**Practical No. 1: Implement a Client/Server using RPC/RMI**

**Aim:**

To develop a distributed Java application using Remote Method Invocation (RMI) to simulate a client-server communication model, enabling a client to invoke methods on a remote object hosted on a server.

**Theory:**

In traditional applications, function calls and data access happen within the same memory space. In distributed systems, components may reside on different physical machines. To facilitate such distributed computing, **Remote Procedure Call (RPC)** and **Remote Method Invocation (RMI)** are used.

**🔹 Remote Procedure Call (RPC)**

* A protocol that allows a program to cause a subroutine to execute in another address space (commonly on another computer).
* RPC abstracts the network communication and makes remote procedure calls appear like local ones.

**🔹 Remote Method Invocation (RMI)**

* Java’s native implementation of RPC but designed for object-oriented systems.
* Allows invoking methods of remote Java objects.
* Built on top of Java Object Serialization.
* Automatically handles communication, object conversion, and marshalling/unmarshalling of parameters.

**📌 RMI Architecture Components:**

1. **Remote Interface:** Defines the methods that can be called remotely.
2. **Remote Object:** Implements the remote interface.
3. **RMI Registry:** Binds remote objects so clients can look them up.
4. **Stub and Skeleton:**
   * **Stub (Client Side):** Acts as a proxy for the remote object.
   * **Skeleton (Server Side):** Receives method calls and passes them to the actual remote object (older versions of Java; now handled internally).

**Conclusion:**

The RMI-based implementation demonstrates a foundational aspect of distributed systems: **inter-machine communication** through remote object invocation. RMI simplifies development by:

* Abstracting lower-level network protocols.
* Supporting object serialization and remote interaction.
* Encouraging modular, object-oriented design for distributed applications.

**Practical No. 2: Simulate Interprocess Communication using a Multithreaded Application**

**Aim:**To simulate interprocess communication (IPC) using threads in Java, demonstrating data exchange, synchronization, and coordination between concurrently executing units within a single process.

**Theory:**

**🔹 Interprocess Communication (IPC)**

In operating systems, IPC is the mechanism that allows processes to communicate and synchronize with each other. While IPC generally involves separate processes, in this practical, we **simulate IPC using multithreading**, where **multiple threads** within the same process share resources and communicate efficiently.

**🔹 Multithreading vs Multiprocessing:**

* **Multiprocessing**: Uses multiple independent processes, each with its own memory.
* **Multithreading**: Uses threads (lightweight processes) that share memory space.

**🔹 Why Multithreading for IPC Simulation?**

* Threads can access shared data structures easily.
* Communication is faster because there’s no need for heavy context switching.
* Still requires careful **synchronization** to avoid **race conditions** and **deadlocks**.

This practical demonstrates how multithreaded programming in Java can effectively simulate interprocess communication. The use of synchronized, wait(), and notify() methods provides insight into:

* **Thread synchronization**
* **Safe shared resource access**
* **Real-time communication coordination**

**Practical No. 3: Implement a Program for Group Communication (LO1)**

**Aim:**

To implement a socket-based group communication system using multithreading in Java, where multiple clients can exchange messages with each other through a centralized server.

**Theory:**

**🔹 What is Group Communication?**

Group communication refers to the **exchange of messages between a group of participants**, typically over a network. It is a key requirement in:

* Collaborative applications (e.g., group chats)
* Distributed systems (e.g., consensus protocols)
* Real-time gaming
* Conference systems

Instead of one-to-one messaging, **group communication** delivers a message from one sender to **multiple receivers**.

**🔹 Technical Concepts Involved:**

1. **Socket Programming:**
   * Uses TCP/IP for establishing client-server communication.
   * Enables bi-directional data flow.
2. **Multithreading:**
   * Each client runs on a separate thread, enabling concurrent communication.
   * The server listens continuously and spawns new threads for each new client.
3. **Broadcasting:**
   * The server forwards a received message to **all other connected clients**.
4. **Thread Synchronization:**
   * Synchronizing access to the shared list of client sockets is necessary to prevent concurrent modification errors.

**🔹 Real-World Analogy:**

Imagine a WhatsApp group: each member sends a message, and all members receive it. That’s group communication in action.

**Implementation:**

We’ll create three components:

1. A **server** to manage connected clients and handle broadcasting.
2. A **client** that sends and receives messages.
3. Proper **threading** to ensure each client works concurrently.

**Conclusion:**

This group communication program effectively simulates a **multi-user chat room** using core concepts of networking and concurrency in Java. It provides valuable insights into:

* Thread handling in server-client architecture
* Managing shared resources across threads
* Designing systems that scale to multiple users

These principles form the backbone of modern communication systems such as Slack, Discord, and video conferencing tools.

