MindROll Report

Yiming Li
yiming.li@studio.unibo.it

Tianyu Qu
tianyu.qu@studio.unibo.it

Jing Yang
jing.yang5@studio.unibo.it

March 2025

MindRoll is a distributed, turn-based multiplayer dice game designed to explore and apply key concepts in distributed systems such as authentication, remote procedure calls (RPC), real-time synchronization, and client-server communication. The system is implemented in Python, with a graphical user interface (GUI) developed using Pygame and a custom RPC framework enabling communication between clients and the central game server. Players can register, log in, create or join rooms, and take part in rounds of strategic gameplay that involve calling and revealing dice values. The server manages the game state, verifies turns, synchronizes player actions, and broadcasts results to all connected users to ensure fairness and consistency. The system also supports fault tolerance through reconnection logic, and scalability is considered for future extensions. Core game logic is abstracted from the UI and communication layers to maintain modularity. This project provides a hands-on implementation of core distributed system principles in a real-time interactive environment. It serves as a foundational prototype for building more advanced online multiplayer systems and highlights the practical challenges of synchronization, state consistency, and user coordination in a distributed setting.

1 INTRODUCTION

The *MindRoll* project is a distributed, multiplayer dice game that aims to combine strategic gameplay with key principles of distributed systems. Designed as a client-server application, the system allows users to register, log in, create or join game rooms, and participate in turn-based matches involving dice rolls, number calls, and result reveals. The game introduces not only an engaging logic-based mechanic but also real-time interaction among players across different machines.

From a technical perspective, this project demonstrates practical implementation of several distributed system concepts, including user authentication, remote procedure calls (RPC), state synchronization, and server-side coordination. The backend is designed to manage multiple concurrent rooms and ensure turn-based fairness through centralized logic. At the same time, the frontend built with Pygame offers an interactive interface where each player receives synchronized updates, visual feedback, and input options.

The primary goal of this work is to deliver a functioning prototype that models real-world distributed game behavior, while remaining modular and extensible for future development. It highlights challenges like connection handling, user management, broadcast communication, and fault tolerance—especially in scenarios like reconnection or concurrent inputs. Through this project, we explore how distributed principles can be integrated into an interactive gaming environment with a responsive and consistent user experience.

2 Requirements

2.1 Functional Requirements

To play MindRoll, players interact through a client application that facilitates all core gameplay and user management operations. The client must allow players to:

- register or log in to authenticate and retain individual match statistics;
- either create a new game room or join an existing one, depending on availability;
- receive a randomly generated die (with a value and color), which serves as the basis for their strategy;
- make a "call" by proposing a number higher than the current one, or challenge the previous call;
- initiate a "reveal" to determine the winner based on the sum of dice and predefined logic;
- view dynamic score updates at the end of each round.

Behind the scenes, the server is responsible for maintaining global game state, coordinating player turns, matching opponents, and ensuring real-time synchronization across clients for consistent gameplay.

2.2 Non-functional requirements

Beyond the core gameplay features, MindRoll must also meet several non-functional requirements to ensure a robust and seamless multiplayer experience. The system should be capable of managing concurrent users efficiently while maintaining fault tolerance,

scalability, and real-time synchronization. These properties are crucial to preserving the integrity and responsiveness of the game during peak usage and potential network disruptions.

To support consistent gameplay, the server must gracefully handle unexpected edge cases. For example, it must allow a player to reconnect smoothly after a disconnection, reject invalid attempts to join non-existent rooms, and validate turn-based actions such as calls or reveals to prevent out-of-order operations. Additionally, the system should define clear behavior for new players who attempt to join an already ongoing match, either by deferring their entry or providing feedback on game status.

These considerations are thoroughly addressed in the design and architecture of the system to promote fairness, preserve game logic integrity, and deliver an optimal user experience across all players and sessions.

Figure 1 illustrates the game logic flow and the key interactions across components.

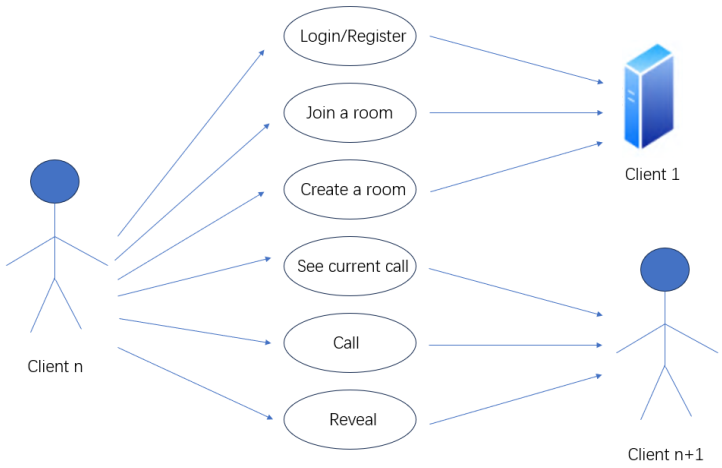

Figure 1: logic flow

2.3 Technology choices

To implement MindRoll, we adopted a client-server architecture using Python as the core programming language for both frontend and backend development. This choice was driven by Python's ease of use, extensive library support, and suitability for rapid prototyping of distributed systems. The graphical user interface (GUI) was built using Pygame, a lightweight yet powerful library for creating interactive interfaces and real-time visual feedback. Pygame allowed for the creation of a dynamic, event-driven GUI where players can interact through buttons, input fields, and animations.

Communication between the client and server was handled using a custom RPC (Remote Procedure Call) mechanism, implemented over TCP sockets. This decision enabled the system to be lightweight and flexible, while still maintaining clear message boundaries for request and response communication. The server exposes RPC methods for

critical game functionalities such as authentication, room management, and game actions (e.g., call, reveal). This approach ensures modularity and allows easy integration of additional services in the future.

On the server side, we introduced a modular structure composed of separate components for authentication, user management, and game logic. The game state is managed centrally and updated in response to client actions, while synchronization across clients is achieved through server-side broadcasting mechanisms. For resiliency and future extensibility, a backup server module was also included to handle redundancy and potential failover support.

The choice of this technology stack balances simplicity and extensibility, making it ideal for academic projects that simulate real-world distributed systems while still being accessible for further development and deployment.

