**Experiment No.10**

**Aim:**

Mini Project (in Accordance with Assignment 1)

Experiment 10 A) Miniproject Demonstration

Experiment 10 B) Assignment based on Miniproject

**Experiment 10 A)**

**Theory:**

This project demonstrates the implementation of a distributed multiplayer Tic Tac Toe game using Java Remote Method Invocation (RMI), a core technology for enabling remote communication in Java-based systems. RMI allows objects running in different Java Virtual Machines (JVMs) to interact seamlessly, making it ideal for building client-server architectures. In this application, a central game server handles all logic and state management, while remote clients interact by invoking server-side methods like making moves or retrieving the game board.

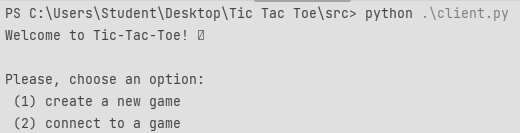
Each client connects to the server through the RMI registry, obtains a reference to the shared game object, and participates in a turn-based session. The server keeps track of player symbols, move sequences, and win conditions, ensuring game integrity. RMI’s built-in support for multithreading enables concurrent sessions, allowing the server to serve multiple games or players simultaneously.

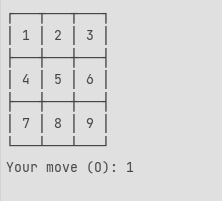
The project highlights several core concepts of distributed systems including synchronous communication, persistent connections, call semantics, and concurrency control. Additionally, it showcases how layered communication (application, stub/skeleton, reference, transport) supports modular and scalable application design. Through this interactive game, students gain hands-on experience in building robust, fault-tolerant networked applications while understanding fundamental concepts of remote procedure calls and distributed software architecture.

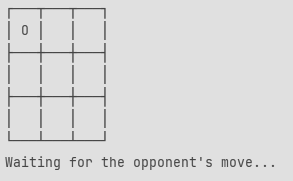
**Conclusion:**

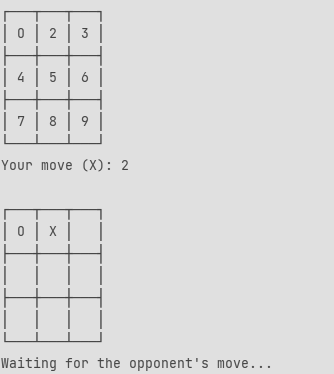
The implementation of Tic Tac Toe using Java RMI successfully demonstrates the principles of distributed computing, client-server architecture, and remote method invocation in a real-time multiplayer environment. By employing a stateful communication model with persistent server activation, the system ensures consistent game state management, low latency, and smooth interaction between players across different machines. The layered communication protocol abstracts the complexity of network communication, making the application modular and maintainable. Furthermore, synchronous request/reply interactions and at-least-once call semantics reinforce reliable gameplay. The server’s ability to handle multiple simultaneous requests and concurrent games adds to the system’s scalability and robustness. Overall, this project provides a practical and interactive foundation for exploring remote procedure calls and distributed system design.

