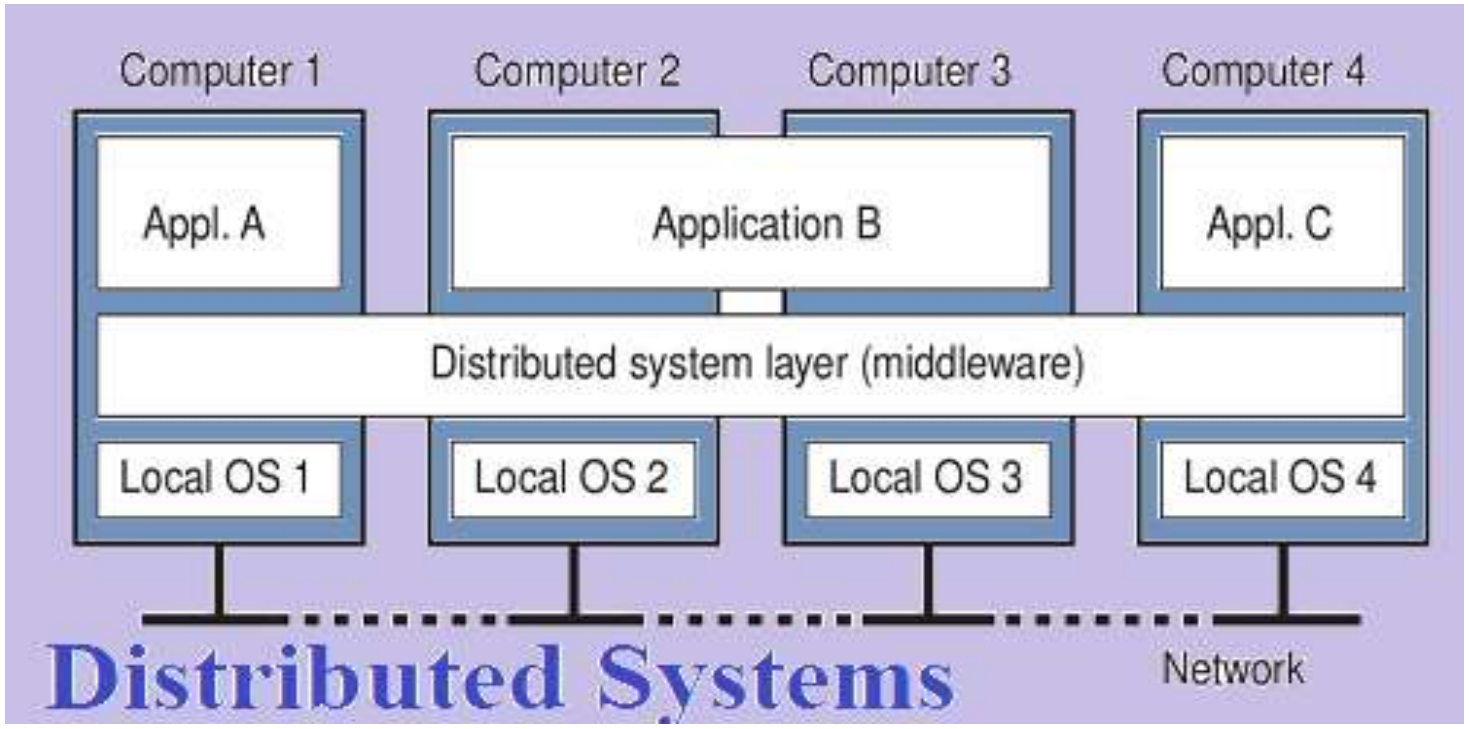
Unit I: Introduction to Distributed Systems

Important Topics : System Models , IPC(RPC,RMI)

Distributed System

- A distributed system is a collection of independent computers that appear to the users of the system as a single coherent system. These computers or nodes work together, communicate over a network, and coordinate their activities to achieve a common goal by sharing resources, data, and tasks.

Distributed System



Characteristics of Distributed Systems

- Resource Sharing
- Concurrency
- Scalability
- Fault Tolerance
- Transparency (Access, Location, Replication, etc.)

