Times asked: 6 times
5 times
4 times
3 times
2 times
1 time

# indicates 5-mark question

View complete PYQ analysis & Previous year papers

# Distributed Computing Question bank

## 1. Introduction to Distributed Systems

1. Explain the different issues (or challenges) faced in distributed systems. #
2. Explain the goals of distributed systems. #
3. Differentiate between NOS, DOS and Middleware in the design of distributed systems. #
4. What is middleware? Explain the services offered by middleware in distributed systems. #

## 2. Communication

5. What is Remote Procedure Call? Describe the working of RPC in detail.
6. Explain the various ordered semantics used for Many-to-Many communication with examples.
7. Explain group communication and its types.

## 3. Synchronization

8. Explain Raymond's tree-based algorithm for distributed mutual exclusion.
9. Explain the Bully election algorithm with a suitable example.
10. What is logical clock? Why are logical clocks required in distributed systems? How does Lamport synchronize logical clock? Which events are said to be concurrent in Lamport's timestamp.
11. What are physical clocks? Explain any one physical clock synchronization algorithm.
12. What is distributed mutual exclusion? Explain how Suzuki–Kasami's broadcast algorithm achieves distributed mutual exclusion.
13. Explain Ricart–Agrawala's algorithm and how it optimizes message overhead in achieving mutual exclusion.
14. Explain Chandy-Misra-Haas Algorithm for distributed deadlock detection.

## 4. Resource and Process Management

15. Explain the key features of a Global Scheduling algorithm. #
16. Explain load estimation policies, process transfer policies, and location policies used in load balancing approaches in distributed systems.
17. Explain code migration and its need in distributed systems. Explain the role of process-to-resource binding and resource-to-machine binding in code migration.
18. Explain how load balancing benefits a distributed system.

## 5. Replication, Consistency and Fault Tolerance

19. What is fault tolerance? Describe different types of failure models.
20. Explain any five data-centric consistency models with example data stores.

## 6. Distributed File Systems

21. What are the desirable features of a good Distributed File System (DFS)?
22. Explain synchronization in Distributed File Systems and its challenges. #
23. Discuss the Google File System (GFS) as a scalable distributed file system.
24. Explain different file caching schemes used in Distributed File Systems.

**Module-wise Marks Weightage and Question Count**

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **2025 Aug** | 15 (2) | 10 (1) | 40 (4) | 25 (3) | 5 (1) | 25 (3) |
| **2025 May** | 15 (2) | 15 (2) | 35 (4) | 25 (3) | 20 (2) | 10 (1) |
| **2024 Dec** | 15 (2) | 15 (2) | 30 (3) | 25 (3) | 20 (2) | 15 (2) |
| **2024 May** | 10 (1) | 10 (1) | 30 (3) | 20 (2) | 30 (3) | 20 (2) |
| **2023 Dec** | 15 (2) | 15 (2) | 30 (3) | 20 (2) | 30 (4) | 10 (1) |
| **2023 May** | 10 (2) | 30 (3) | 40 (5) | 25 (3) | 10 (1) | 10 (1) |
| **Estimate** | **15 (2)** | **15 (2)** | **30-40 (3-4)** | **25 (3)** | **20 (2)** | **10-15 (1-2)** |
| **Total** | 80 | 95 | 205 | 140 | 115 | 90 |

# Asked once:

## 1. Introduction to Distributed Systems

-

## 2. Communication

1. Differentiate between RMI and RPC. #
2. Explain how transparency is achieved in RPC. #
3. Explain various forms of message-oriented communication with suitable example.

## 3. Synchronization

4. Compare two election algorithms and recommend the most efficient one for large-scale distributed systems.
5. Explain the need of election algorithm. #

## 4. Resource and Process Management

6. Explain code migration and its techniques.
7. Describe code migration issues in detail.

## 5. Replication, Consistency and Fault Tolerance

8. Explain how replication helps in achieving fault tolerance. #
9. Explain how Monotonic read consistency model is different than Read your Write consistency model.#
10. Discuss the technique to achieve Process resilience.
11. Discuss design and implementation issues of distributed shared memory.
12. Write a short note on Replication and the types of it.
13. Discuss and differentiate various client consistency models.

## 6. Distributed File Systems

14. Explain the working of Distributed File System with its applications.
15. How are modifications propagated in file caching schemes? #

# Distributed Computing Answer bank

# indicates 5-mark question

## 1. Introduction to Distributed Systems

**1.** Explain the different issues (or challenges) faced in distributed systems. #

// asked for 10 marks once

1. **Communication Issues**
   Problems like network delays, message loss, and unreliable connections can affect communication between machines.
2. **Process Management**
   Includes creating, scheduling, coordinating, and terminating processes across machines.
3. **Data Management**
   Ensuring consistency, storage, replication, and integrity of distributed data.
4. **Fault Tolerance and Reliability**
   System must detect failures and recover quickly using mechanisms like replication and backup.
5. **Security**
   Authentication, authorization, encryption, and privacy protection are necessary to prevent unauthorized access.
6. **Heterogeneity**
   Different hardware, operating systems, and networks must work together seamlessly.
7. **Transparency**
   The system should hide its distributed nature and appear as a single system to users.
8. **Scalability**
   The system should handle growth in users, data, and nodes without affecting performance.
9. **Concurrency Control**
   Managing simultaneous access to shared resources is necessary to avoid conflicts and inconsistencies.
10. **Synchronization**
    Maintaining the correct order of events without a global clock is difficult.

**2.** Explain the goals of distributed systems. #

// asked for 10 marks once

**Goals of Distributed Systems**

1. **Resource Sharing**
   Allows different nodes to share hardware, software, and data over the network, improving overall utilization.
2. **Concurrency**
   Supports multiple users and processes working at the same time.
3. **Scalability**
   The system can grow by adding more nodes to handle increased demand.
4. **Fault Tolerance / Reliability**
   Failure of one node does not cause the entire system to fail.
5. **Openness**
   System follows standard protocols and interfaces, making it easier to extend and integrate with other systems.
6. **Performance**
   Ensures efficient use of resources to provide faster response time and better throughput.
7. **Load Balancing**
   Workload is distributed evenly across nodes to avoid overloading any single system.
8. **Flexibility / Modularity**
   System can be easily modified, upgraded, or expanded without affecting the whole system.
9. **Availability**
   System remains accessible and operational for users most of the time.
10. **Transparency**
    The complexity of multiple machines is hidden, making the system appear as a single unit.

**Failure transparency and Location transparency** (asked once)

**Failure Transparency**

- Hides failures from users so the system continues to operate normally.
- Achieved using redundancy, replication, and recovery mechanisms.

**Example:** If one server fails, another replica handles the request without user noticing.

**Location Transparency**

- Users access resources without knowing their physical location.
- Achieved using naming and directory services.

**Example:** Accessing a file using a path without knowing which server stores it.

**3.** Differentiate between NOS, DOS and Middleware in the design of distributed systems. #

| Parameter | Network OS (NOS) | Distributed OS (DOS) | Middleware |
|---|---|---|---|
| **Degree of Transparency** | Low (Users are aware of multiple machines). | Very High (System appears as a single unit). | Moderate (Hides distribution, but runs on top of host OS). |
| **Same OS on all Nodes** | Not necessary (Heterogeneous). | Usually requires the same OS (Homogeneous) | Not necessary (Designed to bridge different OS). |
| **Number of Copies of OS** | Multiple (One per node). | One logical OS across the system. | Multiple local OS + One shared layer. |
| **Basis for Communication** | File sharing and network services. | Shared memory or message passing. | Remote Procedure Calls (RPC) or RMI. |
| **Resource Management** | Per node (Local). | Global (Centralized or Distributed). | Shared across applications (Distributed). |
| **Scalability** | High. | Limited or moderate. | High (Very flexible). |
| **Openness** | It is open in design. | It is closed in design. | Highly open and extensible. |

**4.** What is middleware? Explain the services offered by middleware in distributed systems. #

**Middleware**

- Middleware is a software layer between OS and applications that enables communication and coordination in distributed systems.
- It hides complexity and provides common services for distributed applications.

**Services offered by Middleware**

1. **Naming Service**
   Helps locate resources or services using names instead of physical addresses, making access easier.
2. **Persistence Service**
   Stores data or objects so they can be reused later, even after system restarts or failures.
3. **Messaging Service**
   Enables communication between systems using message passing (request–response or asynchronous).
4. **Querying Service**
   Allows searching and retrieving resources based on attributes or specific requirements.
5. **Security Service**
   Ensures authentication, authorization, and data protection during communication.

# 2. Communication

5. What is Remote Procedure Call? Describe the working of RPC in detail.

**Remote Procedure Call (RPC)**

- Remote Procedure Call (RPC) is a communication mechanism used in distributed systems that allows a process to call a procedure located on another machine as if it were a local procedure call.
- In RPC, a local process sends parameters to a remote procedure and receives the result.
- The client stub packs the parameters into a message (marshalling).
- The server stub unpacks the data (unmarshalling) and executes the procedure.

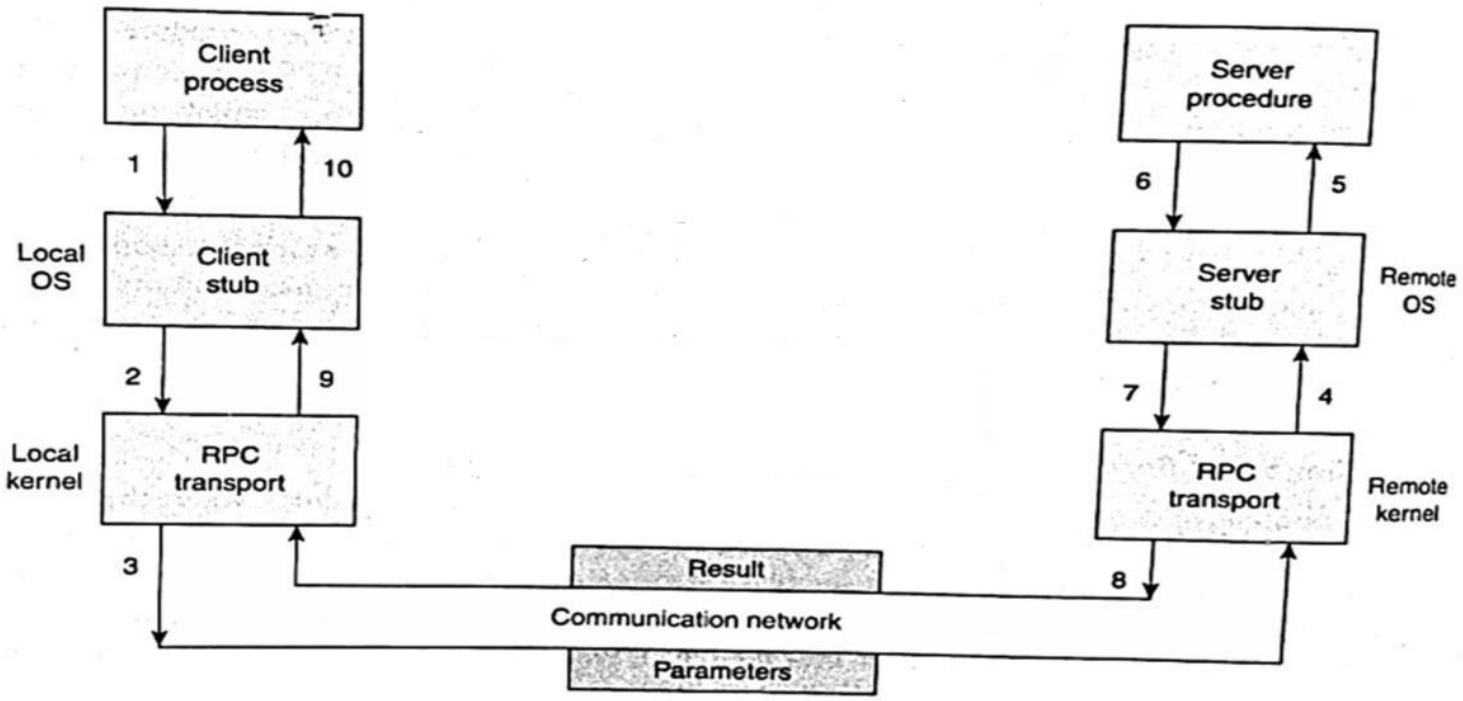



Figure 5.3 Working of RPC.

