UNIT3: Processes
UNIT4: Communication

Yuba Raj Devkota

Distributed System | NCCS

Unit 3. Processes **6 Hrs.**

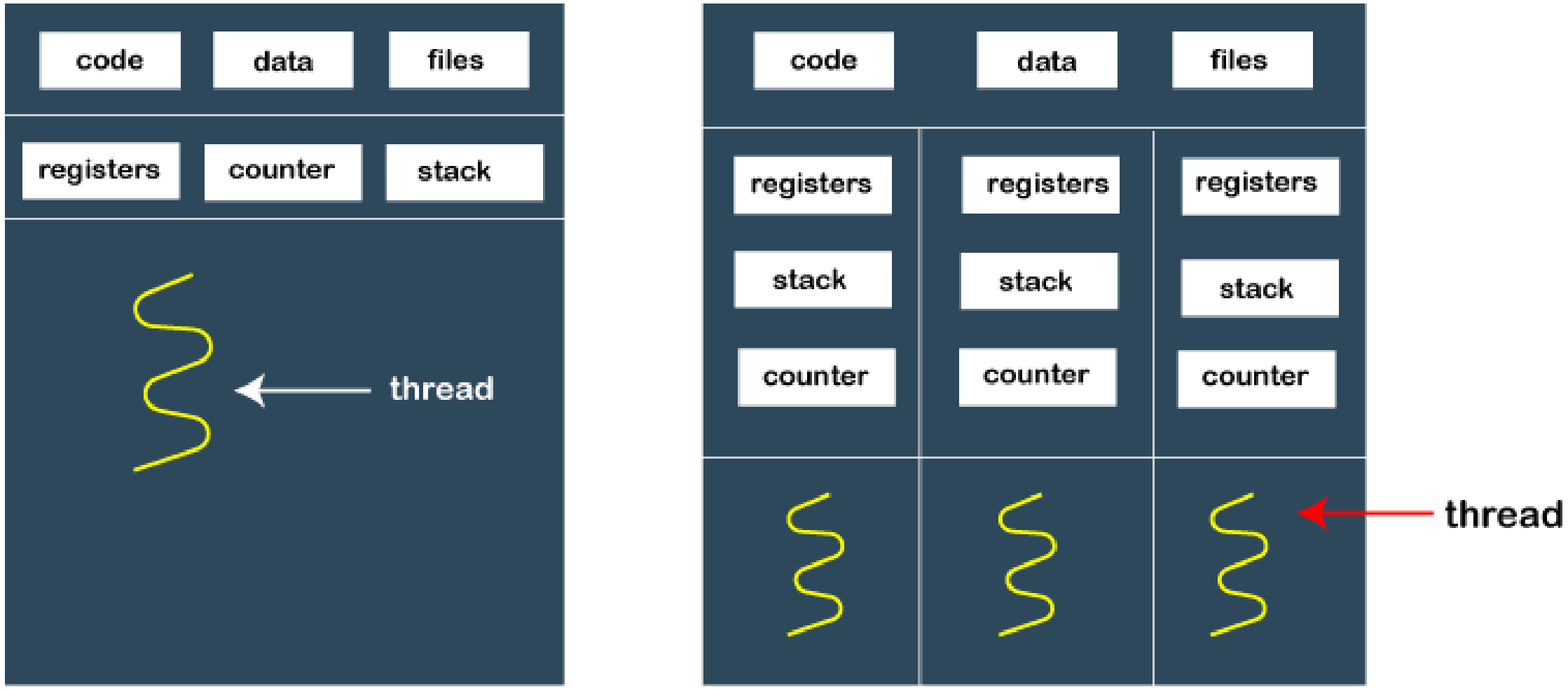
3.1 Threads
3.2 Virtualization
3.3 Clients
3.4 Servers
3.5 Code Migration

Unit 4. Communication **5 Hrs.**

4.1 Foundations
4.2 Remote Procedure Call
4.3 Message-Oriented Communication
4.4 Multicast Communication
4.5 Case Study: Java RMI and Message Passing Interface (MPI)

What is Process?

- **Processes** are isolated units with their own memory and resources, suitable for running independent programs.
- A **Process** is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system, a process can be made up of multiple threads of execution that execute instructions concurrently.
- Key characteristics of a process include:
  - **Isolation**: Processes are isolated from each other. Each process has its own memory space and resources.
  - **Resource Allocation**: Processes are allocated separate resources by the operating system, including memory, file handles, and CPU time.
  - **Execution**: A process executes in its own address space and has at least one main thread of execution.
  - **Communication**: Inter-process communication (IPC) mechanisms are used for processes to communicate with each other since they do not share memory space.



Single-threaded process

Multi-threaded process

What is Thread?

- **Threads** are lightweight units of execution within a process that share the same memory space, allowing for parallel execution and efficient resource usage within a single application.
- A **Thread** is the smallest unit of processing that can be scheduled by an operating system. Threads are sometimes called lightweight processes because they share the same memory space and resources of their parent process.
- Key characteristics of a thread include:
  - **Shared Resources**: Threads of the same process share the same memory space and resources (e.g., file descriptors, global variables).
  - **Concurrent Execution**: Multiple threads within the same process can execute concurrently, allowing for parallelism within a process.
  - **Lighter Context Switching**: Switching between threads is generally faster and more efficient than switching between processes because threads share the same memory space.

Aspect	Process	Thread
Definition	A process is an independent program in execution.	A thread is the smallest unit of execution within a process.
Memory Space	Has its own memory space; separate address space.	Shares memory space with other threads in the same process.
Resource Sharing	Does not share resources with other processes.	Shares resources with other threads of the same process.
Isolation	Processes are isolated from each other.	Threads are not isolated; they share data and resources.
Communication	Requires Inter-process Communication (IPC) mechanisms.	Direct communication is possible since threads share memory.
Overhead	Higher overhead due to the need for separate memory allocation.	Lower overhead; less memory allocation required.
Context Switching	More expensive and slower due to the need to switch memory maps.	Less expensive and faster due to shared memory.
Creation Time	Takes more time to create compared to threads.	Takes less time to create compared to processes.
Stability	A crash in one process does not affect other processes.	A crash in a thread can potentially crash the entire process.
Example Use Case	Running different applications (e.g., a web browser and a text editor).	Performing multiple tasks within the same application (e.g., multiple tabs in a web browser).

Why Process and Thread are useful in Distributed System?

Processes in Distributed Systems

1. **Isolation and Fault Tolerance**: Processes provide isolation between different components of a distributed system. If one process crashes, it does not affect other processes. This isolation enhances fault tolerance and makes the system more robust.
2. **Resource Allocation**: Processes allow for controlled allocation of resources. Each process can be given specific resources, which can be managed independently by the operating system.
3. **Security**: Processes run in separate memory spaces, which helps in maintaining security boundaries. Sensitive data in one process cannot be directly accessed by another process.
4. **Scalability**: Distributed systems often need to scale across multiple machines. Processes can be distributed across different nodes in a network, enabling horizontal scaling and better utilization of distributed resources.
5. **Inter-process Communication (IPC)**: Processes use IPC mechanisms (such as message passing, sockets, or remote procedure calls) to communicate with each other. This is fundamental in distributed systems where processes running on different machines need to coordinate and exchange data.

Threads in Distributed Systems

1. **Concurrency**: Threads allow multiple tasks to be performed concurrently within the same process. This is essential in distributed systems for handling multiple client requests simultaneously, improving responsiveness and throughput.
2. **Resource Sharing**: Threads within the same process share memory and resources, making it easier to manage shared state and data. This is useful in scenarios where tasks need to collaborate or share intermediate results.
3. **Efficiency**: Threads are more lightweight compared to processes, with lower overhead for creation and context switching. This efficiency is beneficial for distributed systems where performance and resource utilization are critical.
4. **Parallelism**: Threads can run on multiple processors or cores, enabling parallel execution of tasks. This parallelism is important for distributed systems that need to process large volumes of data or perform compute-intensive tasks.
5. **Responsiveness**: In interactive distributed applications, threads can be used to handle different parts of the application (e.g., user interface, network communication, and background processing) simultaneously, improving overall responsiveness.

Use Cases in Distributed Systems

1. **Web Servers**: A web server might use multiple processes to handle different clients and multiple threads within each process to manage various tasks such as handling requests, processing data, and communicating with databases.
2. **Microservices**: Each microservice in a distributed system can run as a separate process. Within each microservice, multiple threads can be used to handle incoming requests concurrently, perform background jobs, or manage asynchronous tasks.
3. **Database Servers**: Database systems often use threads to handle multiple client connections and queries simultaneously, improving throughput and responsiveness.
4. **Big Data Processing**: Distributed computing frameworks like Hadoop and Spark use processes to distribute tasks across multiple nodes in a cluster. Each node may run multiple threads to process chunks of data in parallel, enhancing performance and scalability.

