Q. Explain Clock Synchronization

Clock synchronization is about making sure that the clocks on different computers show the same time.

1. **What is it?**
   Clock synchronization is the problem of ensuring that the internal clocks of multiple computers stay in sync with each other.
2. **Why do computers need clocks?**
   Every computer uses its internal clock to keep track of time, and this is important for things like scheduling tasks and tracking events.
3. **Clock Skew:**
   Sometimes, the clocks on different computers don't match exactly, which is called "clock skew."
4. **Why is this an issue?**
   In a distributed system (where many computers work together), it's important that their clocks stay synchronized, because things like data logging or coordination between computers could go wrong if their clocks are out of sync.
5. **Types of Synchronization:**
   There are two main ways to synchronize clocks:
   - **Centralized:** One computer is responsible for synchronizing the others.
   - **Distributed:** All computers work together to keep their clocks in sync.

Q. Explain physical clock synchronization algorithms.

Centralized Clock Synchronization Algorithms:

1. **What is it?**
   - In a centralized system, one computer (called the *time server*) has the correct time. All other computers (called *clients*) sync their clocks to match the time server's clock.
2. **Goal:**
   - Make sure that all the computers' clocks are the same as the time server.
3. **Drawbacks:**
   - **Single Point of Failure:** If the time server fails, everything stops working.
   - **Overload:** The time server has to handle all the work, which can be too much.

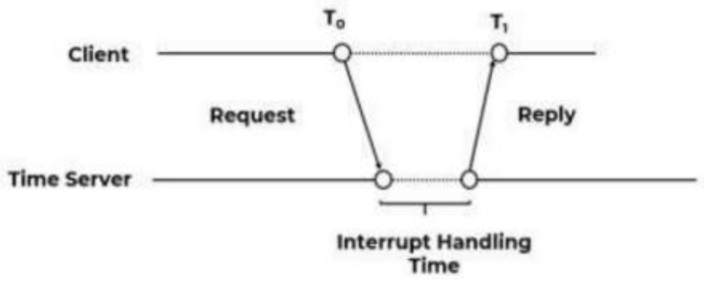
4. **Examples of Centralized Algorithms:**
    - **Cristian's Algorithm**
    - **Berkley Algorithm**

Distributed Clock Synchronization Algorithms:

1. **What is it?**
    - Unlike centralized methods, in a distributed system, all computers (nodes) work together to sync their clocks.
2. **Examples of Distributed Algorithms:**
    - **Network Time Protocol (NTP)**

Q. Cristian's Algorithm

Cristian's Algorithm is a method used to sync the clocks of computers (clients) with a time server. Here's how it works:



1. **How it works**:
    - A **client** (a computer) wants to know the current time, so it sends a **message** to the time server.
    - The **time server** responds with the current time (let's call this time **T**).
2. **Round Trip Time**:
    - **Round Trip Time (RTT)** is the time it takes for the message to go from the client to the server and back.
    - If the network is **fast** (low latency), this method works well.
3. **Steps in the Algorithm**:
    - The client sends a message at **T0** (its own time).
    - The server receives it and sends back the time, **T**.

   - The client receives the response at **T1**.
   - The time it took for the message to travel back and forth is **T1 - T0**.
4. **Adjusting the Client's Clock**:
   - The client estimates the network delay as **(T1 - T0) / 2**.
   - It then adjusts its clock by setting it to **T + (T1 - T0) / 2**.
5. **Final Result**:
   - The client's clock is now synchronized with the server's time, with an adjustment for the round-trip delay.

Advantages:

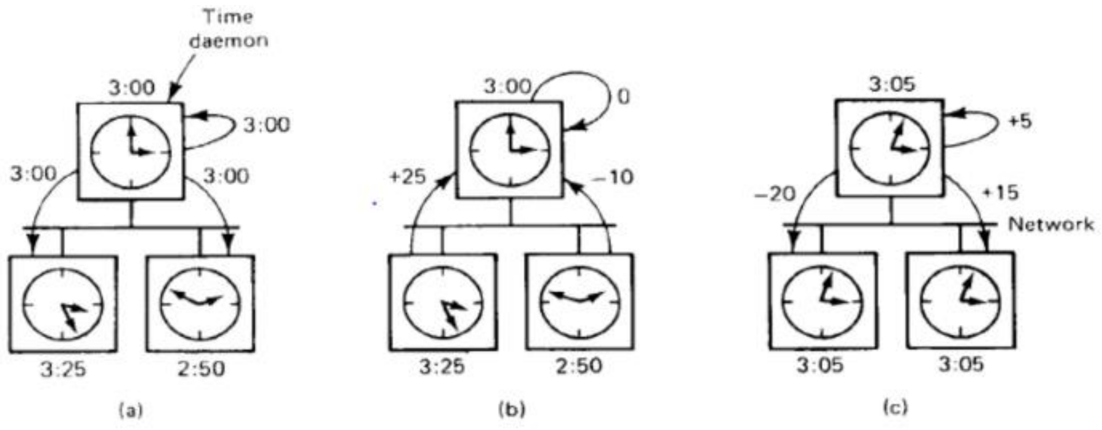
- It's simple and works well in networks with low delay.

Disadvantages:

- It's not very accurate when the round trip time is large, and it only uses one message exchange to estimate time.

Q. Berkley Algorithm

The **Berkley Algorithm** is another method used to sync clocks in a distributed system, but it assumes that **no computer** has a perfectly accurate time. Here's how it works:



(a) The time daemon asks all the other machines for their clock values. (b) The machines answer. (c) The time daemon tells everyone how to adjust their clock.

1. **How it works**:
   - A **time server** sends a message to all the computers in the system, asking them for their current clock values.

    - Each computer replies with its own clock value.

2. **Adjusting for Network Delay**:
    - The time server knows the **approximate time** it takes for a message to travel between computers.
    - Using this information, the server **adjusts** the clock values of all computers to account for any network delays.
3. **Calculating the Average Time**:
    - The server then calculates the **average** of all the clock values from the computers.
    - This average time is treated as the **correct time** for the entire system.
4. **Adjusting Clocks**:
    - The server then adjusts its own clock to this **average time**.
    - Instead of sending the actual time to the computers, the server sends each computer the **amount of time** (either positive or negative) it needs to adjust its clock by.
5. **Final Adjustment**:
    - Each computer will adjust its clock based on the value sent by the server.