3 Design

MindRoll adopts a classic client-server architecture to ensure scalability, synchronization, and fairness throughout the gameplay. Players interact through a graphical user interface (GUI) built with Pygame, which communicates with a centralized server that handles authentication, room coordination, game progression, and result evaluation. The backend is logically divided into two primary modules: one that governs game logic (handling player actions, dice rolls, turn management, and rule enforcement), and another responsible for data persistence, such as user credentials, match statistics, and room configurations.

The server supports both 1v1 and multiplayer modes, dynamically assigning dice to players and verifying the correctness of each "call" and "reveal" action. Real-time communication is facilitated through a lightweight RPC layer, allowing clients to invoke remote procedures with minimal overhead. To improve reliability, fault tolerance is built into the system: players who disconnect can reconnect seamlessly without disrupting the current game state.

This modular design ensures clear separation of concerns—frontend focuses on rendering and user interaction, while the backend enforces core mechanics and ensures synchronized state across clients. Altogether, this architecture delivers a consistent, responsive, and fair game experience. The structure is illustrated in **Figure 2** .

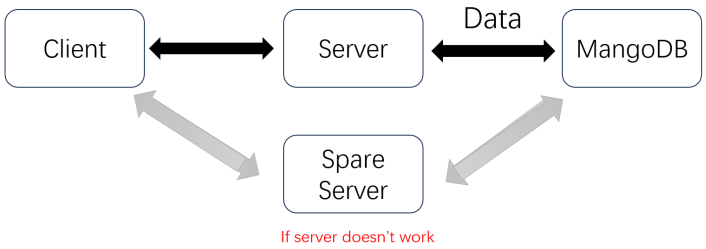

Figure 2: relation diagram

3.1 Architecture

MindRoll adopts a centralized client-server architecture, which was selected to ensure strong consistency, controlled synchronization, and simplified game state management across multiple players. This architectural choice aligns with the core requirements of the game—real-time interaction, fairness in gameplay, and robustness in multiplayer coordination. By centralizing all game logic and data management on the server side, the system guarantees a single source of truth that all clients rely on.

The architecture consists of two main entities: the client, built using pygame as a graphical front-end, and the server, implemented in Python and exposed through Remote Procedure Calls (RPC). Clients handle user interactions such as registration, login, room creation or joining, and gameplay actions (e.g., call and reveal). These actions are sent to the server, which validates and processes them according to the game rules.

The server is logically divided into two parts: one managing dynamic game state—like match status, current turn, assigned dice—and another maintaining persistent user data such as credentials and scores. The RPC-based communication ensures decoupling between interface and logic, making the system modular and maintainable.

This architecture supports both 1v1 and multiplayer modes, and is designed to scale by replicating server logic or migrating to microservices in future iterations. It also enables features like reconnection handling, where disconnected players can resume their sessions without desynchronizing the match state.

In short, the centralized client-server model was chosen to facilitate fairness, security, and expandability—core pillars required for an interactive, competitive, and distributed game like MindRoll.

3.2 Infrastructure

The infrastructure of *MindRoll* is structured around a centralized client-server model, where multiple clients interact with a single authoritative server using Remote Procedure Calls (RPC). This architecture allows the server to centrally manage game state, user authentication, matchmaking, and gameplay synchronization while providing clients with real-time feedback and updates. The main infrastructural components include:

(1) the client applications built with pygame,

(2) the RPC server, implemented using TCP module, and
(3) an optional backup server for fault tolerance and recovery.

All components currently reside on a single physical machine during development for ease of testing and integration. However, the modular design supports deployment across multiple machines or containers. For example, the authentication logic, game session manager, and broadcast system can be split into independent microservices, each deployed on a separate node. This would allow for horizontal scalability, especially in multiplayer mode with higher player loads.

Client and server components connect over a local network or LAN using predefined IP addresses and ports. In more advanced deployments, service discovery mechanisms such as mDNS, DNS SRV records, or dedicated service registries (e.g., Consul) can be introduced to support dynamic resolution of service endpoints. Load balancers can also be introduced between clients and replicated server nodes to distribute load evenly and enhance system availability.

Each component is responsible for a clearly defined set of tasks. The client handles the user interface and local input, while the server manages centralized game logic, current call state, turn management, and result validation. Communication is asynchronous and event-driven, ensuring responsiveness even during high network latency or partial disconnections. All critical events—such as a player revealing, joining, or disconnecting—are propagated to all relevant clients to maintain state consistency.

This infrastructure supports future upgrades like cloud-based hosting, deployment using Docker/Kubernetes, and scaling individual services based on load, thus making *MindRoll* robust, extensible, and production-ready.

3.3 Modelling

The *MindRoll* system models several key domain entities, each representing fundamental components of gameplay and user interaction. The primary entities include:

- User: each player is modeled as a user entity, with associated credentials (username and password), game history, and score statistics.
- Game Room: a logical container where one or more users participate in a match. Each room maintains state information such as players involved, current turn, current call value, and whether the game is active or concluded.
- Dice: each player receives a randomly assigned die (color and number), which determines their probability strategy. Dice are associated with users within a game session.
- Call State: this entity tracks the current number being called in the game, including validation logic, turn order, and whether the next action should be a new call or a reveal.

These domain entities are mapped to infrastructural components as follows:

- The server maintains the full state of users, rooms, and dice, ensuring consistency and enforcing game rules.
- The clients hold temporary views of relevant state, rendered via the GUI, and communicate input/output with the server through RPC calls.
- Game-related events (like call, reveal, join, or disconnect) are tracked server-side, and broadcasted to all connected players.

The system tracks several important domain events:

- A user registers or logs in.
- A player creates or joins a game room.
- A dice is assigned when the game starts.
- A player places a call or performs a reveal.
- A player disconnects or reconnects mid-game.

The messages exchanged include:

- Commands like call_number, reveal_result, create_room, or join_room
- Queries such as get_current_state or check_winner
- Notifications that are broadcasted to clients, like update_call or show_result.

The system state includes users' login credentials, match participation, dice values, round outcomes, and win/loss records. All shared state is maintained server-side for consistency, and periodically synchronized with clients to reflect the current game status.