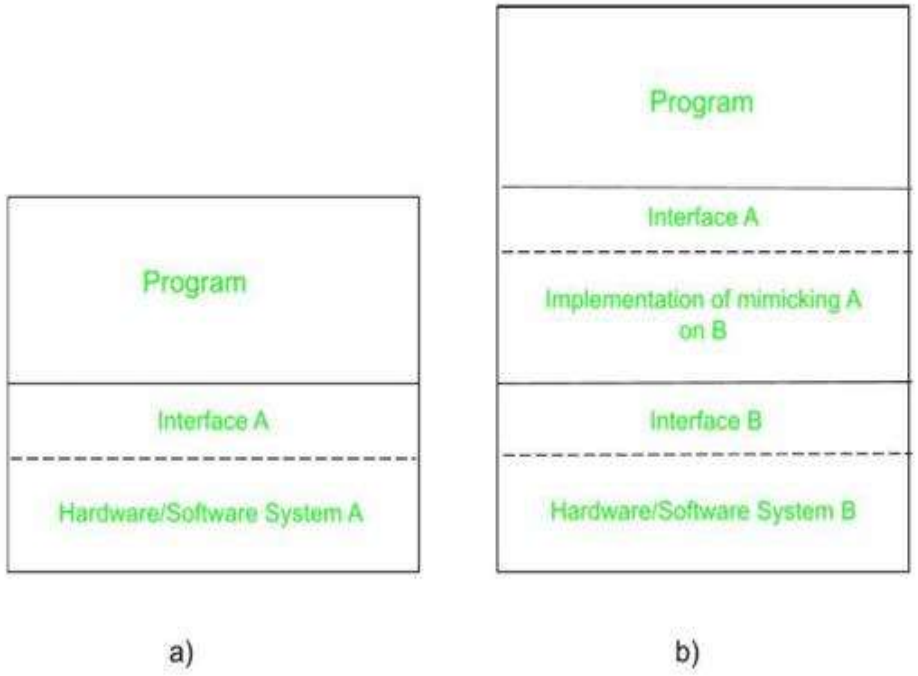
Together, processes and threads enable distributed systems to be more robust, scalable, efficient, and responsive, meeting the complex demands of modern distributed applications.

Virtualization in Distributed System

- A virtual resource, such as a server, desktop, operating system, file, storage, or network, is created through the process of virtualization.
- Virtualization's main objective is to manage workloads by fundamentally altering conventional computing to make it more scalable.
- It is possible to think of threads and processes as ways to do multiple tasks simultaneously. We can construct (parts of) programs that appear to run simultaneously thanks to them. Naturally, this simultaneous execution is fiction on a single-processor computer.
- A single thread or process will only have one instruction at a time because there is only one CPU. The illusion of parallelism is achieved by moving back and forth quickly between threads and processes.
- Resource virtualization is a term used to describe the difference between having a single CPU and being able to pretend there are multiple CPUs.

The Role of Virtualization in Distributed Systems:

- Every (distributed) computer system actually provides a programming interface to higher-level software, as can be seen in the below Fig (a).
- There are many various kinds of interfaces, from the fundamental instruction set that a CPU provides to the enormous selection of application programming interfaces that are included with many modern middleware systems.
- To emulate the behavior of another system, virtualization essentially involves extending or replacing an existing interface, as seen in Fig (b).

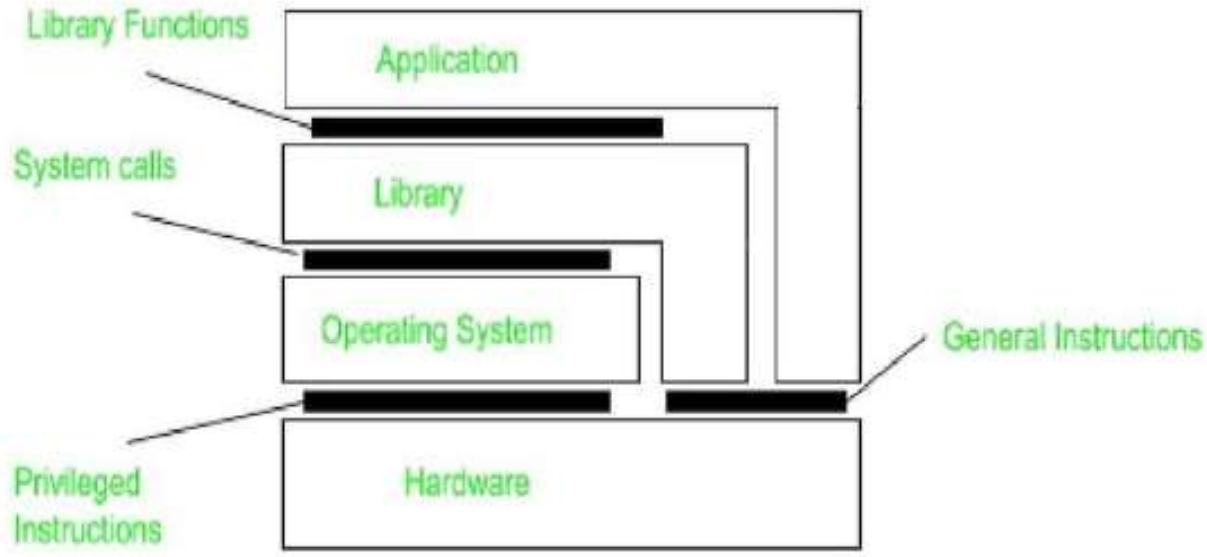


By effectively having each application run on its own virtual computer, possibly with the associated libraries and operating system, which in turn run on a common platform, virtualization can help minimize the diversity of platforms and machines. A high degree of portability and flexibility is offered by virtualization.

Architectures of Virtual Machines :

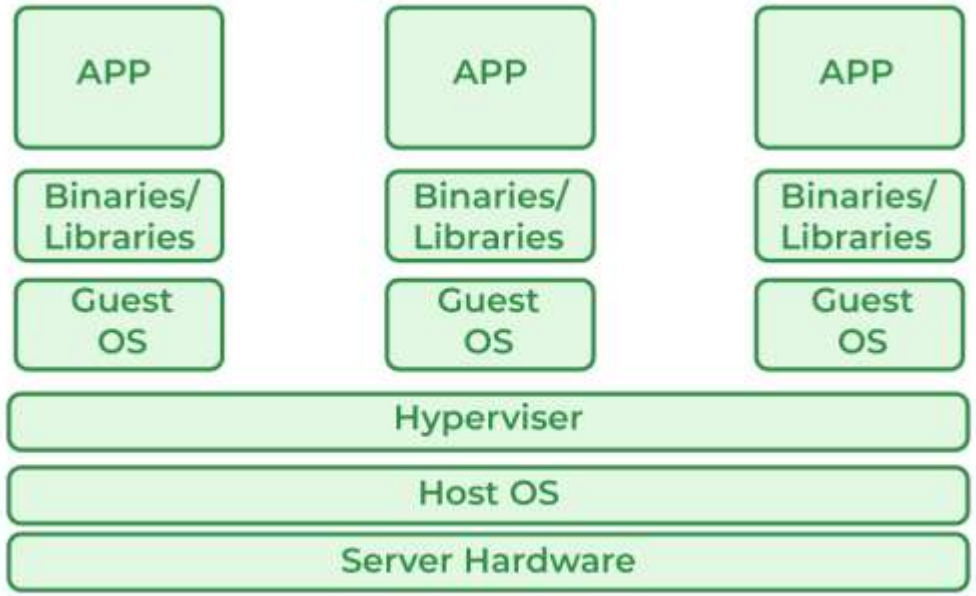
- Understanding the variations in virtualization requires an understanding of the common four types of interfaces that computer systems provide, at four different levels.
  1. A point of contact between hardware and software, made up of commands that can be executed by any application.
  2. A point of contact between hardware and software that is made up of machine instructions that are only accessible to privileged programs, such as an operating system.
  3. An operating system interface that is made up of system calls.
  4. An interface made up of library calls, which often constitutes an API (application programming interface) (API).

The following figure illustrates these various categories. The goal of virtualization is to replicate the functionality of these interfaces.

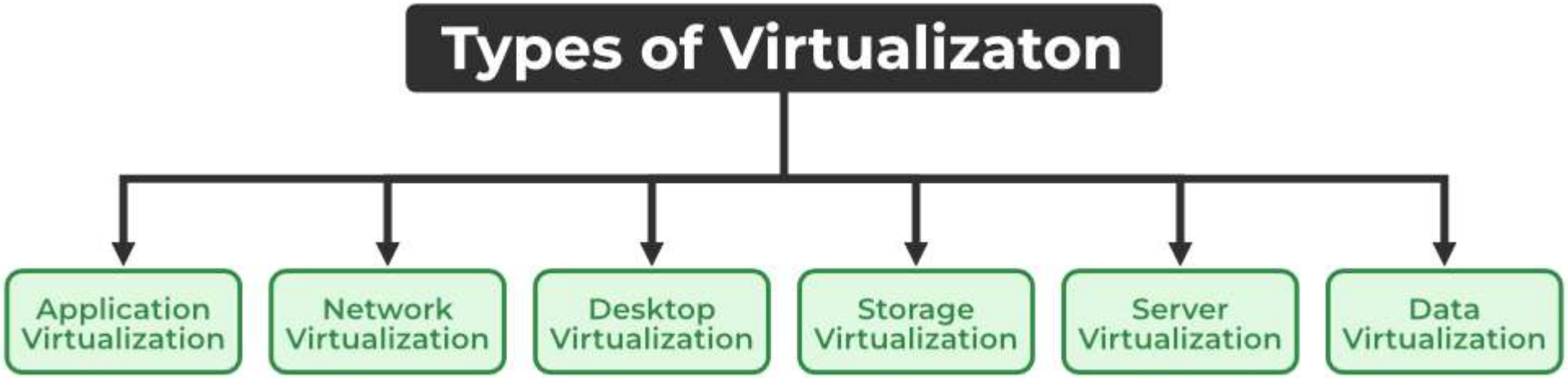


Virtualization is a technique of how to separate a service from the underlying physical delivery of that service. It is the process of creating a virtual version of something like computer hardware.

