Distributed-Computing-Module-2-Important-Topics-PYQs

        - 1. Isomorphism (Partial Ordering)
        - 2. Strong Consistency
        - 3. Event Counting
    - Applications of Vector Time
- 6. Illustrate bully algorithm for electing a new leader. Does the algorithm meet liveness and safety conditions?
  - How It Works:
    - Bully Algorithm Example
    - Safety and Liveness
        - Safety
        - Liveness
- 7. Discuss the method of termination detection by weight throwing in detail.
    - Key Characters:
    - Weights (W):
    - How it works – Rules and Flow:
        - 1. Starting the Work (Rule 1):
        - 2. When a Process Gets a Message (Rule 2):
        - 3. Becoming Idle (Rule 3):
        - 4. When the Manager Gets Weight Back (Rule 4):
    - Why this works?
    - Example
        - Setup:
        - Step 1: Controlling agent starts the computation
        - Step 2: P1 does some work and sends a message to P2
        - Step 3: P2 finishes its work
        - Step 4: P1 also finishes
- 8. Illustrate the working of spanning tree-based termination detection algorithm.
    - Basic Idea:
    - Rules Recap in Simple Terms:
    - Example of the algorithm
        - Step 1
        - Step 2
        - Step 3

        - Step 4
        - Step 5
        - Step 6
    - Time complexitiy
- 9. Explain in detail about Chandy Lamport algorithm.
    - How It Works (Step-by-Step):
    - Example Scenario:
    - When is This Useful?
- 10. What is meant by a consistent global state?
- 11. Explain ring based election algorithm with an example. Does the algorithm meet liveness and safety conditions?
    - How It Works:
    - Example
    - Liveness and safety conditions
        - Safety Condition
        - Liveness Condition
- 12. Explain how logical clock is implemented.
    - 1. Data Structures:
    - 2. Protocol (Rules) for Updating the Clock:
        - R1 – Local Event Rule
        - R2 – Message Rule (Global View Update)
- 13. Clearly mentioning assumptions, explain the rules of termination detection using distributed snapshots.
    - Assumptions
    - Rules of Termination Detection (Using Distributed Snapshots)
        - Rule 1: Unique Last Idle Process
        - Rule 2: Snapshot Request Trigger
        - Rule 3: Granting the Snapshot Request
        - Rule 4: Successful Snapshot
        - Rule 5: Termination Detected via Snapshot

1. Define Termination Detection.

What is Termination Detection?

In a **distributed system**, multiple computers (or processes) work together to solve a problem. Since these processes are running independently and communicating over a network, it's **hard to tell when the entire system has finished its work**.

Termination detection is the process of figuring out **when all processes have finished their tasks**, and there are **no messages still traveling through the network**. Only then can we say, "The system is done!"

Imagine This Scenario:

Let’s say we have **three people** (Process A, Process B, and Process C) working on solving a puzzle. They pass notes to each other when they find clues (these notes are like **messages**). Here’s the problem: **how do they all know when the puzzle is completely solved?**

Even if A finishes their part and stops, B might still be working, or a note might be on its way from C to B. So, how do we detect that everyone has stopped working and no notes are in transit?

Rules for Termination Detection:

1. **No Freezing the Work (No Interference with Computation):**
   - The system should continue solving the puzzle while checking if it's done. We can't pause everyone just to ask, "Are you finished?"
   - **Example:** If A is working, we shouldn’t interrupt them just to ask if they’re done.
2. **No New Ways of Talking (No New Communication Channels):**
   - The people can only send notes through the methods they already use. We can’t suddenly give them walkie-talkies to check if they’re finished.
   - **Example:** If A, B, and C usually pass handwritten notes, we can’t introduce phone calls to check on them.
3. **Every Note Will Arrive (Reliable Message Delivery):**
   - Even if there are delays, every note passed between people will eventually reach its destination.
   - **Example:** If C sends a note to A, even if it takes a long time, it will eventually arrive.
4. **Notes Can Be Late (Finite but Unpredictable Delays):**
   - The notes might take different amounts of time to arrive, but they won’t get lost forever.

- **Example:** A might send a note to B, and it could arrive quickly or take a while, but it will get there.

