

**ADDIS ABABA UNIVERSITY**

**AAiT(SiTE) IT-Stream**

**Distributed System Programming Project**

**Design Documentation**

**Group Members** ……………………………….**ID Numbers**

1. Amanuel Belete…………………..…………….UGR/7564/13

2. Ayan Abas………………………………...........UGR/0999/13

3. Maria Tesfaye………………………..…..………UGR/9011/13

4. Natnael Dejene…………………………..……..UGR/4254/13

5. Yohannes Habtamu….……………………..… UGR/1581/13

**Submitted to**: Mr.Wondimagegn Desta

**Submission Date**: Feb 5, 2025

**Distributed System Design Documentation for *Crazy* Card Game**

**1. Overview**

The *Crazy* card game is an online multiplayer real-time game built using a **distributed system** architecture. The system supports autonomous agents (players) who can interact with shared resources (the game state) and perform real-time updates. The state of the system is distributed across multiple servers and clients, ensuring scalability and fault tolerance.

The main components of the system are:

* **Frontend (React)**: Provides the user interface for players to interact with the game.
* **Authentication Server(Go)**: Manages user authentication and sessions.
* **Game Servers(GO)**: Host the game logic and state for different rooms (each corresponding to a suit).
* **Database (MySQL)**: Stores user data and game state for persistence and recovery.

**2. System Requirements**

The system must meet the following key requirements based on the course's distributed systems objectives:

1. **Multiple Autonomous Agents**: The system should support multiple players interacting with shared game resources. Players (agents) compete in the card game and perform actions that update the game state in real-time.
2. **Distributed State**: The game state is distributed across the game servers. Each game room corresponds to a specific suit, with each room hosting up to three players. The state is stored in the MySQL database to ensure consistency and recovery in case of node failure.
3. **Centralized Authentication**: The only centralized component is the authentication server, which manages player login, user management, and session tracking.
4. **Robustness & Fault Tolerance**:
   * The system should continue operating even if a game server crashes.
   * The system must support state recovery, allowing players to resume their game even after a crash.
   * Each game server communicates with the database to persist game state, and if a failure occurs, the system can recover the last known state.
5. **Real-time Synchronization**: The game state must be synchronized across all players in real-time, ensuring that all players' actions are reflected instantly on their screens.
6. **Performance & Reliability**: The system should deliver low-latency performance and handle failure scenarios gracefully, ensuring continuous gameplay even when a node crashes.

**3. Architecture Design**

The architecture consists of the following components:

* **Client (React Frontend)**:
  + The player interacts with the game through a web interface.
  + The client is responsible for rendering the game UI, displaying cards, and sending player actions to the backend.
* **Authentication Server**:
  + The authentication server is a central service that handles user logins, session management, and authentication.
  + It maintains user credentials and ensures that only authenticated players can participate in the game.
* **Game Servers**:
  + Each game server is responsible for managing one game room (one suit) and its state.
  + The game logic (turn management, card actions, game flow) is implemented in the Go backend.
  + Each game server communicates with the MySQL database to store and retrieve game state for fault tolerance and recovery.
* **Database (MySQL)**:
  + The database stores persistent data about user accounts, session information, and the game state.
  + In case of a failure, the database allows the game to resume from the last consistent state.

**4. Interaction and Workflow**

* **Player Login**:
  1. The player logs in via the frontend, which sends a request to the authentication server.
  2. The authentication server checks the credentials and returns a session token if successful.
  3. The player is now authenticated and can select a game room.
* **Joining a Game Room**:
  1. The frontend sends a request to the backend (game server) to join a specific room.
  2. If the room is available, the player is added to the room, and the game server initializes the game state for that room.
  3. The game state is fetched from the MySQL database and sent to the client, ensuring that the client has the latest game state.
* **Game Play**:
  1. Players take turns performing actions like playing cards, drawing cards, or reversing the play direction.
  2. Each action is sent to the game server via the frontend, which processes the action and updates the game state.
  3. The game server updates the game state and sends it to all other players in the room in real-time.
  4. The game server also stores the updated state in the MySQL database for fault tolerance.
* **State Persistence and Recovery**:
  1. If a game server crashes, the game state is recovered from the database when the server restarts.
  2. The frontend can retrieve the last known state from the game server and continue the game without data loss.

**5. Fault Tolerance and Recovery**

* **Node Failures**:
  + If a game server crashes, it will recover its state from the MySQL database upon restart. This ensures that the game can continue seamlessly after the crash.
* **Database Failures**:
  + The MySQL database is the central point for persisting game states. It ensures that game state is not lost during node crashes. If the database becomes unavailable, players will not be able to make progress, but once the database is back online, the game state can be restored.

**6. Synchronization and Real-Time Updates**

* **Event-driven Architecture**:
  + The game servers utilize an event-driven architecture to ensure that player actions (e.g., playing a card, drawing cards) trigger real-time updates to all other players.
* **WebSockets or HTTP Streaming**:
  + For real-time synchronization, WebSockets is used to push updates to all players in a game room instantly.
  + For authentication HTTP.

**7. Testing and Evaluation**

* **Unit Testing**:
  + Each module (authentication, game server) is thoroughly tested using unit tests to ensure they function correctly under normal conditions.
* **Performance Testing**:
  + The system is tested under load to ensure that it can handle multiple players interacting in real time without performance degradation.

**8. Considerations for Distributed Systems**

* **Consistency**: The system uses a **strong consistency model**, ensuring that all players see the same game state at any given time.
* **Fault Tolerance**: The system is designed with redundancy at both the game server and database levels to ensure continuous operation in the event of a failure.
* **Scalability**: The architecture is scalable, allowing more game servers to be added to handle a larger number of players as needed.

**9. Conclusion**

The *Crazy* card game project is a fun and challenging implementation of a distributed system with an emphasis on fault tolerance, real-time synchronization, and robust performance. The system's design considers both the technical requirements of distributed systems and the gameplay experience, ensuring that it is both reliable and enjoyable for players.

