DISTRIBUTED SYSTEM:

A **distributed system** is a network of independent computers that work together to achieve a common goal.
In this system, multiple nodes (machines) are connected, sharing resources, data, and tasks to improve efficiency, performance, and reliability.
The key feature of a distributed system is that it appears as a single coherent system to users, even though its components are spread across different locations.

CHARACTERISTICS OF DISTRIBUTED SYSTEMS:

1. Resource Sharing: Connecting Resources and Users.
2. Transparency: Communication is hidden from users.
3. Openness: Applications can interact in a uniform and consistent way.
4. Scalability: It will remain effective when there is a significant increase in the number of users and resources.
5. Heterogeneity: Building blocks could be made of different materials and models.
6. Security: How secure is the system against malicious attacks
7. Fault Tolerance: How it deals with failures like message loss, network partitioning, etc.
8. Concurrency: How concurrent shared resources are being used.
9. Quality of Service: Adaptability, availability, reliability, etc.

ADVANTAGES OF DISTRIBUTED SYSTEMS:

1. All the nodes in the distributed system are connected to each other. So nodes can easily share data
with other nodes.
2. More nodes can easily be added to the distributed system i.e. it can be scaled as required.
3. Failure of one node does not lead to the failure of the entire distributed system. Other nodes can still
communicate with each other.
4. Resources like printers can be shared with multiple nodes rather than being restricted to just one.

DISADVANTAGES OF DISTRIBUTED SYSTEMS:

1. It is difficult to provide adequate security in distributed systems because the nodes as well as the connections need to be secured.
2. Some messages and data can be lost in the network while moving from one node to another.
3. The database connected to the distributed systems is quite complicated to handle as compared to a single-user system.
4. Overloading may occur in the network if all the nodes of the distributed system try to send data at once.

Types of Distributed Systems(Architecture):

1. Client/Server Distributed Systems:
In this system, clients request services or resources from a server, and the server responds to the request. Multiple clients can communicate with a single server, which manages and processes requests.
Example: Web applications where the client is a browser, and the server hosts the web pages.

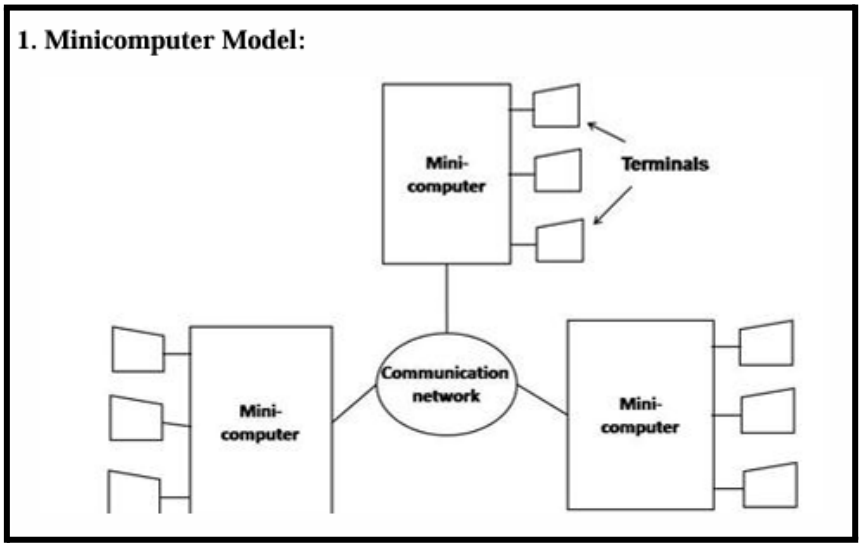
2. Peer-to-Peer (P2P) Distributed Systems:
In P2P systems, all nodes have equal roles, sharing resources and responsibilities without a central server. Each node acts as both a client and a server, collaborating to complete tasks.
Example: File-sharing networks like BitTorrent.

Explain the four distributed computing system models.

The five categories of models for building distributed computing systems are:

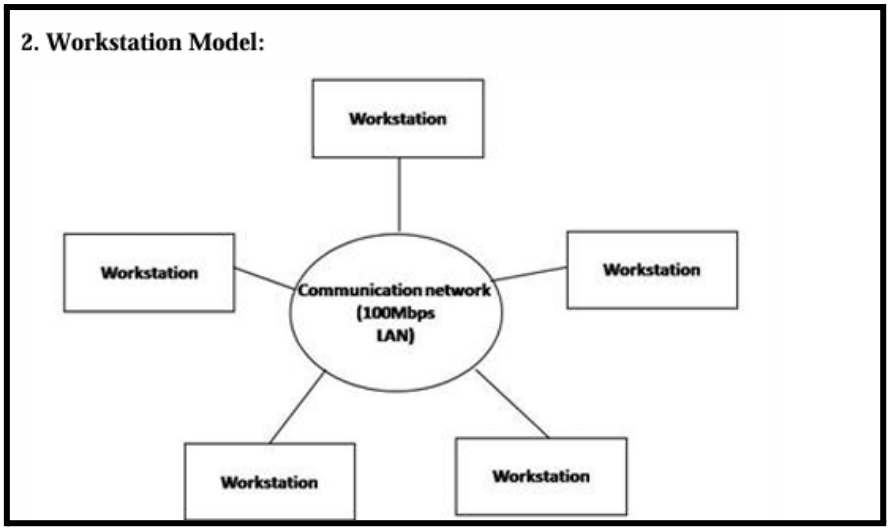
1. Minicomputer Model:

- An extension of the centralized time-sharing system.
- Involves multiple minicomputers interconnected via a communication network, allowing multiple users to log in simultaneously and access remote resources.
- Users can share resources across different machines.
- Example: Early ARPANET.



