Vidyalankar Institute of Technology Accredited A+ by NAAC	**DEPARTMENT OF COMPUTER ENGINEERING**

Assignment No. 03

Semester	B.E. Semester VIII – Computer Engineering
Subject	Distributed Computing Lab
Subject Professor In-charge	Dr. Umesh Kulkarni
Assisting Professor	Prof. Prakash Parmar
Academic Year	2024-25

Student Name	Deep Salunkhe
Roll Number	21102A0014

Title: Designing a distributed system for a collaborative real-time multiplayer gaming platform.

Designing a Distributed System for a Collaborative Real-Time Multiplayer Gaming Platform

1. Introduction

A real-time multiplayer gaming platform requires an efficient distributed system to handle concurrent player interactions, maintain consistency, and ensure minimal latency. The choice of interprocess communication (IPC) mechanisms significantly impacts system performance and scalability. This document discusses the scenarios where Remote Procedure Call (RPC), Remote Method Invocation (RMI), and Message-Oriented Communication (MOC) are most effective. Additionally, we explore strategies for ensuring low latency through stream-oriented communication and synchronization using group communication mechanisms.

2. Interprocess Communication Mechanisms in Multiplayer Gaming

2.1 Remote Procedure Call (RPC)

Use Case: Client-Server Communication for Game Logic Execution

RPC allows clients (players) to invoke functions on remote game servers seamlessly, treating remote calls as local ones.

- **Example:** When a player interacts with an in-game object (e.g., opening a chest), the client sends an RPC request to the game server to execute game logic and return the result.
- **Advantages:**
    - Simplifies client-server interaction.
    - Enables efficient execution of synchronous game logic.
    - Supports cross-platform interoperability (e.g., gRPC for multi-language support).
- **Challenges:**
    - High network latency if overused for frequent updates.
    - May require additional mechanisms for fault tolerance and retries.

2.2 Remote Method Invocation (RMI)

Use Case: Distributed Object Management in Multiplayer Worlds

RMI extends RPC by allowing objects to invoke methods on remote objects, making it useful for object-oriented game components.

- **Example:** A real-time strategy (RTS) game where different servers manage distinct game regions, and units moving between regions invoke remote methods on different servers.
- **Advantages:**
    - Object-oriented approach aligns well with game entity management.
    - Supports dynamic remote object invocation.
- **Challenges:**
    - Language-dependent (Java-based systems typically use RMI).
    - Higher overhead than RPC, making it less suitable for frequent real-time updates.

2.3 Message-Oriented Communication (MOC)

Use Case: Event-Driven Asynchronous Player Actions

Message-Oriented Middleware (MOM) enables loosely coupled communication using message queues or topics, ensuring scalability.

- **Example:** In a battle royale game, player status updates (health, location, actions) are broadcasted asynchronously using a message broker (e.g., Kafka, RabbitMQ, or MQTT).
- **Advantages:**
    - Asynchronous and event-driven nature ensures non-blocking communication.
    - Supports scalability by decoupling components.
    - Ensures resilience with message persistence and replay mechanisms.
- **Challenges:**
    - Potential message delivery delays (must be minimized for real-time interactions).
    - Requires robust message ordering and consistency mechanisms.

3. Ensuring Low Latency with Stream-Oriented Communication

Real-time gaming demands ultra-low latency communication, making stream-oriented communication essential.

3.1 WebSockets for Persistent Bidirectional Communication

- **Example:** Used for player movement updates, attack commands, and chat functionalities.
- **Advantages:**
    - Persistent TCP connections reduce handshake overhead.
    - Enables real-time push notifications to all players.
- **Challenges:**
    - Requires handling dropped connections efficiently.
    - May need load balancing for large-scale deployments.

3.2 UDP for High-Speed, Low-Latency Data Transmission

- **Example:** FPS and racing games use UDP for transmitting position and velocity updates.
- **Advantages:**
    - Faster than TCP as it avoids handshaking and retransmissions.
    - Suitable for real-time physics and movement synchronization.
- **Challenges:**
    - No built-in reliability mechanisms; requires additional error correction.
    - Packet loss may lead to inconsistencies in player positions.

4. Maintaining Consistency and Synchronization Using Group Communication Mechanisms

In multiplayer environments, all players must perceive consistent game states despite network delays.

4.1 Leader Election for State Synchronization

- **Example:** A primary game server acts as the authoritative source for game state synchronization.
- **Advantages:**
    - Ensures a single source of truth for game logic.
    - Reduces conflicts from concurrent updates.
- **Challenges:**
    - Failover mechanisms must be in place in case of leader failure.

4.2 Distributed Locking for Shared Resource Access

- **Example:** Locking mechanisms prevent simultaneous updates to in-game resources (e.g., two players claiming the same loot).
- **Advantages:**
    - Ensures data consistency across distributed nodes.
    - Prevents race conditions in critical operations.

- **Challenges:**
    - Lock contention can cause delays.
    - Requires efficient distributed coordination (e.g., using Zookeeper or Raft consensus algorithm).

4.3 Vector Clocks for Causal Order Synchronization

- **Example:** Used in collaborative gameplay scenarios where player actions must be causally ordered (e.g., turn-based games).
- **Advantages:**
    - Ensures events are applied in the correct order.
    - Helps resolve conflicts arising from concurrent actions.
- **Challenges:**
    - Additional overhead in tracking timestamps.
    - May not be ideal for ultra-fast real-time interactions.

5. Conclusion

A distributed real-time multiplayer gaming platform requires a combination of IPC mechanisms tailored to different components of the system:

- **RPC:** Best for synchronous client-server interactions.
- **RMI:** Ideal for distributed object management.
- **MOC:** Supports event-driven, scalable communication.
- **Stream-Oriented Communication (WebSockets & UDP):** Ensures low latency for real-time gameplay.
- **Group Communication Mechanisms:** Maintain consistency and synchronization across players.

By leveraging these mechanisms appropriately, the system can provide a seamless, responsive, and scalable multiplayer experience while maintaining data consistency and real-time synchronization across players.