- Host Machine: The machine on which the virtual machine is going to be built is known as Host Machine.
- Guest Machine: The virtual machine is referred to as a Guest Machine.

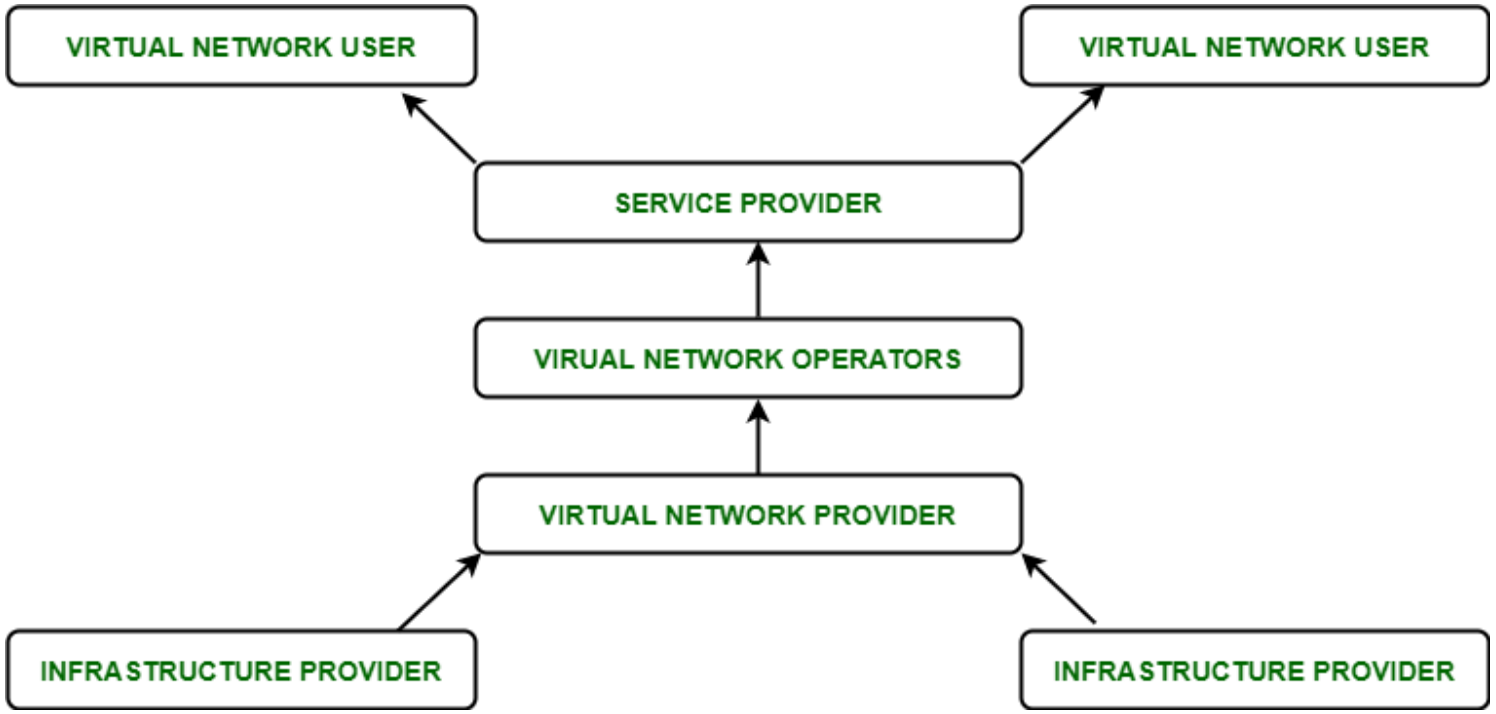


Virtualization



1. Application Virtualization: Application virtualization helps a user to have remote access to an application from a server. The server stores all personal information and other characteristics of the application but can still run on a local workstation through the internet. An example of this would be a user who needs to run two different versions of the same software. Technologies that use application virtualization are hosted applications and packaged applications.

2. Network Virtualization: The ability to run multiple virtual networks with each having a separate control and data plan. It co-exists together on top of one physical network. It can be managed by individual parties that are potentially confidential to each other. Network virtualization provides a facility to create and provision virtual networks, logical switches, routers, firewalls, load balancers, Virtual Private Networks (VPN), and workload security within days or even weeks.

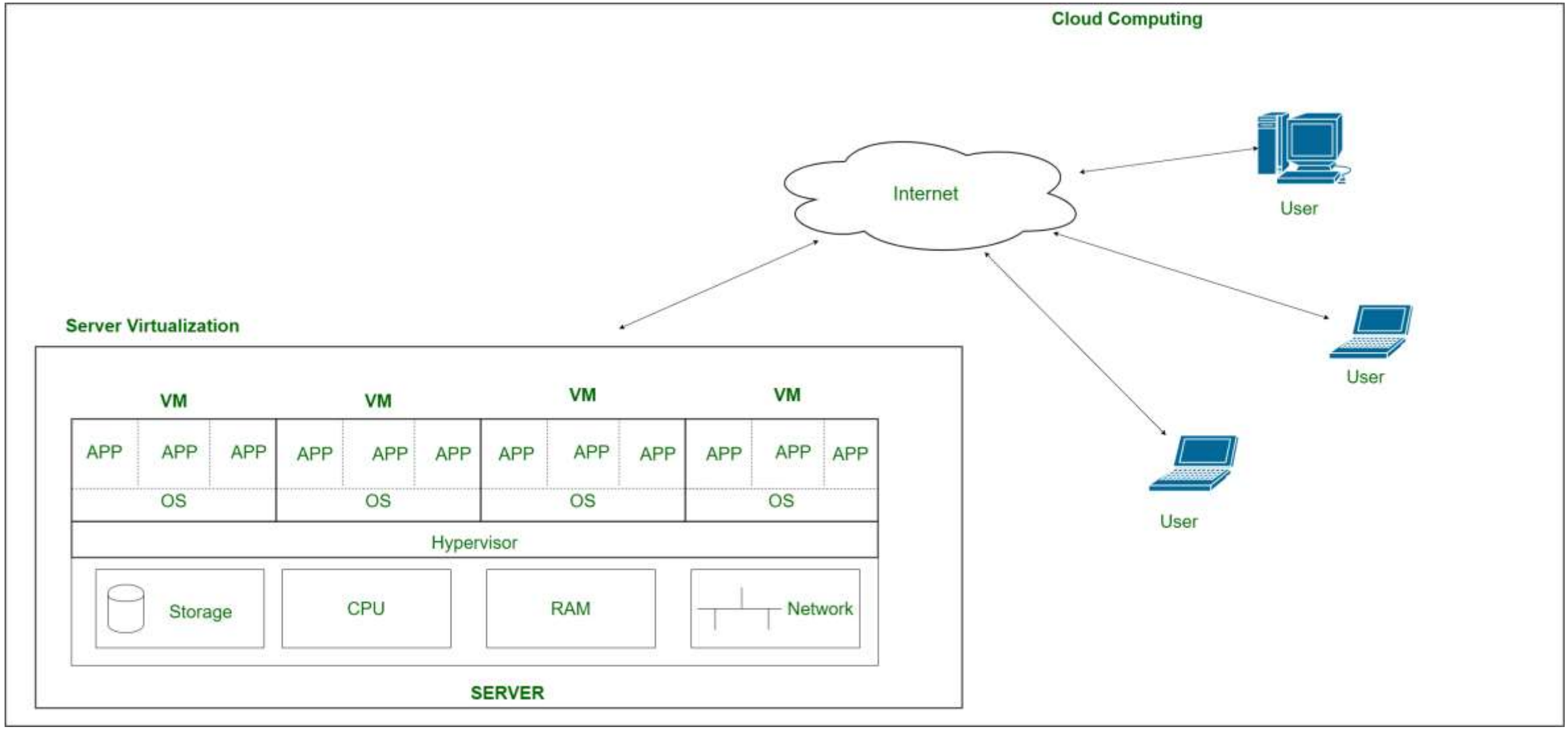


Network Virtualization

3. Desktop Virtualization: Desktop virtualization allows the users' OS to be remotely stored on a server in the data center. It allows the user to access their desktop virtually, from any location by a different machine. Users who want specific operating systems other than Windows Server will need to have a virtual desktop. The main benefits of desktop virtualization are user mobility, portability, and easy management of software installation, updates, and patches.

4. Storage Virtualization: Storage virtualization is an array of servers that are managed by a virtual storage system. The servers aren't aware of exactly where their data is stored and instead function more like worker bees in a hive. It makes managing storage from multiple sources be managed and utilized as a single repository. storage virtualization software maintains smooth operations, consistent performance, and a continuous suite of advanced functions despite changes, breaks down, and differences in the underlying equipment.

5. Server Virtualization: This is a kind of virtualization in which the masking of server resources takes place. Here, the central server (physical server) is divided into multiple different virtual servers by changing the identity number, and processors. So, each system can operate its operating systems in an isolated manner. Where each sub-server knows the identity of the central server. It causes an increase in performance and reduces the operating cost by the deployment of main server resources into a sub-server resource. It's beneficial in virtual migration, reducing energy consumption, reducing infrastructural costs, etc.



Server Virtualization

6. Data Virtualization: This is the kind of virtualization in which the data is collected from various sources and managed at a single place without knowing more about the technical information like how data is collected, stored & formatted then arranged that data logically so that its virtual view can be accessed by its interested people and stakeholders, and users through the various cloud services remotely. Many big giant companies are providing their services like Oracle, IBM, At scale, Cdata, etc.