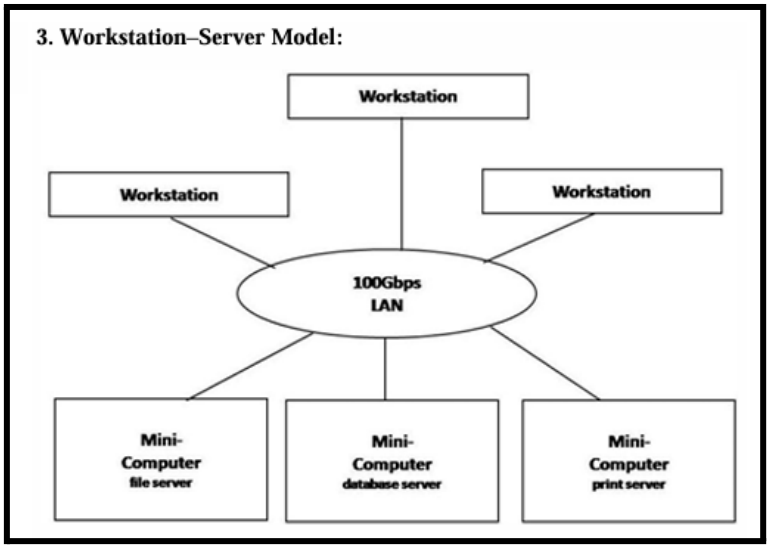
2. Workstation Model:

- A high-speed LAN interconnects several workstations in a distributed system.
- Each workstation is typically a single-user computer with its own disk.
- Idle workstations can be utilized to process jobs from users whose workstations lack processing power.
- Example: Sprite system, Xerox PARC.



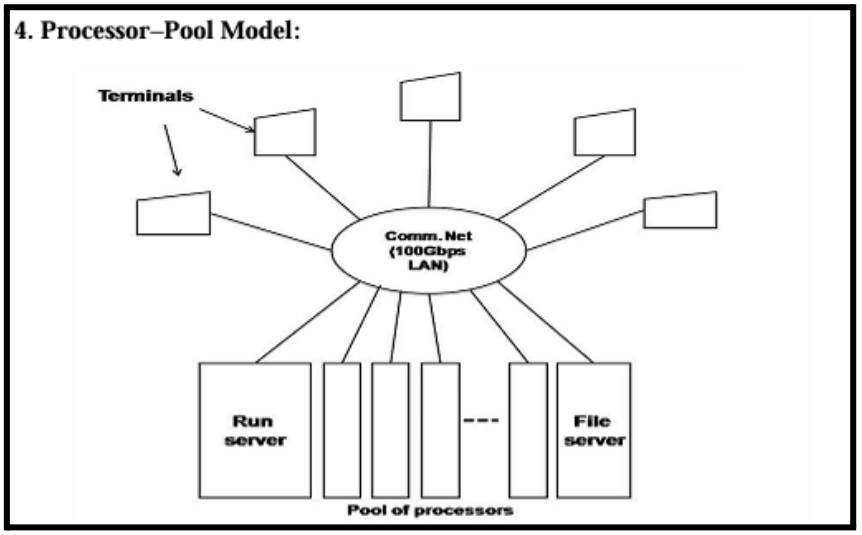
3. Workstation–Server Model:

- Consists of a network of workstations connected by a communication network, either with or without local disks.
- Users perform basic tasks on their home workstations, while dedicated servers handle resource-heavy tasks.
- The user's processes don't need to migrate; servers process the requests and return results.
- Example: The V-System.



4. Processor–Pool Model:

- Involves a pool of processors where users can access large computing power when needed.
- The processors are pooled together and shared dynamically based on demand.
- No home machine for users to log into; resources are shared flexibly.
- Example: Amoeba, Cambridge Distributed Computing System.



5. Hybrid Model:

- Combines the features of the Workstation-Server and Processor-Pool models.
- Provides flexibility in allocating processors dynamically for complex tasks, while allowing interactive tasks to be processed on the user's local workstation.
- Suitable for scenarios with varying computational needs.

Each of these models has specific applications depending on the complexity and scale of the tasks in distributed systems.

Explain any five issues in detail when designing a distributed system.

Design Issues of Distributed Systems

SHOTS-FC

1. Heterogeneity:
   - Distributed systems are made up of different hardware, operating systems, and software developed by various vendors, creating diversity in the environment.
   - Middleware is used to enable seamless communication and integration across these varied systems. Middleware provides a set of services that help different applications communicate with each other across the system.
   - It acts as a bridge between different technologies to ensure smooth interaction in a client-server environment.