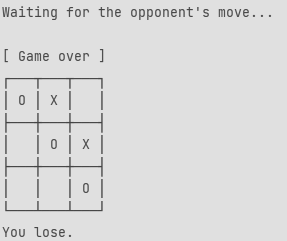
**Code and Output:**

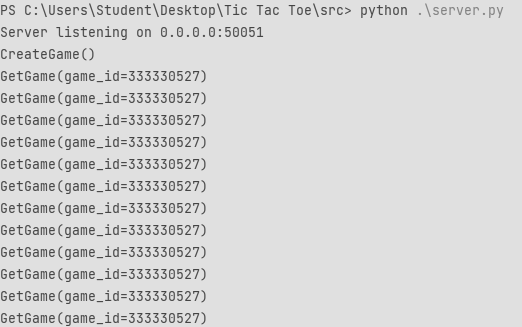
****

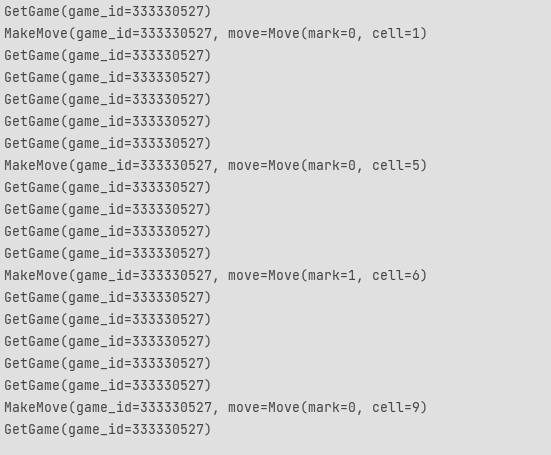
****

****

****

****

****

****

**Experiment No. 10 B)**

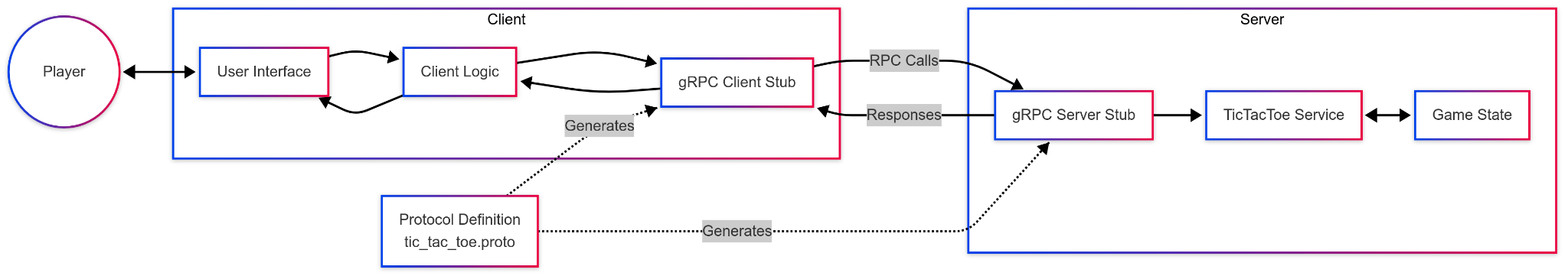
**Project Title: Tic Tac Toe Using gRPC**

### **1. Introduction**

This experiment demonstrates the development of a multiplayer Tic Tac Toe game using **gRPC** to facilitate communication between distributed components. The system follows a client-server architecture where each player interacts through a separate client interface. These clients send and receive game data by making remote procedure calls to a centralized **TicTacToe Server** using gRPC. The server is responsible for processing game logic, maintaining the game state, and ensuring valid moves. It also communicates with a backend **Game Storage** module to persist data. The use of **Protocol Buffers (.proto)** allows for efficient serialization and strong typing of data structures shared between clients and server. This experiment aims to provide hands-on understanding of building real-time, scalable applications using gRPC, and highlights how distributed systems handle synchronization, data consistency, and client-server communication in an interactive environment.