Uses of Virtualization

- Data-integration
- Business-integration
- Service-oriented architecture data-services
- Searching organizational data

Clients and Servers in Distributed System

- In a distributed system, the concepts of clients and servers are fundamental to understanding how tasks are divided, executed, and managed across multiple computers or nodes within the network. Here's an overview of clients and servers in a distributed system:

Clients

1. **Definition**: Clients are the endpoints in a distributed system that request services or resources from other nodes, typically servers. They are often the interfaces through which users interact with the system.
2. **Role**: The primary role of a client is to initiate communication by sending requests to servers and then handling the responses from these servers.
3. **Characteristics**:
    1. **User-Facing**: Clients are usually applications or devices used by end-users (e.g., web browsers, mobile apps).
    2. **Resource Access**: Clients depend on servers for accessing and retrieving data, processing power, and other resources.
    3. **Stateless or Stateful**: Clients can be stateless (where each request is independent) or stateful (where some state is maintained across requests).

Servers

1. **Definition**: Servers are the nodes in a distributed system that provide services or resources in response to requests from clients.
2. **Role**: The primary role of a server is to listen for client requests, process them, and send back the appropriate responses.
3. **Characteristics**:
    1. **Service Provider**: Servers offer various services such as data storage, computation, file serving, or running applications.
    2. **Resource Management**: Servers manage and control access to resources like databases, file systems, and computing power.
    3. **Concurrent Handling**: Servers are designed to handle multiple client requests simultaneously, often through mechanisms like multi-threading or asynchronous processing.

Interaction Between Clients and Servers

1. **Communication Protocols**: Clients and servers communicate over a network using various protocols. Common protocols include:
    1. **HTTP/HTTPS**: Used for web services.
    2. **TCP/IP**: For general data transmission.
    3. **RPC (Remote Procedure Call)**: Allows a program to cause a procedure to execute in another address space.
    4. **SOAP, REST**: Web service protocols.
2. **Request-Response Model**: The interaction is typically based on a request-response model where:
    1. The client sends a request to the server.
    2. The server processes the request.
    3. The server sends back a response to the client.
3. **Middleware**: Middleware can facilitate communication between clients and servers by providing various services like authentication, load balancing, and data transformation.

Examples

1. **Web Browsing**: A web browser (client) requests a web page from a web server. The server processes this request and sends back the web page content.
2. **Email Services**: An email client requests to send or receive emails from an email server. The server processes these requests accordingly.
3. **Database Access**: A client application requests data from a database server, which retrieves the data and sends it back to the client.

Benefits of Client-Server Architecture

1. **Scalability**: Servers can be scaled independently to handle increased load.
2. **Manageability**: Centralized servers make it easier to manage, update, and secure the system.
3. **Resource Optimization**: Efficient use of resources since clients only use what they need and servers optimize resource allocation.

Challenges

1. **Network Dependency**: The system relies heavily on network stability and performance.
2. **Single Points of Failure**: If a critical server fails, it can affect all clients depending on its services.
3. **Security**: Ensuring secure communication and data transfer between clients and servers is crucial.

Code Migration in Distributed System

Code migration in a distributed system refers to the process of moving code (along with its execution state, if necessary) from one machine or node to another within the network. This capability allows for more dynamic and flexible use of resources, improved load balancing, fault tolerance, and easier updates and maintenance. Here's a detailed look at code migration:

Types of Code Migration

1. **Process Migration**: This involves moving an entire process, including its code, data, and execution state, from one machine to another. This can be challenging due to the need to preserve the state and dependencies of the process.
2. **Code Shipping**: Only the code is transferred to a different machine, where it is executed. This is simpler than full process migration as it doesn't require moving the entire execution state.
3. **Mobile Code**: Refers to code that can move between machines at runtime, such as applets, scripts, or other types of executable code.

Levels of Migration

1. **Weak Mobility**: Only the code and some initial data are moved, but the execution state (e.g., program counter, call stack) is not transferred. Examples include Java applets and JavaScript.
2. **Strong Mobility**: The entire execution state of the process is moved along with the code. This allows the process to resume execution seamlessly on the new machine. Examples include some forms of virtual machine migration.

Motivations for Code Migration

1. **Load Balancing**: Distributing the computational load more evenly across the network can optimize resource use and improve performance.
2. **Fault Tolerance**: By moving processes away from nodes that are failing or undergoing maintenance, the system can continue to function without interruption.
3. **Proximity to Data**: Moving code closer to where the data is located can reduce latency and improve performance, especially in data-intensive applications.
4. **Resource Sharing**: Efficiently utilizing available resources across the network by relocating processes to underutilized nodes.
5. **Scalability**: Dynamically allocating resources as demand changes helps the system scale more effectively.
6. **Maintenance and Updates**: Easier deployment of updates and patches by moving the affected processes temporarily.

Mechanisms of Code Migration

1. **Checkpointing and Restarting**: Capturing the state of a process at a point in time (checkpointing) and then restarting it on another machine from that state.
2. **Virtual Machine Migration**: Moving an entire virtual machine, including its running applications, from one physical host to another. Technologies like VMware vMotion and Xen support this.
3. **Mobile Agents**: Autonomous code entities that move between nodes, carrying their own state and executing tasks.

Challenges

1. **State Management**: Ensuring that the execution state is accurately preserved and transferred is complex.
2. **Dependencies**: Managing dependencies like files, libraries, and network connections that the migrating code might need.
3. **Security**: Safeguarding the code during transit and ensuring that the target environment is secure.

4. **Performance Overhead**: The process of migration itself can be resource-intensive and may temporarily degrade system performance.
5. **Consistency**: Ensuring that data consistency is maintained, particularly in distributed databases or applications with shared state.

Examples and Use Cases

1. **Cloud Computing**: Virtual machines and containers in cloud environments are frequently migrated for load balancing and maintenance.
2. **Distributed Databases**: Moving database shards to different nodes to balance load and improve access times.
3. **Web Services**: Distributing microservices across different servers to optimize response times and resource use.
4. **Edge Computing**: Moving processing tasks closer to IoT devices to reduce latency and bandwidth usage.

Foundations of Communication in DS

- The foundation of communication in a distributed system is critical for ensuring that different components within the system can interact seamlessly and perform their intended functions efficiently. Communication in distributed systems involves the exchange of data and control information among various distributed nodes.

Communication Protocols

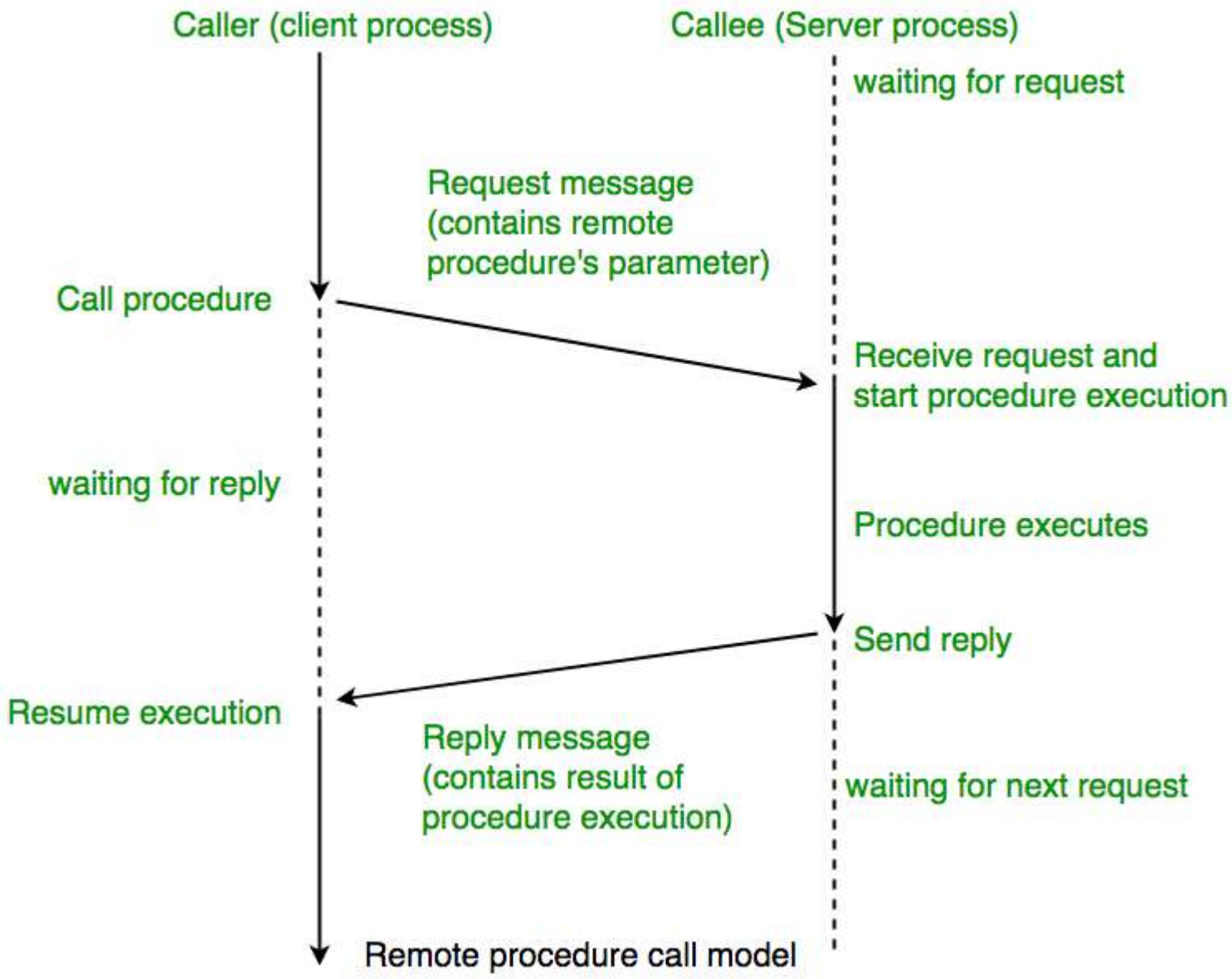
1. **Network Protocols**
    1. **TCP/IP (Transmission Control Protocol/Internet Protocol)**: The backbone of internet and distributed system communication, ensuring reliable and ordered data transmission.
    2. **UDP (User Datagram Protocol)**: Provides faster, but less reliable, data transmission, often used in applications where speed is critical, and occasional data loss is acceptable.
2. **Higher-Level Protocols**
    1. **HTTP/HTTPS:** Commonly used for web-based communication and RESTful services, ensuring secure and standardized data exchange.
    2. **SMTP (Simple Mail Transfer Protocol):** Used for email transmission.
    3. **FTP (File Transfer Protocol):** Used for transferring files between systems.