Advantages:

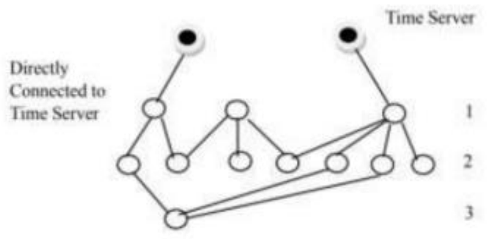
- This method is useful when **no computer** has an accurate time source, as all computers help in finding the average time.

Q. Network Time Protocol (NTP)

NTP is a **protocol** (a set of rules) used to sync the clocks of computers over the internet. It makes sure all computers use the same **reference time**, which is **UTC** (Coordinated Universal Time).

1. **Key Details**:
    - **Application Layer**: NTP works at the application layer, which means it's part of the software that deals with data and communication between computers.
    - **Port 123**: The default port number for NTP is **123**.
    - **UDP**: NTP uses **UDP** (User Datagram Protocol) to send and receive messages over the internet.

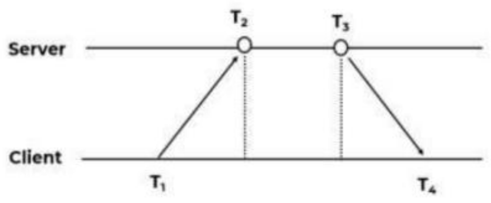
2. **How NTP Works**:
   - NTP helps sync the clocks of computers by using **reference time** (UTC).
   - It is **fault-tolerant**, meaning it can automatically choose the best available time source if one fails.
   - NTP is **scalable**, which means it works well even for very large systems (networks with many computers).
3. **Strata System**:
   - NTP uses a **hierarchical system** called **strata** to organize time servers.
     - **Stratum 1**: These are the **most accurate** time sources and directly connected to the reference time.
     - **Stratum 2**: These get their time from Stratum 1 servers, and so on.
   - Each node (computer or server) in the system exchanges time info with others, either in a **bidirectional** or **unidirectional** manner.



4. **How Time is Shared**:
   - A **client computer** sends a message to a **time server** asking for the time.
   - The server replies with the current time, and the client adjusts its clock based on this info.

Simple Network Time Protocol (SNTP):

- **SNTP** is a **simplified** version of NTP:
  - It's useful for simple systems where **exact time accuracy** isn't as important.
  - **Unicast mode**: SNTP typically works by sending time requests from one computer (client) to another (server), but it can also work in multicast or other modes.



Differences Between NTP and SNTP:

- **NTP** gives **more accuracy** than **SNTP**.
- **SNTP** is easier to set up and works for simpler applications where high accuracy isn't needed.

Limitations of NTP:

1. NTP mainly works well with **UNIX** operating systems.
2. For **Windows**, there can be problems with things like **time resolution**, **reference clocks**, **authentication**, and **name resolution**.

Q. What is a logic clock

A **logical clock** is a way to keep track of **event order** in a **distributed system**, especially when these systems don't have a common clock to rely on. Here's a simpler explanation:

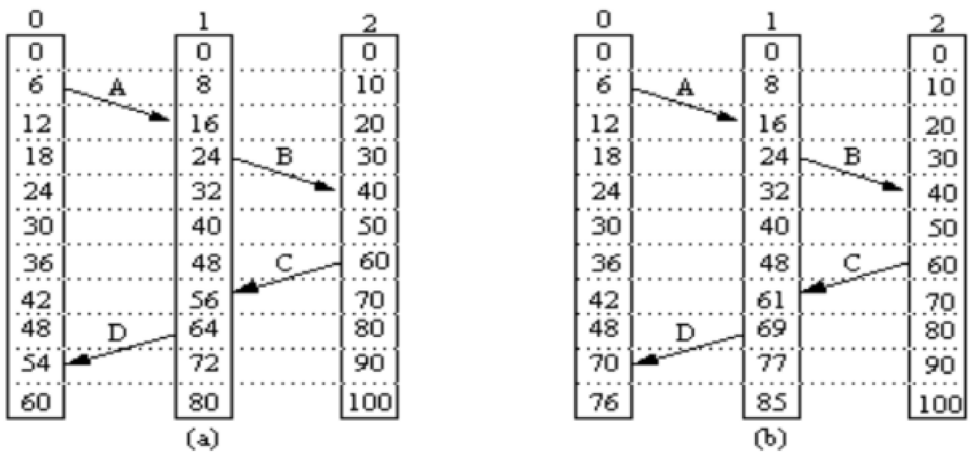
1. **Why Use a Logical Clock?**
    - In distributed systems, there's no guarantee that all computers will have synchronized clocks.
    - A **logical clock** helps us to **order events** and understand their **relationship** without needing an exact, synchronized global time.
2. **How it Works**:
    - Each process (or computer) in the system keeps track of its own **local time** (its own clock) and has information about the **global logical time**.
    - A **logical local time** is used by each process to mark its own events (like a timestamp).
    - When processes send messages to each other, their **global logical time** gets updated.
3. **Key Points**:
    - It helps capture **chronological** and **causal** relationships between events (e.g., what happened first, what happened after).
    - It's useful for things like **distributed algorithms**, **event tracking**, and understanding **computational progress**.

Lamport's Logical Clocks:

1. **The Idea**:

   - In a distributed system, clocks don’t need to be perfectly synchronized. What’s important is knowing **the order of events**.
   - **Lamport’s Logical Clocks** are a method to figure out the **order** of events in such systems.
2. **How Lamport’s Clocks Work**:
   - Lamport clocks give us a **partial ordering** of events using the **"happened-before" relation**.
   - Events that have a **“happened-before”** relationship are ordered, and events that don’t have this relationship are **considered concurrent** (they happened independently).
3. **Rules of Lamport’s Logical Clocks**:
   - **Happened Before Relation**:
     - If **a** happens before **b** in the **same process**, then **a → b**.
     - If **a** sends a message to **b** (across different processes), then **a → b**.
     - If **a → b** and **b → c**, then **a → c** (transitivity).
   - **Logical Clocks Concept**:
     - Since keeping synchronized clocks is hard, we assign **local timestamps** to events based on the **happened-before** rule.