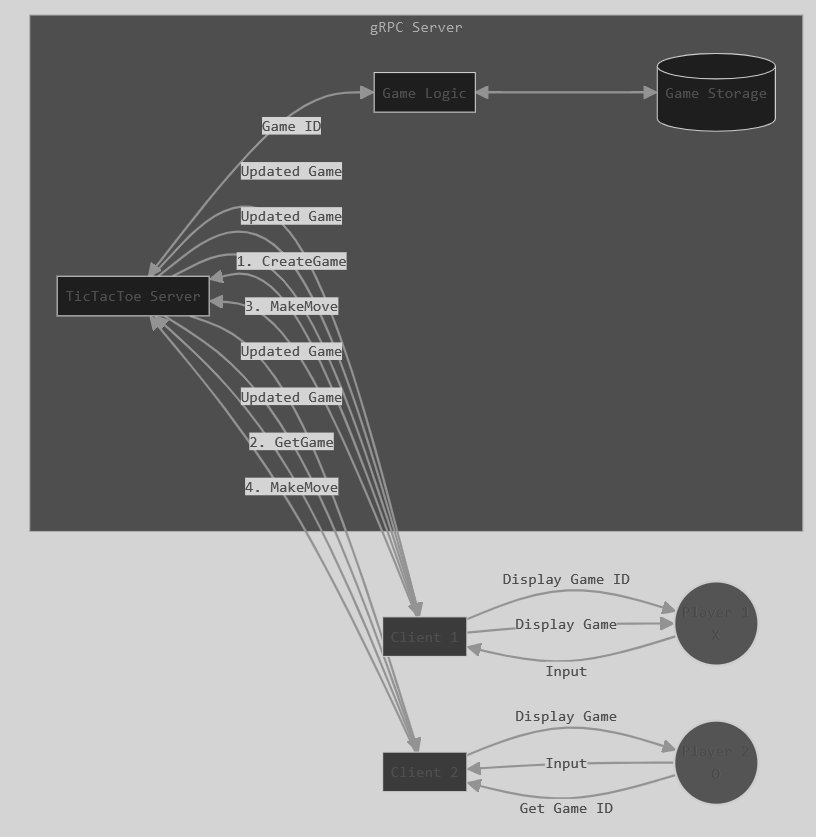
### **2. Design (Block Diagram)**

**2A. High-Level System Architecture for Tic Tac Toe (gRPC-Based)**



This architecture diagram illustrates the client-server interaction in a Tic Tac Toe game using gRPC. The **client** consists of the *User Interface*, *Client Logic*, and a *gRPC Client Stub*, which is auto-generated from the tic\_tac\_toe.proto protocol definition. The **server** includes a *gRPC Server Stub*, *TicTacToe Service*, and *Game State* handler. RPC calls and responses facilitate communication between client and server for turn-based gameplay.

**2B. Tic Tac Toe Game Interaction Flow Using gRPC**



This diagram depicts the interaction between two players (Player 1 (X) and Player 2 (O)), their respective clients, and the gRPC server managing the game. Each player provides input through their client, which communicates with the central TicTacToe Server. The server processes game updates via Game Logic and stores data in Game Storage. Updates are sent back to both clients to reflect the current game state, ensuring a synchronized multiplayer experience through gRPC-based communication.

### **3. Working Concept and Layered Architecture of Communication Protocol**

The core working concept of this experiment revolves around building a **multiplayer Tic Tac Toe game** using a **gRPC-based client-server communication model**. The goal is to enable real-time interaction between two players across a network, with game logic centralized on a remote server and accessed through Remote Procedure Calls (RPCs). This is made efficient and structured through **gRPC (Google Remote Procedure Call)**, which uses **Protocol Buffers (protobufs)** as its interface definition language (IDL).

#### **Working Concept**

The application consists of three main components: the **Client (Player Interface)**, the **gRPC Server (TicTacToe Server)**, and the **Game Storage**. Players interact with the user interface to send game inputs (e.g., move positions) via the client module. The client logic sends these inputs to the gRPC client stub, which then makes an RPC call to the server stub. The server processes the game logic—checking for valid moves, wins, or draws—updates the game state, and sends back the updated game status. The server also interacts with a persistent game storage layer to manage ongoing game sessions.

#### **Layered Architecture**

The communication protocol in this system can be viewed as layered, inspired by the OSI model, although simplified and adapted for application-level communication:

1. **Application Layer**:  
   * Consists of user interfaces and client logic.
   * Handles player input, move validation on the client-side, and display of game state.
2. **RPC Layer (Transport Abstraction Layer)**:  
   * gRPC handles all inter-process communication using HTTP/2 for transport.
   * Supports features like multiplexing, streaming, and flow control.
   * Protocol Buffers define the structure of the requests and responses, providing a language-agnostic format.
3. **Server Application Layer**:
   * The gRPC server receives RPC calls and invokes appropriate service methods.
   * The server implements the defined methods (e.g., StartGame, MakeMove, GetGameState) from the .proto file.
   * It processes logic and returns responses to the clients via the gRPC stub.
4. **Persistence Layer**:  
   * Handles storing and retrieving game sessions.
   * Ensures continuity and state management even if players disconnect.