Remote Procedure Calls (RPCs) and Remote Method Invocation (RMI)

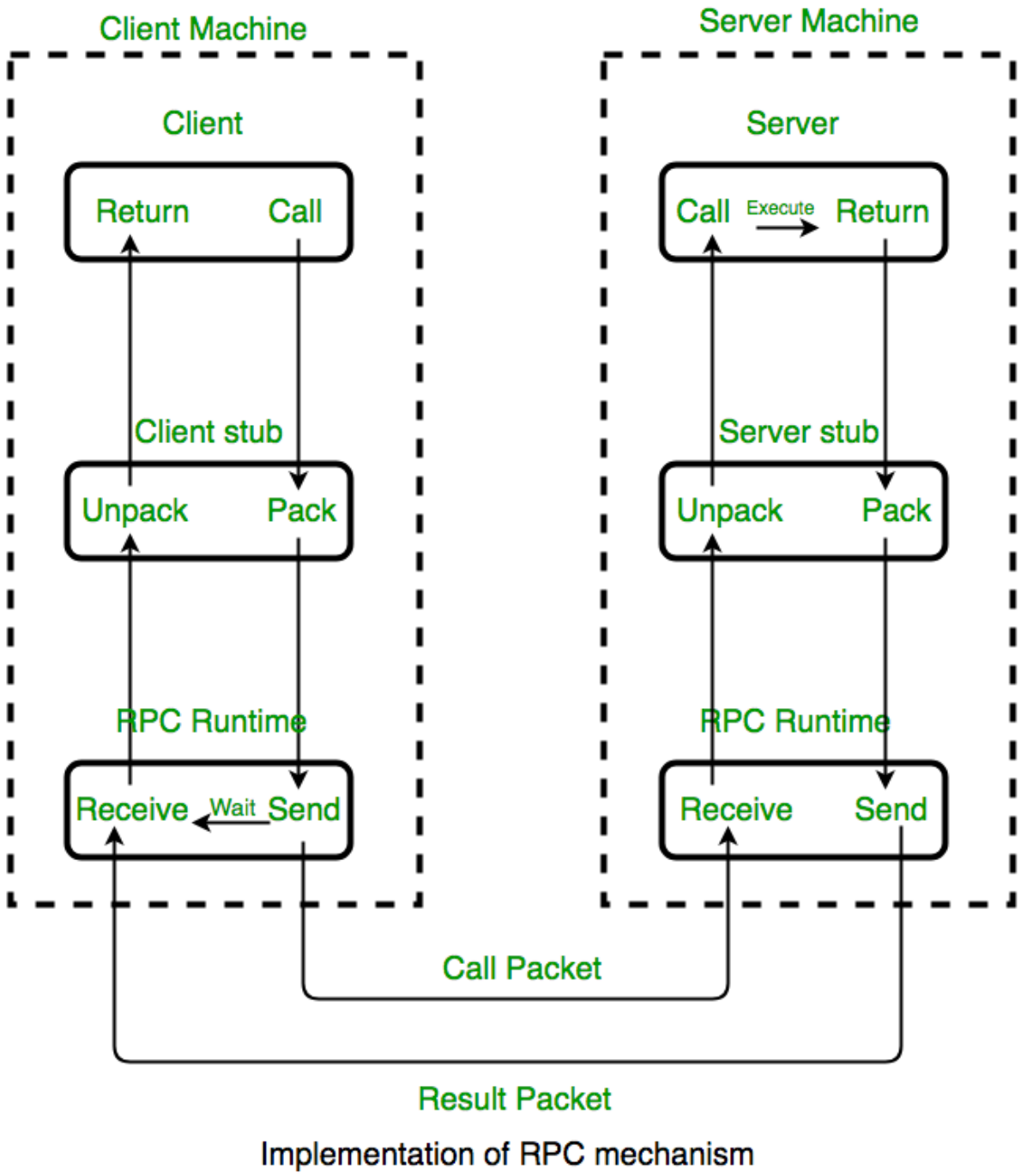
- **RPCs**: Enable a program to cause a procedure to execute in another address space, abstracting the communication details and making remote interactions appear as local function calls.
- **RMI**: Similar to RPCs but specifically designed for object-oriented programming, allowing objects to invoke methods on remote objects.

- **Remote Procedure Call (RPC)** is a powerful technique for constructing **distributed, client-server based applications**. It is based on extending the conventional local procedure calling so that the **called procedure need not exist in the same address space as the calling procedure**. The two processes may be on the same system, or they may be on different systems with a network connecting them.

- The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.
- When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.
- NOTE: RPC is especially well suited for client-server (e.g. query-response) interaction in which the flow of control alternates between the caller and callee. Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again.



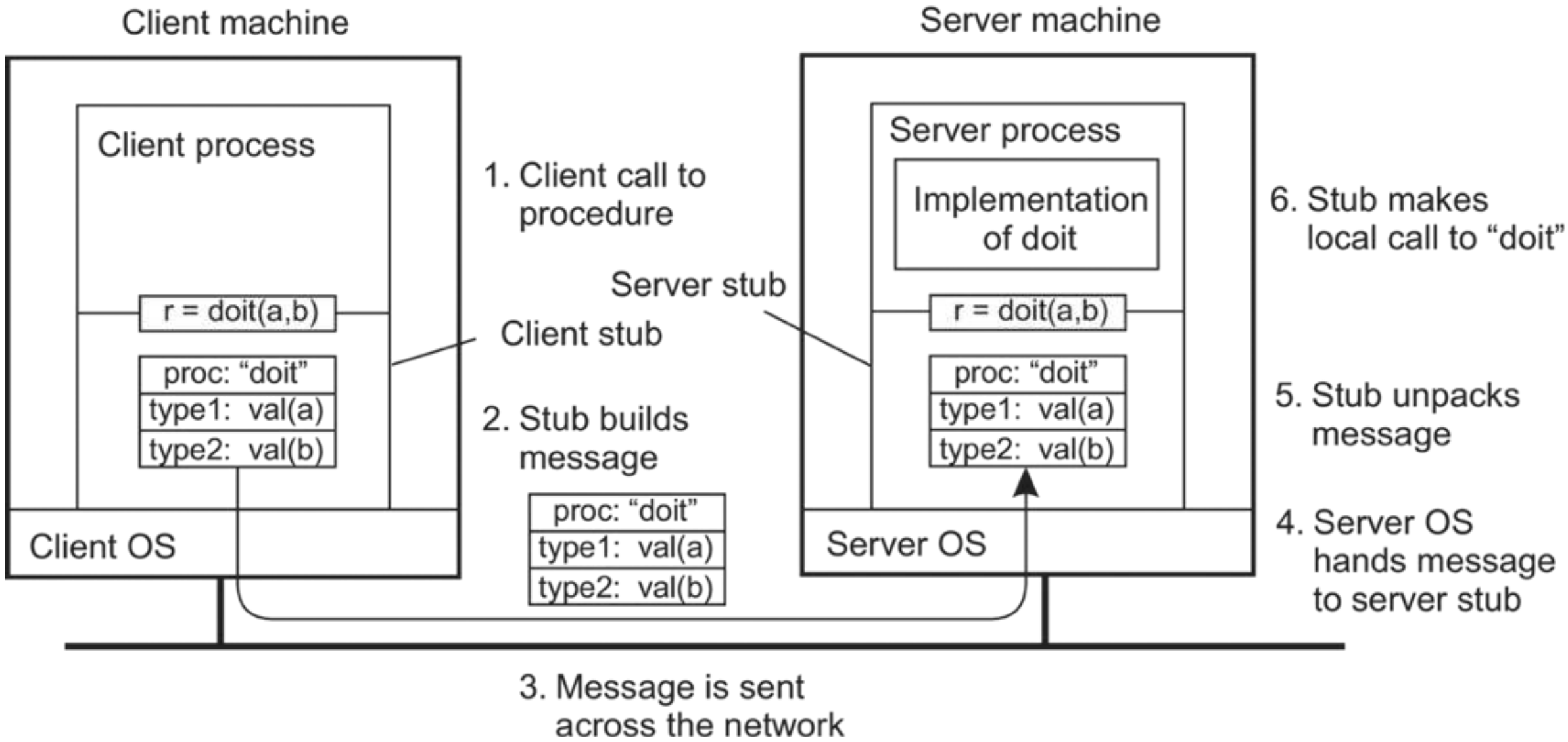
Working of RPC



Implementation of RPC mechanism

- **The following steps take place during a RPC :**

1. A client invokes a **client stub procedure**, passing parameters in the usual way. The client stub resides within the client's own address space.
2. The client stub **marshals(pack)** the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.
3. The client stub passes the message to the transport layer, which sends it to the remote server machine.
4. On the server, the transport layer passes the message to a server stub, which **demarshalls (unpack)** the parameters and calls the desired server routine using the regular procedure call mechanism.
5. When the server procedure completes, it returns to the server stub **(e.g., via a normal procedure call return)**, which marshals the return values into a message. The server stub then hands the message to the transport layer.
6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.
7. The client stub demarshalls the return parameters and execution returns to the caller.