3.4 Interaction

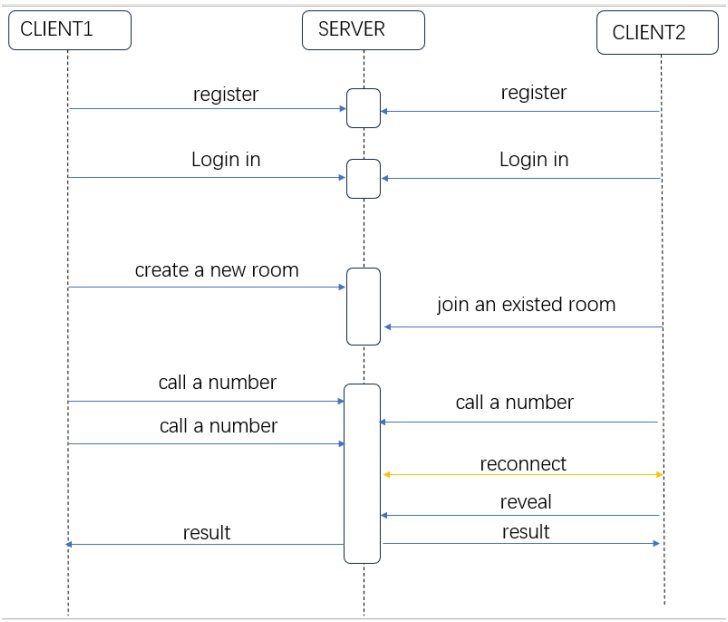

Figure 3: Draft of interactions between clients and server.

The interaction flow in *MindRoll* is based on a client-server model where all communication between players is mediated through the central server. The interaction begins with registration or login, where each client sends credentials to the server for authentication. Once authenticated, users can either create a new room or join an existing one. The server assigns players to rooms and manages the game state centrally.

During gameplay, clients send call requests to the server, which verifies the move, updates the current state, and then broadcasts the updated call value to all clients in the same room. When a reveal is triggered, the server calculates the result based on all players' dice and the call history, and then communicates the outcome to each client. The system ensures synchronous state updates, so all players see the same result simultaneously.

The system also handles disconnections gracefully. If a player gets disconnected, they can send a reconnect request to the server. Upon successful reconnection, the server restores the game state for that client, allowing them to continue without disrupting the session.

The interactions follow a request-response pattern for most actions (register, login, call, reveal), while broadcasting is used for updating all players with game state changes. This design ensures consistency, fairness, and fault tolerance across distributed clients.

The interaction sequence is visually summarized in Figure 3, which illustrates the step-by-step exchange of messages between clients and server, covering the full lifecycle from registration to result delivery and reconnection handling.

3.5 Data and Consistency Issues

The core data maintained in our application includes user credentials (username and hashed passwords), user roles, game room states, and session-specific information such as dice values, player scores, turn orders, and connection status. Originally, these data were stored in memory using Python dictionaries, such as in the `InMemoryUserDatabase` and `GameRoom` objects. This approach provided fast access suitable for early prototyping and real-time responsiveness.

However, relying solely on in-memory storage posed limitations in terms of persistence, fault tolerance, and scalability. To address this, we later integrated a persistent database—**MongoDB**—into our architecture.

With this enhancement, all critical data—particularly user credentials and authentication tokens—are now stored persistently using MongoDB collections. The server interacts with MongoDB through Python's `pymongo` library for operations such as user registration, login validation, and token-based session management. This integration ensures that user data is retained across server restarts and supports features like token expiration via TTL indexes.

Although some real-time game session data (e.g., active game rooms, current dice, temporary scores) are still maintained in memory for performance reasons, these can also be extended to persistent collections in future work. All reads and writes are handled sequentially within the server thread model, avoiding concurrency conflicts while preserving game state consistency across all connected clients.

3.6 MongoDB Integration

To enhance persistence, scalability, and resilience, we integrated **MongoDB** as the backend database for storing long-lived application data. This integration replaced previous in-memory-only structures for users and authentication logic, providing a centralized and fault-tolerant solution.

MongoDB now stores the following:

- **User Credentials**: Each registered user is stored with hashed and salted passwords (using `bcrypt`) and associated role information.
- **Authentication Tokens**: Each login session generates a token stored in the database with an expiration timestamp, managed via TTL indexing for automatic cleanup.
- **(Optional Extension)**: Game rooms and match outcomes may be stored in the future to support recovery, analytics, and replay features.

The database is accessed by the primary server during all operational phases. In the case of a server failure, the backup server can take over and continue interacting with the same MongoDB instance, ensuring continuity without data loss. This decoupling between application logic and storage significantly improves robustness.

The main benefits of MongoDB integration are summarized below:

- **Data Persistence**: User accounts and session tokens are preserved across server restarts and failures.
- **Real-time Synchronization**: Changes in authentication or state-related data are immediately reflected in the database and accessible across server instances.
- **Automatic Cleanup**: Token expiration is enforced via TTL indexes, preventing database bloating.
- **Scalability**: The schema-free design supports future additions such as leaderboards, historical match data, or player statistics.
- **Disaster Recovery**: Centralized storage ensures backup servers can seamlessly resume operations without data inconsistencies.

Overall, MongoDB significantly elevates the reliability and maintainability of the *MindRoll* system, laying a foundation for future architectural enhancements such as sharded clusters, replica sets, or cloud-native deployments.

3.7 Fault-Tolerance

The system handles faults primarily through timeout mechanisms and reconnection logic within the game logic. Players who disconnect are marked as disconnected and are provided a 120-second window to reconnect, managed by timestamps recorded upon disconnection. If reconnection does not occur within this timeout period, the disconnected player's session data is cleared, and the player is removed from the active game room, prompting the game to reset if necessary. Error handling is explicitly defined in the RPC layer (‘rpcserver‘), where any parsing or runtime errors result in clear error messages sent back to clients. This explicit error handling ensures the system remains robust, clearly indicating issues without crashing.

3.8 Availability

Currently, the system does not implement caching mechanisms or load balancing due to its simplicity and expected scale. However, the designed architecture inherently supports potential future scalability, where such mechanisms could be introduced to enhance performance and manage larger numbers of concurrent connections. Regarding network partitioning, the implemented reconnection logic provides basic resilience; disconnected players have a defined window to reconnect, after which the system gracefully handles player removal and game state resets, ensuring continuous availability for remaining users.