**Practical No. 4: Implementation of a Load Balancing Algorithm**

**Aim:**To implement a load balancing algorithm that distributes workload among multiple servers or processes to optimize resource use, minimize response time, and avoid overload of any single resource.

**Theory:**

**What is Load Balancing?:**Load balancing is the process of **evenly distributing incoming network traffic or computation load across multiple servers or resources**. It ensures that no single node is overwhelmed, and system performance remains efficient and reliable.

**🔹 Why is Load Balancing Important?**

* Prevents **bottlenecks** by redistributing workload.
* Enhances **system availability** and **fault tolerance**.
* Reduces **latency** and improves **throughput**.
* Scales applications horizontally (more servers = more capacity).
* **🔹 Use Case in Distributed Systems:**
* Web applications, cloud services, and database systems all rely on load balancing to ensure consistent performance. Even in local applications, task distribution can simulate load balancing logic for parallel processing.

This practical demonstrates the **Round Robin Load Balancing Algorithm** effectively distributing computational jobs among multiple servers. It reinforces key distributed system principles:

* Efficient task assignment
* Balanced resource utilization
* Modular architecture for scalability

Understanding load balancing not only aids in building efficient distributed applications but also strengthens your concepts in cloud computing, network management, and backend systems.

**Practical No. 5: Simulate the Functioning of Lamport’s Logical Clock (LO2)**

**Aim:**

To simulate the working of **Lamport’s Logical Clock algorithm** to establish a partial ordering of events in a distributed system and demonstrate the concept of causality among processes.

**Theory:**

**🔹 Background:**

In a **distributed system**, there’s no global clock to synchronize all processes. So, it’s hard to determine the order of events. To solve this, **Lamport’s Logical Clock** was introduced by Leslie Lamport in 1978.

It helps in assigning **logical timestamps** to events, allowing processes to determine the order of execution and causality.

**🔹 Lamport’s Logical Clock Rules:**

1. **Each process maintains a local logical clock**.
2. **Before an event** (internal or send), the process **increments its clock**.
3. **When sending a message**, it includes its clock value.
4. **On receiving a message**, the receiver sets its clock to:

This ensures **“happened-before” relation** (→) is respected.

**🔹 Types of Events:**

* **Internal event**: Occurs within a single process.
* **Send event**: Sending a message to another process.
* **Receive event**: Receiving a message from another process.

**🔹 Real-World Analogy:**

Imagine multiple users chatting in a group where everyone’s watches are not in sync. Each message carries a timestamp, and users adjust their watch if someone else’s timestamp is ahead. This is similar to Lamport clocks.

**Conclusion:**

This simulation of **Lamport’s Logical Clock** demonstrates how events in a distributed system can be ordered **without relying on a synchronized clock**. It introduces the concept of **causality** and **partial ordering** that is crucial for building fault-tolerant distributed applications.

Understanding Lamport clocks is fundamental in distributed databases, blockchain, event-driven systems, and algorithms like distributed snapshots and vector clocks.

**Practical No. 6: Implement an Election Algorithm**

**Aim:**

To implement a **process election algorithm** in a distributed system to select a **coordinator (leader)** among multiple processes. The aim is to ensure that even if the current coordinator fails, a new one is elected without human intervention.

**Theory:**

**🔹 What is an Election Algorithm?**

In distributed systems, an election algorithm is used when:

* A **coordinator (leader)** process crashes or becomes unreachable.
* Other processes must **elect a new coordinator** dynamically to continue managing resources or synchronization.

Election algorithms help maintain system stability and coordination in a **fault-tolerant and decentralized** way.

| **Algorithm Name** | **Key Idea** |
| --- | --- |
| **Bully Algorithm** | Highest ID process takes over. Lower ID processes “inform” higher ones. |
| **Ring Algorithm** | Processes are arranged in a logical ring. Election message circulates. |

**🔹 Working of Bully Algorithm:**

1. Any process can initiate an election if it notices the coordinator has failed.
2. It sends **ELECTION** messages to **all processes with higher IDs**.
3. If **no higher process responds**, it becomes the coordinator.
4. If a higher process responds, that process takes over the election.
5. Eventually, the process with the **highest ID becomes the coordinator** and announces itself.

**Implementation:**

We simulate a system of n processes, each with a unique ID. One of them is chosen to fail, and another initiates the election.

**Conclusion:**

The **Bully Algorithm** helps in maintaining **coordination and fault tolerance** in distributed systems. When the coordinator fails, any active process can start an election to ensure system reliability.

Key learnings:

* Distributed systems must handle failures autonomously.
* The process with the **highest priority (ID)** becomes the new leader.
* Election algorithms are foundational for **cloud orchestration**, **blockchain consensus**, and **cluster management systems** like Kubernetes.