This layered architecture not only decouples concerns but also enhances modularity, making it easier to maintain, scale, and extend. By using gRPC, the system achieves low-latency communication and strong contract-based interaction between clients and server.

### **4. Communication Model Justification**

### **4A. Client/Server Design: Stateful**

The TicTacToe project adopts a stateful client-server design to ensure continuity and synchronization throughout the multiplayer game session. In a stateful design, the server maintains the game state for each active session, storing information such as the players’ identities, the current board status, turn order, and win/draw conditions. This allows both clients (players) to stay in sync, ensuring that every move updates the shared game state on the server.

Each client connects to the server using gRPC, a high-performance RPC framework that facilitates real-time communication. When a player initiates or joins a game, a unique session is created or joined, and the server maintains this session’s context in memory. Subsequent client actions, like making a move or querying the game status, are processed based on this stored context.

This design ensures reliable coordination between clients. For instance, if Player 1 makes a move, the server updates the board and informs Player 2 accordingly. Since the server is aware of the entire session state, it can validate turns, detect winning conditions, and prevent illegal moves. The stateful nature provides a seamless and interactive experience, making it ideal for multiplayer games where persistent and synchronized data is essential.

### **4B. Server Creation Semantic**

The server creation semantic in this experiment refers to the structure, logic, and setup involved in initializing and running the gRPC-based TicTacToe server. The server is built using the gRPC framework, which allows the server to expose defined services through remote procedure calls. The service definitions are written in a .proto file using Protocol Buffers (protobuf) syntax. This file specifies the available methods (e.g., StartGame, MakeMove, GetGameState) and the message types exchanged between clients and the server.

During server creation, the .proto file is compiled using the protoc compiler to generate server-side code in the target language (e.g., Python, Go, or Java). This generated code provides the base classes and interfaces for the server implementation. The actual game logic is implemented by extending these base classes, where the server methods are defined to process incoming requests and send responses.

Once implemented, the gRPC server is instantiated by binding it to a specific IP address and port. It listens for incoming RPC calls from clients, processes them using the implemented logic, and returns the game status accordingly. The server is multithreaded, allowing it to handle multiple game sessions or moves concurrently for a seamless multiplayer experience.

### **4C. Persistent and Transient Communication**

In the Tic Tac Toe project, communication between the client and server is handled using **gRPC**, which supports both **persistent** and **transient** communication models depending on the game phase and context.

**Persistent communication** occurs through **bi-directional streaming RPC**, where both clients (players) maintain an open channel with the server. This persistent connection allows the server to **stream real-time updates** to both clients throughout the game, including opponent moves, game state changes, and win/loss notifications. Since the game relies on constant synchronization, this persistent stream ensures minimal delay and a smooth user experience.

On the other hand, **transient communication** is used during **initial setup stages**, such as when a player sends a request to create or join a game. These are **single-request–single-response** operations where the connection is closed after the server responds. For example, the CreateGameRequest and JoinGameRequest messages in the .proto file represent transient interactions used to initiate a game session.

Combining both models, the system achieves an efficient balance: transient communication handles **lightweight one-time operations**, while persistent streams **maintain game continuity**. This hybrid design enhances scalability, responsiveness, and the real-time nature of multiplayer gameplay.

### **4D. Synchronous / Asynchronous Communication**

The Tic Tac Toe project efficiently leverages both **synchronous** and **asynchronous** communication models to ensure smooth gameplay and real-time interaction between two players via gRPC.

**Synchronous communication** is employed during the initial phase of the game, where the client makes blocking requests to the server—for instance, when a player sends a CreateGameRequest or JoinGameRequest. In these cases, the client waits for a response from the server before proceeding, ensuring that the game session is properly initialized and players are successfully matched before gameplay begins.

**Asynchronous communication** comes into play once the game session is active. The gameplay uses **bi-directional streaming RPCs**, allowing the server and clients to exchange game states (like player moves and game-over notifications) in real-time without blocking. This ensures that both players receive updates simultaneously, enhancing the responsiveness and fluidity of the multiplayer experience.