**Steps of RPC:**

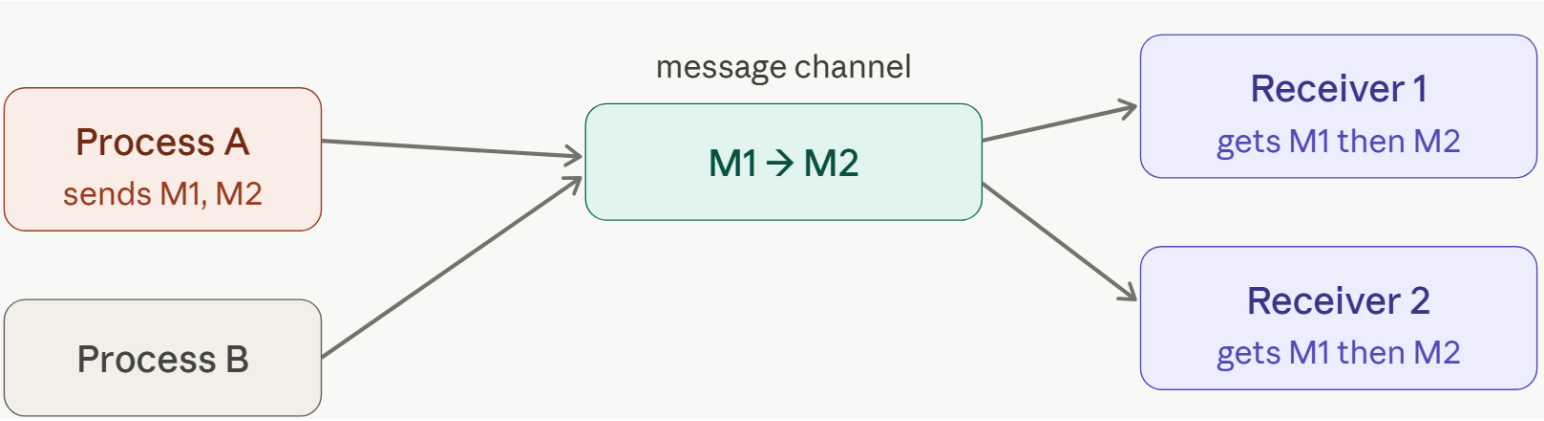
1. The client procedure calls the client stub like a normal procedure call.
2. The client stub marshals (packs) the parameters into a message.
3. Client OS transmits the message over the network to the server (remote) OS.
4. Server OS passes the message to the server stub.
5. Server stub unmarshals (unpacks) the parameters and calls the server procedure.
6. The server procedure performs the task and returns the result to the server stub.
7. The server stub packs the result into a message and calls the server OS.
8. The server's OS sends the message back to the client's OS over the network.
9. The client's OS passes the message to the client stub.
10. The client stub unpacks the result and returns it to the client procedure.

6. Explain the various ordered semantics used for Many-to-Many communication with examples.

**1. FIFO Ordering (First-In-First-Out)**

- Messages sent by a single sender are received in the same order by all receivers.
- Ordering is maintained per sender, not across different senders.
- Ensures basic consistency for messages from the same sender, making communication predictable.

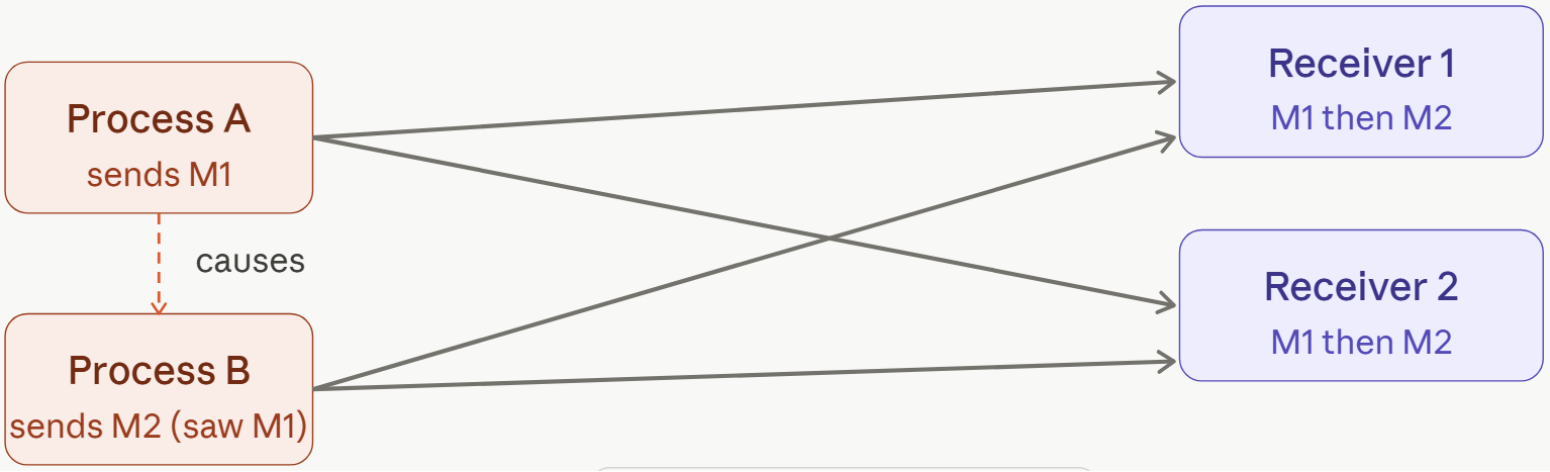
**Example:** If Process A sends M1 then M2 → All receivers get M1 before M2.




**2. Causal Ordering**

- Messages that are causally related (dependent) must be delivered in the same order.
- If one message influences another, the order is preserved.
- Preserves the logical relationship between events, so cause-and-effect is never violated.

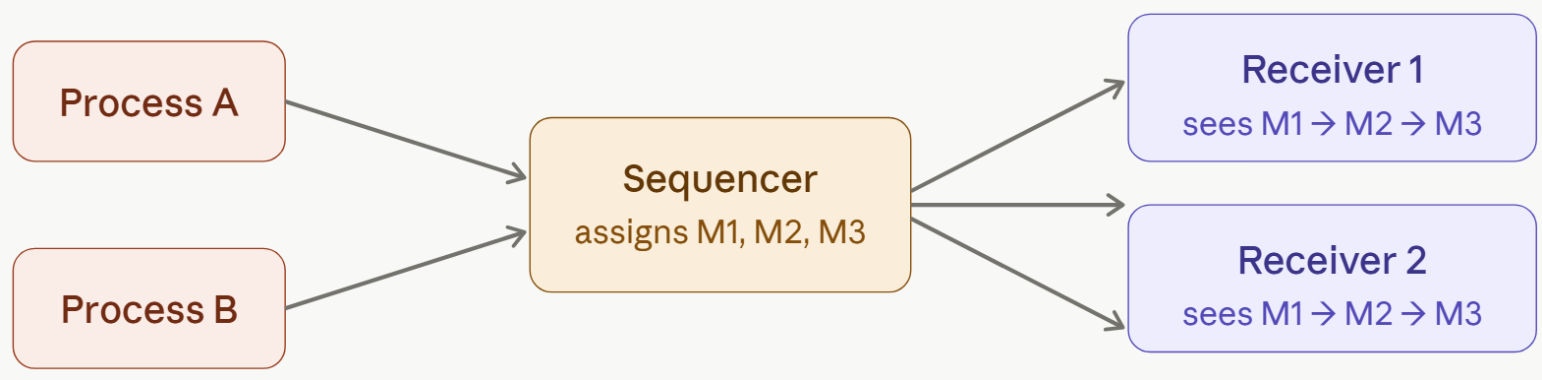
**Example:** If M1 causes M2 → All processes must receive M1 before M2.



**3. Total Ordering (Global Ordering)**

- All messages are delivered in the same order at all processes, regardless of sender.
- Ensures a consistent global sequence of messages.
- Guarantees that all processes see the exact same sequence of messages, avoiding inconsistencies.

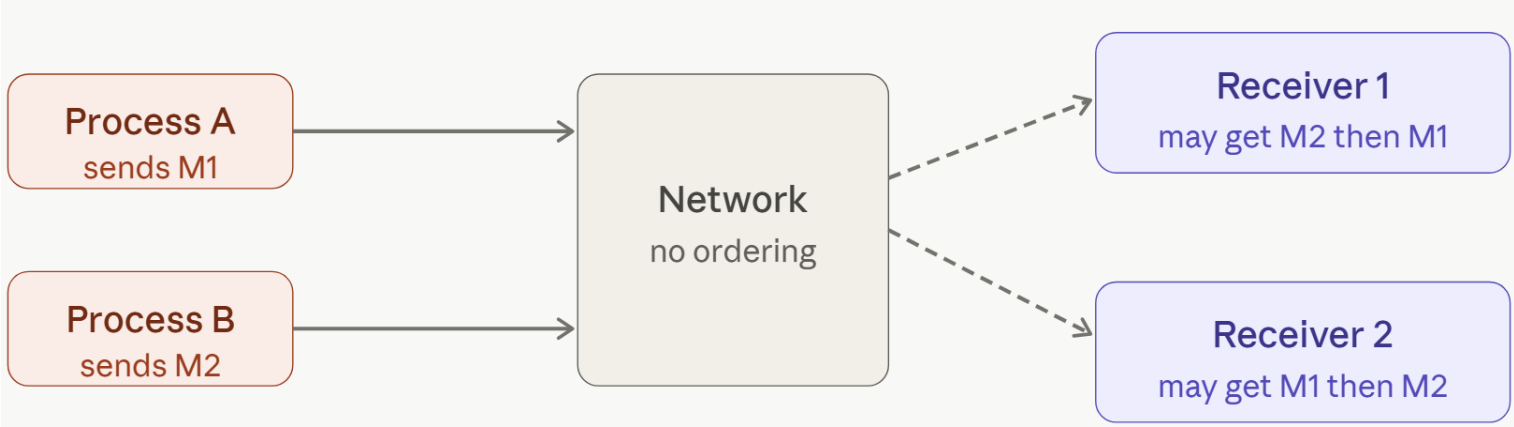
**Example:** If one process sees M1 before M2 → All processes must see M1 before M2.




**4. Unordered / No Ordering**

- Messages can be delivered in any order.
- Provides maximum flexibility and speed, but without any guarantee of message sequence.

**Example:** M1 and M2 may arrive in any order at different processes.



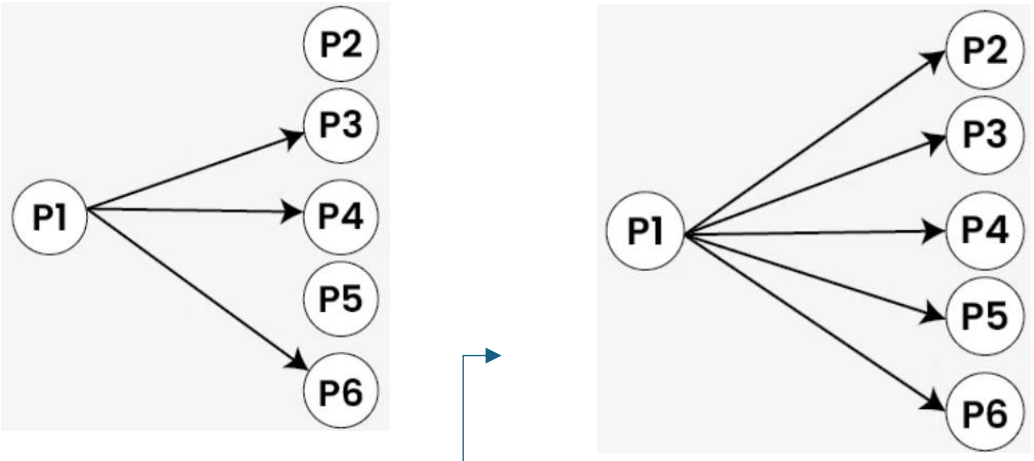
**7.** Explain group communication and its types.

**Group Communication**

- Group communication is a communication mechanism in distributed systems where a process sends a message to multiple processes simultaneously.
- It is mainly used to improve coordination, reliability, and efficiency among distributed processes.

**Types of Group Communication**

1. **Multicast Communication (1:M or One-to-Many)**
   A message is sent to a selected group of processes; only group members receive it.
   **Example:** A server pushing updates to all subscribed clients (e.g., stock updates)




2. **Broadcast Communication (One-to-All)**
   A message is sent to all processes in the distributed system regardless of group membership.
   **Example:** A router broadcasting its presence to all nodes on a network.
3. **Many-to-One Communication (M:1)**

   Multiple senders send messages to a single receiver. It is used for collecting or aggregating data.
   **Example:** Sensors sending data to a central server.

**Properties of Group Communication**

1. **Atomicity (All-or-Nothing Property)**
   A message sent to a group is delivered to all members or to none of them.
2. **Message Ordering**
   Ensures that all group members receive messages in the same sequence.
3. **Reliability**
   Ensures that messages are delivered correctly without loss.
4. **Fault Tolerance**
   If a member fails during communication, the system continues functioning correctly for remaining members.

# 3. Synchronization

8 Explain Raymond's tree-based algorithm for distributed mutual exclusion.

**Raymond's tree-based algorithm**

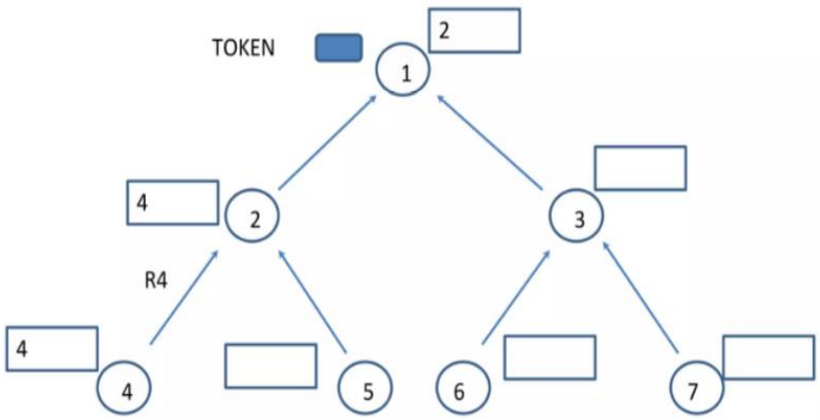
- Raymond's algorithm is a token-based distributed mutual exclusion algorithm.
- It organizes processes in a logical tree structure, where only the process holding the token can enter the critical section.