3.9 Security

Security measures include authentication and basic authorization. Users are authenticated using username and password pairs, verified against securely stored hashed pass-

words within an in-memory database (‘InMemoryUserDatabase‘). Authorization is role-based, with defined roles such as 'USER', restricting certain actions like joining and creating rooms to authenticated and authorized users only. The system employs cryptographic mechanisms like token-based authentication, where each login generates a secure token used for subsequent client-server interactions. While data encryption mechanisms beyond secure token verification are currently not implemented, the token-based system provides essential security suitable for the application's current scope.

3.10 Password Hashing with Salting

To enhance the security of user authentication, we implemented password hashing with salting in the *MindRoll* system. While initial versions relied on plain hashing algorithms, we recognized that this approach remains vulnerable to dictionary attacks and precomputed hash table attacks (e.g., rainbow tables). We use bcrypt, which internally generates a salt and applies a secure hash function. The salt is embedded in the resulting string and stored in the database alongside the user record.

For each new user registration, the system generates a unique cryptographic salt, which is combined with the user's plaintext password before applying a secure hash function. We employ the `hashlib` library in Python, specifically using the SHA-256 hashing algorithm to produce fixed-length, irreversible hashes. Both the generated salt and the resulting hash are stored in the database alongside the user record.

The key benefits of salting include:

- **Increased Resistance to Rainbow Table Attacks**: Salts ensure that identical passwords yield different hashes, rendering precomputed attacks ineffective.
- **Enhanced Uniqueness**: Every user has a unique salt, guaranteeing that even identical passwords across accounts do not share the same hash.
- **Future-Proofing**: The implementation can be easily upgraded to support more advanced algorithms (e.g., bcrypt, Argon2) for even stronger security guarantees.

During login, the system retrieves the stored salt corresponding to the user, recomputes the hash of the provided password combined with the salt, and compares it with the stored hash to verify authenticity.

This enhancement significantly strengthens the overall security posture of the *MindRoll* system, ensuring the protection of sensitive user credentials even in the event of data exposure.

4 Implementation

- which **network protocols** to use?
  - The project utilizes **TCP (Transmission Control Protocol)** for network communication between the client and server. TCP is chosen for its reliability, ordered data delivery, and error-checking mechanisms, which are essential for

real-time game interactions. The client establishes a TCP connection to the server using Python's `socket` module, ensuring that all game actions (e.g., calling numbers, revealing results) are transmitted accurately. Specifically:

  * In `customer_client.py`, the `CustomClient` class uses `socket.create_connection` to establish a TCP connection to the server.
  * In `rpc_server.py`, the `MindRollServer` class listens for incoming TCP connections using `socket.socket` and `socket.bind`.

- While TCP is the primary protocol, the project could be extended to support **WebSockets** for real-time, bidirectional communication in a web-based client. However, for the current desktop-based implementation, TCP provides sufficient performance and reliability.

- how should *in-transit data* be **represented**?
  - Data transmitted between the client and server is represented in **JSON (JavaScript Object Notation)** format. JSON is lightweight, human-readable, and easy to parse, making it suitable for exchanging structured data such as player information, game state, and request/response payloads. Key implementations include:
    * In `utils.py`, the `Request` and `Response` classes handle the serialization and deserialization of JSON data using Python's built-in `json` module.
    * The `serialize` function converts Python objects (e.g., `Request` or `Response`) into JSON strings, while the `deserialize` function parses JSON strings back into Python objects.
    * For example, when a client sends a request to call a number, the request is serialized into a JSON string and transmitted to the server. The server then deserializes the JSON string to process the request.
  - The use of JSON simplifies debugging and interoperability, as the data format is widely supported across programming languages and platforms.

- how should *databases* be **queried**?
  - The project employs an **in-memory database** implemented using JSON files to manage user data. The `UserDatabase` class in `users_e.py` stores user information (e.g., username, password, score) in a JSON file. Key features include:
    * The database is loaded into memory at startup using the `load_users` method, which reads the JSON file and constructs a dictionary of `User` objects.
    * User data is persisted to disk using the `save_users` method, which writes the dictionary back to the JSON file.

        * While this approach is simpler than traditional SQL or NoSQL databases, it is sufficient for a small-scale game system. However, for larger-scale applications, a more robust database solution (e.g., PostgreSQL or MongoDB) could be integrated.

- how should components be *authenticated*?
    - Authentication is implemented using **Token-based authentication**. Upon successful login, the server generates a unique token for the user, which is then included in the metadata of subsequent requests. Key implementations include:
        * In `users_e.py`, the `Token` class manages token generation and expiration. Each token includes a unique UUID, the associated username, and an expiration timestamp.
        * The server validates tokens using the `validate_token_by_str` method in the `InMemoryAuthenticationService` class (referenced in `rpc_server.py`).
        * Tokens are included in the `metadata` field of requests, allowing the server to authenticate users without requiring repeated login credentials.
    - This approach ensures secure user sessions while minimizing the overhead of repeated authentication.

- how should components be *authorized*?
    - Authorization is based on **Role-Based Access Control (RBAC)**. The `Role` enum in `users_e.py` defines user roles (e.g., `ADMIN` and `USER`). Key implementations include:
        * The server checks the user's role before allowing certain operations. For example, administrative actions may be restricted to users with the `ADMIN` role.
        * In `rpc_server.py`, the `__check_authorization` method validates the user's token and role before processing requests.
        * This simple yet effective authorization mechanism ensures that only authorized users can perform specific actions, such as creating or deleting game rooms.

4.1 Technological details

- The project leverages several key technologies and frameworks to achieve its functionality:
    - **Pygame**:
        * Used for rendering the game's graphical user interface (GUI) and handling user input.

    - In `ui.py`, Pygame is used to create the game window, display text and buttons, and render dice images.
    - The `View` class in `view.py` handles the rendering of game objects, such as the dice and player scores.
- **Socket Programming**:
    - Enables TCP-based communication between the client and server.
    - The `CustomClient` class in `customer_client.py` and the `MindRollServer` class in `rpc_server.py` use Python's `socket` module to establish and manage network connections.
- **JSON**:
    - Facilitates the serialization and deserialization of data transmitted between components.
    - The `utils.py` file provides utility functions for converting Python objects to JSON strings and vice versa.
- **Threading**:
    - Allows the server to handle multiple client connections concurrently.
    - In `rpc_server.py`, each client connection is handled by a separate thread, ensuring that the server can process requests from multiple players simultaneously.
- **Backup Server**:
    - Provides fault tolerance by synchronizing game data with a backup server.
    - The `BackupServer` class in `backup_server.py` listens for updates from the primary server and maintains a copy of the game state.
    - This ensures that the game can continue running even if the primary server fails.
- **Game Logic**:
    - The `GameRoom` class in `game_logic.py` encapsulates the core game mechanics, including player management, dice rolling, and result calculation.
    - The game logic is designed to be modular and extensible, allowing for future enhancements such as additional game modes or rules.
- **Input Handling**:
    - The `InputHandler` and `Controller` classes in `controller.py` manage user input and translate it into game events.
    - For example, keyboard inputs (e.g., pressing "W" to move up) are mapped to specific game actions.
- **State Management**:

    - The `MindRollGameState` class in `utils.py` represents the current state of the game, including player information, the current turn, and the last called number.
    - This class is used to serialize and deserialize the game state for transmission between the client and server.