By combining synchronous communication for setup and asynchronous streaming for gameplay, the system ensures **reliability and interactivity**. Players experience minimal latency, and the game state remains consistent across clients. This dual-mode communication strategy is critical for maintaining a real-time, competitive, and user-friendly gaming environment.

### **4E. Call Semantics**

In distributed systems, **call semantics** define how remote procedure calls (RPCs) behave in terms of message delivery guarantees. In the context of the Tic Tac Toe game, which uses **gRPC**, the call semantics play a crucial role in ensuring a consistent and reliable communication flow between the client and server.

This project primarily implements **"at-most-once"** and **"at-least-once"** call semantics based on the operation type:

* **At-most-once** semantics are used for critical actions like CreateGame and JoinGame, where duplicating a request could lead to inconsistent game state or multiple game rooms. gRPC handles this by assigning unique request IDs and using retries with idempotency checks to avoid duplicated processing.
* **At-least-once** semantics may appear in bi-directional streams (like during gameplay moves), ensuring that messages are delivered even if retransmission is needed due to a network glitch. However, this may occasionally lead to duplicate messages, so the game logic must handle idempotency (e.g., ignoring repeated moves).
* **Exactly-once** semantics, while ideal, are hard to guarantee over unreliable networks without complex protocols. Instead, the system handles retries, acknowledgments, and state-checking mechanisms to simulate similar reliability.

This careful use of call semantics ensures consistent, real-time interaction while minimizing the risk of data loss or corruption during gameplay.

### **4F. Concurrent Access to Multiple Servers**

In a real-time multiplayer game like Tic Tac Toe using gRPC, enabling **concurrent access to multiple servers** enhances scalability, reliability, and performance. While the basic setup may involve a single gRPC server handling all requests, production-level systems often distribute traffic across multiple servers using **load balancing** and **replication** strategies.

Each player’s client communicates with a server instance to create or join a game. Through **stateless service design**, multiple backend gRPC servers can serve user requests concurrently. Game sessions are tracked via unique game IDs, and centralized storage (like an in-memory database or Redis) maintains consistent state across all servers. This ensures that a move made on one server is reflected immediately if the next request is routed to another server.

To manage **race conditions and consistency**, server instances use locking mechanisms or atomic operations when updating shared resources like game state. Additionally, message queues or pub/sub systems can be implemented to synchronize data between instances, preventing desynchronization.

This architecture allows for **high availability**, where one server’s failure doesn’t impact overall gameplay, and **horizontal scaling**, where servers can be added during peak usage to balance the load across all incoming client connections efficiently.

### **4G. Serving Multiple Requests Simultaneously**

In a real-time multiplayer Tic Tac Toe game, the gRPC server must be capable of **handling multiple player requests at the same time** to ensure smooth gameplay and user experience. This is achieved through **concurrent request handling**, a core feature of gRPC servers, which are typically multithreaded and event-driven.  
When two players are connected in a session, they may send moves or status checks simultaneously. The gRPC server assigns each incoming request to a separate thread or coroutine, depending on the programming model used (e.g., asyncio in Python, goroutines in Go). This allows multiple clients to interact with the server concurrently without blocking each other’s operations.  
Internally, the server uses **thread-safe data structures** or synchronization mechanisms (like mutexes or locks) to handle shared resources such as the game state. For example, if two clients try to update the same game board simultaneously, the server ensures consistency by processing one update at a time in a controlled manner.  
This design not only supports multiple ongoing matches but also ensures responsiveness and minimizes latency. By serving multiple requests in parallel, the system becomes scalable and reliable, offering a better user experience in competitive multiplayer settings.