Examples of Distributed Systems

- Client-Server Model (e.g., Web Browser ↔ Web Server)
- Peer-to-Peer Model (e.g., BitTorrent)
- Grid Computing (e.g., SETI@home)
- Cloud Computing (e.g., AWS, Google Cloud)

Advantages of Distributed Systems

- Improved performance via parallelism
- Resource and data sharing
- High availability and fault tolerance
- Scalability and modular growth

System Models

- System models help us understand, design, and analyze distributed systems by providing structured views. Generally, they are divided into
- Architectural Models
- Fundamental Models

System Models

- A system model describes how the components of a distributed system are organized, interact, and communicate.
- Since distributed systems involve multiple computers connected via a network, models are needed to:
- Understand structure.
- Handle failures.
- Manage communication & security.

System Models

- Architectural Models: Client-Server, Peer-to-Peer, Layered
- Fundamental Models: Interaction, Failure, and Security Models

Architectural Models

- These models describe the structure of the system — how components are placed and interact.
- **Types of Architectural Models**
- **Client–Server Model**
  - Clients request services, servers respond.
  - Example: Web browser (client) ↔ Web server.
- **Peer-to-Peer (P2P) Model**
  - No fixed client/server roles; all nodes act as both.
  - Example: BitTorrent, blockchain.
- **Tiered Model (n-tier architecture)**
  - Multiple layers (Presentation, Logic, Database).
  - Example: Web apps (frontend, backend, database).
- **Hybrid Models**
  - Mix of client-server and P2P.
  - Example: Skype (supernodes + clients).

Fundamental Models

- These describe the basic properties and issues of distributed systems.(delays, failure, security threats)
- **Types of Fundamental Models**
- **Interaction Model**
  - Defines how processes communicate.
  - Considers **latency, bandwidth, failure of messages**.
- **Failure Model**
  - Describes possible failures in distributed systems:
    - **Crash failure**: process stops.
    - **Omission failure**: message lost.
    - **Timing failure**: response too early/late.
    - **Arbitrary failure**: arbitrary/incorrect results.
- **Security Model**
  - Focuses on **threats and defenses** in communication and data.
  - Example: eavesdropping, denial of service, spoofing, encryption.

Networking and Internetworking in the context of Distributed systems

Aspect	**Networking (within a network)**	**Internetworking (across networks)**
Scope	Single LAN / local network	Multiple networks interconnected
	Switches, hubs	Routers, gateways
Protocols	Ethernet, Wi-Fi, ARP	TCP/IP, UDP, BGP, ICMP
Addressing	MAC address, private IP	Public IP, global addressing
Example	A cluster of servers in a lab	The Internet (connecting global nodes)

Interprocess Communication (IPC) in Distributed Systems

- Interprocess Communication (IPC) the mechanisms that allow processes to exchange data and synchronize their actions.
- In a single machine (centralized system), processes can communicate through shared memory or message passing.
- In a distributed system, processes run on different machines, so IPC is more complex because <u>there’s no physically shared memory communication relies mainly on message passing via network.</u>

Characteristics of Inter-process Communication in Distributed Systems

There are mainly five characteristics of inter-process communication in a distributed environment/system.

- **Synchronous System Calls**: In synchronous system calls both sender and receiver use blocking system calls to transmit the data which means the sender will wait until the acknowledgment is received from the receiver and the receiver waits until the message arrives.
- **Asynchronous System Calls**: In asynchronous system calls, both sender and receiver use non-blocking system calls to transmit the data which means the sender doesn't wait from the receiver acknowledgment.
- **Message Destination:** A local port is a message destination within a computer, specified as an integer. A port has exactly one receiver but many senders. Processes may use multiple ports from which to receive messages. Any process that knows the number of a port can send the message to it.
- **Reliability :** It is defined as validity and integrity.
- **Integrity:** Messages must arrive without corruption and duplication to the destination.

Distributed Objects

- In a distributed system, different processes may run on separate machines connected via a network.
- A distributed object is an object whose state and methods can be accessed across process or machine boundaries, as if it were a local object.
- The main goal: location transparency – the client program doesn’t care whether the object is local or remote.