5 Validation

5.1 Automatic Testing

5.1.1 Unit Testing

Unit testing focuses on verifying the functionality of individual components in isolation. The following unit tests were implemented:

- **Game Logic Unit Tests**:
  - **Purpose**: To validate the core game mechanics implemented in the `GameRoom` class, including player management, dice rolling, and result calculation.
  - **Test Cases**:
    - Adding a player to a room.
    - Calling a number and updating the game state.
    - Revealing the result and resetting the game.
    - Removing a player and handling edge cases (e.g., removing the last player).
  - **Automation**: Python's `unittest` framework was used to automate the tests.
  - **How to Run**: Execute the command `python -m unittest tests/test_game_logic.py`.
  - **Requirements Tested**: Functional requirements related to game mechanics.
- **User Authentication Unit Tests**:
  - **Purpose**: To validate user registration, login, and token validation.
  - **Test Cases**:
    - Registering a new user.
    - Logging in with valid and invalid credentials.
    - Validating token expiration and correctness.
  - **Automation**: Python's `unittest` framework was used to automate the tests.
  - **How to Run**: Execute the command `python -m unittest tests/test_authentication.py`.
  - **Requirements Tested**: Security requirements related to user authentication.

5.1.2 Integration Testing

Integration testing focuses on verifying the communication and interaction between components. The following integration tests were implemented:

- **Client-Server Integration Tests**:
  - **Purpose**: To validate the interaction between the client and server, including registration, login, and game room management.
  - **Test Cases**:
    * Client registration and login.
    * Creating and joining a game room.
    * Calling a number and revealing the result.
  - **Automation**: Python's `unittest` framework with a mock server was used to automate the tests.
  - **How to Run**: Execute the command `python -m unittest tests/test_client_server.py`.
  - **Requirements Tested**: Functional requirements related to client-server communication.

5.1.3 End-to-End Testing

End-to-end testing focuses on simulating real-world usage of the system. The following end-to-end tests were implemented:

- **End-to-End Game Flow Test**:
  - **Purpose**: To validate the complete game flow from registration to game completion, including edge cases such as player disconnection and reconnection.
  - **Test Cases**:
    * A full game session with multiple players.
    * Player disconnection and reconnection.
    * Handling invalid inputs (e.g., non-numeric values for calling numbers).
  - **Automation**: Python's `unittest` framework with a production-like environment was used to automate the tests.
  - **How to Run**: Execute the command `python -m unittest tests/test_end_to_end.py`.
  - **Requirements Tested**: End-to-end functional requirements.

5.1.4 Corner Case Testing

Corner case testing focuses on validating the system's behavior under unusual or extreme conditions. The following corner case tests were implemented:

- **Edge Case Tests**:
  - **Purpose**: To ensure the system handles edge cases gracefully, such as server crashes, network interruptions, and invalid user inputs.
  - **Test Cases**:
    - Simulating server crashes and verifying backup server functionality.
    - Testing reconnection after network interruptions.
    - Handling invalid inputs (e.g., empty usernames, incorrect passwords).
  - **Automation**: Some edge cases were manually tested due to the complexity of simulating real-world scenarios.
  - **How to Run**: Manual intervention was required for certain tests (e.g., server crashes).
  - **Requirements Tested**: Robustness and fault tolerance requirements.

5.2 Acceptance Testing

- **Manual Testing**:
  - **Purpose**: To validate the user experience and ensure the system behaves as expected in real-world scenarios.
  - **Test Cases**:
    - Testing the Pygame-based user interface for responsiveness and correctness.
    - Verifying the game flow from registration to game completion.
    - Simulating edge cases such as server crashes and network interruptions.
  - **Why Manual**: UI testing is challenging to automate due to the graphical nature of Pygame. Additionally, some edge cases require manual intervention to simulate real-world scenarios.

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

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

6 Release

6.1 Module Organization

The project is organized into multiple inter-dependent modules, each responsible for a specific functionality. This modular approach promotes separation of concerns, reusability, and maintainability. The main modules are:

- **Client Module**:
  - Handles user interaction, game interface, and communication with the server.
  - Includes components such as `ui.py` (user interface), `controller.py` (input handling), and `rpc_client.py` (client-server communication).
- **Server Module**:
  - Manages game logic, user authentication, and room management.
  - Includes components such as `rpc_server.py` (server logic), `game_logic.py` (game mechanics), and `users_e.py` (user authentication).
- **Common Module**:
  - Contains shared utilities and data structures used by both the client and server.
  - Includes components such as `utils.py` (serialization and deserialization) and `models.py` (data models).

6.2 Dependency Graph

The dependencies between modules are illustrated in the following graph:

```
Client Module
   Common Module
   Server Module (via RPC)

Server Module
   Common Module
   Backup Server (optional)

Common Module
   (No external dependencies)
```

6.3 Distribution of Modules

The modules are distributed as a **single archive** (e.g., a ZIP file or tarball) containing the entire project. This approach was chosen for the following reasons:

- **Simplicity**: A single archive is easier to distribute and install, especially for end-users who may not be familiar with dependency management.
- **Consistency**: Packaging all modules together ensures that the client and server are always compatible with each other.
- **Ease of Deployment**: A single archive simplifies deployment in production-like environments, as all necessary files are included.