**Working of the Algorithm**

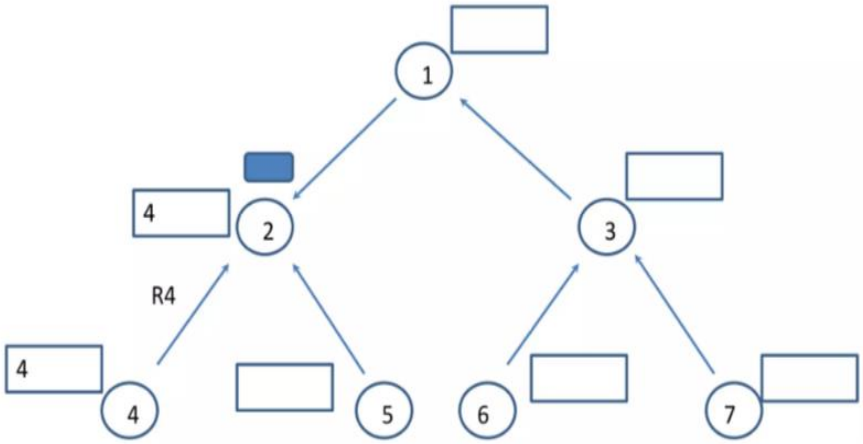
- When a process wants to enter the critical section and does not have the token, it sends a request to its parent.
- The request is forwarded along the tree until it reaches the node holding the token.
- Each node keeps track of pending requests in a queue.
- When the token holder receives a request, it passes the token to the requesting node.
- Once a process receives the token, it enters the critical section.
- After completing execution, it passes the token to the next node in its queue (if any).

**Example:**

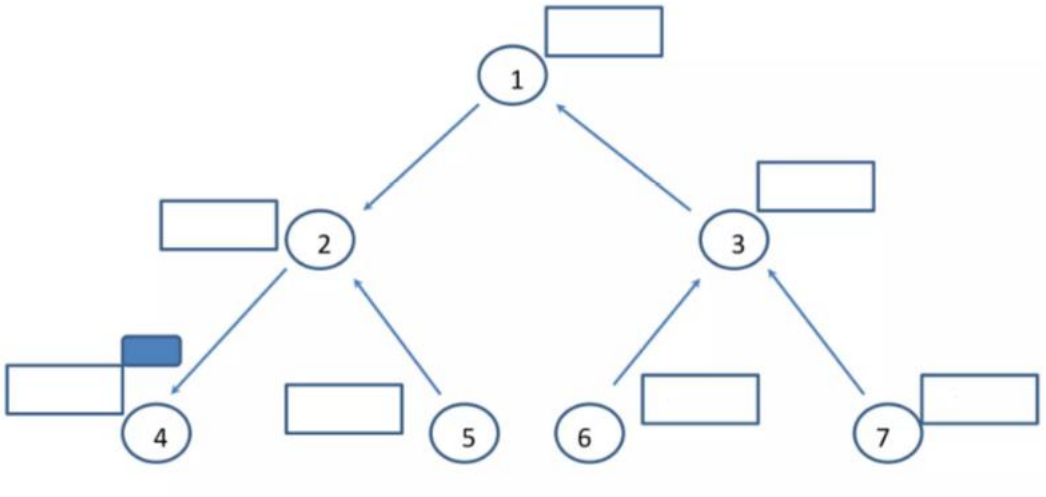
- Initially, node 1 holds the token.
- Node 4 wants to enter the critical section, so it sends a request to its parent (node 2).




- The request is forwarded from node 2 to node 1, and both nodes update their queues.
- Node 1 (token holder) processes the request and passes the token to node 2.



- Node 2 then forwards the token to node 4, which receives it and enters the critical section.




- After completing execution, node 4 will pass the token to the next requesting node if present.

**Conclusion**

- Raymond's algorithm achieves distributed mutual exclusion by using a single token that is passed through a logical tree structure.
- Requests are forwarded through parent pointers, so they reach the token holder efficiently without unnecessary messages.
- The token is then passed to requesting nodes in an orderly (queue-based) manner, ensuring fairness.
- Since only the node holding the token can enter the critical section, mutual exclusion is maintained across the system.

9. Explain the Bully election algorithm with a suitable example.

**Bully election algorithm:**

- The Bully Algorithm is a coordinator election algorithm used in distributed systems.
- It is used when the coordinator (leader) crashes, and a new coordinator must be selected.
- The process with the highest process ID always becomes the coordinator, which is why it is called the "Bully" algorithm i.e. the biggest process wins.

**Working of Bully Algorithm**

Assume every process has:

- A unique ID number
- Knowledge of all other process IDs

When a process detects that the coordinator has failed:

1. The process sends an ELECTION message to all processes with higher IDs.
2. If no process replies, it becomes the coordinator.
3. If any higher process replies with OK, it stops the election.
4. The higher process then starts a new election.
5. Finally, the highest active process becomes the coordinator.
6. The winner sends a COORDINATOR message to all other processes announcing itself as leader.

**Example:**

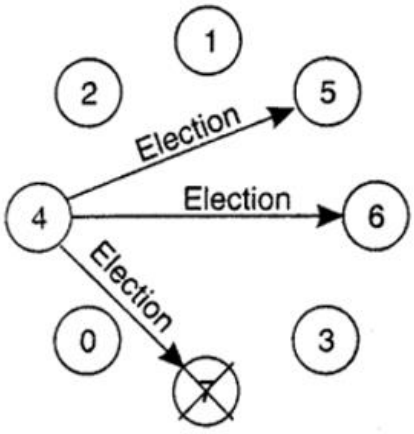
Assume 8 processes numbered 0 to 7. Process 7 is the coordinator.

**Step 1: Coordinator Failure**

- Process 7 crashes.

**Step 2: Election Initiation**

- Process 4 detects the failure.
- It sends ELECTION messages to higher ID processes 5, 6, 7. (a)



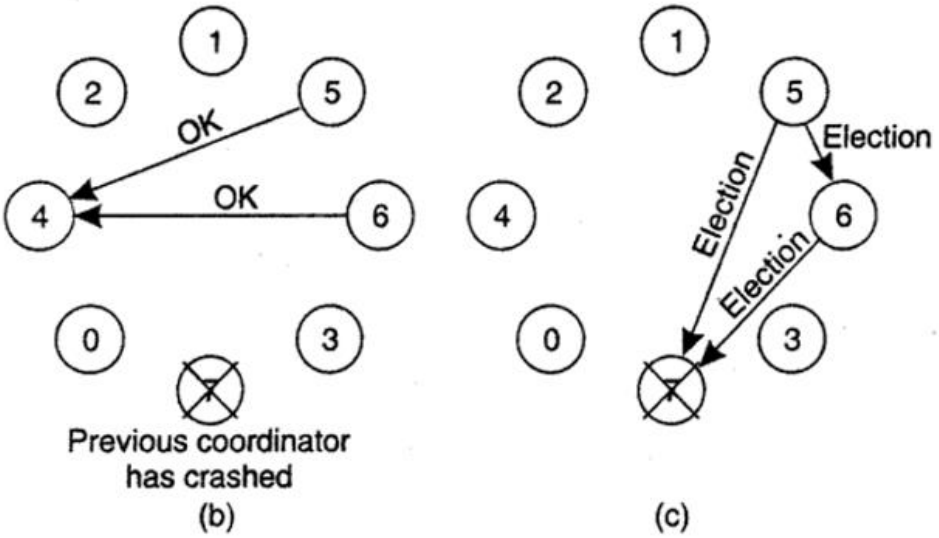
(a)

**Step 3: Higher Processes Respond**

- Processes 5 and 6 reply with OK (since they have higher IDs). (b)
- Process 4 stops the election.

**Step 4: New Election by Higher Process**

- Process 5 starts a new election.
- It sends ELECTION messages to 6 and 7. (c)
- Process 6 replies with OK. (d)

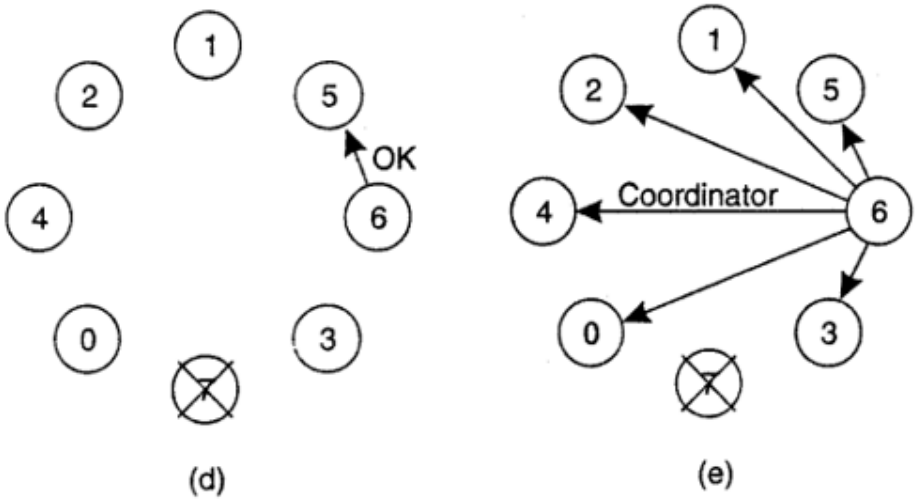



**Step 5: Final Election Attempt**

- Process 6 sends an ELECTION message to 7.
- No reply is received (since 7 has crashed).

**Step 6: New Coordinator Selected**

- Process 6 becomes the new coordinator.
- It sends a COORDINATOR message to all lower processes (0–5). (e)



**10.** What is logical clock? Why are logical clocks required in distributed systems? How Lamport does synchronizes logical clock? Which events are said to be concurrent in Lamport's timestamp.

**Logical Clock**

- A logical clock is a mechanism used to order events in a distributed system without using physical time.
- It is based on Lamport's "happened-before" relation to maintain event ordering.
- It assigns timestamps to events so that their sequence can be understood across different processes.

**Why Logical Clocks are Required**

- Distributed systems do not have a global clock.
- Physical clocks can be out of sync, leading to incorrect ordering of events.
- Logical clocks help maintain causal ordering of events across processes.
- They ensure that communication between processes follows the correct sequence of events.

**Rules for Synchronization**

1. **Increment Rule (Internal Event):**
   Before executing any event, increment the logical clock by 1.
2. **Send Rule:**
   When sending a message, increment the clock and attach its timestamp to the message.
3. **Receive Rule:**
   When a message with timestamp T is received,
   update the clock as:

$$Clock = \max(\text{local clock}, T) + 1$$

This ensures that if event A happened-before event B, then
**Timestamp(A) < Timestamp(B)**.

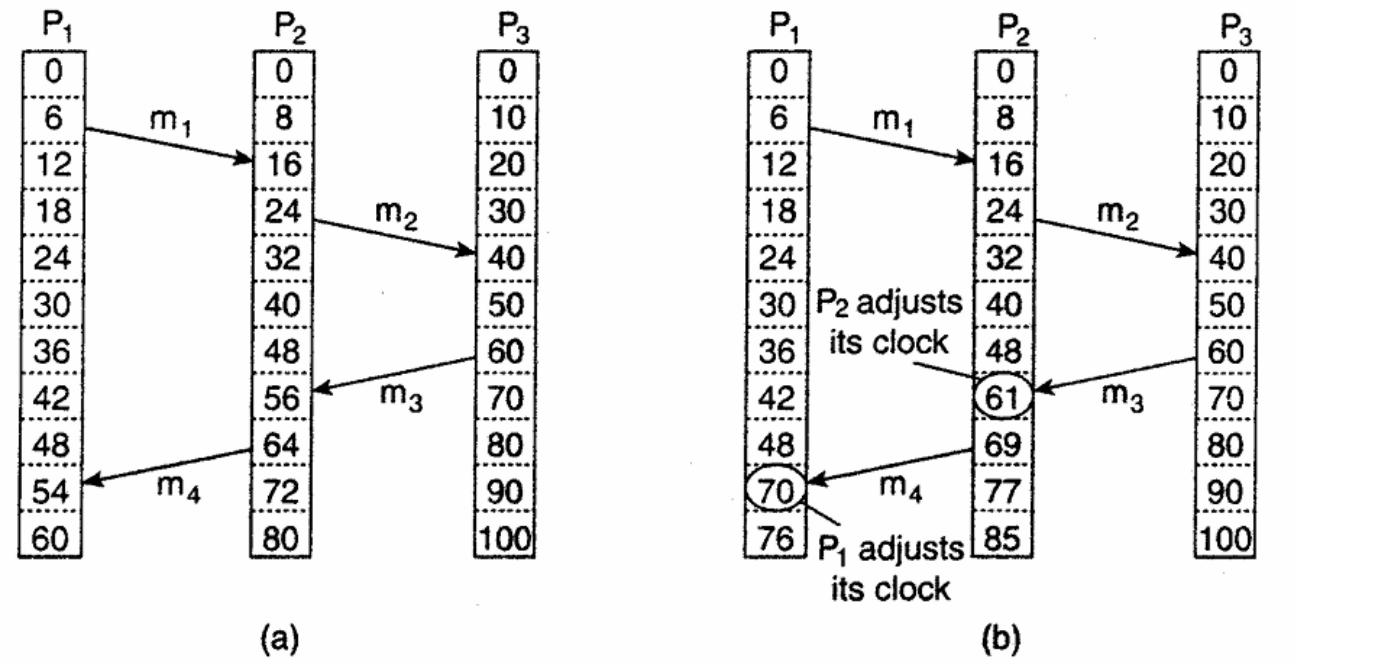
**Example:**

Consider three processes: P1, P2, and P3, each with its own logical clock running independently.