Interprocess Communication

- Message Passing: Send/Receive across network
- Shared Memory: Memory mapped access

Communication paradigms

How entities communicate in a distributed system, three types of communication paradigm

- Interprocess communication;
- Remote invocation;
- Indirect communication.

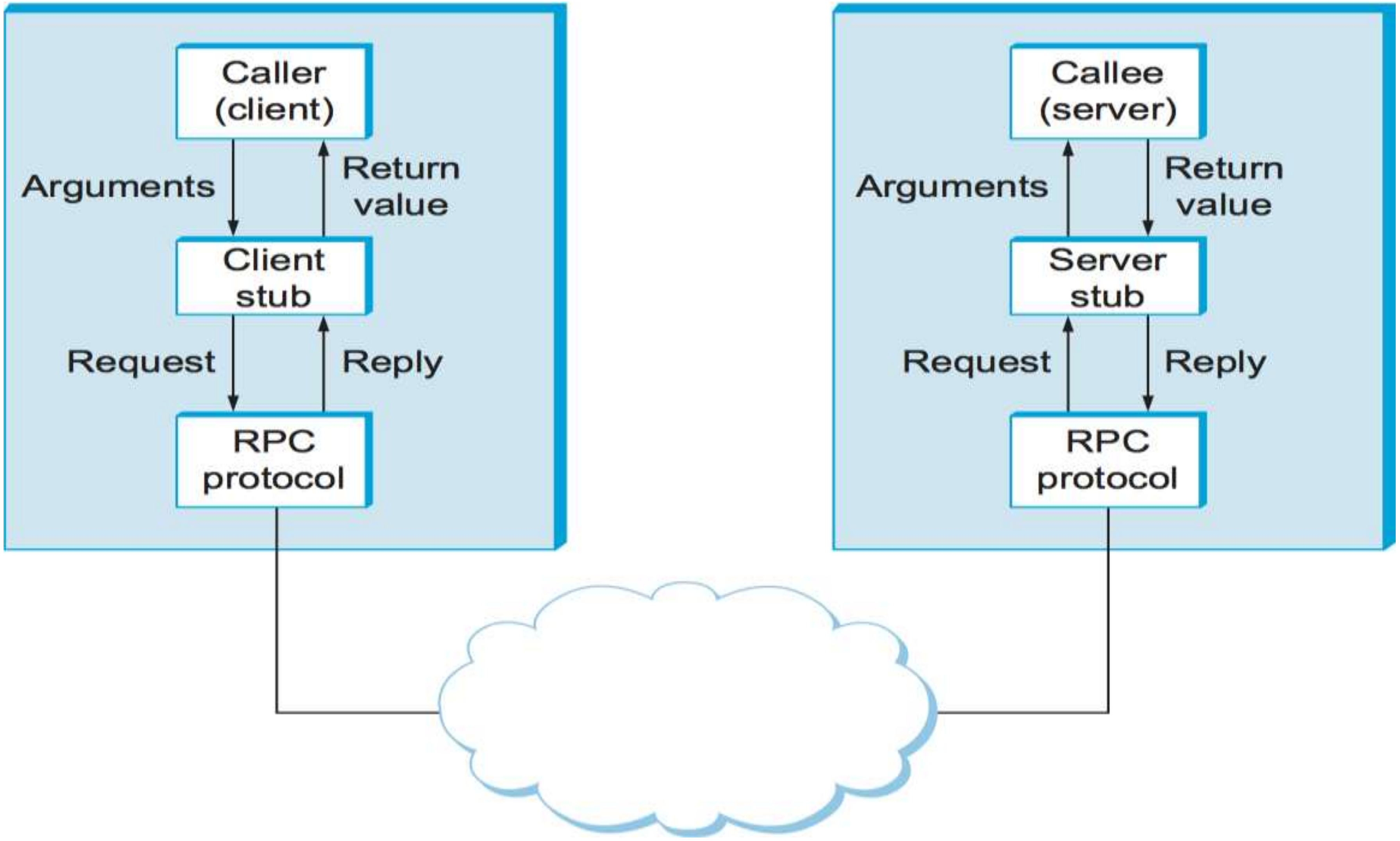
Remote Procedural Call

- An RPC aims at hiding most of the intricacies of message passing, and is ideal for client-server applications
- Paper by Birrell and Nelson (1984) introduced a completely different way of handling communication

Remote Procedural Call

- A Remote Procedure Call (RPC) is a communication protocol that allows a program to execute a procedure (function) on a remote system (server) as if it were running locally on the client.
- The client program calls a procedure, but instead of executing locally, the request is sent over a network to a server.
- The server executes the procedure and sends the result back.
- The process is designed to make remote interaction transparent to the programmer
- **Stub:** A helper code that hides the complexity from the programmer. On the client side, it converts function calls into messages (marshalling/unmarshalling) and works with the runtime to connect to the server.

