enter within timeout.
- `[6] Exit`: quit the client.

The system continuously updates state. A disconnected player is marked absent and may reconnect within the timeout. Otherwise, they are removed and cards redistributed. (as illustrated in Figure 12 and then in Figure 13 to show the updated view on the other client).

When cards run out (or only one player remains), the server announces "GAME OVER" and displays final scores, as shown in Figure 14.

Descriptive screenshots:

- Figure 8 – shows the client startup and creation of a new game, including the generation and display of the Game ID.
- Figure 9 – depicts a second player joining the game by entering the Game ID.
- Figure 10 – highlights the host starting the game, dealing cards, and displaying the current state.
- Figure 11 – illustrates a player performing a valid move by selecting a card from their hand.
- Figures 12 and 13 – document the disconnection simulation and the resulting update of the game state.
- Figure 14 – presents the final phase of the game, showing the determination of the winner and the display of the scores.

8 Final Reflections & Future Work

This project delivers a robust and maintainable platform, thanks in large part to its immutable game-state model, which ensures consistent, thread-safe updates, and to the clear separation of concerns among components responsible for move logic, state management, and disconnection handling. Leveraging modern technologies such as Scala

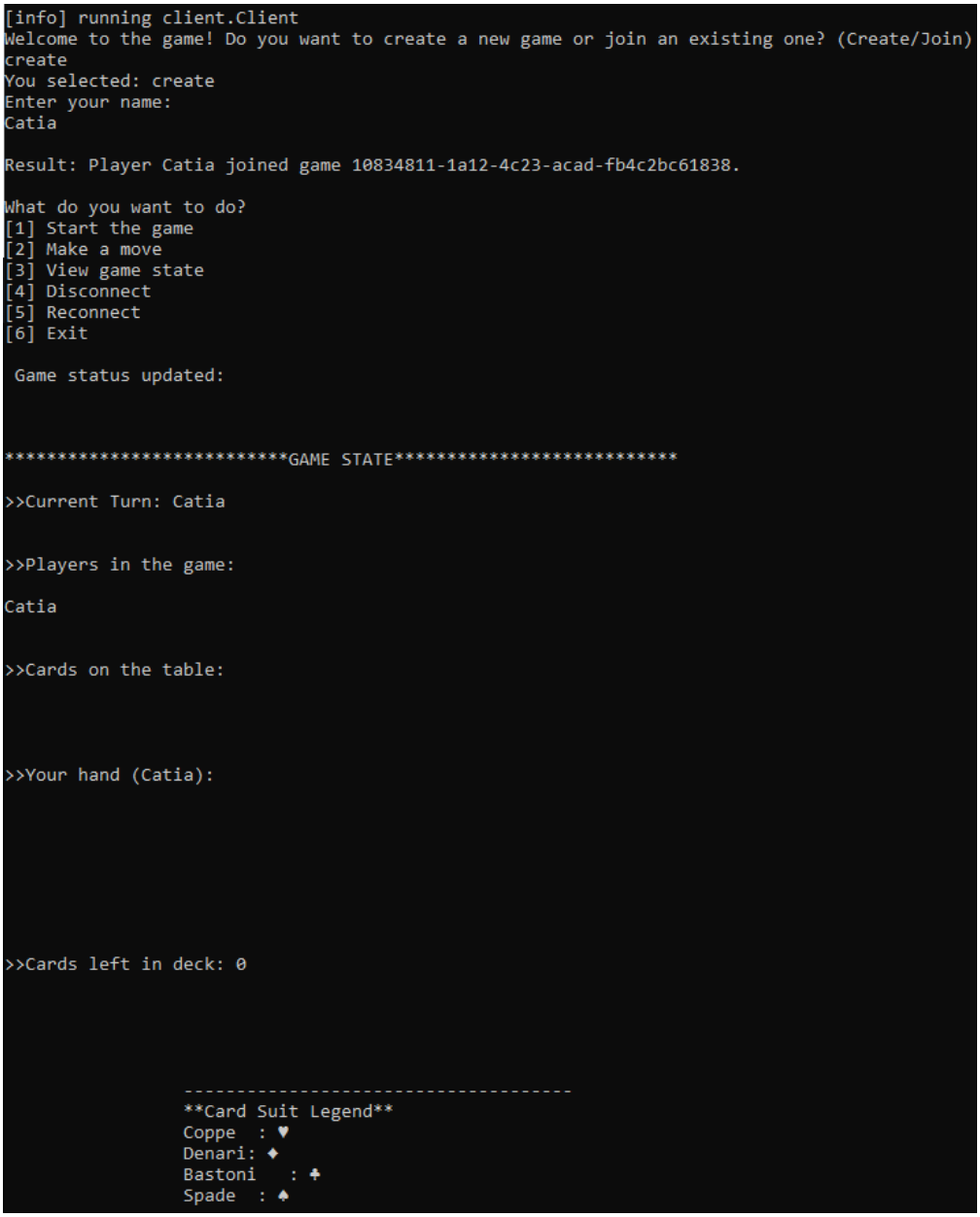

Figure 8: Client start and new game creation with Game ID display

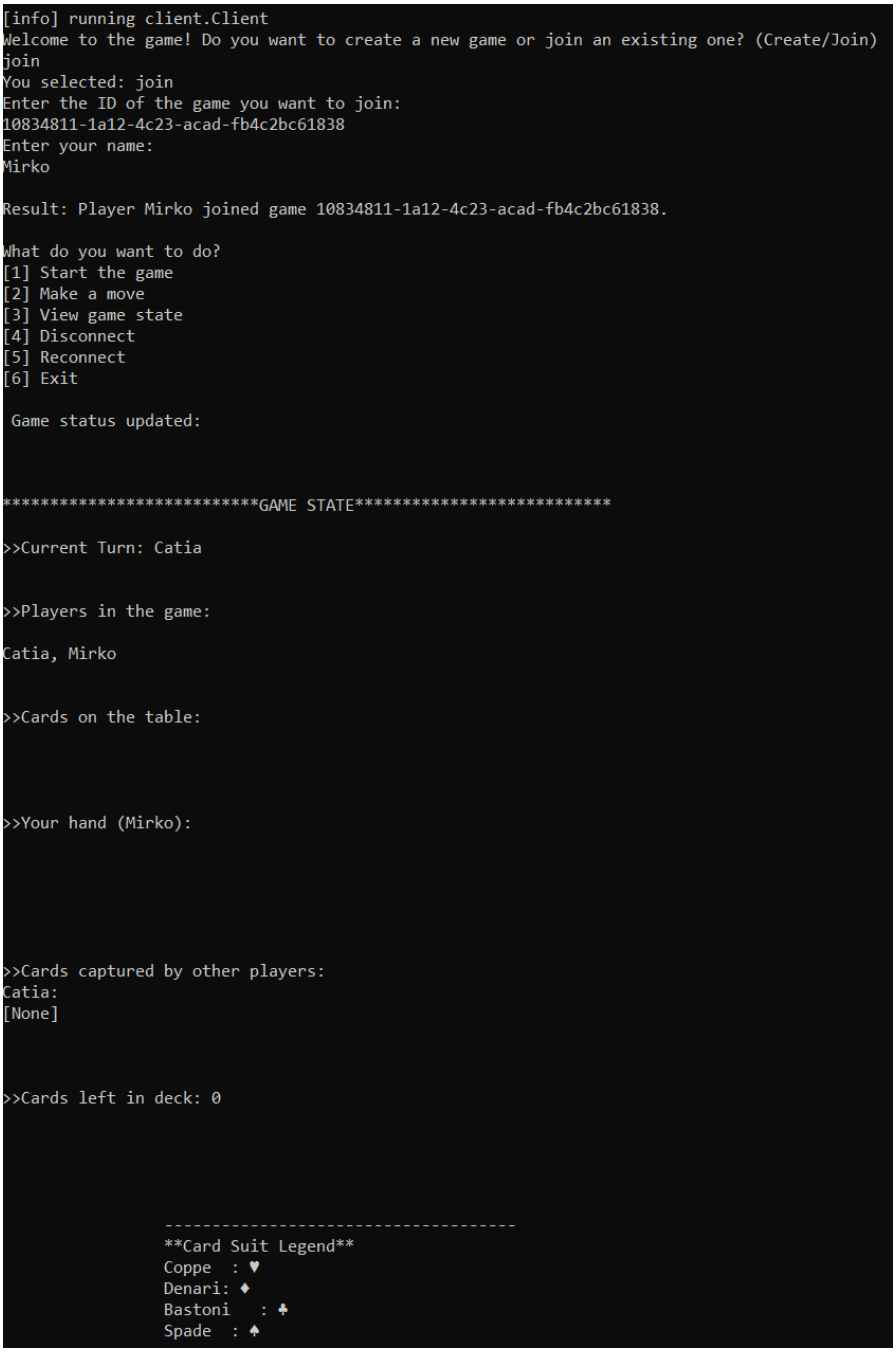

Figure 9: Joining an existing game by entering the Game ID

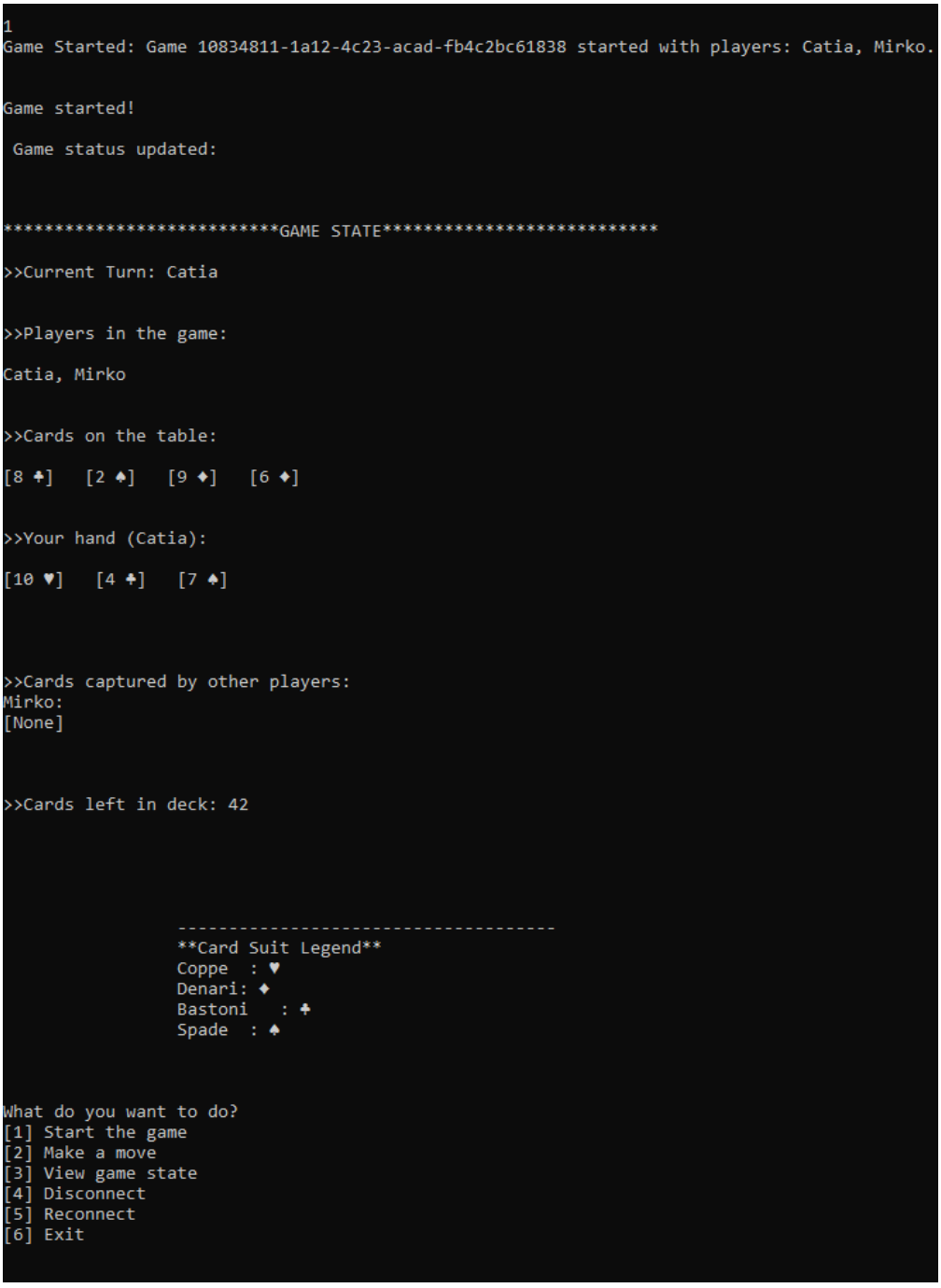

Figure 10: Host starts the game and sees the dealt hands and table cards