Consider message $m_3$ :

- P3 sends $m_3$ at timestamp 60.
- When P2 is about to receive $m_3$, its local clock shows 56.

Logically, a message **cannot be received at a time earlier than it was sent**.
Receiving at 56 while it was sent at 60 would violate causal ordering.




**Clock Adjustment (Receive Rule)**

So when P2 receives $m_3$, it updates its clock using:

$$\max(\text{local clock, received timestamp}) + 1$$
$$\max(56,60) + 1 = 61$$

Thus, P2 updates its clock to 61.

Now the receive event occurs after the send event, which is correct.

Similarly, for message $m_4$, if P1's local clock is 54 and it receives $m_4$ with timestamp 69, it updates its clock as:

$$\max(54,69) + 1 = 70$$

Thus, P1 adjusts its clock to 70, maintaining proper causal ordering.

**Concurrent Events in Lamport Timestamp**

Two events are concurrent if:

- Neither event happened-before the other.
- There is no message communication between them.
- Their timestamps do not imply causal relation.

**Example:**
If P1 performs an event at time 5, and P2 performs an event at time 4, and there is no message between them, then these two events are concurrent.

**11.** What are physical clocks? Explain any one physical clock synchronization algorithm.

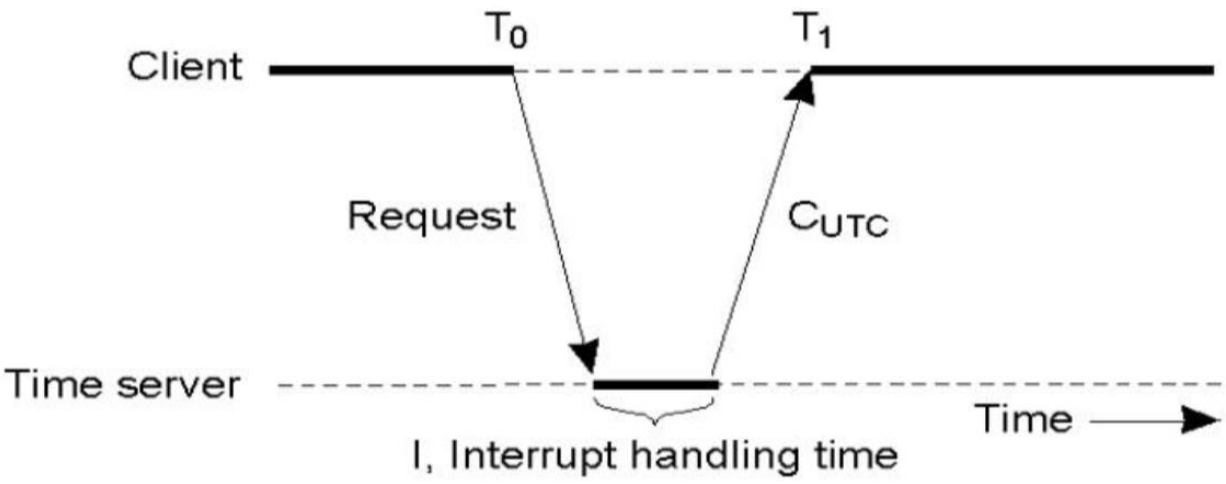
**Physical Clocks**

- Physical clocks measure real (wall-clock) time using the system's hardware clock.
- In distributed systems, different machines may run at slightly different times due to clock drift.
- So, clocks need to be synchronized regularly to keep time consistent across all nodes.
- To solve clock drift, distributed systems use synchronization algorithms such as:
  - **Cristian's Algorithm**
  - **Berkeley Algorithm**

**Cristian's Algorithm**

- A client synchronizes its clock by sending a request to a time server.
- The server is assumed to maintain a correct and accurate time.
- The client then adjusts its clock using the server's time and the estimated network delay to get a closer value.

**Working of Cristian's Algorithm with an example:**




- The client sends a time request to the server at time $T_0$.
- The server receives the request, reads its current time, and sends it back to the client (including a small processing delay).
- The client receives the response at time $T_1$.
- The client calculates the round-trip time:

$$RTT = T_1 - T_0$$

- **Example:**
  - Assume $T_0$ = 10 ms and $T_1$ = 70 ms
  - RTT = 70 − 10 = 60 ms
  - Estimated one-way delay = 60 / 2 = 30 ms

- The client updates its clock as:

$$\text{New Time} = \text{Server Time} + \frac{RTT}{2}$$

- If server time = 50 ms, then:

$$\text{New Time} = 50 + 30 = 80 \text{ ms}$$

- Thus, the client sets its clock to 80 ms, ensuring it is synchronized with the server.

12. What is distributed mutual exclusion? Explain how Suzuki–Kasami's broadcast algorithm achieves distributed mutual exclusion.

**Distributed Mutual Exclusion**

- Distributed Mutual Exclusion ensures that only one process at a time can enter the critical section in a distributed system.
- It is required because processes run on different machines without shared memory, so coordination must be done using messages.

**Suzuki–Kasami Broadcast Algorithm**

- It is a token-based algorithm where a process can enter the critical section only if it holds a unique token.
- Processes request access by broadcasting messages to all other processes.

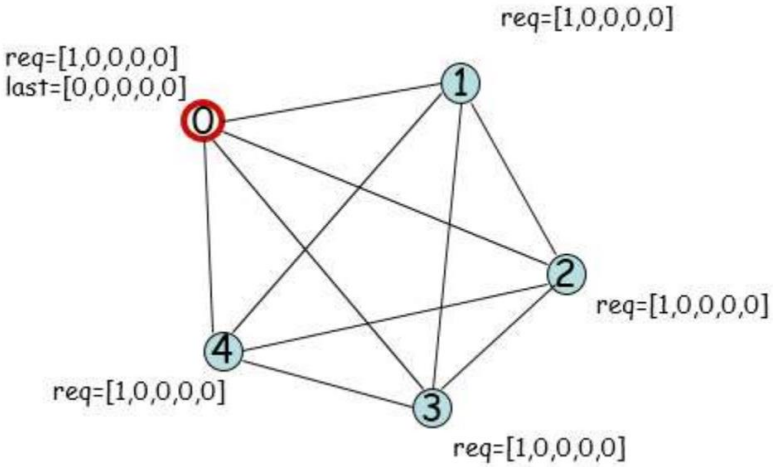
**Data Structures Used**

- **RN[ ] (Request Number)**: Stores latest request number of each process
  - Example: req = [1,1,1,0,0] → P0, P1, P2 have requested
- **Token contains:**
  - **LN[ ] (Last Served)**: Last request served for each process
    Example: last = [1,0,0,0,0]
  - **Queue (Q)**: Order of processes waiting for token.
    Example: Q = (1,2)

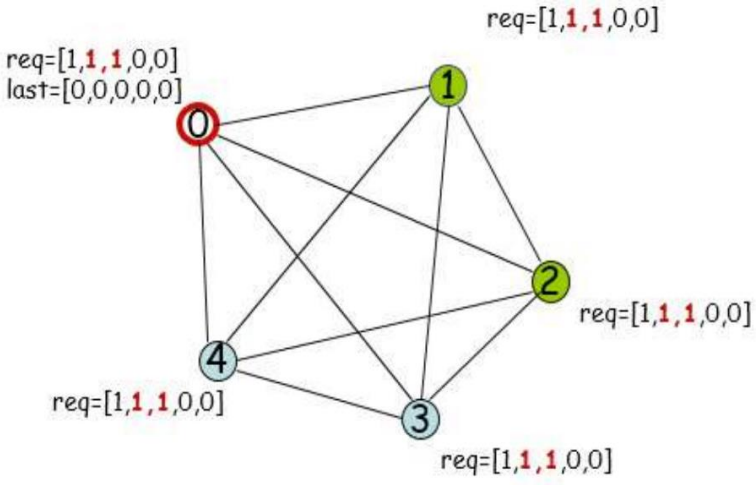
**Working of Suzuki–Kasami Algorithm with an example:**

**Step 1 - Initial State**

- Process P0 holds the token and enters critical section.
- RN and LN arrays are initialized.




Step 1




Step 2

**Step 2 - Request Broadcast**

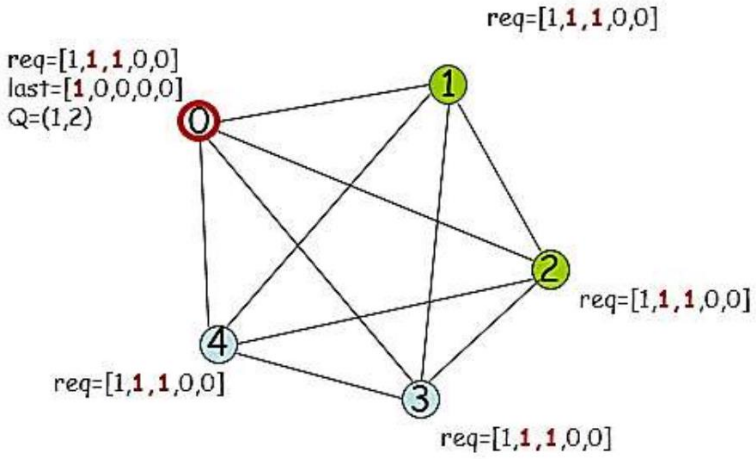
- Processes P1 and P2 request to enter CS.
- They increment RN[i] and broadcast request messages.
- All processes update their RN[] arrays.

**Step 3 - Queue Formation**

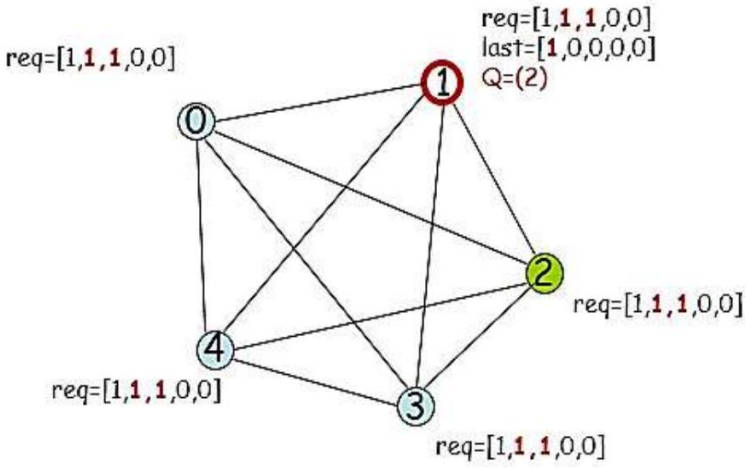
- Token holder (P0) finishes execution and checks requests.
- It applies:

$$RN[i] = LN[i] + 1$$

- Processes satisfying condition are added to queue Q = (1,2).




Step 3




Step 4

**Step 4 - Token Transfer to P1**

- P0 sends token to P1 (first in queue).
- P1 receives token and enters critical section.

After execution, the process updates LN[i] = RN[i] and passes the token to the next process in queue (2), continuing the same procedure.

**Conclusion**

- Suzuki–Kasami achieves mutual exclusion using a single token that controls access to the critical section.
- Requests are broadcast so all nodes know who is waiting.
- The token is passed in FIFO order using a queue, ensuring fairness and no starvation.
- Since only the token holder can enter the critical section, mutual exclusion is strictly maintained.

**13.** Explain Ricart–Agrawala's algorithm and how it optimizes message overhead in achieving mutual exclusion.

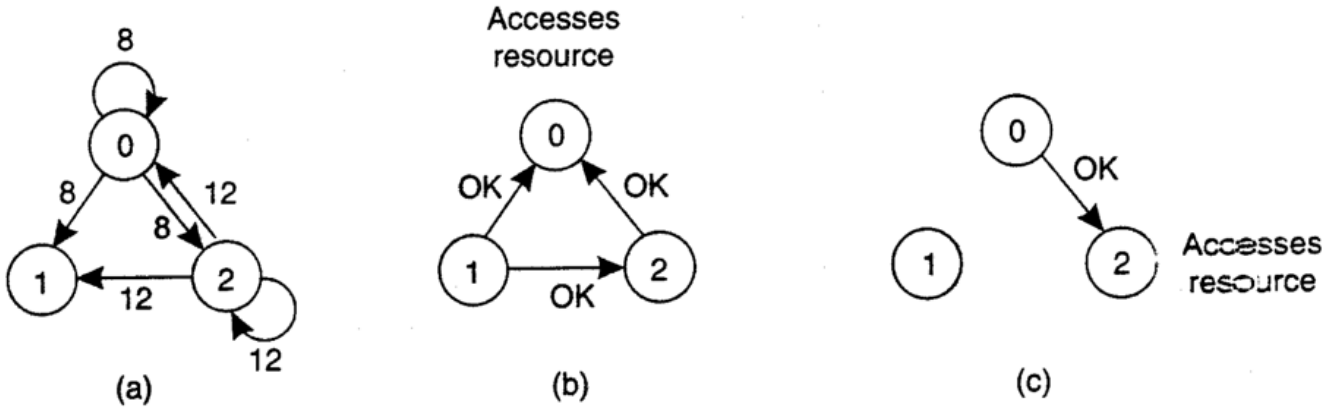
**Ricart–Agrawala Algorithm**

- It is a distributed mutual exclusion algorithm based on Lamport timestamps.
- It ensures that only one process can enter the critical section at a time.
- It improves Lamport's mutual exclusion algorithm by reducing message overhead.