Example to Understand:



Let’s imagine we have **three processes** (let’s call them 0, 1, and 2):

1. **Initial Clock Values**:
   - Process 0: 6
   - Process 1: 8

    - Process 2: 10

2. **Messages Sent Between Processes**:
    - Process 0 sends **message A** to Process 1 at time 6. Process 1 receives it at time 16. (Takes 10 ticks to transfer).
    - Process 1 sends **message B** to Process 2 at time 16. Process 2 receives it at time 40. (Takes 16 ticks to transfer).
    - Process 2 sends **message C** to Process 1 at time 64. It takes 16 ticks, so Process 1 will receive it at time 65 or later.
    - Process 1 sends **message D** to Process 0 at time 69. It takes 10 ticks, so Process 0 will receive it at time 70 or later.
3. **Problem Solved**:
    - Without logical clocks, events might overlap in a confusing way (e.g., receiving a message before it was sent).
    - Lamport's **happened-before relation** ensures that events happen in the correct order (messages arrive in the expected time frames).

Q. What is the requirement of Election algorithm in Distributed Systems? Describe any one Election algorithm in detail with an example

In **distributed systems**, there needs to be a **coordinator process** to manage and control tasks. **Election algorithms** are used to choose which process becomes the **coordinator**. Here's what's needed for an election algorithm:

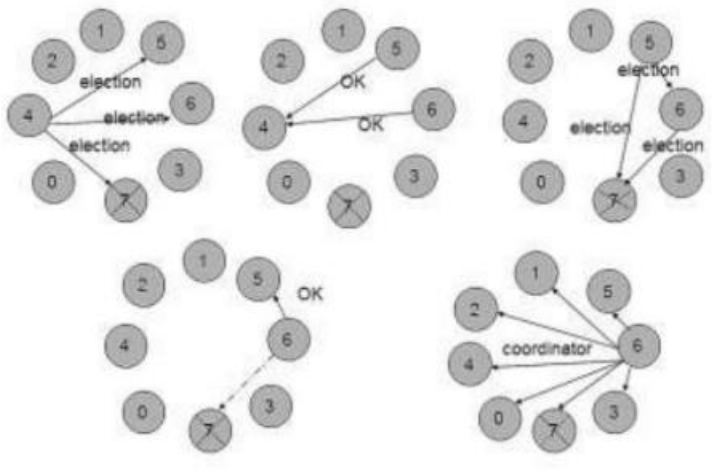
1. **Unique IDs**: Every process in the system must have a **unique ID** (like a process number or network address).
2. **Knowledge of All IDs**: Each process should know all the **other process IDs** in the system, but it doesn't need to know which ones are up or down.
3. **Highest ID as Coordinator**: The process with the **highest ID number** will be chosen as the coordinator.
4. **Multiple Elections**: There can be **multiple elections** happening at the same time, but only one will successfully elect a coordinator.
5. **Elections when Needed**: Elections happen either when the system starts, or if the **coordinator crashes** or leaves.

Bully Algorithm for Election

The **Bully Algorithm** is one method used to elect a coordinator. Here's how it works:

1. **Assumptions**:
   - Each process has a **unique ID**.
   - Every process knows the IDs of other processes.
   - The system is **synchronous** (things happen in order and on time).
   - The process with the **highest ID** is elected as the coordinator.
2. **How it Works**:
   - If a process notices the coordinator is **not responding**, it **calls an election**.
   - It sends a message to all **higher-numbered processes** in the system.
   - If no one replies, the calling process becomes the **new coordinator**.
   - If a higher-numbered process responds, the calling process stops its election and lets the higher-numbered process take over.
   - This higher-numbered process then **calls an election** if it hasn't already done so.
   - This process continues until the highest process becomes the coordinator.
   - The **final winner** sends a message announcing itself as the new coordinator.

Example of Bully Algorithm:



Let's say there are 4 processes with IDs 4, 5, 6, and 7. Here's how the election works:

1. Process 4 notices the coordinator is down and calls an election.
2. Process 4 sends a message to processes 5, 6, and 7 (those with higher IDs).
3. Process 7 replies, meaning process 4 must stop its election.

4. Process 7 now calls an election and sends messages to processes 5 and 6.
5. If process 5 replies, process 7 stops and process 5 calls the election.
6. This continues until **process 7 wins** as the coordinator (since it has the highest ID).
7. Process 7 announces it is the new coordinator.

The algorithm ensures that the **highest-numbered process** wins and becomes the new coordinator.

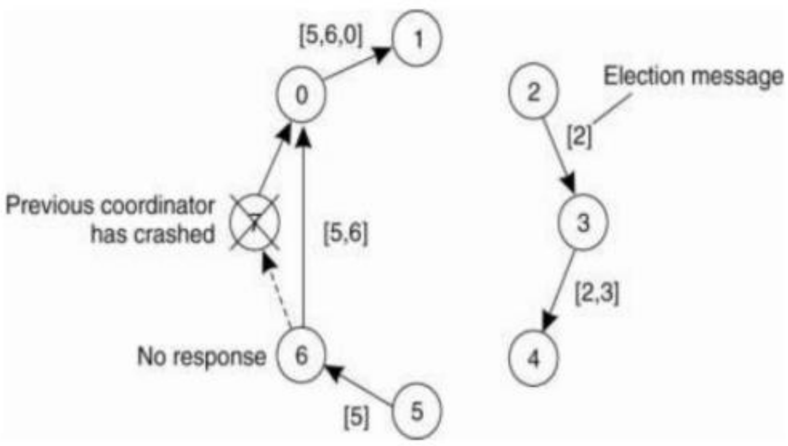
Q. Explain Ring Algorithm and Token ring algorithm.

Ring Algorithm

The **Ring Algorithm** is used for **election** in distributed systems where processes are arranged in a logical ring (circle). Here's how it works:

1. **Ring Structure**: The processes are arranged in a ring, and each process knows the order of the processes in the ring.
2. **Fault Tolerance**: If a process fails, the other processes can skip over it and continue the election.
3. **Election Process**:
    - If a process thinks the coordinator has crashed, it starts an **election**.
    - The process creates an election message with its own ID and sends it to its next process in the ring.
    - Each process adds its own ID to the message and forwards it to the next process.
    - If the message reaches the starting process (the one that started the election), and its own ID appears in the list, it means the election is complete.
    - The process that started the election sends a **coordinator message** to announce the new coordinator with the highest ID.
4. **Multiple Elections**: It's fine to have two elections happening at the same time, and they will eventually complete without issues.