2. Openness:
   - A distributed system is considered "open" if it can easily expand by adding new resources and making them available to users.
   - The openness of the system depends on having standardized and well-defined interfaces, allowing new components to interact with existing ones.
   - Open systems allow resource sharing across heterogeneous hardware and software platforms, making it more flexible and adaptable.

3. Scalability:
   - Scalability refers to the system's ability to efficiently handle an increase in users or resources.
   - A well-designed distributed system should be able to expand without losing performance, even when there is a significant increase in demand.
   - It ensures the system can grow smoothly without requiring drastic changes in architecture or operation.

4. Security:
   - Ensuring the security of data and resources in a distributed system is one of the primary challenges.
   - Security measures include:
     - Confidentiality: Protecting data from unauthorized access.
     - Integrity: Ensuring data remains accurate and unaltered.

- Availability: Ensuring authorized users can access the system when needed.
- Encryption techniques are commonly used to protect shared resources and communication from potential threats.

5. Failure Handling:
- In distributed systems, failures are more complex because one part of the system may fail while others continue to function normally.
- Handling failures requires strategies to detect and recover from hardware or software faults, ensuring the system can continue working despite issues in some components.
- Effective failure handling ensures that the system remains reliable and avoids data corruption or loss of functionality.

6. Concurrency:
- Concurrency occurs when multiple users or processes try to access the same resources at the same time.
- Distributed systems must manage concurrent access to resources (like reading, writing, or updating data) to prevent conflicts or errors.
- Mechanisms like locks and synchronization are used to ensure safe access in concurrent environments.

7. Transparency:
- Transparency in distributed systems means users and application developers should perceive the system as a single, unified entity rather than a collection of independent components.
- The system should hide the complexity of distribution, so users don't need to worry about where services or data are located.
- This simplifies the interaction with the system and improves user experience.

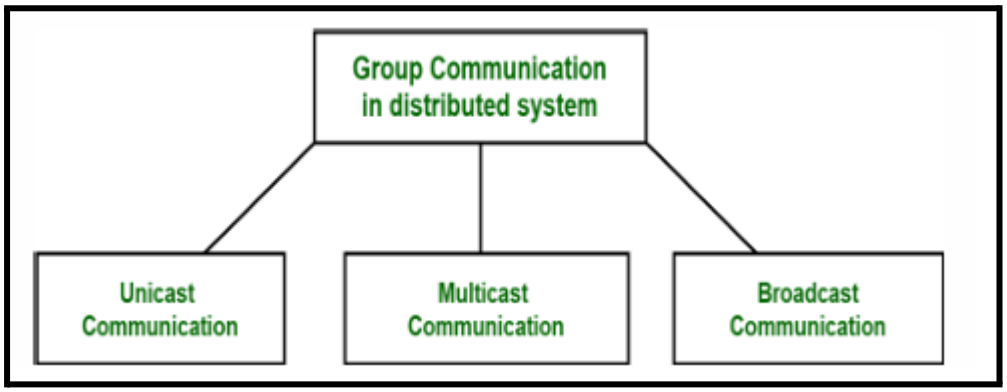
By addressing these design issues, distributed systems can become more efficient, reliable, and secure.

What is group communication in the context of distributed systems?

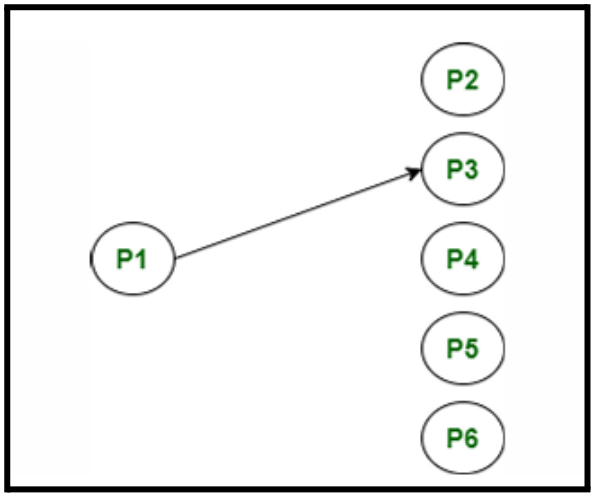
Group Communication in Distributed Systems refers to the interaction and coordination among multiple nodes (servers or processes) within a group to exchange data or perform actions collaboratively.
Messages are sent to the entire group, and all members receive the message.
Nodes can join or leave the group dynamically, and communication is often facilitated through multicast, where messages are directed to multiple recipients at once.