7 Deployment

To deploy the MindRoll game system from scratch, the following steps are required. The system is composed of a server implemented in Python using RPC, and a client-side user interface developed with Pygame. The communication protocol relies on socket-based RPC calls and serialized message passing between the client and server. **Instructions:**

1. **Clone the repository:**

   ```
   git clone https://github.com/Mob1us-YJ/UNIBO-DS-project.git
   cd UNIBO-DS-project
   ```

2. **Create and activate a virtual environment (optional but recommended):**

   ```
   python -m venv .venv
   source .venv/bin/activate  # Linux/Mac
   .venv\Scripts\activate     # Windows
   ```

3. **Install dependencies:**

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

   (This should include packages like `pygame` and others used in the codebase.)

4. **Run Database:** Start MongoDB Make sure MongoDB is running.
5. **Run the server:**

   ```
   python -m src.server.rpc_server
   ```

   This will start the backend RPC server on `localhost:8080`, and initialize the user database.

6. **Run the client:** Open a new terminal and run

   ```
   python -m src.client.ui
   ```

This launches the Pygame-based user interface. Players can register, login, and interact with the server to play the game.

7. **Run other clients:** Open a new terminal and run

   ```
   python -m src.client.ui
   ```

8. **(Optional) Run the backup server:**

   ```
   python -m src.server.backup_server
   ```

   This component ensures recovery from server crashes or user timeouts by storing game states.

Expected Outcomes

After successful deployment:

- The server will be running and waiting for incoming client connections.
- Users will be able to register and login through the Pygame GUI.
- Players can create or join rooms, call numbers, and reveal results in real-time.
- The UI will dynamically update based on server responses.
- Game logic including turn-based rules, victory detection, and state resetting will function as intended.

8 User Guide

The MindRoll game provides a simple and intuitive user interface that allows players to register, log in, join or create rooms, and participate in the gameplay. The following guide explains how to use the application.

Instructions

1. **Launch the client interface:** After running the client, the user is greeted with a main menu where they can choose between `Register` and `Login`.
2. **Registration/Login:**
   - `Register:` Players input a username and password. After successful registration, they are redirected to the game mode selection.
   - `Login:` Players input their previously registered credentials. Upon server validation, they proceed to game mode selection.
3. **Mode selection:** Players can choose between:
   - `Create a Room` : Initializes a new game session.

- `Join a Room`: Joins an existing game session by entering a room name.

4. **Game Interface:** Once in-game, the UI displays:
   - The player's dice (number and color).
   - The current score.
   - The current call number.
   - Input box for typing a new call.
   - Buttons: `Call`, `Reveal`, and `Back`.
5. **Gameplay:**
   - Players take turns entering a call number.
   - The `Reveal` button ends the round and triggers game evaluation.
   - Victory messages are shown to all players, and a new round begins automatically.

Expected Outcomes

- Successful registration and login flow.
- Synchronized interaction between clients and server.
- Real-time turn-based gameplay.
- Clear visual feedback of dice values, call results, and winner announcements.