Example: If process 5 starts the election in a ring of processes [5,6,0,1,2,3,4], the message will circulate through all the processes until process 5 sees its own ID, completing the election.

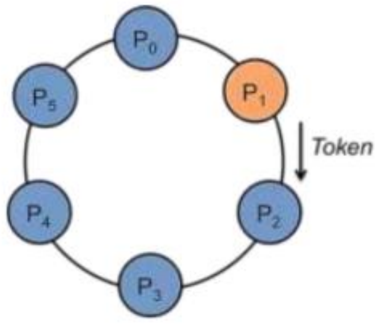


Token Ring Algorithm

The **Token Ring Algorithm** is used to achieve **mutual exclusion** (ensuring that only one process can access a critical section at a time) in a distributed system.

1. **Ring Setup**: The system is arranged in a logical ring, and each process is assigned a position in the ring. A token is passed around the ring.
2. **Token Circulation**:
   - Process P1 starts with the token.
   - The token circulates from one process to the next in a loop.
3. **Critical Section (CS)**:
   - When a process gets the token, it checks if it wants to enter the **Critical Section**.
   - If it does, the process enters the CS, performs its task, and then exits.
   - After exiting the CS, it passes the token to the next process.
4. **Passing the Token**:
   - If a process doesn't need to enter the CS, it simply passes the token to the next process.
   - The token keeps circulating around the ring until a process wants to enter the CS.
5. **No Starvation**: The algorithm ensures that every process gets a chance to access the CS, preventing starvation (where a process might never get to enter the CS).

Example: If process P1 has the token and wants to access a shared resource (critical section), it will do its work, then pass the token to the next process, allowing others to do the same.



Advantages and Disadvantages:

- **Ring Algorithm**:
    - **Advantage**: It's simple and ensures that the highest ID process is selected as coordinator.
    - **Disadvantage**: It may take time for the message to complete a full circle in the ring.
- **Token Ring Algorithm**:
    - **Advantage**: It avoids **starvation** because each process gets a fair chance to access the critical section.
    - **Disadvantage**: **Detecting and regenerating a lost token** can be difficult if it gets lost in the ring.

Q. Explain the centralized algorithms for Mutual Exclusion

Mutual exclusion is a concept used to make sure that only **one process** can access a shared resource (like a printer) at a time. The goal is to **prevent conflicts** where multiple processes try to use the same resource simultaneously.

There are two important conditions for mutual exclusion:

1. **Mutual Exclusion**: Only one process can use the resource at any given time.
2. **No Starvation**: Every process must eventually get access to the resource when it asks for it.

There are two main types of algorithms for mutual exclusion:

1. **Centralized Algorithm**
2. **Distributed Algorithm**

Centralized Algorithm for Mutual Exclusion:

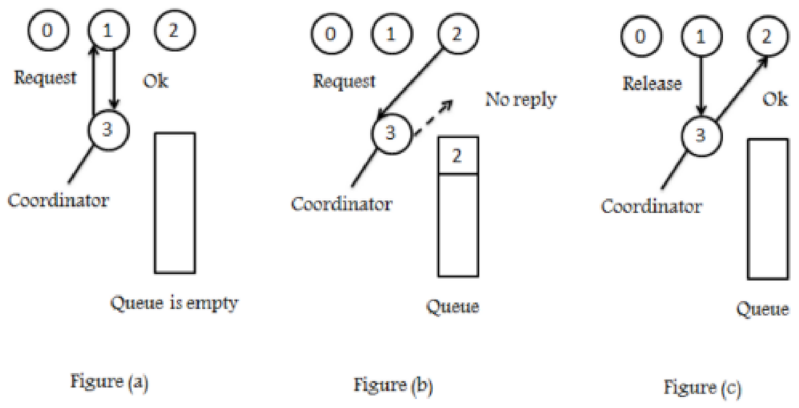
In the **centralized approach**, there is one **coordinator** process that manages access to the shared resource.

How it works:

1. **Coordinator**: One process is chosen as the coordinator (this could be the machine with the highest network address).
2. **Requesting Access**:
   - When a process wants to use the shared resource (e.g., enter a critical section), it **sends a request** to the coordinator, asking for permission to access the resource.
3. **Granting Permission**:
   - If **no one else** is using the resource, the coordinator sends back a **"permission granted"** reply.
   - When the process receives the permission, it enters the critical section (the resource).
4. **Blocking Another Request**:
   - If another process wants to enter the same critical section while one process is already using it, the coordinator **denies the request** (by not replying or explicitly sending a "permission denied" message).
5. **Releasing the Resource**:
   - Once a process finishes using the resource, it **notifies the coordinator** that it's done and releases the resource.
6. **Granting Permission to Next Process**:
   - After receiving the release message, the coordinator takes the first process waiting in the queue and grants it permission to enter the critical region.

Example:

1. **Process 1** wants to enter the critical section and asks the coordinator for permission.
2. The coordinator replies with **"permission granted"** since no one else is using the resource.
3. **Process 2** also requests the resource. The coordinator cannot grant permission immediately because **Process 1** is already using it.
4. When **Process 1** finishes and releases the resource, the coordinator grants permission to **Process 2**.



Advantages of Centralized Algorithm:

1. **No Starvation**: Every process will eventually get access to the resource when it's their turn.
2. **Simple**: It's easy to implement and manage because there's a single coordinator.

Disadvantages:

1. **Single Point of Failure**: If the coordinator crashes, the entire system fails because there's no one to manage access.
2. **Performance Bottleneck**: In large systems, the coordinator can become a bottleneck, causing delays as all requests must go through it.

Q. Explain Ricart-Agrawala algorithm for Mutual Exclusion.

The **Ricart-Agrawala Algorithm** is a **distributed approach** for mutual exclusion in systems where processes are spread across different machines. This method ensures that only one process can access a shared resource (critical section) at a time.

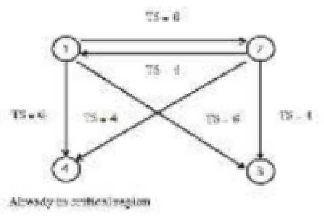
How it works:

1. **Requesting Access to Critical Section**:
   - When a process wants to enter a critical section, it **broadcasts a request** message to all other processes.
   - The message includes:
     - The **process ID** of the requesting process.
     - The **name of the critical section** it wants to enter.
     - A **timestamp** generated by the requesting process (to order the requests).