Client machine
Server machine
Client process
Server process
Implementation of doit
r = doit(a,b)
r = doit(a,b)
proc: “doit”
type1: val(a)
type2: val(b)
proc: “doit”
type1: val(a)
type2: val(b)
Client OS
Server OS
1. Client call to procedure
Server stub
Client stub
2. Stub builds message
proc: “doit”
type1: val(a)
type2: val(b)
6. Stub makes local call to “doit”
5. Stub unpacks message
4. Server OS hands message to server stub
3. Message is sent across the network

RPC ISSUES :

1. **RPC Runtime:**
    1. RPC run-time system is a library of routines and a set of services that handle the network communications that underlie the RPC mechanism. In the course of an RPC call, client-side and server-side run-time systems' code handle **binding, establish communications over an appropriate protocol, pass call data between the client and server, and handle communications errors.**
2. **Stub:** The function of the stub is to **provide transparency to the programmer-written application code**.
    1. **On the client side**, the stub handles the interface between the client's local procedure call and the run-time system, marshalling and unmarshalling data, invoking the RPC run-time protocol, and if requested, carrying out some of the binding steps.
    2. **On the server side**, the stub provides a similar interface between the run-time system and the local manager procedures that are executed by the server.
3. **Binding: How does the client know who to call, and where the service resides?**
    1. The most flexible solution is to use dynamic binding and find the server at run time when the RPC is first made. The first time the client stub is invoked, it contacts a name server to determine the transport address at which the server resides.
    2. Binding consists of two parts: Naming and Location

Advantages of Remote Procedure Call

Some of the advantages of RPC are as follows –

- Remote procedure calls support process oriented and thread oriented models.
- The internal message passing mechanism of RPC is hidden from the user.
- The effort to re-write and re-develop the code is minimum in remote procedure calls.
- Remote procedure calls can be used in distributed environment as well as the local environment.
- Many of the protocol layers are omitted by RPC to improve performance.

Disadvantages of Remote Procedure Call

Some of the disadvantages of RPC are as follows –

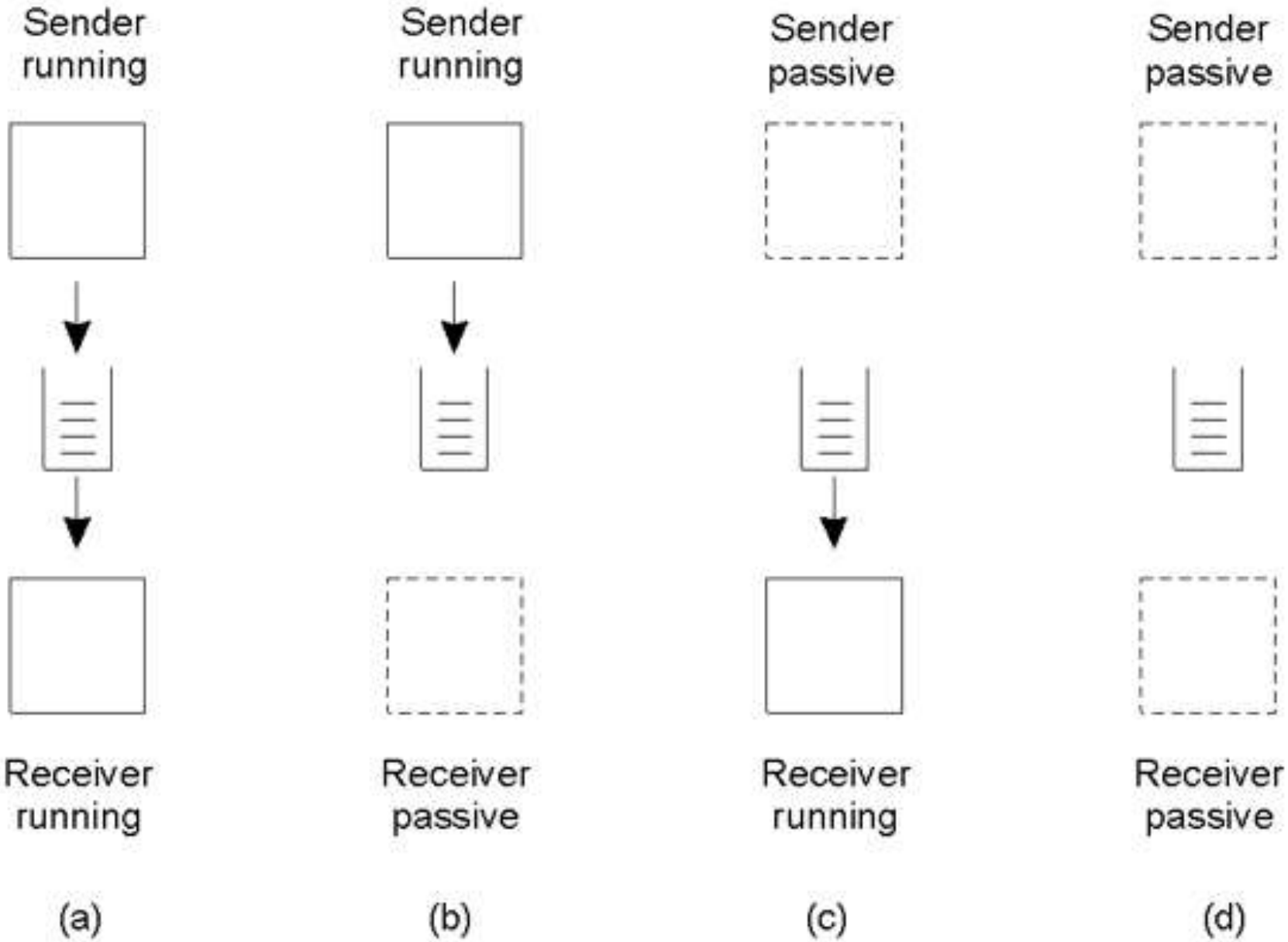
- The remote procedure call is a concept that can be implemented in different ways. It is not a standard.
- There is no flexibility in RPC for hardware architecture. It is only interaction based.
- There is an increase in costs because of remote procedure call.

Message Oriented Communication in DS

Message Oriented Communication (MOC) is a fundamental paradigm in distributed systems that involves the exchange of messages between processes or systems. This communication method is pivotal for enabling different parts of a distributed system to coordinate and perform tasks. Here's an overview of its key concepts, benefits, and applications:

Key Concepts

1. **Messages**: Units of data exchanged between systems. Messages can contain various types of information, such as commands, queries, or data updates.
2. **Message Queues**: Temporary storage for messages in transit. Queues help decouple the sender and receiver, allowing for asynchronous communication.
3. **Message Brokers**: Intermediary services that manage the routing and delivery of messages. Examples include RabbitMQ, Apache Kafka, and Amazon SQS.
4. **Producers and Consumers**: Producers send messages to the system, and consumers receive and process them.
5. **Asynchronous Communication**: Producers and consumers operate independently, improving system responsiveness and scalability.
6. **Synchronous Communication**: Both parties must be available at the same time, often requiring a direct reply or acknowledgment.



a) Both sender and receiver is active

b) Sender is active but receiver is not active – msg can't be delivered

Applications

1. **Microservices Architecture**: Services communicate via messages, allowing for scalable and resilient applications.
2. **Event-Driven Systems**: Systems react to events (messages) in real-time, suitable for applications like monitoring, alerts, and user notifications.
3. **Enterprise Integration**: Different enterprise applications can communicate and integrate via message queues, facilitating data exchange and process coordination.
4. **Data Streaming**: Large volumes of data can be processed and analyzed in real-time, critical for applications like analytics and IoT.
5. **Workflow Management**: Complex workflows can be managed by coordinating tasks through messages, ensuring reliable execution and coordination.