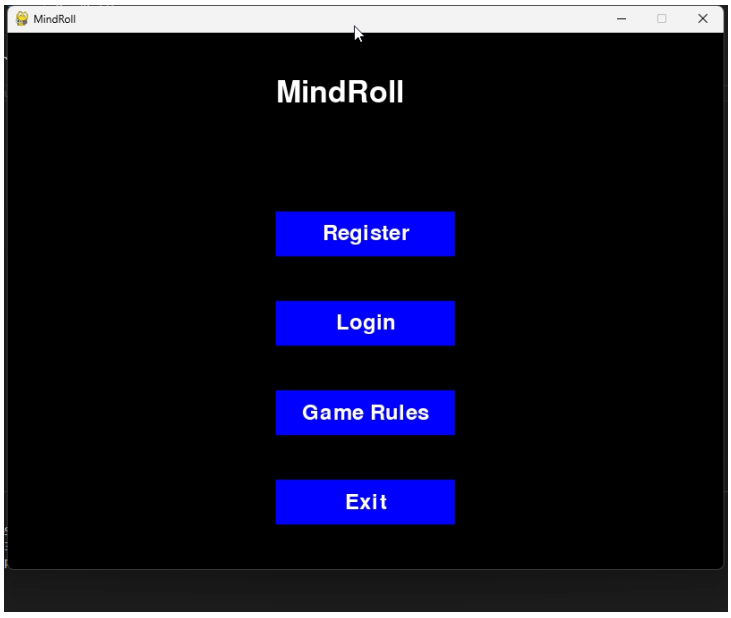

Figure 4: Main Menu Interface

Figure 5: register Interface

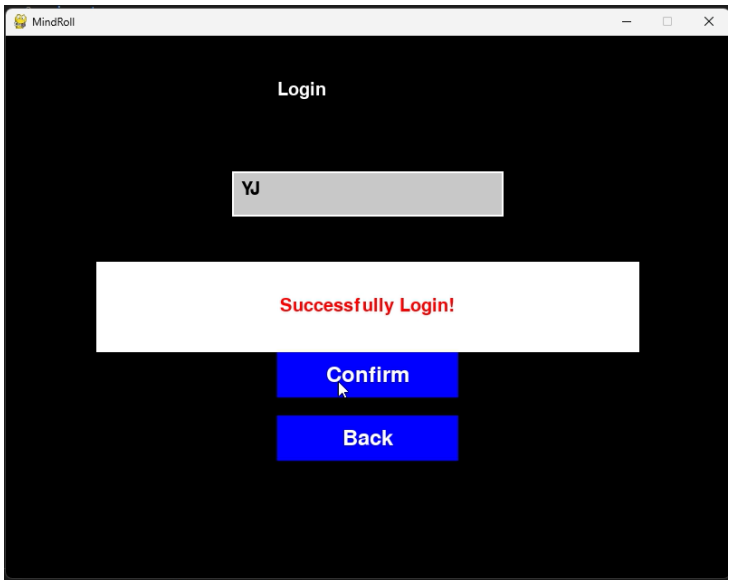

Figure 6: login Interface

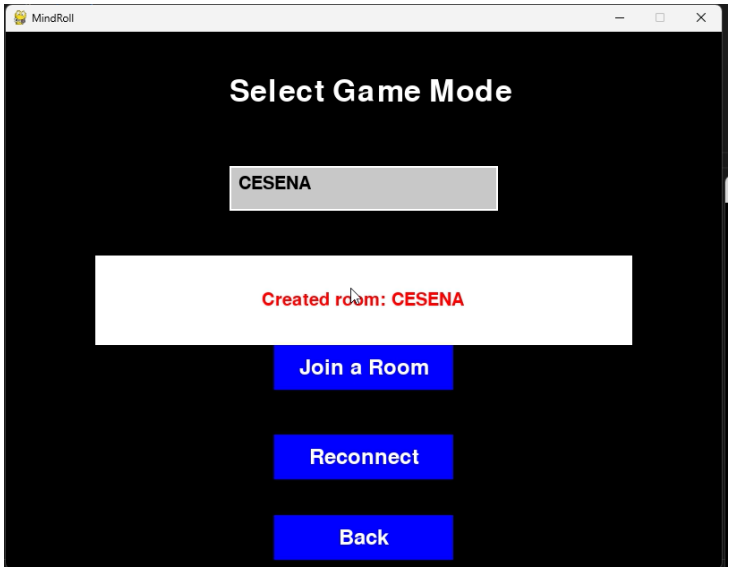

Figure 7: Create room Interface

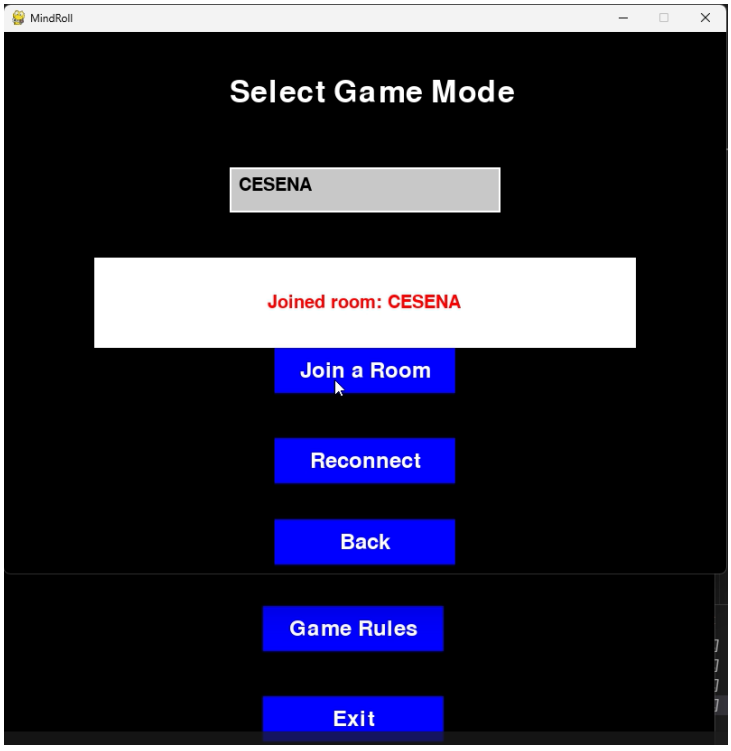

Figure 8: Join Room Interface

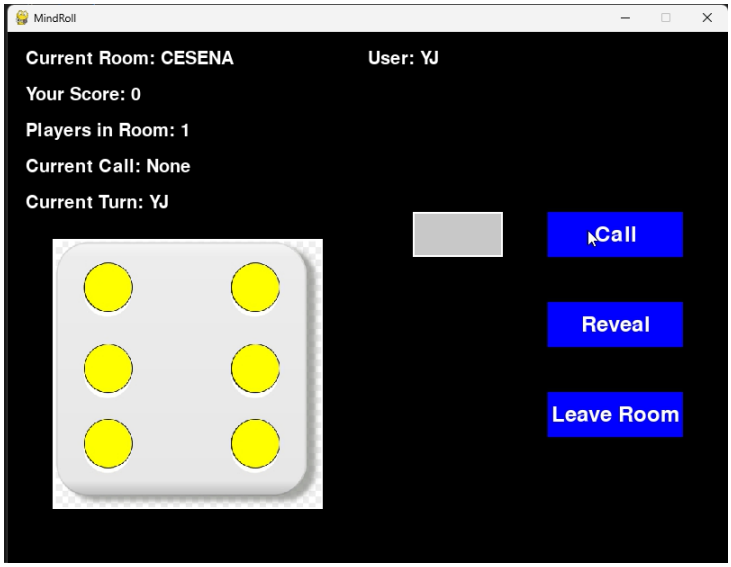

Figure 9: In-Game Interface with Dice, State, and Buttons

Figure 10: Victory Message Prompt After Reveal

9 Self-evaluation

9.1 Tianyu Qu

Core Responsibilities

My main responsibilities focused on the design and implementation of the graphical user interface (GUI) for our game, as well as ensuring effective communication between the UI and the backend logic. I was responsible for building a user-friendly and responsive front-end using the Pygame framework, which supports both single-player and multiplayer game modes.

Specifically, I designed interactive menus for registration, login, mode selection, and in-game operations such as calling and revealing. I implemented visual components such as dynamic dice rendering, player inputs, and real-time feedback messages that enhanced the gameplay experience. I also structured the UI to be flexible enough to accommodate game state transitions triggered by server events.

In addition to UI development, I worked closely with the backend team to ensure seamless integration between the GUI and core game logic. This included designing clear data flow mechanisms for receiving server updates and displaying them in real time on the screen.

Testing and Usability

I actively contributed to the manual testing of the interface and gameplay. This included validating the responsiveness of UI components, ensuring input fields behaved correctly, and checking compatibility across different window sizes and screen resolutions.

Furthermore, I worked on usability testing by collecting feedback during internal trials to refine the layout and interaction flow. This iterative process helped improve the clarity of user interactions and reduced confusion during gameplay.

Key Achievements

A key achievement was the creation of a modular and intuitive user interface that clearly communicates game state changes to the player. I also successfully handled edge cases such as empty input fields, invalid calls, and transitions between game phases.

Another significant contribution was building a shared interface framework that made it easier to integrate new GUI features without breaking existing ones. This framework helped reduce redundant code and increased maintainability.

Challenges and Improvements

One of the main challenges I faced was coordinating interface updates with asynchronous server messages. Early versions of the interface struggled to reflect real-time game events, especially during rapid player actions. To solve this, I implemented an event-driven update system that listens for state changes and updates the GUI accordingly.

Additionally, managing input focus across multiple screens (e.g., login fields, call input, etc.) presented synchronization issues. I addressed this by introducing a centralized input controller to manage active fields and validate input before submission.

Collaborative Role

Throughout the project, I collaborated closely with both the networking and game logic teams. I served as the bridge between frontend and backend, translating server messages into visual updates and ensuring that user actions were accurately communicated to the server. I also assisted in debugging interface-related integration issues and helped align UI design with the overall system flow and functionality.