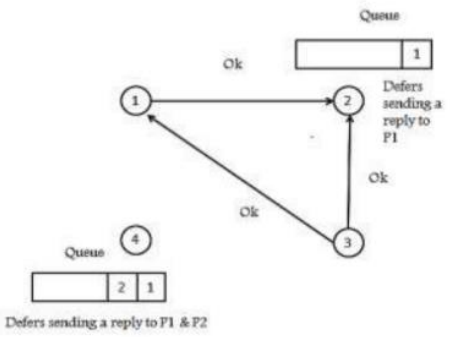
2. **Receiving the Request**: Each process that receives the request checks the following conditions:
   - **If it's already in the critical section**: It does **not reply** to the requesting process.
   - **If it's waiting for the critical section**: It compares its own timestamp with the request's timestamp.
     - If its own timestamp is **later** (greater), it **replies** to the requesting process.
     - If its own timestamp is **earlier**, it **does not reply**.
   - **If it's not interested in the critical section**: It immediately **replies** with an "OK" message.
3. **Entering the Critical Section**:
   - A process can enter the critical section when it has received **"OK" replies** from all other processes, meaning no one else is trying to enter or has higher priority.

Example:

1. **Processes Involved**: There are 4 processes: **1**, **2**, **3**, and **4**.
   - Process **4** is in the critical section.
   - Processes **1** and **2** want to enter the critical section.

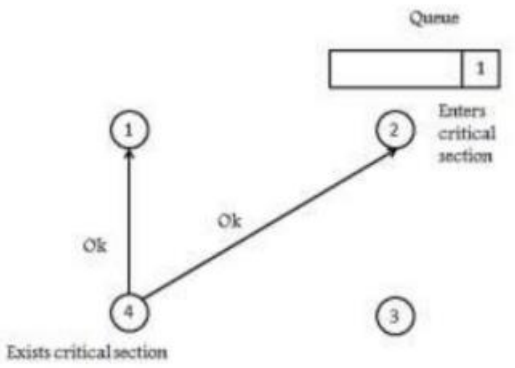


2. **Requesting Access**:
   - Process **1** sends a request with timestamp **6**.
   - Process **2** sends a request with timestamp **4**.

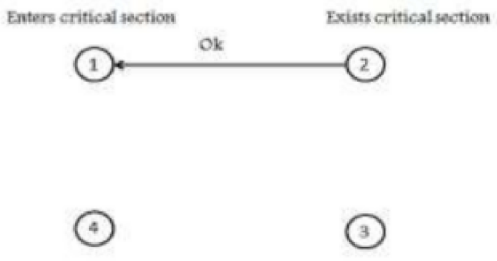


Queue
1
Ok
Defers sending a reply to P1
Ok
Ok
Queue
2 1
Defers sending a reply to P1 & P2

3. **Responses**:
   - Since **Process 4** is in the critical section, it doesn't reply to **1** or **2** and queues their requests.
   - **Process 3** is not in the critical section, so it sends **"OK"** replies to both **1** and **2**.
   - **Process 2** sees that its own timestamp (4) is smaller than **Process 1's** timestamp (6), so it queues **Process 1**'s request and sends an **"OK"** reply to **Process 2**.



4. **Critical Section Entry**:
   - When **Process 4** exits the critical section, it sends an **"OK"** message to all processes in its queue.
   - **Process 2** then gets the **"OK"** message from all processes and enters the critical section.
   - After **Process 2** finishes and exits, it sends an **"OK"** message to **Process 1**, which then enters the critical section.



Advantages:

- **No starvation**: Every process eventually gets a chance to enter the critical section.

Disadvantages:

- **Message broadcasting**: All processes must send and receive broadcast messages, which can be a problem if network bandwidth is limited.

Q. Explain Lamport's Distributed Mutual Algorithm

Lamport's **Distributed Mutual Exclusion Algorithm** is used to ensure that only one process can access a shared resource (like a printer or database) at a time in a distributed system. It's a **contention-based** algorithm, meaning that processes compete to get access to the resource.

Key Features:

1. **Non-Token Based**: Unlike some algorithms (like the Token Ring), this one doesn't use a token for controlling access. Instead, processes send messages to request permission to access the critical section.
2. **Timestamp-Based**: Lamport's algorithm uses **timestamps** to order the requests for the critical section and ensure a fair access order.
3. **FIFO**: The algorithm requires that messages be delivered in **FIFO (First In, First Out)** order between processes.

How the Algorithm Works:

1. **Requesting the Critical Section**:
   - When a process **Pi** wants to enter the critical section, it sends a **request message** with its **process ID (i)** and a **timestamp (Ti)** to all other processes in the system.
   - Pi also places its request in its own **queue** ordered by the timestamp.
2. **Receiving a Request**:
   - When another process **Pj** receives a request from **Pi**, it sends an **acknowledgment (ack)** back to Pi with its own timestamp, showing that it has received the request.
   - Every process maintains a queue called **request_queue** that holds the requests it receives, ordered by their timestamps.
3. **Entering the Critical Section**:
   - Once **Pi** has received an acknowledgment from **every other process** with a timestamp **larger** than its own timestamp, it can enter the critical section.
   - Pi's request must have the **earliest timestamp** in its queue (meaning no other process has a higher priority).
4. **Releasing the Critical Section**:
   - When Pi finishes its work in the critical section, it removes its request from its queue.

   - Pi sends a **release message** to all other processes, informing them that it is leaving the critical section.

5. **Handling Release Messages**:
   - When a process receives a **release message**, it removes the corresponding request from its own queue.
   - This may allow another process's request, which now has the earliest timestamp, to enter the critical section.

Example:

- **Process Pi** sends a request with timestamp **Ti** to all processes.
- **Process Pj** responds with an acknowledgment (ack) and adds Pi's request to its queue.
- **Pi** can enter the critical section only when it has received **acknowledgments** from every other process and its request has the earliest timestamp.
- Once Pi finishes, it sends a **release message** to all other processes to allow the next process with the earliest timestamp to enter.

Advantages:

- **Fairness**: Processes are granted access to the critical section in the order of their requests based on timestamps.
- **No Starvation**: Every process eventually gets access to the critical section.

Disadvantages:

- **Message Overhead**: Since each process must send a message to all others, the algorithm can have significant communication overhead.