Implementations

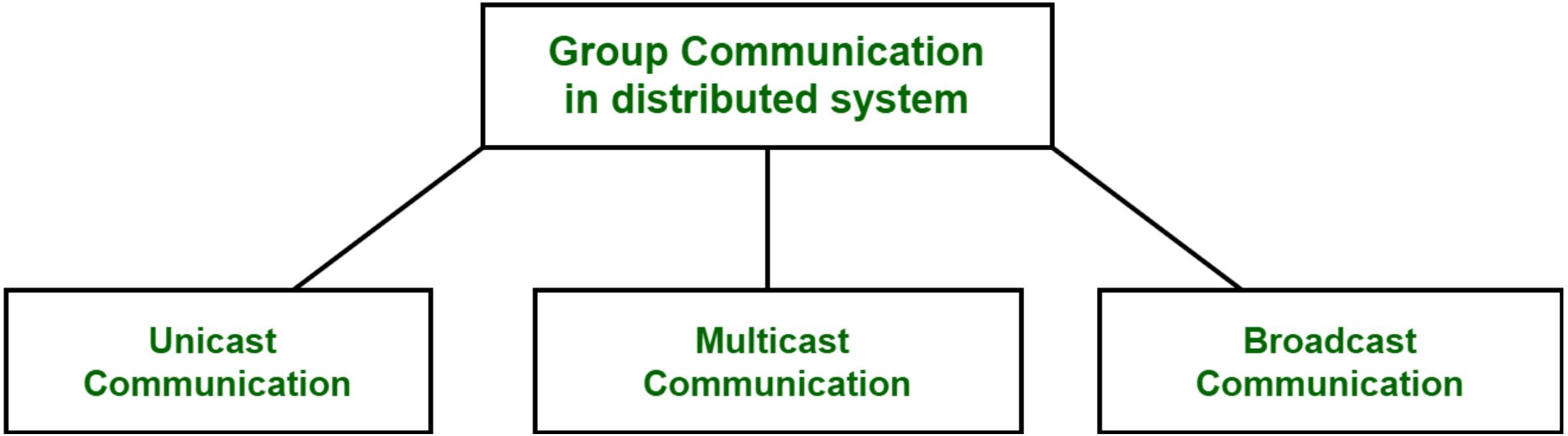
- **RabbitMQ**: A widely-used message broker implementing the Advanced Message Queuing Protocol (AMQP). It supports multiple messaging patterns and high availability.
- **Apache Kafka**: Designed for high-throughput, distributed streaming. Kafka is often used for real-time data pipelines and streaming applications.
- **Amazon SQS**: A fully managed message queuing service by AWS, providing reliable and scalable messaging infrastructure.
- **ZeroMQ**: A high-performance asynchronous messaging library aimed at use in scalable distributed or concurrent applications.

Benefits

1. **Decoupling**: Systems can be developed, deployed, and scaled independently, improving maintainability and flexibility.
2. **Scalability**: Systems can handle varying loads by adding or removing consumers without affecting the producers.
3. **Reliability**: Messages can be persisted in queues, ensuring delivery even if the receiver is temporarily unavailable.
4. **Fault Tolerance**: Systems can recover from failures by reprocessing messages from queues.

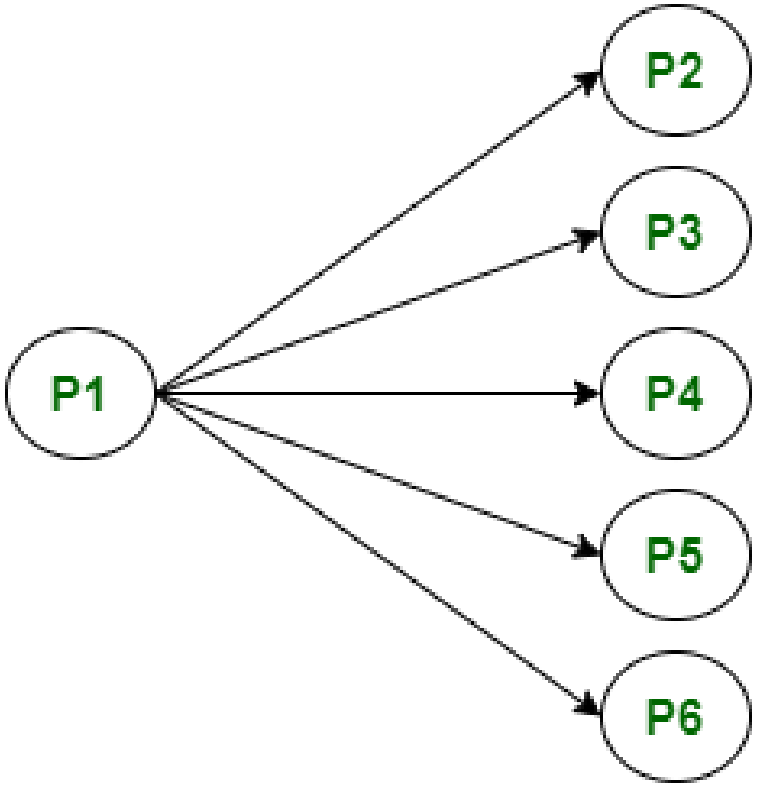
Group Communication in distributed Systems

- **Communication** between two processes in a distributed system is required to exchange various data, such as code or a file, between the processes. When one source process tries to communicate with multiple processes at once, it is called **Group Communication**. A group is a collection of interconnected processes with abstraction. This abstraction is to hide the message passing so that the communication looks like a normal procedure call. Group communication also helps the processes from different hosts to work together and perform operations in a synchronized manner, therefore increasing the overall performance of the system.



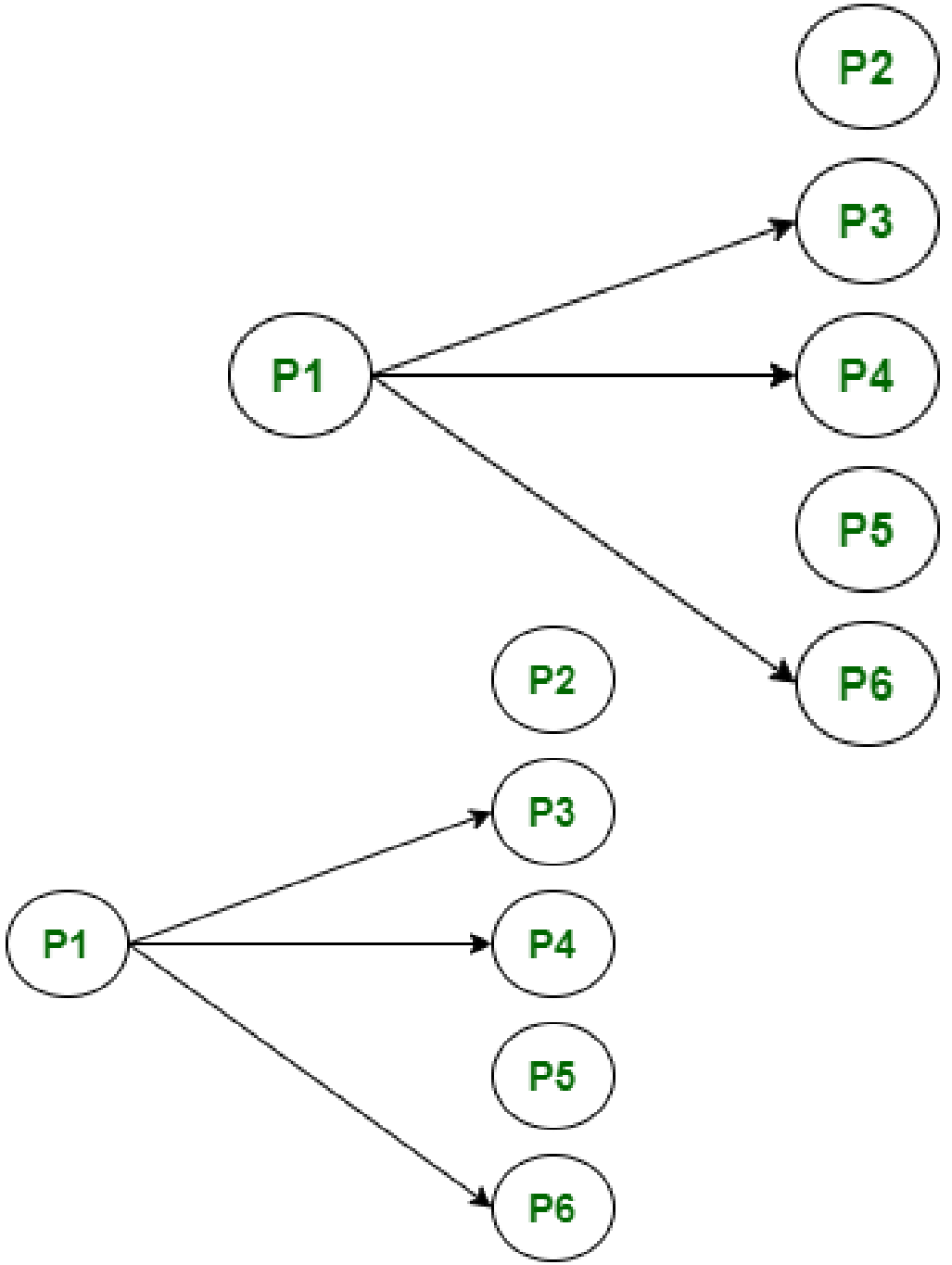
Broadcast Communication

- When the host process tries to communicate with every process in a distributed system at same time. Broadcast communication comes in handy when a common stream of information is to be delivered to each and every process in most efficient manner possible.
- Since it does not require any processing whatsoever, communication is very fast in comparison to other modes of communication.
- However, it does not support a large number of processes and cannot treat a specific process individually.



- **Multicast Communication :** When the host process tries to communicate with a designated group of processes in a distributed system at the same time. This technique is mainly used to find a way to address problem of a high workload on host system and redundant information from process in system. Multitasking can significantly decrease time taken for message handling.

- **Unicast Communication :** When the host process tries to communicate with a single process in a distributed system at the same time. Although, same information may be passed to multiple processes. This works best for two processes communicating as only it has to treat a specific process only. However, it leads to overheads as it has to find exact process and then exchange information/data.