Key Concepts:

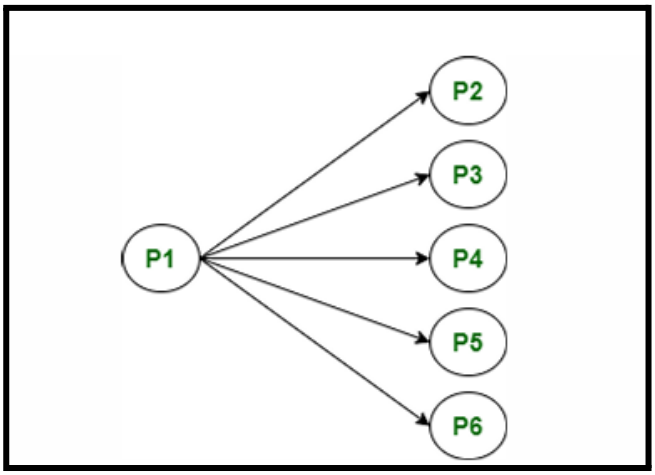
1. Communication Modes:



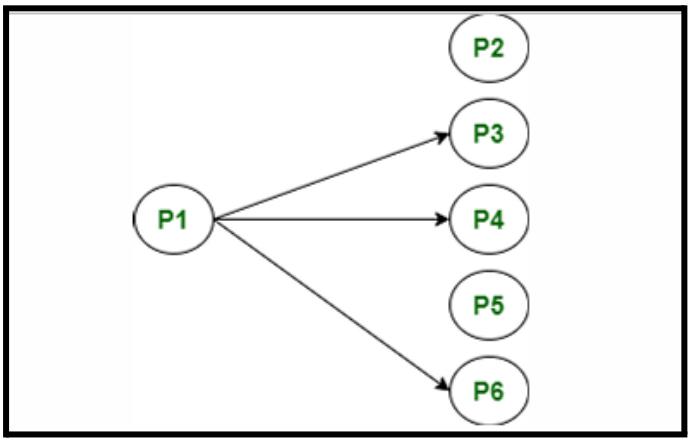
- **Unicast:** Communication between two individual processes (point-to-point).



- **Broadcast:** Communication where a message is sent to all processes in the network.



- **Multicast**: Communication where a message is sent to a specific group of processes.



2. Types of Group Communication:

- Closed Groups: Only members within the group can send messages to each other, and a process can send a message to itself.
- Open Groups: Members and non-members can send messages to the group.
- Overlapping Groups: A process can belong to multiple groups.
- Non-overlapping Groups: Processes belong to only one group at a time.

3. Message Ordering:

- FIFO Ordering: Messages are delivered in the order they were sent from the same sender.

- Causal Ordering: Messages are delivered in accordance with their causal relationships (if one message depends on another, that order is maintained).
- Total Ordering: All messages are delivered in the same order to all processes, ensuring consistency.

4. Reliability:
- Integrity: Ensures messages are delivered without duplication.
- Validity: Ensures that if a message is sent, it will be delivered.

Examples:
- Multicast communication in distributed databases for data replication.
- Broadcast communication in distributed sensor networks, where information is shared across all sensors.

Remote Procedure Call:

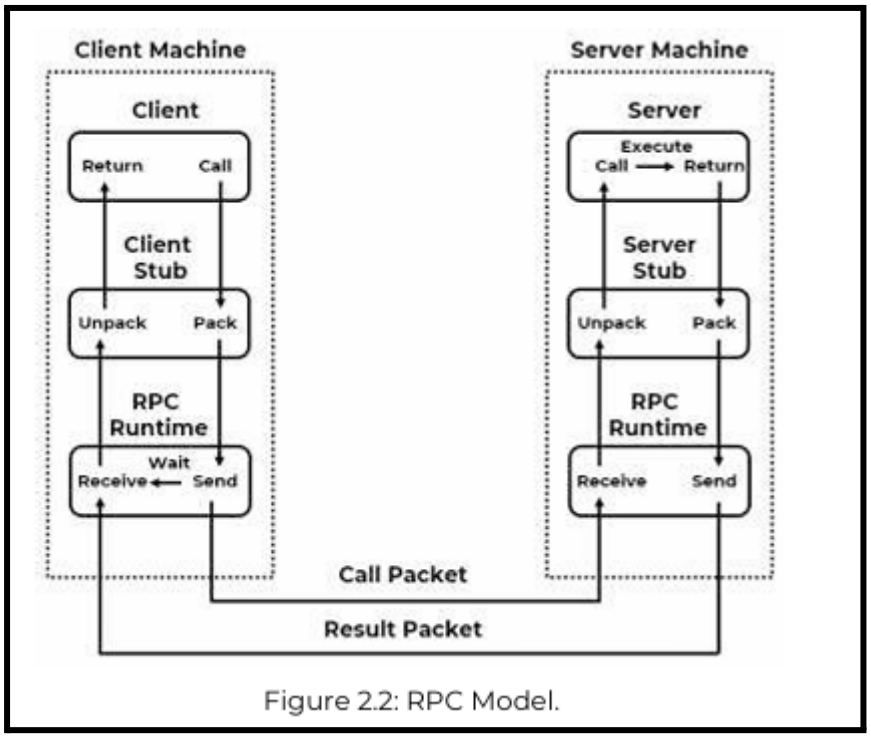


Figure 2.2: RPC Model.

The implementation of an RPC mechanism typically involves the following five elements:

- Client: The user process that initiates a remote procedure call.

- Client Stub:
  - Receives a call request from the client, packs a specification of the target procedure and its arguments into a message.
  - On receipt of the result from the procedure execution, it unpacks the result and passes it to the client.

- RPC Runtime: Manages the transmission of messages across the network between client and server machines, including tasks such as encryption, routing, and acknowledgment.