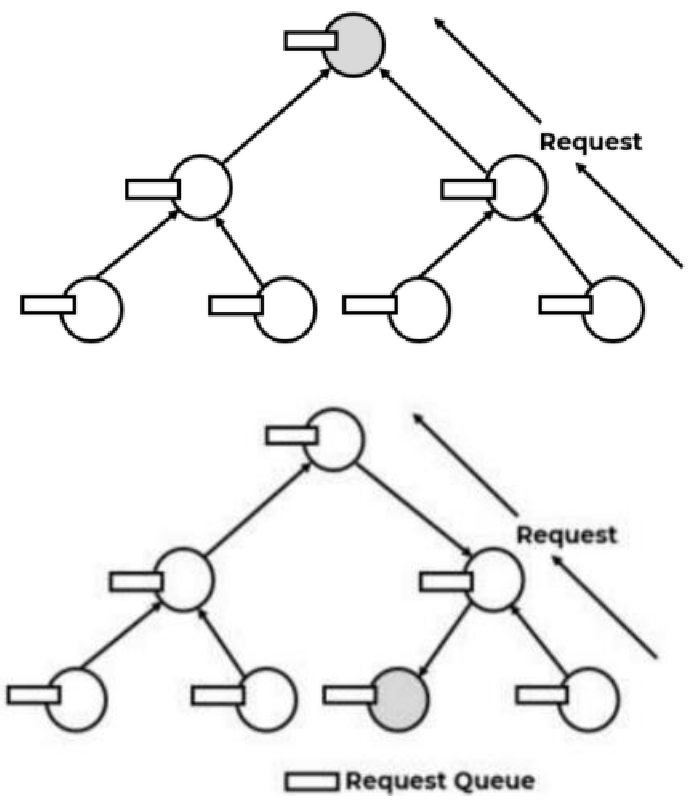
Q. Raymond's Tree based algorithm

Raymond's Tree-Based Algorithm is a **token-based** approach used for mutual exclusion in distributed systems. It organizes processes in a **tree structure**, where each process is a node in the tree.

Key Features:

1. **Token-Based**: There is one token that controls access to the critical section.
2. **Tree Structure**: Processes are arranged in a **directed tree**, with one root node that holds the token.

3. **Parent-Child Relationship**: Each process (node) has a parent, and every node maintains a **FIFO request queue**.
4. **Token Movement**: The token moves along the tree, and the process holding the token can enter the critical section.



How It Works:

1. **Requesting the Token**:
    - When a process (node) wants to enter the critical section, it adds itself to its request queue and sends a request to its **parent** node.
    - The request keeps moving up the tree until it reaches the **root**, which holds the token.
    - As the request moves up, each parent adds the request to its queue, ensuring that all nodes in the tree are aware of the request.
2. **Releasing the Token**:
    - Once the process holding the token has finished using the critical section, it sends the token to the node at the **front** of its request queue (the first waiting process).
    - The token holder then removes that node from its queue and updates its **parent pointer** to point to the new token holder.

    - The process receiving the token checks if it is the one that requested the critical section:
        - If it’s the process that requested the critical section, it enters the critical section.
        - If it’s not the requesting process, it forwards the token to the next process in the queue.

3. **Parent Pointer Updates**:
    - As the token moves, each process updates its **parent pointer** to always point to the current token holder. This allows the process to follow the token by checking its parent pointer.

Example of the Token Flow:

- Process **P1** wants to enter the critical section, so it requests the token by adding itself to the request queue and sending the request up the tree.
- The request continues moving up until it reaches the **root** (the token holder).
- Once the token holder is done, it forwards the token to the first process in the request queue.
- The process that receives the token checks if it’s the one that requested access to the critical section and enters the critical section if so.

Advantages:

- **Efficient Token Movement**: The token moves in a tree structure, and only one token is passed around, ensuring only one process can access the critical section at a time.
- **Reduced Overhead**: The structure allows for fewer messages compared to a fully connected network, making it more efficient in large systems.

Disadvantages:

- **Single Token**: Only one token is used for mutual exclusion, and if the token is lost, it can cause problems.
- **Complexity in Managing Parent Pointers**: The parent pointers need to be updated correctly whenever the token moves.

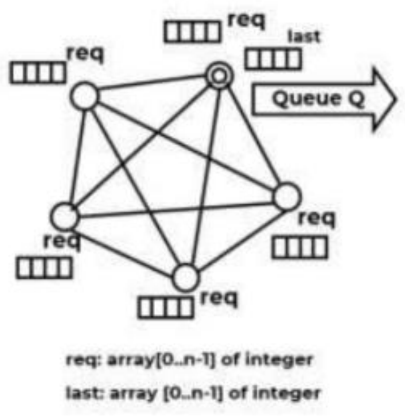
Q. Write a Suzuki-Kasami's Broadcast Algorithm. Explain with example.

Suzuki-Kasami's Broadcast Algorithm is a **token-based** algorithm used to achieve **mutual exclusion** in distributed systems. It's designed to control access to shared resources so that only one process can enter the critical section (CS) at a time. This algorithm is an improvement to the **Ricart-Agrawala** algorithm and is efficient in terms of message passing between processes.

Key Concepts:

1. **Token**: Only the process holding the token can enter the critical section.
2. **Requests**: A process sends out **request messages** to all other processes to ask for permission to enter the critical section.
3. **Arrays**: Each process has two arrays:
   - **Req[i]**: Tracks the latest request number from each process.
   - **Last[i]**: Tracks the last time each process entered the critical section.

How the Algorithm Works:


req
req
last
Queue Q
req
req
req
req: array[0..n-1] of integer
last: array [0..n-1] of integer

1. **Requesting the Critical Section (CS)**:
   - If a process doesn't have the token, it **increments** its request number and **broadcasts** a request message to all other processes.
   - When a process receives the request, it updates its **Req[j]** (the request number for the requesting process) and checks if it has the token.
   - If it has the token and the requesting process is asking in the correct order (i.e., **Req[i] = Last[i] + 1**), it sends the token to the requesting process.
2. **Releasing the Critical Section**:
   - When a process finishes its work in the critical section, it updates its **Last[i]** to match the current **Req[i]** (request number).
   - Then, it checks the request queue:

        - It adds the IDs of processes waiting for the token to the **token queue** if their request number matches the required sequence (**Req[j] = Last[j] + 1**).
    - If there are processes in the queue, the process sends the token to the next process in line (the one at the front of the queue).
3. **Executing the Critical Section (CS)**:
    - A process can **enter the critical section** only when it holds the token.