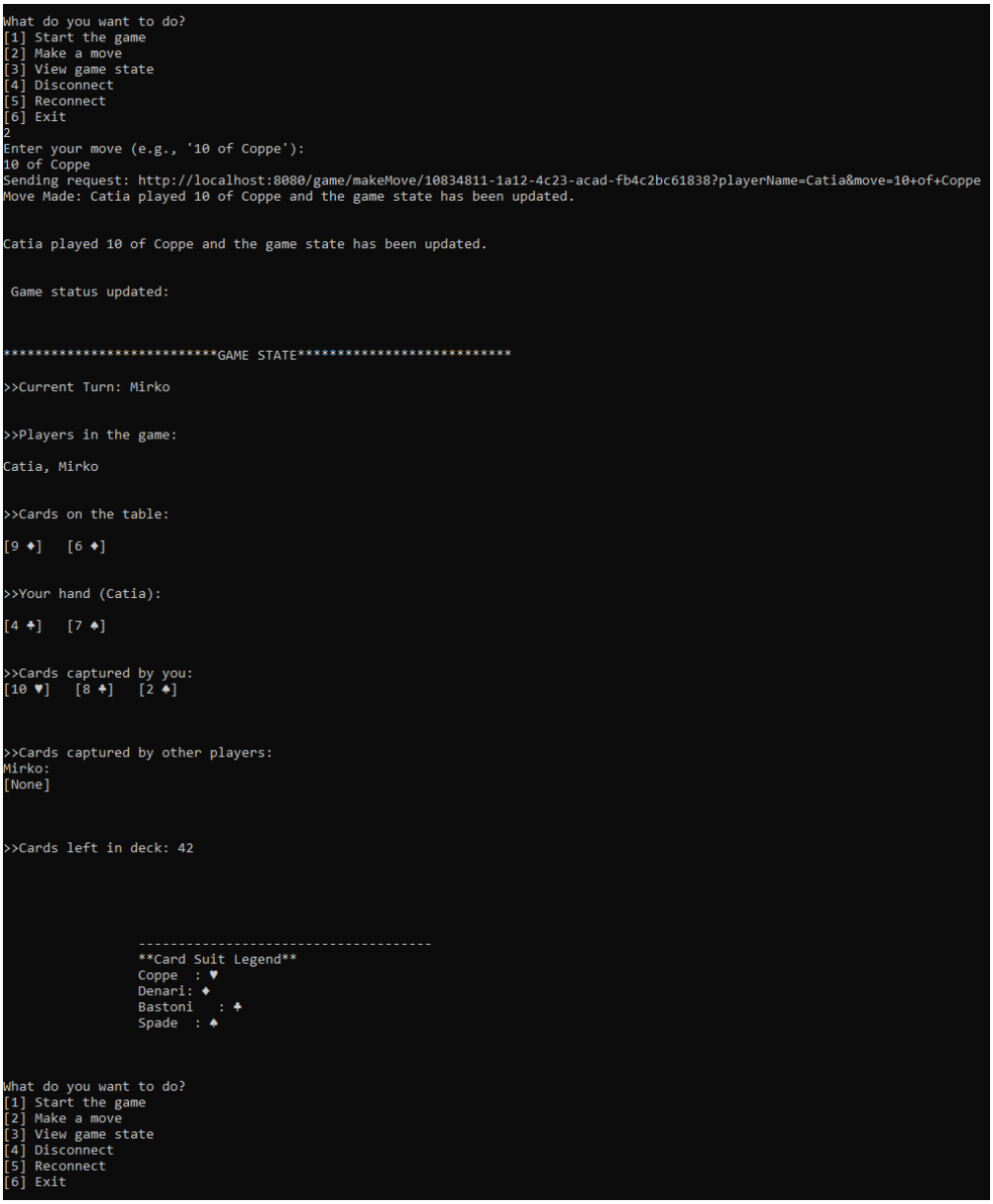

Figure 11: Player makes a move: selecting and playing a card from hand

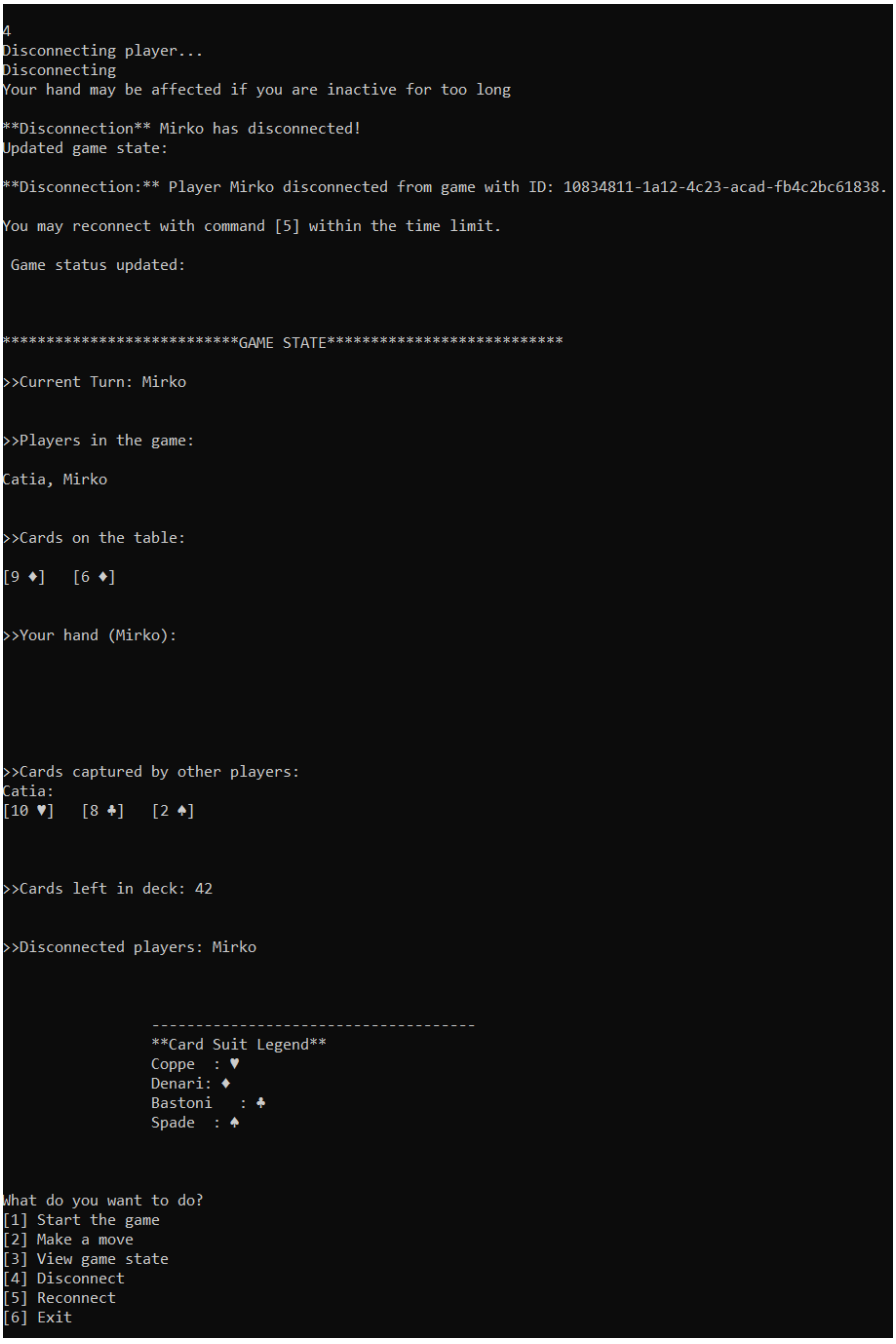

Figure 12: Player disconnection simulation; absent player marked in state

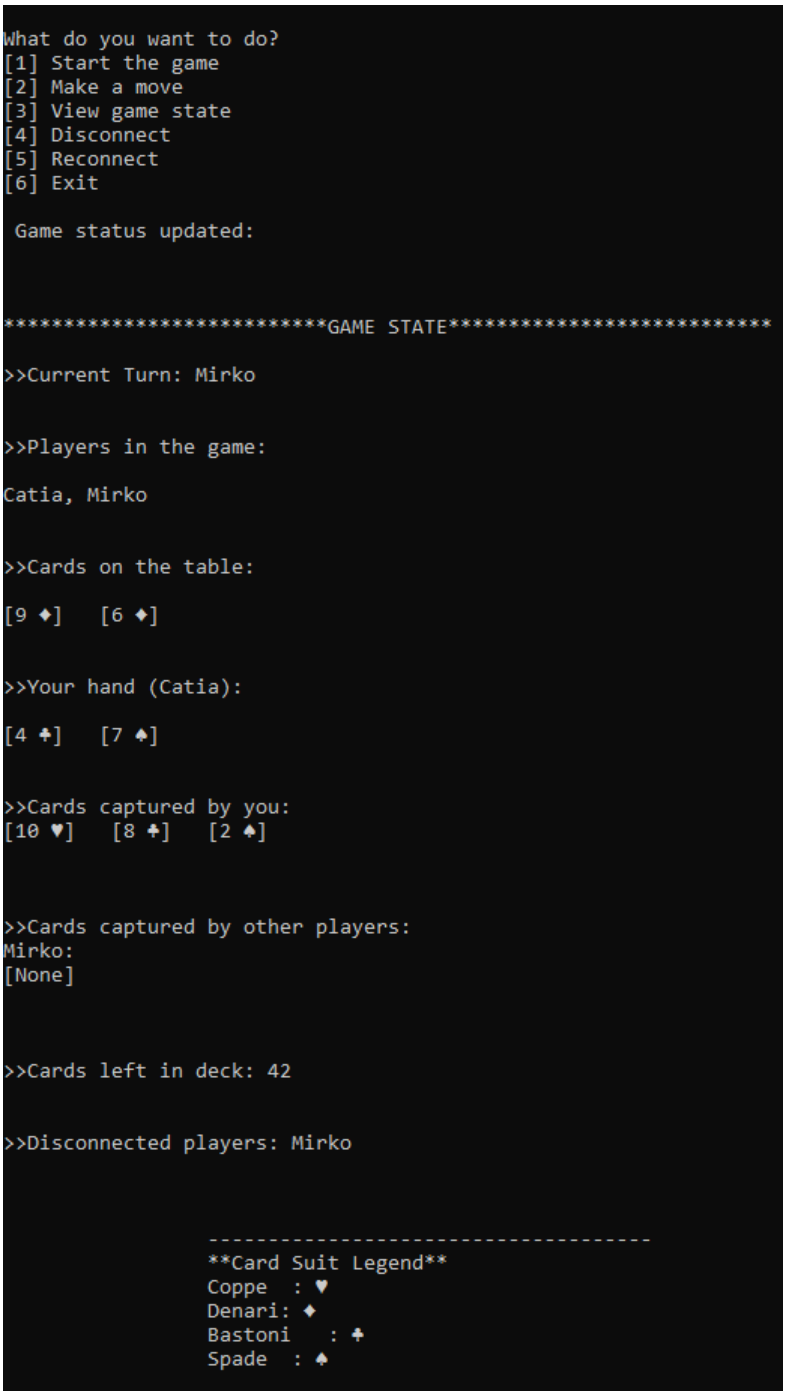

Figure 13: Other player's view after a peer disconnects

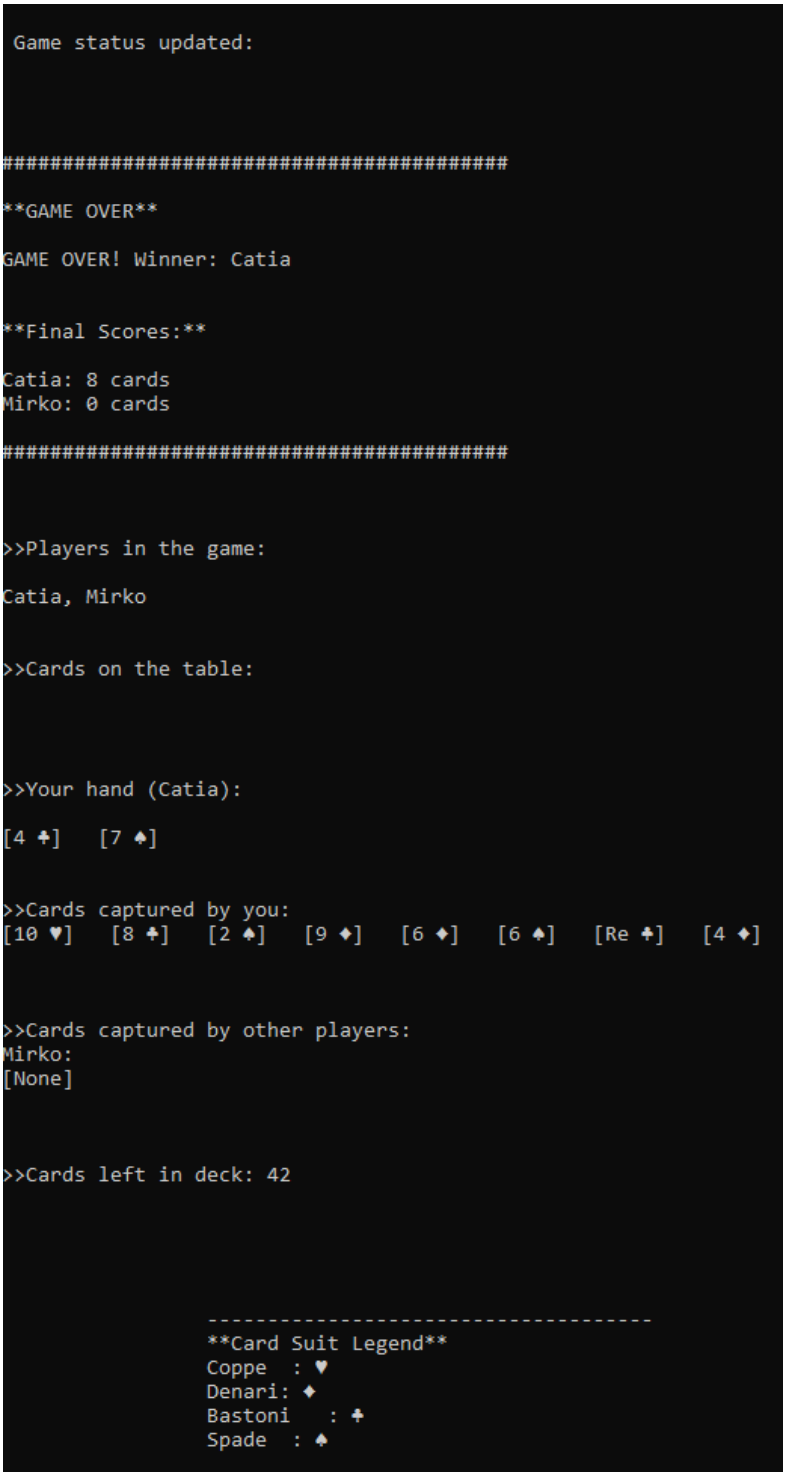

Figure 14: Game conclusion: “GAME OVER” and final score display

and Akka HTTP further enhances responsiveness and scalability under normal operating conditions.

Looking ahead,opportunities to broaden the system's appeal and production readiness have been identified. Introducing a graphical or web-based user interface would make the game more accessible to a wider audience, while adding a persistent storage layer such as a database would safeguard game state against server crashes or restarts. Finally, although our actor-based design scales well, a centralized server can become a bottleneck under heavy load; adopting clustering technologies (for example, Akka Cluster) and distributing the load across multiple instances or geographic regions would significantly boost performance and resilience.

Pursuing these enhancements will not only refine the user experience but also prepare the system for large-scale, production-grade deployment.