### **4H. Reducing Per-Call Workload of Servers**

To ensure efficiency and scalability in a gRPC-based Tic Tac Toe multiplayer game, it's essential to **reduce the server's per-call workload**. Each client request—such as making a move or joining a game—triggers a remote procedure call (RPC) to the server. If each call required complex processing or repeated resource-intensive operations, the server would quickly become a bottleneck.  
One way to reduce this workload is through **in-memory caching** of frequently accessed data, such as the current game state or active player list, instead of querying persistent storage for each request. This greatly improves response time and reduces computation overhead.  
Another approach is **preprocessing requests on the client side**. For example, validating the move before sending it to the server ensures only legitimate data is processed, saving the server from unnecessary checks.  
Additionally, using **streaming RPCs** for ongoing communication (instead of new calls for every move) minimizes connection overhead. For long sessions, the server can maintain a lightweight session token or game instance identifier, avoiding repeated authentication or setup.  
By minimizing redundant tasks and optimizing communication patterns, the system becomes more responsive and can handle a higher number of simultaneous users without server overload.

### **4I. Reply Caching of Idempotent Remote Procedures**

In distributed systems like the gRPC-based Tic Tac Toe multiplayer game, idempotent remote procedures—those that return the same result when called multiple times with the same parameters—can benefit significantly from reply caching. Caching the server’s response to such calls reduces processing overhead and improves system responsiveness.  
For example, if a player repeatedly requests the current state of the game board without any change in moves, the server can return a cached reply instead of recomputing or re-fetching the state from memory or storage. Since the result remains the same until a new move is made, this caching is both valid and efficient.  
This caching mechanism can be implemented using simple in-memory data structures like hash maps, where the key is a combination of the request parameters (e.g., game ID and move count), and the value is the cached response. When the game state changes (due to a new move), the corresponding cache entry is invalidated.  
By caching replies to idempotent procedures like getGameStatus() or getCurrentPlayer(), the server handles repeated requests more efficiently, reducing latency and improving scalability—especially useful under high-concurrency gameplay scenarios.

### **4J. Proper Selection of Timeout Values**

In a real-time multiplayer game like the gRPC-powered Tic Tac Toe application, selecting appropriate **timeout values** for client-server communication is essential to ensure a responsive and seamless user experience. Timeout values dictate how long a client should wait for a server response before considering the request failed.  
If the timeout is set **too short**, legitimate responses might be discarded due to network latency or brief server slowdowns, leading to unnecessary retries or game disruptions. Conversely, **too long a timeout** can delay failure detection and freeze the UI, frustrating the player and reducing gameplay interactivity.  
For example, during a makeMove() request, a 2-3 second timeout might be ideal—quick enough to detect unresponsive behavior but lenient enough for minor delays. On the other hand, periodic requests like getGameStatus() can tolerate slightly longer timeouts since they don’t block critical operations.  
gRPC supports **per-call timeout settings**, allowing the client to specify timeout durations for different RPCs based on their urgency. This fine-tuning helps maintain a balance between responsiveness and reliability.  
Proper timeout management also plays a role in **resource optimization** on both client and server sides, avoiding prolonged connections and freeing resources for active users in a competitive game environment.

### **4K. Proper Design of RPC Protocol Specification**

Designing a proper **Remote Procedure Call (RPC) protocol specification** is crucial for ensuring clarity, efficiency, and scalability in distributed systems like the gRPC-based Tic Tac Toe game. A well-structured protocol defines clear communication semantics between the client and server, handling game creation, player moves, and state synchronization effectively.  
In this project, the .proto file serves as the core of the protocol specification. It includes message definitions like CreateGameRequest, MoveRequest, GameState, and services such as GameService with clearly defined RPC methods like CreateGame, JoinGame, MakeMove, and GetGameStatus. Each RPC is purpose-specific, reducing ambiguity and simplifying the client-server interactions.|  
Properly defining input/output message formats ensures that data such as player IDs, move positions, and game outcomes are consistently structured and easily serialized. The use of enums for game status and results improves type safety and minimizes interpretation errors.  
Versioning the protocol specification also supports backward compatibility and smooth future upgrades. Moreover, clear documentation within the .proto file allows developers to understand the expected behaviors, reducing development time and integration errors.  
Ultimately, a well-designed RPC protocol forms the foundation of reliable communication, enabling smooth, synchronized gameplay across multiple clients in a real-time multiplayer environment.