Example:

Let's walk through the example of the algorithm:

1. **Step 1**: Initial State:
    - Process 0 sends a request to all processes and gets the token.
2. **Step 2**:
    - Now, Process 1 and Process 2 send requests to enter the critical section.
3. **Step 3**:
    - Process 0 finishes its work in the critical section and prepares to release the token.
4. **Step 4**:
    - Process 0 sends the token (along with its request and last entry information) to Process 1.
5. **Step 5**:
    - Now, Process 0 and Process 3 send requests to enter the critical section.
6. **Step 6**:
    - Finally, Process 1 sends the token to Process 2 when it's done using the critical section.

Advantages:

- **Fairness**: Ensures that every process can eventually access the critical section without starvation.
- **Efficient**: Reduces the number of messages by using the token-based system.

Disadvantages:

- **Token Loss**: If the token is lost, the system must have a recovery mechanism.
- **Message Overhead**: Broadcasting request messages to all processes can lead to overhead in large systems.

Q. Compare and contrast mutual exclusion algorithms

Parameter	**Centralized Algorithm**	**Distributed Algorithm**	**Token Ring Algorithm**
Election	One process is chosen as the coordinator.	There's no central coordinator. Events are ordered.	A token is passed around, and the holder enters the critical section.
Messages per entry/exit	3 messages required to enter or exit critical section.	2(n-1) messages required to request permission.	Varies, depending on the position of the token.
Message delay	Delay is around 2 messages.	Delay is 2(n-1) messages.	Delay varies from 0 to n-1 messages.
Mutual Exclusion	Ensures only one process can enter the critical section.	Mutual exclusion is guaranteed without deadlock.	Mutual exclusion is guaranteed.
Starvation	No starvation (each request gets served).	No starvation (requests are eventually granted).	No starvation (every process will eventually get the token).
Complexity	Simple and easy to implement.	More complicated to set up.	Simple if processes are in a ring.
Used for	General resource allocation in smaller systems.	Small, fixed groups of processes.	Systems where processes form a ring.
Problems	Single point of failure (coordinator can fail).	Multiple points of failure; hard to recover lost tokens.	Token can get lost, making recovery difficult.
Cost/Expenses	Low cost (less overhead).	Higher cost due to message exchange.	Low cost (simple setup).
Robustness	More robust (as only one coordinator exists).	Less robust (multiple failure points).	More robust, but token loss can be an issue.

Q. Comparison: Token-Based vs Non-Token-Based Algorithms

Feature	Token-Based Algorithm	Non-Token-Based Algorithm
How It Works	Uses a "token" that is passed around to decide which process can enter the critical section.	Relies on messages and timestamps to manage access to the critical section without a token.
Example Algorithms	Raymond's Tree, Suzuki-Kasami, Token Ring	Lamport's Algorithm, Ricart-Agrawala
Message Exchange	Only requires passing the token between processes.	Requires multiple messages for each request and response.
Message Overhead	Low message overhead (only token passing).	Higher message overhead due to multiple requests and replies.
Fairness (Starvation)	No starvation (token ensures every process gets a chance).	No starvation if implemented well (requests are handled based on timestamps).
Fault Tolerance	If the token is lost, it may require recovery mechanisms.	No token, but multiple messages and processes could fail if not managed properly.
Simplicity	Simple to implement in a ring or tree structure.	More complex due to message exchanges and ordering.
Scalability	Works well in fixed topologies (e.g., ring or tree).	Works in distributed systems but with higher overhead in large systems.
Examples of Systems	Used in systems with fixed topology like distributed systems.	Used in more flexible, decentralized systems where a central token is not needed.

Q. Explain Deadlock

Deadlock refers to a situation in which a group of processes are blocked because each process is holding a resource and waiting for another resource held by another process. This creates a cycle of waiting processes, and no process can proceed.

In distributed systems, deadlock handling becomes more complex because:

- Resources and processes are distributed across different nodes.

- Information about resource allocation and process states is scattered, making it difficult to monitor and control.

Deadlock in Distributed Systems:

- In centralized systems, deadlock is easier to handle because resources and processes are controlled at a single location.
- In distributed systems, deadlocks are more complex to detect and resolve because processes are spread across various machines, and the state of resources may not be easily visible.

Strategies for Handling Deadlock:

In distributed systems, deadlock can be managed using one of the following strategies:

1. **Avoidance**: Resources are allocated cautiously to avoid deadlocks.
2. **Prevention**: Constraints are imposed on how processes request resources to prevent deadlocks.
3. **Detection and Recovery**: Deadlocks are allowed to happen, but the system uses algorithms to detect and recover from them.

Types of Deadlock in Distributed Systems

1. **Resource Deadlock**:
   - **What happens?** A process waits for a resource held by another process, and that process is waiting for a resource held by the first process, creating a circular wait.
   - **Example**:
     - **Process 1** has resources R1 and R2 but needs R3 to continue.
     - **Process 2** has R3 and needs R1 to continue.
     - Neither process can proceed because each is waiting for the other to release a resource.
2. **Communication Deadlock**:
   - **What happens?** A set of processes are blocked, waiting for messages from other processes, but no messages are being sent between the processes.
   - **Example**:
     - **Process 1** wants to communicate with **Process 2**.
     - **Process 2** wants to communicate with **Process 3**.
     - **Process 3** wants to communicate with **Process 1**.

        - None of the processes can proceed because they are all waiting for each other.

Deadlock Detection in Distributed Systems

In distributed systems, deadlock detection is more challenging than in centralized systems due to the distributed nature of resources and processes. The system must use specialized algorithms to detect cycles in the wait-for graphs (WFGs) to identify deadlocks.

Requirements for Deadlock Detection:

1. **Progress**: The detection method should be able to detect all deadlocks.
2. **Safety**: The method must be reliable and detect deadlocks accurately.

Deadlock Detection Techniques:

1. **Centralized Approach**:
    - **How it works**: One node (the **deadlock-detection coordinator**) maintains a global wait-for graph and is responsible for detecting deadlocks.
    - **Process**:
        - The coordinator periodically checks the global wait-for graph for cycles (which indicate deadlocks).
        - If a cycle is found, the coordinator picks a victim process and rolls it back to break the deadlock.
    - **Disadvantages**:
        - **Single point of failure**: If the coordinator fails, the whole system might be affected.
        - **High workload**: The coordinator has to monitor all processes and resources.
2. **Hierarchical Approach**:
    - **How it works**: Combines both centralized and distributed approaches.
    - **Process**: A central node oversees deadlock detection for clusters of nodes or resources.
    - **Advantages**: This approach reduces the load on a single node compared to the centralized method.
3. **Distributed Approach**:
    - **How it works**: Multiple nodes collaborate to detect deadlocks. Each node maintains its own local state and contributes to global deadlock detection.