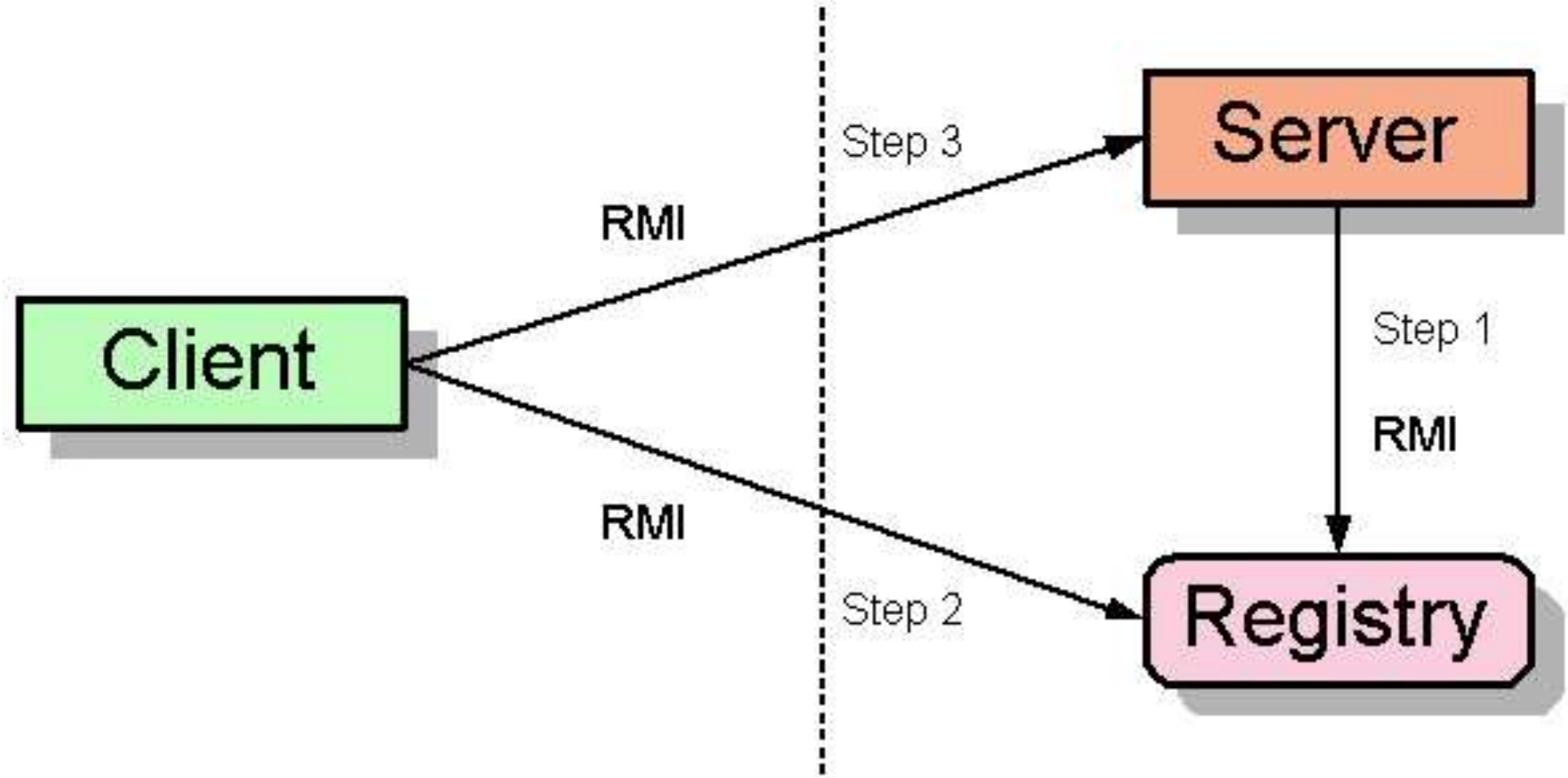


Case Study: Java RMI and Message Passing Interface (MPI)

Java RMI

- **Overview**
  - Java Remote Method Invocation (RMI) is a Java API that allows an object to invoke methods on an object running in another Java Virtual Machine (JVM). RMI provides a straightforward approach to building distributed applications in Java by abstracting the underlying network communication.

- **Application Example**
  - **Distributed Financial Application:** A financial services company developed a distributed trading system using Java RMI. The system consists of several components:
  - **Trading Engine**: Handles trade execution and runs on a dedicated server.
  - **Client Applications**: Traders use these applications to place orders.
  - **Data Servers**: Provide real-time market data to the trading engine and clients.


Server
Step 3
RMI
Client
Step 1
RMI
RMI
Step 2
Registry

Implementation Details

- **Trading Engine**: Exposes remote methods for placing orders and querying trade status.
- **Client Applications**: Invoke remote methods on the trading engine to execute trades and retrieve information.
- **Data Servers**: Use RMI to push market data updates to the trading engine and clients.

Advantages

- **Simplicity**: Java RMI provides a high-level abstraction, making it easy to implement remote method calls.
- **Integration**: Seamless integration with Java applications, leveraging existing Java libraries and tools.
- **Security**: Built-in support for SSL/TLS, allowing secure communication.

Challenges

- **Performance**: RMI can introduce latency due to network communication and serialization overhead.
- **Scalability**: Managing numerous client connections and ensuring load balancing can be complex.

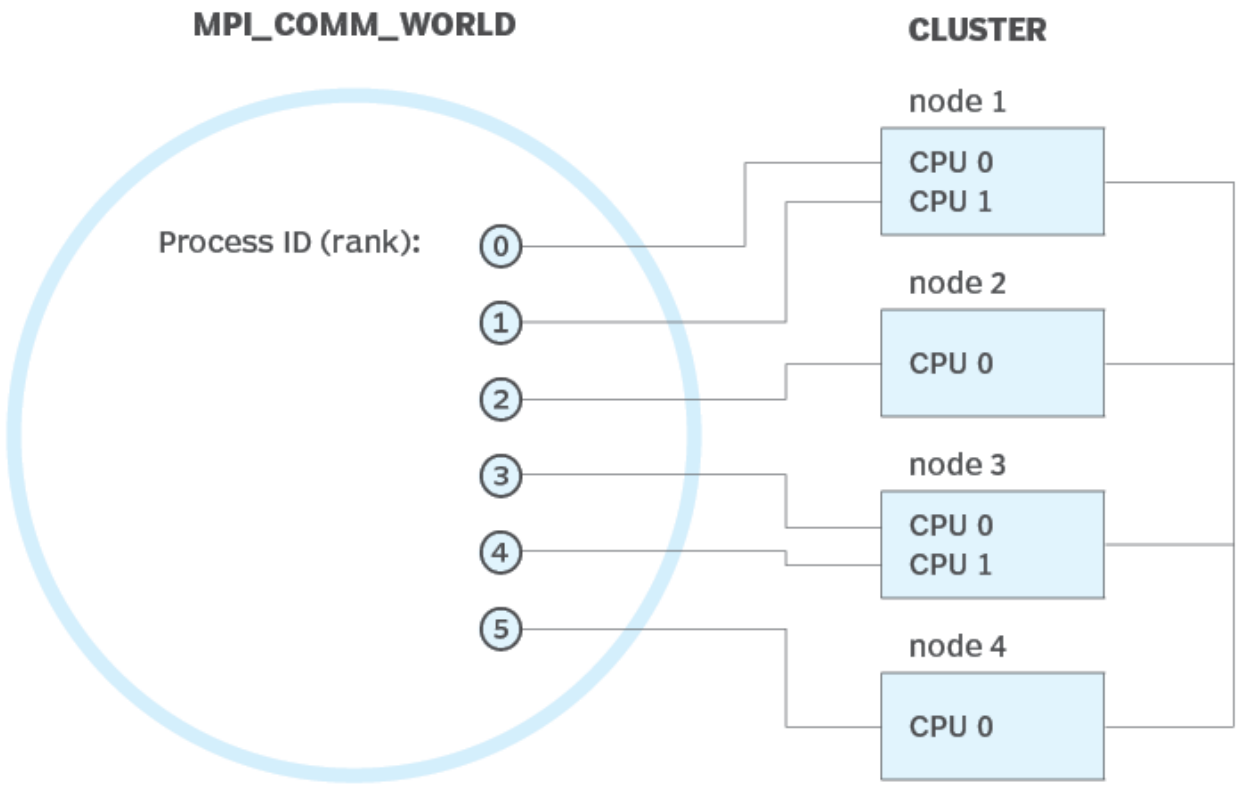
Message Passing Interface (MPI)

MPI is a standardized and portable message-passing system designed to function on a variety of parallel computing architectures. MPI enables processes to communicate with each other by sending and receiving messages, making it suitable for high-performance computing (HPC) applications.

Application Example

- **Scientific Research Cluster**
  - A research institution implemented a distributed system for running large-scale simulations in computational fluid dynamics (CFD). The system consists of a cluster of high-performance computers, each running a part of the simulation.
- **Implementation Details**
  1. **Simulation Tasks**: Divided into smaller tasks that run concurrently on different nodes.
  2. **Inter-Process Communication**: Nodes use MPI to exchange intermediate results and synchronize their computations.
  3. **Load Balancing**: MPI's dynamic process management ensures efficient use of computational resources.

Message passing interface (MPI)



An MPI COMM process containing multiple nodes in four clusters shows how a rank is given to each CPU.

Assignment -2 / Old Questions

1. Explain the Classes of Failures in Request-Reply Communication.
2. How does mutual authentication work? Explain.
3. Define virtualization. Explain different types of virtualization in detail.
4. Explain Message-oriented communication in detail.
5. Explain about code migration.
6. What are the steps involved in calling a remote procedure? Discuss the design issues for RPC.
7. What is meant by inter process communication? How inter process communication is used in distributed system?
8. What is virtualization in distributed system explain different virtualization methods
9. Explain layered protocol communication in distributed system
10. Differentiate between RPC and RMI.