- Server Stub: Performs similar tasks as the client stub but on the server side.

- Server: Executes the appropriate procedure and returns the result to the client.

Execution of RPC (Remote Procedure Call)

1. Client initiates a request: The client calls a procedure as if it's local, but it's handled by the client stub, which packages the procedure name and arguments into a message (marshalling).

2. RPC runtime and network transmission: The RPC runtime sends the message across the network to the server stub, which unpacks the message (unmarshalling) and invokes the procedure on the server.

3. Return of results: The server executes the procedure, and the result is sent back through the server stub and RPC runtime to the client stub, which unpacks and returns the result to the client.

Advantages of Remote Procedure Call

1. Remote procedure calls support both process-oriented and thread-oriented models.
2. The internal message-passing mechanism of RPC is hidden from the user.
3. Minimal effort is required to re-write or re-develop code when using RPC.
4. Remote procedure calls can be used in both distributed and local environments.

Disadvantages of Remote Procedure Call

1. Remote procedure call is a concept that can be implemented in different ways; it is not a standardized method.
2. RPC lacks flexibility for various hardware architectures, as it is interaction-based.
3. The use of remote procedure calls increases costs due to communication overhead.

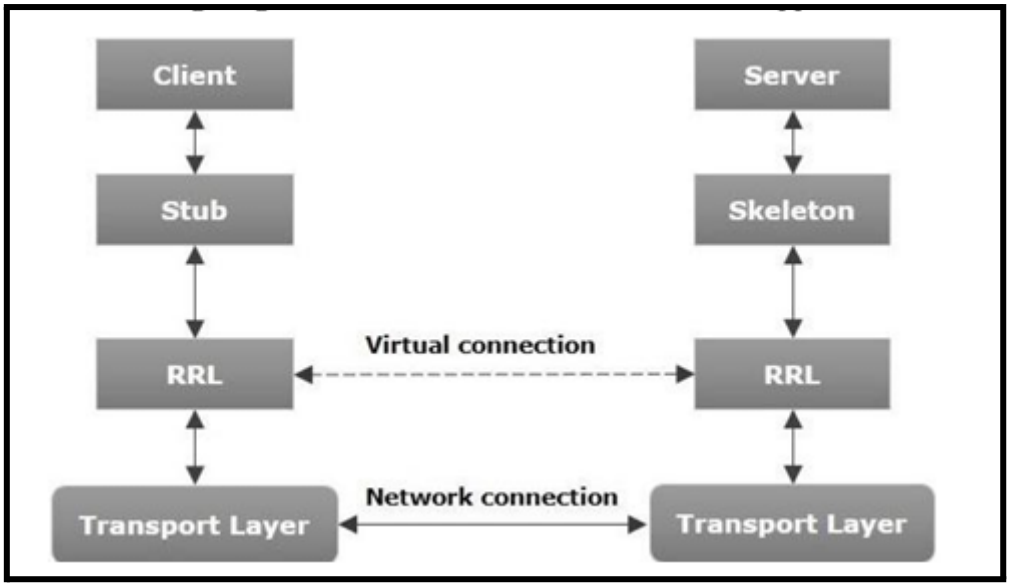
Remote Method Invocation (RMI)

Definition:
RMI stands for Remote Method Invocation, which allows an object residing in one system (JVM) to access or invoke an object running on another JVM. RMI is commonly used in distributed applications to facilitate remote communication between Java programs and is part of the `java.rmi` package.

Goals of RMI:
- To minimize the complexity of distributed applications.
- To preserve type safety.
- Support distributed garbage collection.
- Minimize the difference between working with local and remote objects.

Architecture of an RMI Application:
An RMI application consists of two main programs: the server and the client.
- Server program: Creates a remote object and provides access to the client via a reference to this object.
- Client program: Requests the remote object and invokes its methods.

The architecture consists of the following layers:



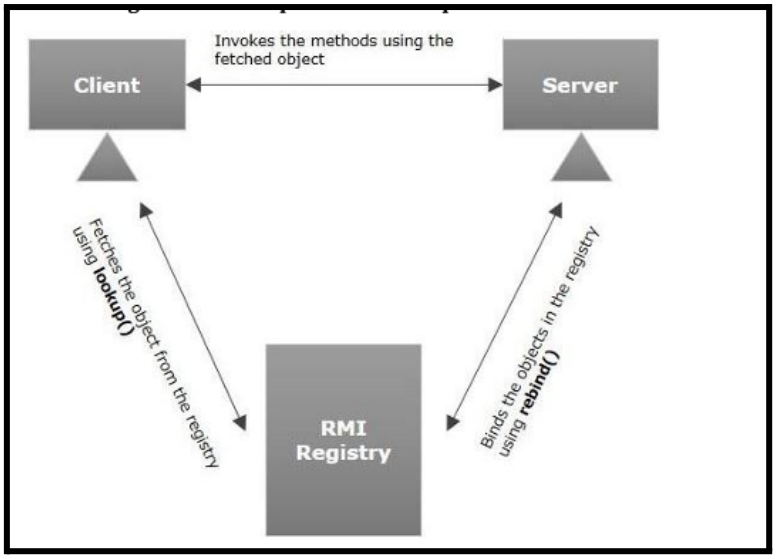
1. Transport Layer:
   - Responsible for establishing and managing connections between the client and the server.
   - Manages existing connections and sets up new ones.