Remote Procedural Call



Working Principle of RPC

- **Client Stub Creation:**
  The client code calls a procedure through a stub (a small piece of code that represents the remote procedure locally).
- **Marshalling of Parameters:**
  The client stub marshals (packs) parameters into a message to send over the network.
- **Transport & Communication:**
  The packed message is sent via a transport protocol (like TCP or UDP) to the server.
- **Server Stub:**
  The server stub unmarshals (unpacks) the message to retrieve parameters and then calls the actual procedure on the server.
- **Execution & Response:**
  The server executes the requested function, marshals the result, and sends it back to the client.
- **Client Receives Result:**
  The client stub receives and unmarshals the result, passing it to the client application as if it was a local call.

- **Remote Procedure Call (RPC)** allows a program to execute a procedure on a remote machine as if it were local. Normally, RPC is synchronous , the client sends a request and waits for the server to respond.
- But in **event-driven systems**, sometimes the client should not be blocked while waiting. **That’s where events and notifications come in.**

Events in RPC

- **Event** is an action or change in state (e.g., a file update, a new message arrival, completion of a task).
- In RPC, events are used to trigger remote procedures.
- Instead of polling (asking repeatedly), the client can register to be informed when an event occurs.
- Example: A stock trading app may raise an event when a stock price changes; the server notifies the subscribed clients.

Notifications in RPC

- **Notification** is a lightweight RPC where the client just sends information to the server (fire-and-forget) without expecting a reply.
- Useful for one-way communication.
- Saves resources because no need to block/wait for response.
- Example: A logging service where multiple applications send log data as notifications to a central server.

Working Principle

Events

- The client subscribes to an event at the server.
- When the event occurs, the server sends a callback (remote call back to client).
- The client’s handler processes it.

Notifications:

- The client sends a request to the server.
- The server executes it but does not send back a result.
- Used when acknowledgment is not required.

Remote Method Invocation

- **Remote Method Invocation (RMI)** is a Java API that allows an object residing in one Java Virtual Machine (JVM) to invoke methods of an object running on another JVM (possibly on a different machine).
- It is used for distributed computing where objects can communicate across networks, while hiding the complexity of sockets and networking from the programmer.

Working Principle of RMI

Client–Server Model

- RMI works on the client–server architecture.
- The **client** calls a method of a remote object as if it is local.
- The **server** hosts and provides access to the remote object.

Steps in RMI Communication

- **Remote Interface**: Defines the methods that can be invoked remotely.
- **Stub (Client-side Proxy)**: Acts as a gateway on the client side. It forwards the method call to the remote object.
- **Skeleton (Server-side Proxy)**: Receives the request from the stub, invokes the actual method on the server object, and returns the result. (In modern Java versions, skeleton is handled internally.)
- **RMI Registry**: A special service that keeps track of remote objects and allows clients to look them up by name.
- **Communication**: Uses TCP/IP and Java Remote Protocol (JRMP) for data transfer.

Working Principle of RMI

Execution Flow

- Client looks up the remote object in the RMI Registry.
- Stub sends the method call request (with parameters) across the network.
- Skeleton receives the request, calls the actual method, and sends back the result.
- Stub receives the result and gives it to the client program.