**Working of Ricart–Agrawala Algorithm:**

- When a process wants to enter the critical section, it sends a REQUEST (timestamp, process ID) to all other processes.
- When a process receives a request:
  - **If it is not interested**, it replies immediately.
  - **If it is already in the critical section**, it delays the reply and queues the request.
  - **If it also wants the critical section**, it compares timestamps:
    - The smaller timestamp gets priority.
    - The loser waits and queues the request.
- A process enters the critical section only after receiving replies from all other processes.
- After finishing, it sends replies to all queued requests, allowing others to proceed.

**Example:**




- Process P0 sends a request with timestamp 8, while P2 sends with timestamp 12.
- Process P1 is not interested, so it sends OK to both P0 and P2.
- Process P2 compares timestamps, sees that P0 has a lower timestamp (higher priority), and sends OK to P0.
- Process P0 receives all replies and enters the critical section (CS).
- Meanwhile, P0 queues P2's request for later.
- After completing execution, P0 sends OK to P2.
- Process P2 then receives permission and enters the critical section.

**How It Optimizes Message Overhead**

In **Lamport's algorithm**:

- 3 types of messages are used: REQUEST, REPLY, RELEASE
- Total messages required = $3(N - 1)$

In **Ricart–Agrawala algorithm:**

- Only 2 types of messages: REQUEST and REPLY
- Total messages required = $2(N - 1)$

The RELEASE message is eliminated because a process simply delays its REPLY to other processes until it exits the Critical Section, thereby reducing message overhead.

## 14. Explain Chandy-Misra-Haas Algorithm for distributed deadlock detection.

The Chandy–Misra–Haas algorithm is a distributed deadlock detection algorithm. It uses an edge-chasing (probe-based) method to detect deadlocks without building a global Wait-For Graph.

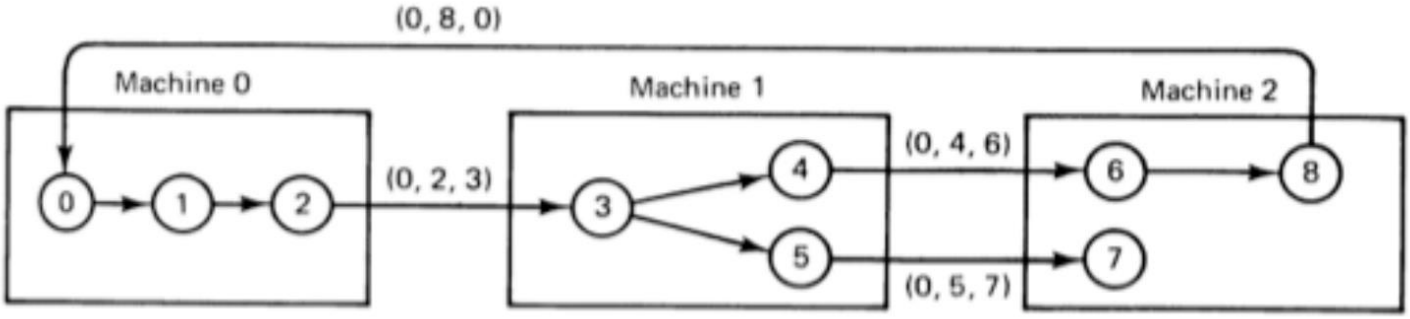
**Working of the Algorithm:**

Instead of sending large graphs over the network, the algorithm sends a small message called a probe along the waiting chain between processes.

The probe format is Probe(i, j, k):

- i: The process ID that initiated the deadlock detection.
- j: The process ID currently sending this probe.
- k: The process ID receiving this probe.

**Example**




Assume a distributed system with 3 machines (Machine 0, Machine 1, Machine 2) and 9 processes (0 through 8).

**Step 1:** Process 0 is blocked and suspects a deadlock. It initiates a probe. Since it is waiting on Process 1, it sends **Probe(0, 0, 1)**.

**Step 2:** Process 1 is blocked waiting on Process 2. It forwards the probe as **Probe(0, 1, 2)**.

**Step 3:** Process 2 forwards the probe to Machine 1 as **Probe(0, 2, 3)**.

**Step 4:** Process 3 is waiting on two processes (4 and 5). It duplicates the probe and sends:

- Path A: **Probe(0, 3, 4)**
- Path B: **Probe(0, 3, 5)**

**Step 5 (Path B ends):** Process 5 forwards the probe to Process 7 as **Probe(0, 5, 7)**.
Since Process 7 is not waiting on any process, it discards the probe. No deadlock on this path.

**Step 6 (Path A continues):** Process 4 forwards the probe to Machine 2 as **Probe(0, 4, 6)**.
Process 6 forwards it to Process 8 as **Probe(0, 6, 8)**.

**Step 7 (Deadlock Detected):** Process 8 is waiting on Process 0. It sends **Probe(0, 8, 0)** back to Machine 0.

**Conclusion:** Process 0 receives **Probe(0, 8, 0)**. Since the receiver ID (0) matches the initiator ID (0), a cycle exists. Therefore, Process 0 declares a deadlock.

# 4. Resource and Process Management

**15.** Explain the key features of a Global Scheduling algorithm. #

Global Scheduling Algorithm is a scheduling method in distributed systems that makes decisions by considering all nodes in the system, assigning processes to different processors to balance load and improve overall performance.

**Key Features of Global Scheduling algorithms:**

**1. Efficiency**

- Ensures that system resources like CPU and memory are used properly without waste.
- Improves overall system performance and reduces idle time.

**2. Fairness**

- Makes sure tasks are shared equally among all nodes.
- Prevents overloading of some nodes while others remain idle.

**3. Fault Tolerance**

- Allows the system to keep working even if some nodes fail.
- Tasks are shifted to other nodes to maintain execution.

**4. Scalability**

- System should work efficiently even when more nodes or processes are added.
- It should be able to handle growth without affecting performance.

**5. Stability**

- Maintains consistent performance even when system load changes.
- Avoids frequent changes in scheduling decisions.

**6. No Prior Knowledge**

- Does not require any advance information about processes.
- Works well in dynamic environments where data is unknown.

**7. Low Overhead**

- Requires less communication between nodes while making decisions.
- Reduces computation cost of scheduling decisions.

16. Explain load estimation policies, process transfer policies, and location policies used in load balancing approaches in distributed systems.

**1. Load Estimation Policy**

- It defines how the load of a node is calculated in the system.
- It is a key component of load balancing, as all decisions depend on accurate load estimation.
- Load can be measured using number of processes, CPU utilization, or resource usage.
- It helps identify whether a node is overloaded or underloaded.

**Types of Estimation:**

- **Static:** Load is fixed and does not change during execution.
- **Dynamic:** Load changes over time based on system state.

**Methods of Estimation:**

- **Process-based method:** Counts number of active processes on a node.
- **CPU-based method:** Uses CPU busy time or utilization.

**Advanced Estimation:**

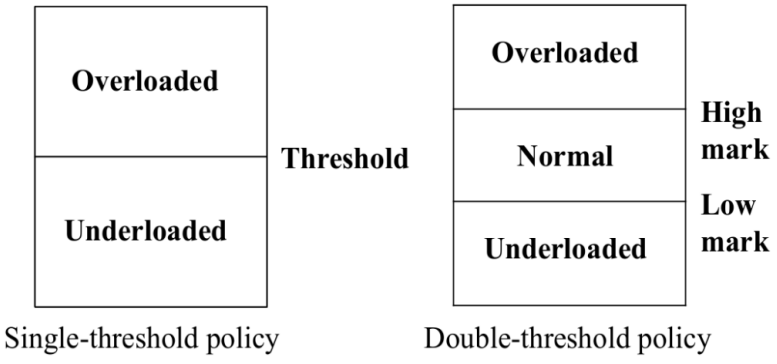
- **Memoryless method:** Assumes all processes have equal remaining time.
- **Past repeats method:** Assumes future load is similar to past behavior.
- **Distribution method:** Uses statistical distribution of service time.

**2. Process Transfer Policy**

- It decides when a process should be transferred from one node to another.
- It works based on threshold values to detect overload or underload.
- If a node becomes overloaded, it transfers processes to other nodes.
- If a node is underloaded, it accepts processes from others.

**Types of threshold policies:**

- **Single threshold policy:**
  - One limit is used to decide overload/underload.
  - May cause instability after transfer.
- **Double threshold policy (High-Low):**
  - Uses two limits (high and low).
  - More stable and avoids frequent transfers.



**3. Location Policy** *(asked once)*

- It decides where to transfer the process (which node to select).
- It selects a suitable node based on load, distance, and communication cost.
- It ensures that transfer improves performance and does not increase overhead.

**Methods of location policy:**

- **Threshold method:** Randomly selects nodes and checks if they can accept load.
- **Shortest method:** Chooses node with minimum load among selected nodes.
- **Bidding method:**
    - Nodes bid based on their capacity.
    - Best node is selected to execute the process.
- **Pairing method:** Fixed sender-receiver pairs for process transfer.

17. Explain code migration and its need in distributed systems. Explain the role of process-to-resource binding and resource-to-machine binding in code migration. %

**Code Migration**

- Code migration is the process of moving code (or a process) from one machine to another in a distributed system for execution.
- Code migration improves performance and flexibility, but its feasibility depends on binding between processes and resources.

**Need for Code Migration:**

- **Improves performance:** Processes can run closer to resources, reducing execution time.
- **Load balancing:** Workload is shared across nodes to avoid overload.
- **Better resource utilization:** Idle resources can be used effectively.
- **Flexibility:** Processes can run on different machines when needed.
- **Supports mobile computing:** Allows execution across different systems and devices.
- **Fault tolerance:** Processes can be moved away from failed or slow nodes.

**Process-to-Resource Binding**

- It defines the relationship between a process and the resources it needs during execution.
- It affects whether a process can be migrated or not.

**Types:**

- **Strong Binding:**
  - The process is tightly bound to a specific resource and cannot be easily moved.
  - Example: Process using a local hardware device.
- **Weak Binding:**
  - The process is loosely bound and can access resources from different locations.
  - Weak binding supports migration, while strong binding limits migration.
  - Example: Process using remote or shared resources.

**Resource-to-Machine Binding**

- It defines how resources are attached to specific machines in the system.
- It determines whether resources can move along with the process.

**Types:**

- **Unattached Resources:**
  - Resources are not tied to any specific machine and can be accessed from anywhere.
  - Example: Web services.

- **Fastened Resources:**
  - Resources are tied to a machine but can be moved if required.
  - Example: Files that can be transferred.
- **Fixed Resources:**
  - Resources are permanently tied to a specific machine and cannot be moved.
  - Example: Hardware devices like printers.

Migration is easier with unattached/fastened resources but difficult with fixed resources.

**18.** Explain how load balancing benefits a distributed system.

**Load Balancing**

- Load balancing is the process of distributing workload evenly across multiple nodes in a distributed system.
- It ensures that no single node is overloaded while others remain idle, improving overall system performance.

**Benefits of Load Balancing**

1. **Improves resource utilization**
   Ensures all nodes are properly used instead of some being overloaded and others idle.
2. **Reduces response time**
   Tasks are executed faster as workload is evenly distributed.
3. **Increases throughput**
   More tasks are completed in less time due to parallel execution.
4. **Avoids overload and failure**
   Prevents any single node from becoming a bottleneck or crashing.
5. **Provides scalability**
   System can handle more nodes and increased workload efficiently.
6. **Improves reliability and availability**
   If one node fails, other nodes can continue processing tasks.
7. **Efficient handling of peak loads**
   System can manage sudden increases in workload without performance degradation.
8. **Better system stability**
   Balanced workload reduces chances of system slowdown and ensures smooth operation.

# 5. Replication, Consistency and Fault Tolerance

**19.** What is fault tolerance? Describe different types of failure models.

Fault tolerance is the ability of a distributed system to continue operating correctly even when some of its components fail. In such systems, failures are common due to network issues, hardware faults, or software errors. Fault tolerance ensures that the system remains reliable and available, and that failures do not affect overall performance significantly.

**Types of Failure Models:**

**1. Crash Failure**

- A process stops working completely and does not respond to any requests.
- It remains inactive until restarted and is easier to detect.

  Example: Server crash due to power failure.

**2. Omission Failure**

- A process fails to send or receive messages due to network or internal issues.
- This leads to incomplete communication between nodes.

**Types:**

- **Send omission:** Message not sent.
- **Receive omission:** Message not received.

  Example: Packet loss in network.

**3. Timing Failure**

- A process responds too early or too late, violating time constraints.
- Can affect system performance, especially in real-time systems.

  Example: Delayed response in real-time system.

**4. Response Failure**

- A process gives incorrect output even though it is running.
- May return wrong values or perform incorrect operations.

**Types:**

- **Value failure:** Wrong output.
- **State transition failure:** Incorrect state change.

  Example: Incorrect calculation result.

**5. Arbitrary failures (Byzantine Failure)**

- A process behaves unpredictably and sends incorrect or conflicting data.
- Difficult to detect as behaviour may appear normal.