2. Stub:
   - A proxy representation of the remote object on the client side.
   - Acts as a gateway for the client program to communicate with the server.

3. Skeleton:
   - Resides on the server side and communicates with the stub to pass requests to the remote object.

4. Remote Reference Layer (RRL):
   - Manages references from the client to the remote object.

Working of an RMI Application:
- When the client calls a remote object, the request is first received by the stub on the client side.
- The stub forwards the request to the RRL.
- The RRL on the client side calls the `invoke()` method of the remote object, which is passed to the server-side RRL.
- On the server side, the request is passed to the skeleton, which invokes the appropriate method on the server-side object.
- The result is returned through the skeleton, RRL, and stub back to the client.



Marshalling and Unmarshalling:
- Marshalling: When a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message (primitive types are packed, and objects are serialized) before being sent over the network.
- Unmarshalling: On the server side, the packed parameters are unpacked, and the method is invoked.

RMI Registry:
- The RMI registry is a namespace where all server objects are registered.
- When a server creates an object, it registers it using the `bind()` or `reBind()` method with a unique name called the bind name.
- To invoke a remote object, the client fetches the reference from the registry using its bind name via the `lookup()` method.

Difference between **Remote Procedure Call (RPC)** and **Remote Method Invocation (RMI)**

Feature	RPC (Remote Procedure Call)	RMI (Remote Method Invocation)
Programming Paradigm	Procedural-based	Object-oriented (Java-based)
Language Dependency	Language-independent (works with multiple languages like C, Python)	Java-specific
Data Handling	Handles simple data types (e.g., integers, strings)	Handles complex objects through object serialization
Communication Mechanism	Protocols like HTTP, TCP/IP, or UDP	Java RMI Protocol (TCP/IP)
Object Support	Does not inherently support objects	Fully supports objects and remote method invocation
Data Serialization	Manual serialization of complex data structures	Automatic serialization and deserialization of objects
Ease of Use	Simpler to implement but limited in flexibility	More complex, but allows the passing of entire objects and methods
Use Case Example	Remote procedure for fetching data from a server	Invoking methods on remote Java objects (e.g., bank account object)
Server Interaction	Client calls remote procedures (functions)	Client calls remote methods on objects
Performance	Relatively faster due to simple data handling	May have overhead due to object serialization and Java-specific communication
Abstraction	Abstracts procedure calls between client and server	Abstracts method calls between client and server objects
Platform Compatibility	Cross-platform	Requires both client and server to run Java

Unit 2:

ELECTION ALGORITHM:

1. Many algorithms used in distributed systems require a coordinator.
2. In general, all processes in the distributed system are equally suitable for the role.
3. Election algorithms are designed to choose a coordinator.
4. Any process can serve as coordinator.
5. Any process can “call an election”.
6. There is no harm in having multiple concurrent elections.
7. Elections may be needed when the system is initialized, or if the coordinator crashes or retires.
8. An example of election algorithm is the Bully Algorithm.

Assumptions and Requirement of Election Algorithm:

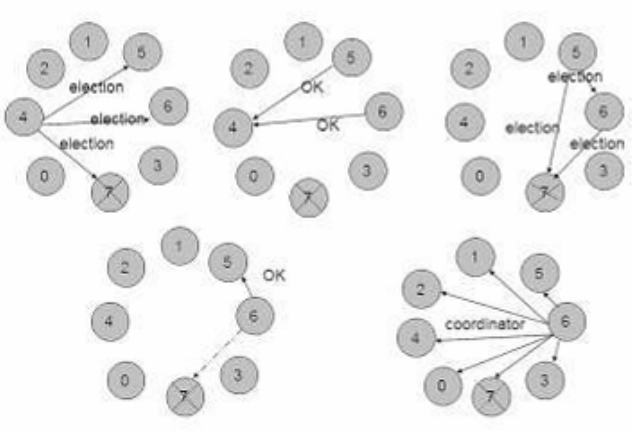
1. Every process/site has a unique ID.
2. Example: The network address or a process number.
3. Every process in the system should know the values in the set of ID numbers, although not which
processors are up or down.

BULLY ALGORITHM:

1. This algorithm is planned on assumptions that:
   a. Each process involved within a scenario has a process number that can be used for unique identification.
   b. Each process should know the process numbers of all the remaining processes.
   c. Scenario is synchronous in nature.
   d. A process with the highest process number is elected as a coordinator.

2. Process ‘p’ calls an election when it notices that the coordinator is no longer responding.
3. Process ‘p’ sends an election message to all higher-numbered processes in the system.
4. If no process responds, then p becomes the coordinator.
5. If a higher-level process (q) responds, it sends ‘p’ a message that terminates p’s role in the algorithm.
6. The process ‘q’ now calls an election (if it has not already done so).
7. Repeat until no higher-level process responds.
8. The last process to call an election “wins” the election.
9. The winner sends a message to other processes announcing itself as the new coordinator.