- **Advantages**: No single point of failure, and the load is distributed across nodes, improving scalability and reliability.

Q. Chandy-Misra-Haas Algorithm

The **Chandy-Misra-Haas (CMH) algorithm** is used to detect deadlocks in distributed systems using an approach called **edge chasing**. Here's how it works:

Key Concepts

1. **Probe Message**:
    - A special message called a **probe** is used for deadlock detection.
    - The probe is a triplet (i, j, k), where:
        - **i**: The process that starts the probe.
        - **j**: The process being probed.
        - **k**: The process to which the probe is being sent.
2. **Wait-For Graph (WFG)**:
    - The **probe** message travels through the **Wait-For Graph** (WFG), which shows which processes are waiting for others.
    - If the probe message forms a cycle (i.e., it comes back to the original process **i**), it indicates a **deadlock**.
3. **Dependent Processes**:
    - A **dependent** process is one that depends on another process for execution. For example, if Process P1 is waiting for a resource held by Process P2, then P1 is **dependent** on P2.
    - **Locally dependent** means both processes are on the same site.

Algorithm Steps:

1. **Initiating the Probe**:
    - If a process **P** is **locally dependent** on itself (i.e., it's waiting for itself or resources it cannot get), it **declares a deadlock**.
    - Otherwise, process **P** checks for the following:
        - **P** is waiting on **Pj** (i.e., Process P is blocked and waiting for Pj).
        - **Pj** is waiting for **Pk**.

        - Processes **P** and **Pk** are on different sites (distributed system).
        - If all these conditions are true, **P** sends a **probe** message (i, j, k) to the home site of **Pk**.

2. **Receiving the Probe**:
    - When **Pk** (the recipient process) receives the probe, it checks the following conditions:
        - **Pk** is blocked (waiting for resources).
        - **P** (initiating the probe) has not yet replied to **Pj**.
        - If both conditions are true, **Pk** sets **dependent[i]** to true and checks if **k == i**:
            - If **k == i**, then **P** is deadlocked.
            - Otherwise, **Pk** checks if **Pk** is dependent on any other process and sends a probe for further checking.
3. **Deadlock Detection**:
    - If the probe message loops back to the initiating process **i**, then it **declares a deadlock** because it means there's a cycle in the wait-for graph.

Advantages:

- **No special data structures** are needed, just a small probe message and a simple Boolean array.
- **Low computation** and **low overhead** for each site.
- **No false deadlocks** (phantom deadlocks). The algorithm is accurate.
- Efficient, as it doesn't require building complex graphs or passing them around between sites.

Disadvantages:

- **Not all sites** may know about all the processes involved in a deadlock, making resolution tricky.
- **Proof of correctness** for the algorithm can be difficult.
- **Communication delay** might slow down deadlock detection, especially in large systems with many processes.

Performance:

- The algorithm may require up to **m(n-1)/2** messages, where **m** is the number of processes and **n** is the number of sites in the system.

- The time to detect a deadlock is proportional to **O(n)**.

Example:

Imagine three processes **P1**, **P2**, and **P3** on three different sites. If **P1** is waiting for **P2**, **P2** is waiting for **P3**, and **P3** is waiting for **P1**, this forms a circular dependency, i.e., a **deadlock**.

The Chandy-Misra-Haas algorithm would send probes through the system to check if this cycle exists, and if the probe returns to the initiating process (e.g., **P1**), it would declare that a deadlock has occurred.

Q. Maekawa Mutual exclusion algorithm

Maekawa's Algorithm is a **quorum-based** approach for mutual exclusion in distributed systems. Unlike other algorithms (like Lamport's or Ricart-Agrawala), which require a process to request permission from all other processes, Maekawa's algorithm only asks for permission from a **subset** of processes, called a **quorum**.

Key Concepts:

- **Quorum**: A **subset of processes** that a site requests permission from. Each site needs permission from its quorum to enter the critical section.
- **Messages Used**:
    1. **REQUEST**: A process sends this to its quorum to ask for permission to enter the critical section.
    2. **REPLY**: A process sends this to approve another process's request.
    3. **RELEASE**: A process sends this to notify others that it has exited the critical section.

Steps of the Algorithm:

1. **Requesting Permission**:
    - When a site (say **Si**) wants to enter the critical section, it sends a **REQUEST** message to all the sites in its **quorum**.
2. **Responding to a Request**:
    - If a site (**Sj**) receives a **REQUEST** message from **Si**, it sends a **REPLY** message to **Si**, but only if it hasn't already replied since the last time it sent a **RELEASE** message. Otherwise, it adds **Si's** request to its **queue**.
3. **Entering the Critical Section**:

   - A site can enter the critical section only when it has received a **REPLY** from **all** the sites in its quorum.

4. **Exiting the Critical Section**:
   - After finishing its work in the critical section, the site sends a **RELEASE** message to all the sites in its quorum.
   - When a site (**Sj**) receives a **RELEASE** message from another site (**Si**), it:
     - Sends a **REPLY** to the next site in its **queue** (if there's a request waiting).
     - If the queue is empty, it updates its status to show that it hasn't replied to any requests since the last **RELEASE**.

Properties of the Request Set (Quorum):

For Maekawa's algorithm to work:

1. Each pair of sites must have at least **one common site** in their request sets.
2. Every site is part of exactly **K request sets**.
3. The number of sites, **N**, must satisfy the equation:
   N=K(K−1)+1N = K(K - 1) + 1

Message Complexity:

- For each critical section entry, Maekawa's algorithm requires **3√N** messages. This includes:
  - **N request messages**
  - **N reply messages**
  - **N release messages**

Drawbacks:

- **Deadlock-prone**: Since the requests are not prioritized by timestamps, a site may end up waiting forever if another site holds the resources it needs.

Performance:

- **Synchronization delay**: The delay is about twice the message propagation time.
- **Message overhead**: Requires **3√N** messages per critical section execution.