  Example: Hacked node sending false data.

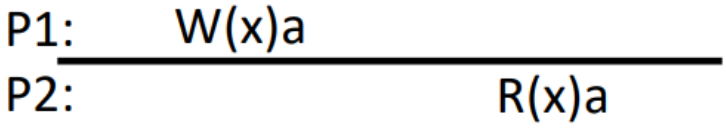
**20.** Explain any five data-centric consistency models with example data stores.

**1. Strict Consistency**

- A read always returns the latest written value, so all processes see the same data at the same time.
- It follows real-time ordering of operations.
- Very difficult to implement in practice due to network delays and lack of a global clock.

**Example Data Store:**

- If P1 does W(x)=a, then any process (like P2) doing R(x) will always read a immediately.
- This shows that all processes always see the most up-to-date value instantly.

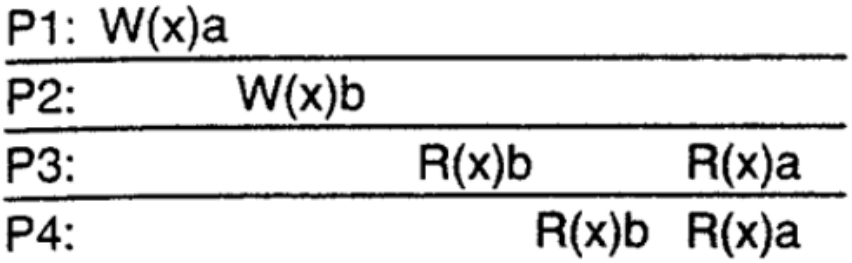



**2. Sequential Consistency**

- All processes see operations in the same order, though not necessarily in real-time.
- The execution appears as if all operations happened one after another in a sequence.
- Easier to implement than strict consistency and commonly used.

**Example Data Store:**

- If P1 does W(x)=a and P2 does W(x)=b, all processes see either:
  W(x)=a → W(x)=b OR W(x)=b → W(x)=a (same order for all).
- Even if timing differs, the overall sequence remains consistent.

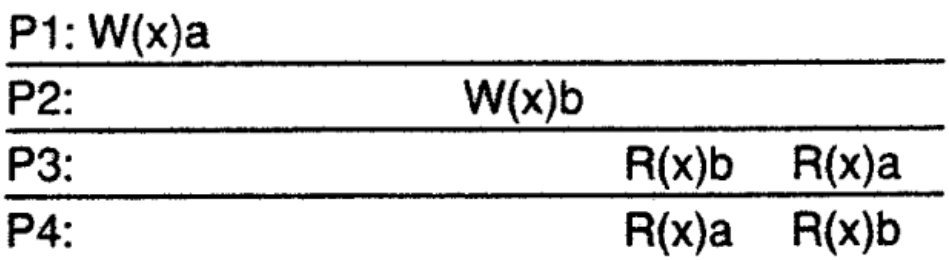



**3. Causal Consistency**

- Ensures that causally related operations are seen in the same order by all processes.
- Independent operations may be seen in different orders across processes.
- Maintains the cause-and-effect relationship between events, making it more flexible.

**Example Data Store:**

- If P1: W(x)=a and then P2 (after seeing a) does W(x)=b, all must see:
  W(x)=a → W(x)=b
- Unrelated operations may still appear in different orders.

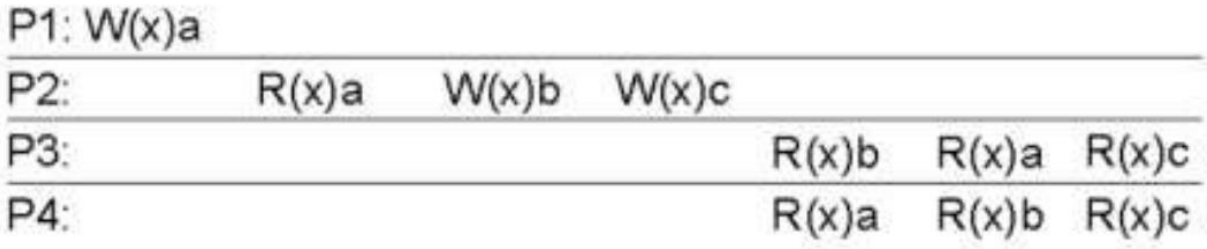



**4. FIFO Consistency (PRAM)**

- Writes from a single process are seen by all in the same order they were sent.
- Writes from different processes may be seen in different orders.
- Simpler and more efficient since it does not enforce global ordering.

**Example Data Store:**

- If P1 does W(x)=a → W(x)=b → W(x)=c, all processes see them in that same order.

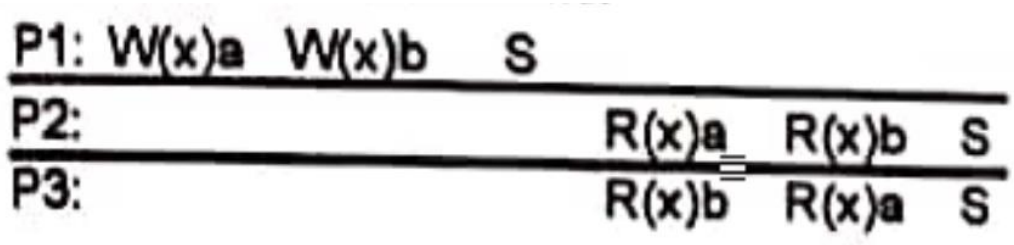



**5. Weak Consistency**

- Updates are not immediately visible to all processes, so different nodes may see different values temporarily.
- Data becomes consistent only after a synchronization operation.
- Reduces communication overhead and improves performance, making it suitable where strict consistency is not required.

**Example Data Store:**

- If P1 does W(x)=a → W(x)=b, other processes may still read old values like R(x)=a or earlier.
- Only after a synchronization (S) will all processes read the latest value



# 6. Distributed File Systems

**21.** What are the desirable features of a good Distributed File System (DFS)?

A Distributed File System is a file system that allows multiple users across multiple machines to access, share, and store files over a network as if they were stored locally.
It should provide fast, reliable, and secure file access while hiding the complexity of distribution.

**Desirable features of DFS**

**1. Transparency**

- Files should appear as if stored locally, hiding their actual location.
- Includes location and access transparency for ease of use.

**2. Scalability**

- The system should handle increase in users, data, and nodes efficiently.
- Performance should remain stable even as the system expands.

**3. Reliability**

- The system should continue to function even if some nodes fail.
- Data should not be lost due to node or network failures.

**4. Availability**

- Files should be accessible at all times with minimal downtime.
- Even during failures, users should still be able to access data.

**5. Security**

- Files should be protected from unauthorized access.
- Includes authentication, access control, and data protection.

**6. Performance**

- File access should be fast and efficient.
- Techniques like caching and replication help improve speed.

**7. Consistency**

- All users should see the correct and updated version of files.
- The system should properly handle updates across nodes.

**8. Fault Tolerance**

- The system should recover from failures automatically without affecting users.
- Backup and recovery mechanisms should ensure smooth operation.

**22.** Explain synchronization in Distributed File Systems and its challenges. #

Synchronization in DFS ensures that shared files remain consistent when multiple users access and modify them. It helps maintain correct and updated data across all nodes.

**Synchronization in DFS:**

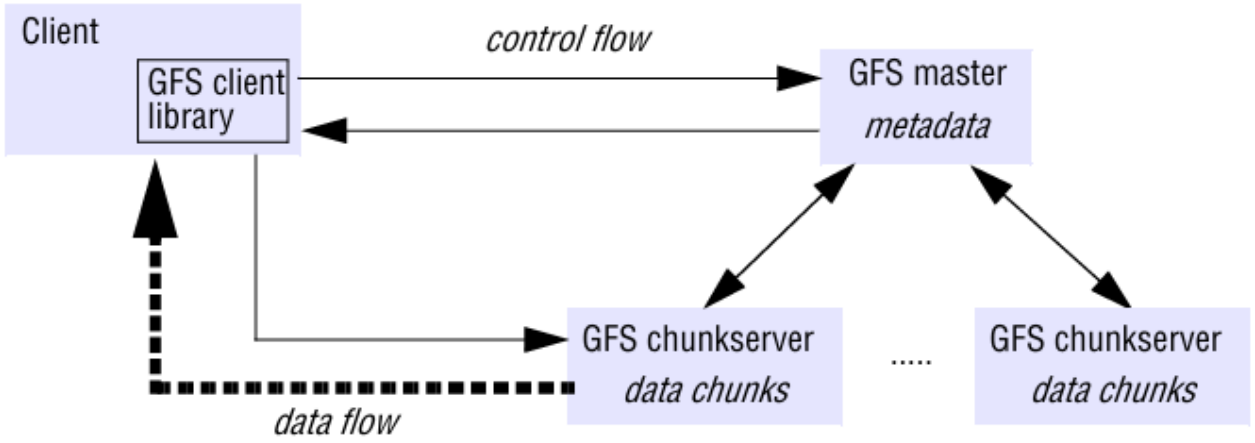
- **Maintains consistency:** Ensures all users see the latest and correct file data across the system.
- **Handles multiple users:** Allows many users to access the same file safely without causing conflicts.
- **Uses file locking:** Controls access using read and write locks to avoid simultaneous updates.
- **Updates shared data:** Changes made by one user are reflected to other users in the system.

**Challenges:**

- **Simultaneous access:** Multiple users editing the same file can cause conflicts or data loss.
- **Network delay:** Updates may take time to reach all nodes, causing temporary inconsistency.
- **Old data in cache:** Some nodes may use outdated copies of files.
- **Scalability issues:** Maintaining synchronization becomes difficult as number of users increases.

**23.** Discuss the Google File System (GFS) as a scalable distributed file system.

The Google File System (GFS) is a distributed file system developed by Google to store and manage very large amounts of data across multiple machines. It is designed to be scalable, reliable, and efficient for large-scale data processing.




**Architecture of GFS**

- **Master:**
  Manages file information like file names and chunk locations.
  In the diagram, it controls the system and guides the client to the correct chunk servers.
- **Chunk Servers:**
  Store the actual data in the form of chunks.
  As shown in the diagram, they directly send data to the client.
- **Client:**
  First contacts the master to find where data is stored.
  Then directly communicates with chunk servers to read/write data.

**Features supporting scalability:**

- **Large file support:** Files are divided into large fixed-size chunks, making it easier to handle very large data efficiently.
- **Distributed storage:** Data is stored across many machines, allowing the system to scale as more storage is needed.
- **Replication:** Each chunk is stored on multiple chunk servers, ensuring data safety even if one server fails.
- **High throughput:** GFS is optimized for handling large data operations rather than small fast requests.
- **Fault tolerance:** The system automatically detects failures and recreates lost data using replicas.
- **Automatic management:** The master continuously monitors chunk servers and manages data distribution.

**24.** Explain different file caching schemes used in Distributed File Systems.

**File Caching**

- File caching means storing frequently used files temporarily (at client or server) so they don't have to be fetched again and again from the network.
- This helps in faster access and better performance, but the system must ensure that cached data stays consistent.

**Types of File Caching Schemes**

**1. Based on Location of Cache**

**Client-side Caching**

- Files are stored in the client's local memory or disk after first access, so future requests can be served locally.
- This reduces network traffic and server load, but needs proper consistency handling if the file is updated somewhere else.

**Server-side Caching**

- Frequently accessed files are kept in the server's cache (main memory) to avoid repeated disk access.
- This improves response time for multiple clients, but does not reduce communication between client and server.

**2. Based on Update Strategy**

**Write-through Cache**

- Every time a file is modified, the change is immediately sent to the server along with updating the cache.
- This keeps data consistent and reliable, but increases network traffic and slows down write operations.

**Write-back (Delayed Write) Cache**

- Changes are first made in the local cache and sent to the server later (on file close or after some time).
- This improves performance and reduces network usage, but may cause temporary inconsistency or data loss if a failure occurs.