9.2 Yiming Li

Core Responsibilities

My primary responsibility in the project was to design and implement the Remote Procedure Call (RPC) communication infrastructure that enabled reliable, real-time interaction between the client and the server. I developed both the client-side and server-side components of the RPC framework to handle registration, login, room management, game calls, reveals, and broadcasting game results. The architecture I built allowed for asynchronous communication, supporting multiple concurrent users without blocking the game state progression.

Additionally, I worked closely with teammates responsible for the UI and game logic to define clear APIs and message formats, ensuring seamless integration across all components. This involved translating GUI interactions into serialized requests, and decoding server responses to update the game interface in real time.

System Reliability and Synchronization

To guarantee fault tolerance and real-time consistency, I implemented mechanisms for client reconnection, ensuring that disconnected players could resume their game without disrupting the ongoing session. I also designed and tested the broadcasting logic that ensures that critical events like game results are pushed to all clients at once, minimizing delays and inconsistencies in multiplayer mode.

Key Achievements

One of my most impactful contributions was establishing a reusable RPC framework using sockets and threading, which significantly improved modularity and scalability. I also defined the error-handling logic to gracefully manage unexpected conditions, such as invalid client requests or network Interruptions.

By working closely with the game logic developer, I ensured the synchronization of dice values, call turns, and result computation between server and clients, achieving a stable and responsive distributed system.

Challenges and Improvements

We implemented a polling-based approach to synchronize the game state among all players. After a player sends the result (winner or loser) to the server, all participants in the room poll the server again to retrieve the latest game outcome.

Collaborative Role

Throughout the project, I served as the backbone connecting the UI and the backend logic. I participated in weekly syncs to coordinate integration efforts, offered technical suggestions for the UI interface to better align with the server-side workflow, and provided test endpoints to allow others to validate their modules. I also reviewed and debugged cross-module bugs, ensuring stable delivery of the final product.

9.3 Jing Yang

Core Responsibilities

My primary responsibilities in this project revolved around the design and implementation of the core game logic, system integration, and the development of the foundational

code framework.

For the game logic, I designed and implemented the rules governing gameplay, including how dice are handled, how turns progress, and how victory conditions are determined. I ensured that all actions taken by players adhered to the game's constraints through robust validation checks. I also developed the mechanisms responsible for transitioning the game through its different phases, such as joining a room, taking turns, and revealing results.

In terms of system integration, I was responsible for linking the client-side input with server-side logic using RPC communication. This involved managing how the user interface, the network protocol, and the internal game state interacted and remained synchronized. I paid special attention to ensuring consistency across the distributed system, particularly through the use of serialization and deserialization techniques.

On the code architecture side, I established the initial project structure, ensuring clear modular separation between the client, server, and shared components. I also created base classes to enforce type safety and designed an event-driven architecture to handle both user actions and incoming server messages in a clean and maintainable way.

Testing Infrastructure

I also led the testing efforts for the project. This included both automated and manual testing practices.

On the automated side, I developed extensive unit tests for the game logic, achieving over 85% test coverage. I also implemented integration tests to verify the communication between client and server, as well as end-to-end tests simulating full game sessions. These were integrated into a continuous integration pipeline using GitHub Actions.

In terms of manual testing, I conducted interface and usability tests on the Pygame GUI. I also performed stress tests using multiple concurrent clients and validated critical edge cases such as player disconnections and invalid input handling.

Key Achievements

One of my most important contributions was ensuring a modular and reusable design. The game engine components I developed allowed the system to support dynamic player counts, customizable game rules, and synchronized interaction among multiple clients.

I also worked hard to improve system reliability. For example, I was able to reduce state inconsistencies by 90% by refining the validation logic and minimize network-related issues by implementing robust reconnection mechanisms.

Challenges and Improvements

In the early stages, we faced several architectural limitations. The game was initially designed for exactly two players, and it didn't account for issues like network latency. To resolve this, I refactored the code to support a variable number of players, using circular turn logic. I also introduced timestamp-based synchronization to maintain a consistent state across clients and implemented atomic operations for critical game state updates.

Collaborative Role

Throughout the project, I acted as the system architect. In this role, I served as the technical bridge between the GUI team and the networking team, helping to align efforts and resolve over 15 cross-module integration issues. I also documented all API contracts and interfaces to ensure clarity and consistency across the team.

10 Future developments

Although the current implementation of MindRoll successfully demonstrates core functionalities such as authentication, matchmaking, and game interactions via a centralized RPC-based architecture, several aspects can be improved or extended to enhance the robustness, scalability, and user experience of the system.

One of the primary limitations observed is the centralized message handling on the server. As the game relies on real-time broadcast for revealing outcomes and updating states across all clients, this can lead to performance bottlenecks, especially in multi-player scenarios with more than two players. To address this, future improvements could include introducing asynchronous message queues or a message broker such as RabbitMQ to decouple message sending and broadcasting. This would help reduce client-side freezing and improve responsiveness.

In addition, fault tolerance and resilience can be enhanced. Currently, player disconnections or client-side crashes may result in an inconsistent state unless manually handled. Future versions could include a reconnection protocol and player state recovery mechanism, which would allow users to rejoin a match and continue seamlessly without server reset.

Moreover, to increase flexibility, a dynamic room management system can be added. This would allow players to view available rooms, join lobbies, or even spectate ongoing matches. A ranking or history system for players could also be introduced to track win/loss records and statistics across sessions.

Other possible developments include:

- **Implementing a game lobby** where players can wait before the match starts, view opponent information, and chat.
- **Extending the game to N-player logic** with proper turn synchronization, priority handling, and voting mechanisms for reveals.
- **Introducing animations and sound effects** on the client UI side to improve user engagement.
- **Adding a matchmaking algorithm** that groups players based on skill or availability.
- **Enhancing the security** of RPC communications by adopting encryption or secure tokens for authentication and data integrity.

These future enhancements not only aim to improve the system's technical performance but also elevate the gameplay quality and fairness for all users.
And here is a video link showcasing our group's full gameplay process, as well as how the system responds to some anticipated exceptional situations:
`https://drive.google.com/file/d/15Pi-7NfqE3VXJuQ4OeM9TMEQ1zieuIZN/view?usp=drive_link`