RMI Architecture

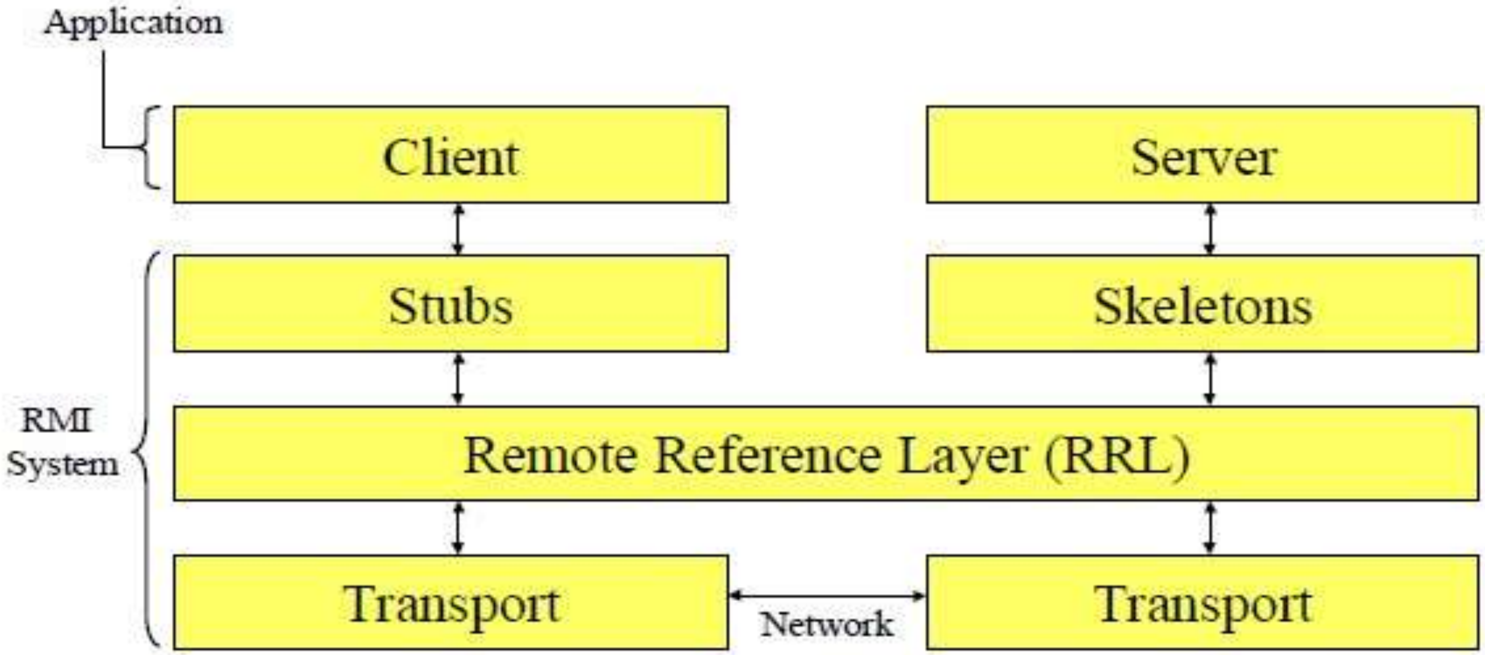


Figure 2. RMI Architecture Overview

Aspect	RPC (Remote Procedure Call)	RMI (Remote Method Invocation)
Definition	Allows calling a procedure on a remote machine as if it were local.	Allows invoking methods on remote Java objects across JVMs.
Abstraction Level	Procedure/function-based.	Object-oriented (method + object-based).
Language Dependency	Language-independent (C, C++, Python, etc.).	Java-specific (works only with JVMs).
Communication Data Types	Handles primitive data types; complex objects require manual serialization.	Supports automatic object serialization (can pass objects as arguments/return values).
Binding	Usually static (fixed procedure signatures).	Dynamic (stub & skeleton classes, runtime binding).
Middleware/Implementation	Implemented via ONC RPC, XML-RPC, JSON-RPC, gRPC, etc.	Implemented via java.rmi package.
Use Cases	Microservices, distributed systems, cross-language communication.	Distributed Java applications, EJB, RMI-based servers.
Flexibility	Can be used between heterogeneous systems (different languages/platforms).	Limited to Java ecosystem.

Summary & Key Takeaways

- Distributed systems = networked systems with shared resources
- Client-server, P2P, Grid, and Cloud architectures
- Communication via message passing or shared memory
- Java RMI simplifies remote object interaction in Java