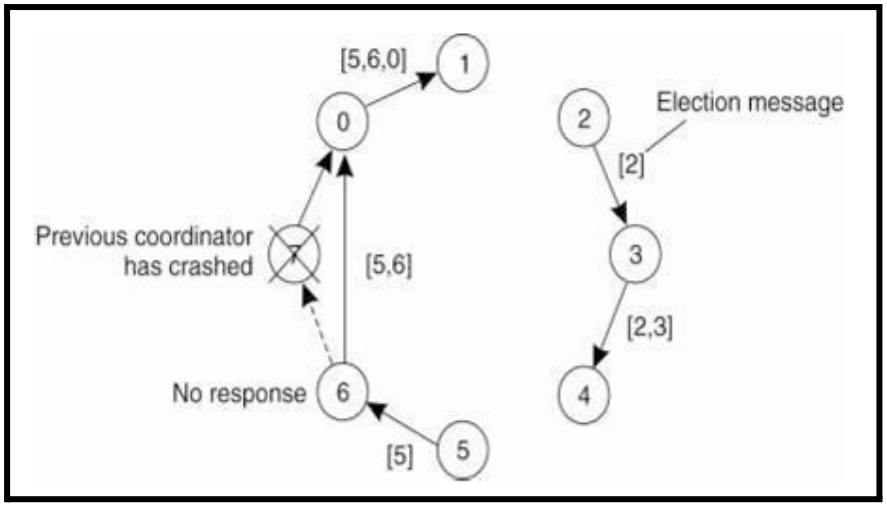
Example:



1. Process (4) calls an election when it notices that the coordinator is no longer responding.
2. Process (4) sends an election message to all higher-numbered processes in the system.
3. If no process responds, then Process (4) becomes the coordinator.
4. If a higher-level process (5, 6) responds, it sends process (4) a message that terminates (4)'s role in the
algorithm.
5. Now process (5) now calls an election (if it has not already done so).
6. Repeat until no higher-level process responds.
7. The last process to call an election "wins" the election.
8. The winner sends a message to other processes announcing itself as the new coordinator.

Ring algorithm:

1. The ring algorithm assumes that the processes are arranged in a logical ring, and each process knows the order of the ring of processes.
2. Processes are able to "skip" faulty systems: Instead of sending to process j, they send to j + 1. Faulty systems are those that don't respond within a fixed amount of time.
3. Process 'P' thinks the coordinator has crashed and builds an ELECTION message which contains its own ID number.
4. Process 'P' sends the ELECTION message to its first live successor.
5. Each process that receives the ELECTION message adds its own ID number and forwards it to the next process in the ring.
6. It is acceptable to have multiple elections at once in this algorithm.
7. When the message returns to the origin process 'P,' it sees its own process ID in the list and knows that the circuit is complete.
8. Process 'P' circulates a COORDINATOR message with the highest process number in the list as the new coordinator.
9. Both 2 and 5 elect process 6 as the new coordinator in these example orders:
   - [5,6,0,1,2,3,4]
   - [2,3,4,5,6,0,1]



Inter-Process Communication (IPC)

In distributed systems refers to the methods used for processes running on different machines to communicate and coordinate their activities. Common IPC mechanisms include message passing, where processes exchange data over the network, and Remote Procedure Call (RPC), where one process can execute a procedure on another machine.
These mechanisms allow processes to collaborate and share information effectively across a distributed environment.

Mutual Exclusion

Mutual Exclusion in distributed computing refers to a mechanism that ensures that multiple processes or nodes do not access a shared resource simultaneously, thereby avoiding conflicts or inconsistent states. It's a critical concept for maintaining data integrity and ensuring that concurrent processes or transactions operate in a synchronized manner.

1. Centralized Algorithm:

- A single coordinator node manages access to the shared resource.
- All nodes must request access from the coordinator, which grants or denies the request.
- Pros: Simple, easy to implement.
- Cons: Single point of failure, possible bottleneck.

2. Distributed Algorithm (Ricart-Agrawala Algorithm):

- All nodes cooperate to achieve mutual exclusion by exchanging messages.
- A node requests permission from all other nodes before accessing the shared resource.
- Once all nodes grant permission, it can proceed.
- Pros: No single point of failure.
- Cons: High message overhead.

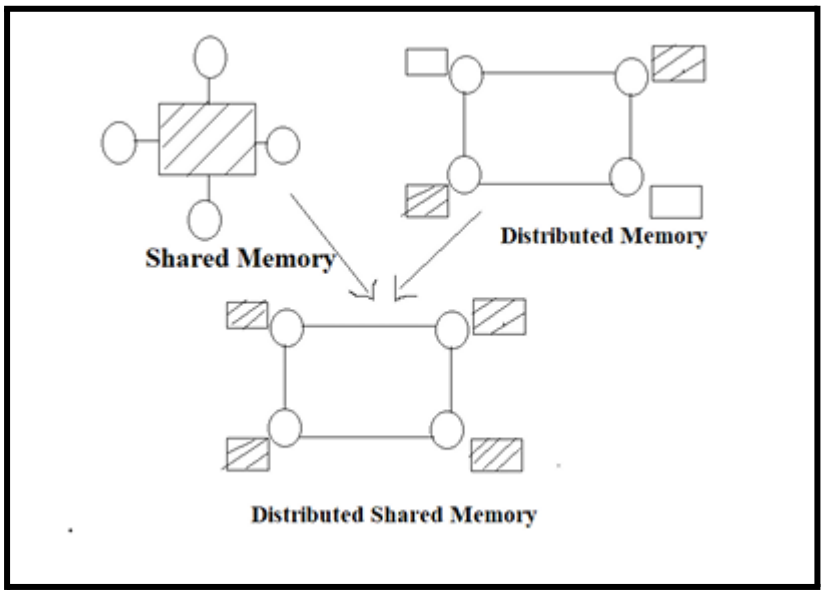
Key Challenges:

- Deadlock: Multiple processes wait indefinitely for a resource, causing a system halt.
- Starvation: Some processes may be denied access to the resource indefinitely.
- Fault Tolerance: Distributed systems must handle node failures and network issues while maintaining mutual exclusion.

Unit3:

Distributed Shared Memory:

Definition: Distributed Shared Memory is a mechanism allowing end-user processes to access shared data without using inter-process communications. The DSM system is to make inter-process communication transparent to end users. By using DSM variables can shared directly and the cost of communication is invisible.



Types of DSM:

1. On-Chip Memory
2. Bus-Based Multiprocessors
3. Ring-Based Multiprocessors
4. Switched Multiprocessors

Issues in Designing and Implementing DSM Systems:
(DR-MH-GST)

1. Granularity
2. Structure of Shared Memory
3. Memory Coherence and Access Synchronization
4. Data Location and Access
5. Replacement Strategy
6. Thrashing
7. Heterogeneity

1. Granularity:
Granularity refers to the block size of shared memory. When data is shared and transferred across the network, block faults (either word or page faults) may occur. The issue lies in the size of the memory chunk that contains the word.

2. Structure of Shared Memory:
This refers to the layout of shared data in memory. The structure of shared memory depends on the different applications supported by the Distributed Shared Memory (DSM) system.

3. Memory Coherence and Access Synchronization:
A DSM system allows duplicates of shared data items, with copies of the data available in the main memories of several nodes. To address this issue, consistency must be ensured for shared data located in the main memories of two or more nodes.

4. Data Location and Access:
Shared memory requires shared data, and in a DSM system, it is essential to locate and retrieve the data accessed by a user process.

5. Replacement Strategy:
This strategy involves replacing a data block with a new one. When the local memory of a node is full, a cache miss at that node triggers the retrieval of a data block from a remote node, prompting a replacement.

6. Thrashing:
In a DSM system, memory blocks migrate between nodes based on demand, leading to thrashing. Different techniques are used to minimize thrashing. For example, application-controlled locking algorithms can be used to manage access to shared data patterns.

Suppose two variables, a and b, access the same page; in such cases, pages can be accessed using different variables while keeping the data consistent.

7. Heterogeneity:
Heterogeneity in DSM systems applies to operating systems, computer hardware, networks, and the system's implementation by developers. Heterogeneity provides an environment like client-server, functioning as middleware with a set of services that interact with end users.

To achieve consistency in distributed shared memory is important. But then why is Strong Consistency not that popular? Discuss.

Strong consistency in Distributed Shared Memory (DSM) systems ensures that every read returns the most recent write, but it's not popular due to several drawbacks

While this seems like an ideal solution for maintaining consistency, there are several reasons why strong consistency is not that popular in practical implementations

1. High Latency and Performance Overhead:
Strong consistency requires synchronization across all nodes after every write, leading to delays and slower performance, especially in distributed networks.

2. Scalability Issues:
Maintaining strong consistency across a large number of nodes becomes inefficient and can lead to a scalability bottleneck.

3. Reduced Availability:
According to the CAP theorem, strong consistency sacrifices availability during network partitions, making the system less fault-tolerant.

4. Resource-Intensive:
Ensuring strong consistency requires significant computational and network resources, which is impractical for many real-time applications.

5. Unnecessary for Many Applications:
Many applications, like social media or e-commerce, can tolerate some inconsistency and opt for eventual or causal consistency models, which offer better performance without strict consistency guarantees.

In practice, systems prioritize performance, availability, and scalability over strict consistency, making strong consistency less popular.

PYQ:

a) Discuss RPC execution with a diagram. (2M)

b) Difference between RMI and LMI. (2M)

c) Explain any two Distributed computing models in detail. (2M)

d) With the help of a diagram, explain the Message Passing mechanism. (2M)

e) What is Inter-process communication in Distributed Systems? (2M)

f) Define the working of Java RMI. (2M)

Q.2 (Attempt one of the following) – 5 Marks

a) How would you alter physical clocks to logical clocks with the help of an algorithm? (5M)

OR

b) For instance, if we have five processes, one must be selected as coordinator. Who can be one and discuss various scenarios? (5M)

Q.3 (Attempt one of the following) – 5 Marks

a) To achieve consistency in distributed shared memory is important. But then why is Strong Consistency not that popular? Discuss. (5M)

OR

b) Discuss issues in implementing DSM Systems. (5M)