# Asked once:

## 2. Communication

**1. Differentiate between RMI and RPC. #**

| Parameter | RPC (Remote Procedure Call) | RMI (Remote Method Invocation) |
|---|---|---|
| **Definition** | RPC allows a program to call a remote procedure as if it were local. | RMI allows a program to call a method of a remote object. |
| **Programming Style** | RPC is procedure-oriented. | RMI is object-oriented. |
| **Language Support** | RPC can be used with different programming languages. | RMI is mainly used in Java. |
| **Communication** | RPC calls remote functions. | RMI calls methods of remote objects. |
| **Data Handling** | RPC passes simple data types. | RMI can pass objects as parameters. |
| **Complexity** | RPC is simpler to implement. | RMI is more complex. |
| **Object Support** | RPC does not directly support objects. | RMI supports objects and remote objects. |
| **Platform** | RPC is more platform independent. | RMI is mainly used in Java-based systems. |

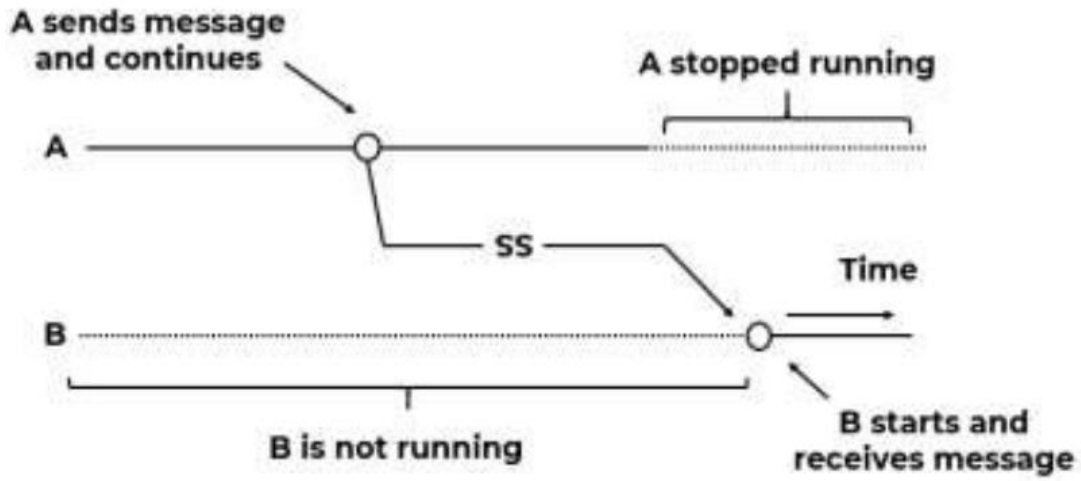
**2. Explain how transparency is achieved in RPC. #**

- RPC achieves transparency with the use of client-server stubs, marshalling/ unmarshalling, and hidden communication mechanisms.
- The client calls the client stub as if it were a local function, without knowing the procedure is remote.
- The client stub packs the parameters (marshalling) and sends the request over the network.
- The server stub receives and unpacks the data (unmarshalling), then calls the actual procedure.
- After execution, the result is sent back through the same process and returned to the client.
- All network communication and complexities are handled internally, so the user is unaware of remote execution.
- In this way, RPC makes a remote procedure call appear like a simple local function call, achieving transparency.

## 3. Explain various forms of message-oriented communication with suitable example.

### 1. Persistent Asynchronous Communication

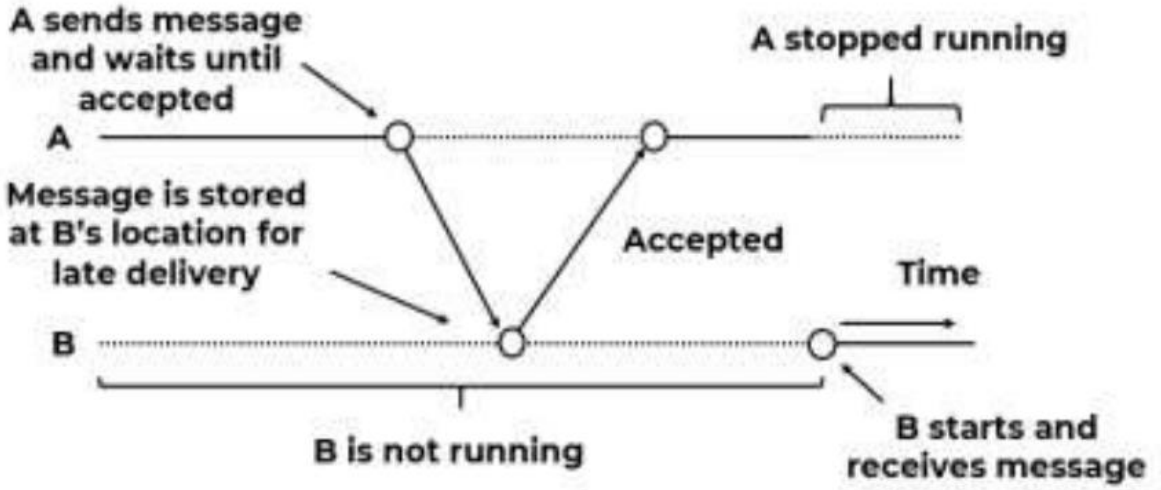
- Messages are stored in the system until the receiver becomes available, and the sender continues execution immediately.




- In the diagram, process A sends the message and continues execution without waiting.
- Process B is initially not running, so the message is stored (shown by the system/server line).
- When B starts later, it receives the message from storage.

### 2. Persistent Synchronous Communication

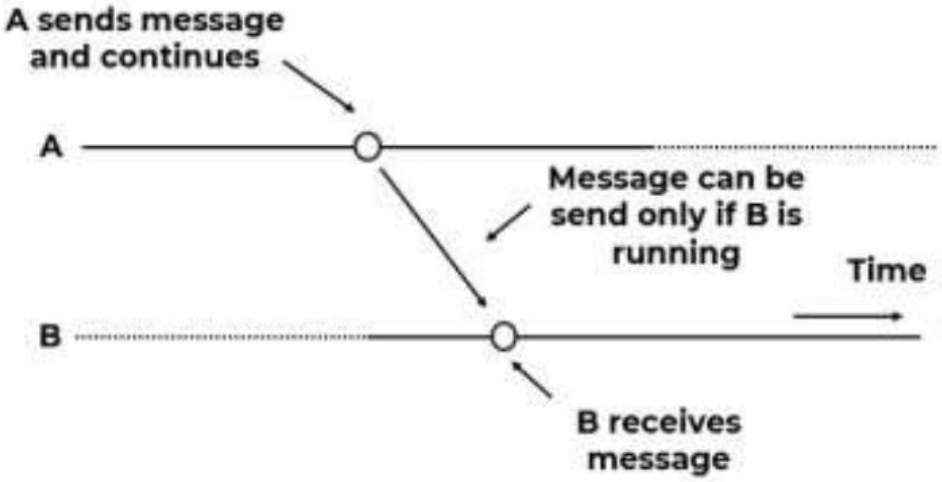
- Messages are stored by the system and the sender waits until the message is safely accepted (stored) before continuing.




- In the diagram, process A sends the message and waits until it is accepted.
- The message is stored at B's location (or communication server) even though B is not running.
- Once the message is stored, A continues execution, and later when B starts, it receives the message.

### 3. Transient Asynchronous Communication

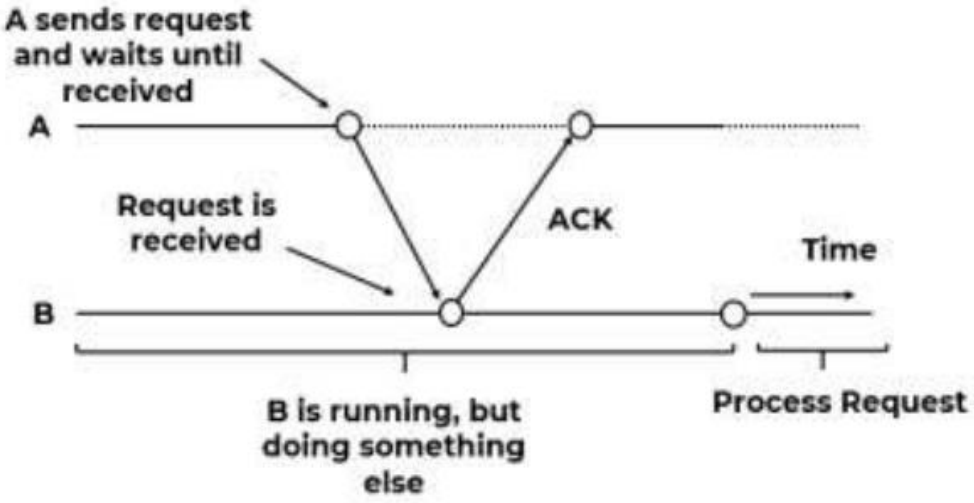
- Messages are delivered only if the receiver is active at that time, and the sender doesn't wait.




- In the diagram, process A sends the message and continues immediately.
- The message is delivered only if B is running at that moment.
- If B is not active, the message would be lost (no storage).

### 4. Receipt-based Transient Synchronous Communication

- The sender waits until the message is received (ACK), but not until it is processed.




- In the diagram, process A sends a request and waits.
- Process B receives the message and sends an acknowledgment (ACK).
- Once A gets the ACK, it continues execution, even though B may still be processing the request.

# 3. Synchronization

**4. Compare two election algorithms and recommend the most efficient one for large-scale distributed systems.**

| Parameter | Bully Algorithm | Ring Algorithm |
|---|---|---|
| **Basic Idea** | The process with the highest ID becomes coordinator. | Processes pass an election message in a ring to select leader. |
| **Structure** | Fully connected system | Logical ring structure |
| **Message Type** | Uses ELECTION, OK, COORDINATOR messages. | Uses ELECTION (with ID) and COORDINATOR messages. |
| **Message Flow** | Messages sent to all higher-ID processes. | Message moves step-by-step around the ring. |
| **Who Starts Election** | Any process detecting coordinator failure. | Same, any process detecting failure. |
| **Winner Selection** | Highest ID process that gets no reply wins. | Process whose own ID returns after full ring traversal wins. |
| **Knowledge Required** | Each process knows all other process IDs. | Each process knows only its next neighbour. |
| **Scalability** | Poor for large systems. | Better suited for larger systems. |
| **Speed** | Faster in small systems (parallel messaging). | Slower (message must complete full ring). |
| **Number of messages** | $O(n^2)$ in worst case. | O(n) per election round. |

**Ring Algorithm is preferred for large-scale distributed systems due to:**

- **Lower message complexity (O(n) vs $O(n^2)$)**: Uses fewer messages, so it doesn't overload the network as the system grows.
- **Minimal knowledge requirement**: Each process only needs to know its neighbour, making it easier to manage in large systems.
- **More stable elections**: Avoids repeated restarts, so the election process remains smooth and predictable.
- **Better scalability**: Works efficiently even when the number of processes becomes very large.

## 5. Explain the need of election algorithm. #

**Need of Election Algorithm**

- In distributed systems, a coordinator (leader) is needed to manage tasks like resource allocation, synchronization, and decision making.
- If the coordinator fails or crashes, the system must choose a new one to keep working smoothly.
- Election algorithms help in selecting a new leader among active processes in a reliable way.
- They ensure that only one coordinator is chosen, avoiding confusion or multiple leaders.
- This helps maintain system stability and continuous operation without manual effort.

# 4. Resource and Process Management

## 6. Explain code migration and its techniques.

**Code Migration**

- Code migration is the process of moving code (or a process) from one machine to another in a distributed system for execution.
- Code migration improves performance and flexibility, but its feasibility depends on binding between processes and resources.

**Techniques of Code Migration**

**1. Process Migration (Strong Mobility)**

- The entire process (code + execution state + data) is moved to another machine.
- Execution resumes from the same point, without restarting.

**Example:** A running job is shifted from an overloaded system to another machine and continues execution.

**2. Code Migration (Weak Mobility)**

- Only the code is transferred, and execution starts from the beginning on the new machine.
- No execution state is transferred.

**Forms:**

- **Code push** → Sender sends code.
- **Code fetch** → Receiver requests code.

**Example:** A browser downloading and executing a script.

**3. Migration based on Resource Binding**

- When code is migrated, it may depend on resources (files, devices, etc.), which affects migration.

**Process-to-Resource Binding**

- Defines how tightly a process is connected to its required resources.
- **Strong Binding:**
  - Process is tightly linked to a resource → Difficult to migrate
  - Example: Process using a local device
- **Weak Binding:**
  - Process can access resources from different locations → Easy to migrate
  - Example: Process using remote/shared resources

## 7. Describe code migration issues in detail.

**1. Heterogeneity**

- Different machines may have different hardware and operating systems, so code may not run the same everywhere.
- Without proper compatibility support, the migrated code may fail or behave incorrectly.

**2. Resource Binding**

- A process may depend on local resources like files, memory, or devices.
- If these are tightly bound, it becomes difficult or impossible to migrate the process.

**3. Communication Issues**

- After migration, the process may still need to communicate with other processes.
- Network delays or failures can affect coordination and performance.

**4. Security**

- Migrated code may be unsafe or malicious, creating security risks.
- Proper authentication and secure execution mechanisms are required.

**5. Fault Tolerance**

- Failures during migration can lead to loss of process or incomplete execution.
- The system must have recovery mechanisms to handle such failures.

**6. Synchronization**

- Keeping data consistent when multiple processes interact becomes more difficult after migration.
- It may lead to conflicts or inconsistent states.

**7. Naming and Location Issues**

- After migration, the system must track where the process has moved.
- This requires proper naming and directory services.

**8. Scalability**

- Managing migration across many nodes becomes more complex as the system grows.
- Performance may decrease if not handled efficiently.

# 5. Replication, Consistency and Fault Tolerance

## 8. Explain how replication helps in achieving fault tolerance. #

- Replication means maintaining multiple copies of data or services on different nodes in a distributed system.
- It helps achieve fault tolerance by ensuring that if one node fails, other replicas can continue to provide the service.

**How Replication Helps**

- **Redundancy:** Multiple copies ensure the system can continue even if one replica fails.
- **Failure Handling:** Requests are redirected to another replica when a node crashes.
- **Data Backup:** Copies of data are stored at different locations, preventing data loss.

**Example**

- In a distributed database, if one server fails, another replica provides the same data without affecting users.

## 9. Explain how Monotonic read consistency model is different than Read your Write consistency model. #

| Aspect | Monotonic Read Consistency | Read-Your-Write Consistency |
|---|---|---|
| **What it ensures** | Once a value is read, the client will not see an older value later. | After writing, the client will always see its own latest update. |
| **Relation of operations** | Between consecutive reads. | Between a write and the next read. |
| **User experience** | Data only moves forward, no going back to older versions. | User always sees their own changes immediately. |
| **Main purpose** | Avoids confusion due to inconsistent reads. | Ensures correctness of user updates. |
| **Example** | After reading version 5, next reads will be version ≥ 5. | After updating a profile, user sees updated data immediately. |

## 10. Discuss the technique to achieve Process resilience.

Process resilience is the ability of a distributed system to continue working even when failures occur. It ensures reliability and reduces loss of work during system crashes.

**Techniques to Achieve Process Resilience:**

**1. Redundancy (Replication)**

- Multiple copies of a process are kept on different nodes.
- If one process fails, another copy continues execution without interruption.
- This improves availability and fault tolerance.

**2. Checkpointing**

- The system saves the state of a process at regular intervals.
- In case of failure, the process restarts from the last checkpoint instead of the beginning.
- This helps save time and avoid repeating work.

**Types:**

- **Coordinated checkpointing:** All processes save their state together.
- **Uncoordinated checkpointing:** Each process saves its state independently.

**3. Message Logging**

- All messages exchanged between processes are recorded.
- This helps in recreating the process state during recovery.
- It works together with checkpointing for accurate recovery.

**4. Process Restart**

- When a process fails, it is restarted on the same or another node.
- It uses checkpoint and log data to resume execution.
- This ensures the system keeps running smoothly.

**5. Failure Detection**

- The system continuously monitors processes using techniques like heartbeat signals.
- Detects failures quickly and starts recovery actions.

**6. Recovery Mechanisms**

- Combines checkpointing and message logging to restore process state.
- This ensures minimal loss of work after failure.

## 11. Discuss design and implementation issues of distributed shared memory.

Designing a Distributed Shared Memory (DSM) system involves managing data sharing across multiple nodes while ensuring consistency and performance.

Various issues must be considered to achieve efficient communication, synchronization, and reliability.

**1. Granularity**

- Granularity refers to the size of data blocks shared across the system.
- Small blocks increase communication overhead; large blocks may transfer unnecessary data.

**2. Structure of Shared Memory Space**

- It defines how data is organized and accessed in the memory.
- The structure depends on the application requirements and type of data being used.

**3. Memory Coherence and Access Synchronization**

- Ensures that all nodes see consistent and updated data at all times.
- Uses synchronization methods like semaphores or locks to avoid data conflicts.

**4. Data Location and Access**

- System should allow users to easily find and access data across the network.
- Uses data location mechanisms to access data across the network.

**5. Replacement Strategy**

- It decides which data block should be replaced when memory is full.
- A good strategy helps in efficient memory usage and better system performance.

**6. Thrashing**

- Occurs when data is moved frequently between nodes, increasing overhead.
- It reduces system performance and should be avoided.

**7. Heterogeneity**

- Different nodes may have different hardware and operating systems.
- The system should handle these differences to ensure smooth operation.

## 12. Write a short note on Replication and the types of it.

**Replication:**

- Replication is the process of maintaining multiple copies of data across different nodes in a distributed system.
- It is used to improve fault tolerance, performance, and data availability.

**Benefits of Replication**

- **Reliability:** Data remains available even if a node crashes or becomes unreachable.
- **Performance:** Requests are served faster by accessing nearby or less loaded replicas.
- **Scalability:** System can handle more users by distributing workload across replicas.

**Types of Replication:**

**1. Eager (Synchronous) Replication**

- All replicas must update and confirm before the client gets a response, ensuring strong consistency.
- Involves high coordination overhead, so performance is slower but data is always consistent.

**Example:** Banking systems where all replicas must agree before completing a transaction.

**2. Lazy (Asynchronous) Replication**

- Updates are applied to the primary first and propagated to other replicas later in the background.
- Faster performance but may lead to temporary inconsistency between replicas.

**Example:** Social media posts appearing with slight delay across regions.

**3. Passive Replication (Primary–Backup)**

- One node acts as the primary handling all requests, while backups are updated accordingly.
- If the primary fails, a backup takes over, ensuring fault tolerance and continuity.

**Example:** Master database with replica servers.

**4. Active Replication**

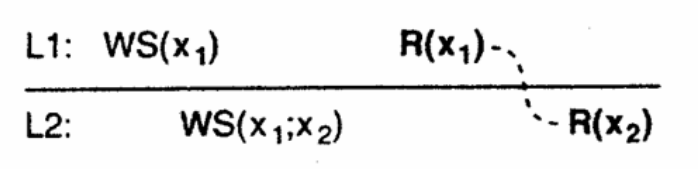
- All replicas process the same request simultaneously and maintain identical state.
- Requires strict ordering of operations to ensure consistent results across all replicas.

**Example:** Multiple servers executing the same transaction in parallel.

## 13. Discuss and differentiate various client consistency models.

### 1. Monotonic Read Consistency

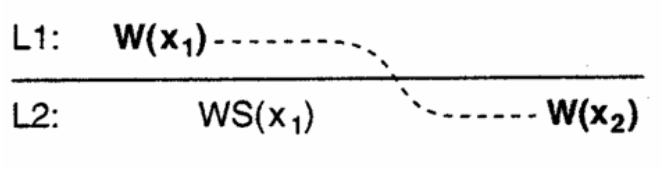
- Once a client reads a value, it will not see older values in future reads.
- A process reads data from one replica (L1) and later reads from another replica (L2).




- In the diagram, after reading $x_1$, the next read returns $x_2$ (same or newer value).
- This ensures the client never goes back to an older version even when switching replicas.

### 2. Monotonic Write Consistency

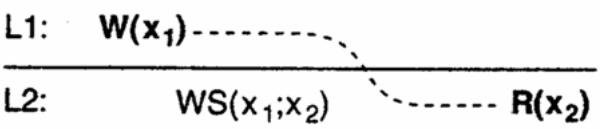
- Write operations by a client are executed in the same order they were issued.
- A process performs write $W(x_1)$ followed by $W(x_2)$.




- In the diagram, the system ensures $W(x_1)$ is applied before $W(x_2)$ across replicas.
- This maintains correct order and avoids older writes overriding newer ones.

### 3. Read-Your-Write Consistency

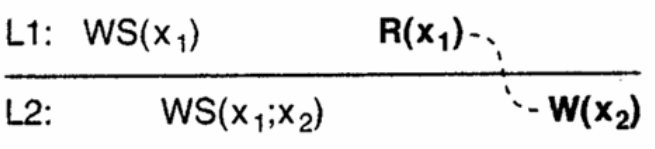
- After a client writes a value, it will always see its own updated value in future reads.
- A process writes $W(x_1)$ and later performs a read.




- In the diagram, the read returns $x_1$ or a newer value, not an older one.
- This ensures the client always sees its own latest update.

### 4. Write-Follows-Read Consistency

- A write operation is based on the latest value previously read by the client.
- A process first reads $R(x_1)$ and then performs a write $W(x_2)$.




- In the diagram, the write is applied on a replica that already has $x_1$.
- This ensures the write is based on the most recent version of data.

| Parameter | Monotonic Read | Monotonic Write | Read-Your-Write | Write-Follows-Read |
|---|---|---|---|---|
| **Operation Relation** | Read → Read | Write → Write | Write → Read | Read → Write |
| **Main Idea** | Future reads should not return older values | Writes must be applied in the same order. | Client must see its own latest update. | Writes must be based on latest read. |
| **Guarantee** | No backward reads. | No out-of-order writes. | Own updates are visible. | Writes use most recent data. |

# 6. Distributed File Systems

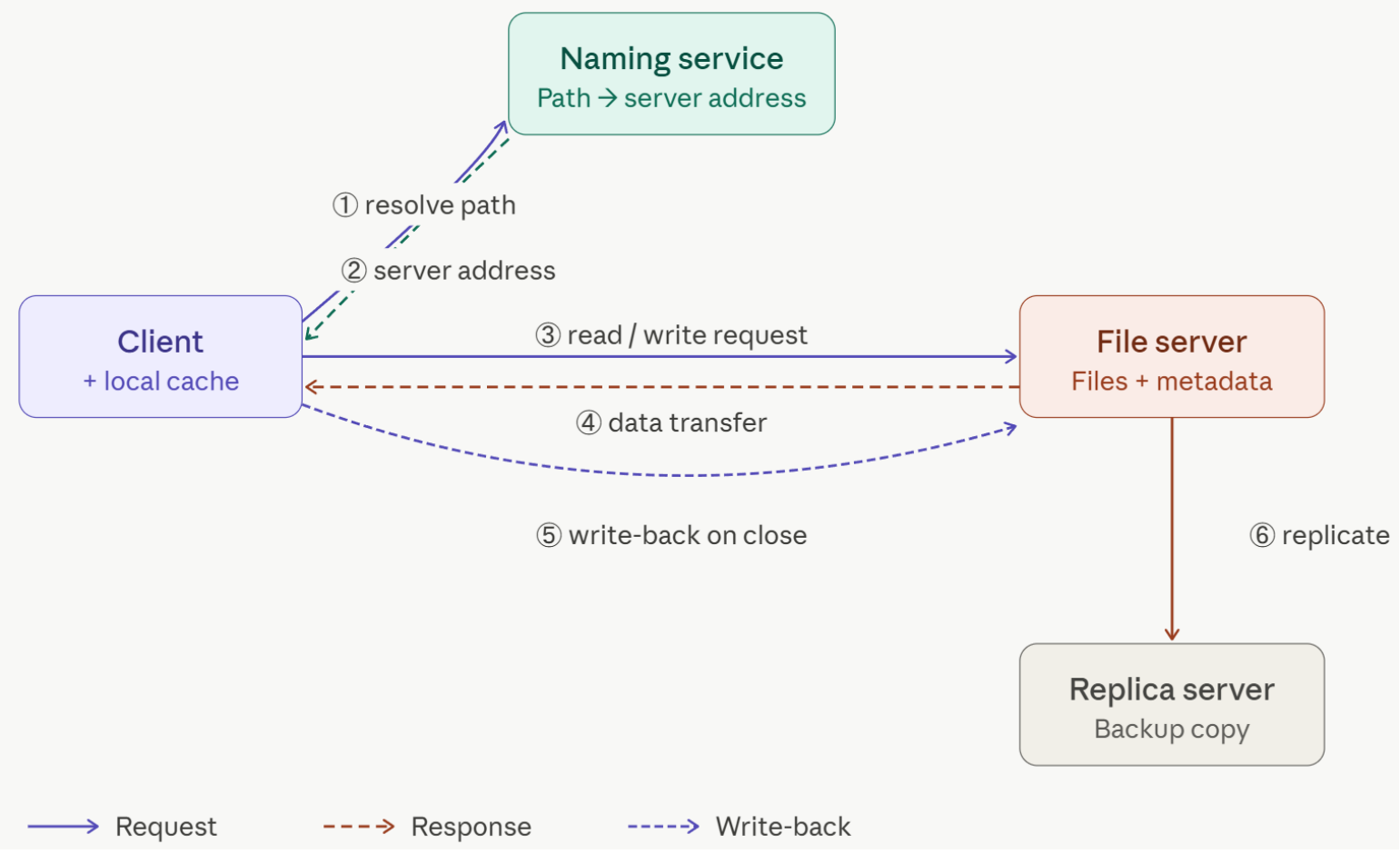
## 14. Explain the working of Distributed File System with its applications.

**Distributed File System (DFS)**

- A Distributed File System is a file system that allows multiple users across multiple machines to access, share, and store files over a network as if they were stored locally.
- The physical location of the file is transparent to the user; they interact with one unified file namespace regardless of where the data actually resides.

**Architecture of a DFS**

- **Client -** Requests file operations and sends them to the server, and may use local cache to improve access speed.
- **File Server -** Stores files and handles read, write, and access control operations.
- **Naming / Directory Service -** Maps file names to their actual storage location, hiding physical details from the user.




**Working of Distributed File System**

- **File Request** - Client performs a file operation (open/read/write) and first checks its local cache for the data.
- **Name Resolution** - If data is not in cache, client contacts the naming service which maps the file path to the server address.
- **Server Request** —-Client sends the read/write request directly to the file server using the obtained address.

- **Data Transfer** - File server processes the request and sends file data back to the client.
- **Write-back** - If the file is modified, changes are sent back to the server on close (write-back).
- **Replication** - The file server updates replica servers to maintain availability and fault tolerance.

**Applications of Distributed File System**

- **Cloud Storage Systems** - Used in platforms like Google Drive and Dropbox for storing and accessing files over the internet.
- **Big Data Processing** - Systems like Hadoop use DFS to store and process large datasets across multiple nodes.
- **Enterprise File Sharing** - Enables employees to access and share files within an organization from different locations.
- **Backup and Disaster Recovery** - Maintains copies of data across nodes to prevent data loss and ensure recovery.

## 15. How are modifications propagated in file caching schemes? #

1. **Write-through**
   - Whenever a file is modified, the update is immediately sent to the server.
   - Keeps data always consistent, but slows down performance due to frequent updates.
2. **Write-back (Delayed Write)**
   - Changes are first made in the local cache and sent to the server later.
   - Improves speed, but there can be temporary mismatch between cache and server data.
3. **Invalidate Approach**
   - When one copy is updated, other cached copies are marked invalid.
   - Next time they are used, fresh data is fetched from the server.
4. **Update Approach**
   - When a file is modified, the updated data is sent to all cached copies.
   - Keeps all copies consistent, but uses more network bandwidth.

~ *AJ*