5. **People are Either Working or Waiting (Process States):**
   - At any time, a person is either **active** (working on the puzzle) or **idle** (waiting, doing nothing).
   - **Example:** A is active when solving clues. A becomes idle when they have no more clues to solve.
6. **Rules for Switching Between Working and Waiting (Transition Rules):**
   - **Active to Idle:** A person can stop working when they finish their part.
     - **Example:** B finishes their clue and stops.
   - **Idle to Active:** A person can only start working again if they get a note from someone else.
     - **Example:** A was idle but starts working again when they receive a new clue from C.
   - **Only Active People Can Send Notes:** If you're not working, you can't send notes.
     - **Example:** If A is idle, they can't send any more messages until they become active again.
7. **Special "Are You Done?" Notes (Control Messages):**
   - To detect if everyone is finished, special notes (control messages) are used to ask, "Are you done?" But these should not mess up the regular notes being passed.
   - **Example:** Besides the puzzle clues, people pass special notes asking, "Have you finished your clues?" without mixing them up with puzzle hints.
8. **Notes Don't Have to Be in Order (No FIFO Requirement):**
   - Notes sent between the same people don't have to arrive in the order they were sent.
   - **Example:** If A sends two notes to B, the second one might arrive before the first.
9. **Sending and Receiving are Instant (Atomic Actions):**
   - When someone sends or receives a note, it happens completely, without half-finished actions.
   - **Example:** You can't half-send a note; it's either sent or not. Similarly, you can't half-receive a note.

How Does This Help in Detecting Termination?

- **Goal:** We want to find out when **everyone has stopped working** and **no notes are in transit**.

- If everyone is idle and no messages are left to be delivered, we can safely say the work is done.

Example of Termination Detection:

Let’s say Process A finishes its task and goes idle. But before declaring that everything is done, we need to make sure:

1. B and C have also finished their tasks.
2. No messages are still traveling between A, B, and C.

Only when both conditions are true can we detect that the entire system has terminated.

2. What are the basic properties of scalar time.

Imagine you have a bunch of computers (or processes) working together but not sharing a clock. They need to **agree on the order of events** (like "who sent a message first?").

To do this, **Lamport** came up with the idea of **Scalar Time** in 1978.

Each process has its own **logical clock** — just a simple counter (like `C1`, `C2`, etc.). Whenever something important happens (like sending or receiving a message), the process **updates its counter** using two simple rules.

Rules to Update Clocks

R1. Local Event: When a process does something (like sending a message), it increases its counter by 1.

> E.g., if `C1 = 5`, and process does something, it becomes `C1 = 6`.

R2. Message Received: When a process receives a message, it compares its own clock with the timestamp in the message and takes the **maximum + 1**.

> If `C2 = 3`, and it gets a message with time `5`, it sets `C2 = max(3,5)+1 = 6`.

This way, time **always moves forward**!

Properties of Scalar Time

1. Consistency Property

If one event happens before another (in real life), scalar time will reflect that.

2. Total Ordering

- We can use scalar timestamps to **totally order** events across the whole system.
- But sometimes... **two events in different processes might have the same timestamp**
- We use **process IDs** to break ties.
  - Example:
    - If two events have timestamp `5`,
    - one happened in Process 1, and the other in Process 2,
    - we write their time as `(5,1)` and `(5,2)`.
    - Since 1 < 2, we say `(5,1)` happened first.
    - So the final timestamp is a **tuple**: `(time, process_id)`

3. Event Counting

Because the clock increases with each event, you can **count events** using the scalar clock values.

4. No Strong Consistency

Scalar time is not *perfectly accurate*. It **does not guarantee** that if two events have different timestamps, one definitely happened before the other. It only helps with a rough, logical ordering.

3. What are the rules used to update clocks in scalar time representation?

R1. Local Event: When a process does something (like sending a message), it increases its counter by 1.

> E.g., if `C1 = 5`, and process does something, it becomes `C1 = 6`.

R2. Message Received: When a process receives a message, it compares its own clock with the timestamp in the message and takes the **maximum + 1**.

> If `C2 = 3`, and it gets a message with time `5`, it sets `C2 = max(3,5)+1 = 6`.

This way, time **always moves forward**!

4. Specify the issues in recording a global state.

In a distributed system (multiple computers or processes working together), a **global state** is like taking a photo of the whole system at one moment. It includes:

- Each process's state (what it's doing, local variables, etc.)
- Each communication channel's state (messages in transit)

Issue 1: Which messages should be included in the snapshot?

- Imagine
  - Process A sends a message to Process B.
  - Process B takes its snapshot **before** it receives the message.
- Question: Should that message be part of the snapshot?
  - We need a clear rule to decide:
  - Is the message already in the process's state?
  - Or is it "in transit" and should be recorded in the **channel**?
  This is tricky because **messages can fly across the network at any time**, and we need to **capture a consistent picture**.

Issue 2: When should each process take its snapshot?

Without a global clock, processes can't all take snapshots at the same moment.
So

- How do we decide **when each process should capture its state**?
- If they all do it at random times, the snapshot might not make sense (it might show events out of order).

We need a **coordinated way** (like a special message or protocol) to tell each process **when it's the right time** to take its snapshot, based on what's happening in the system.

5. Define vector time.

Imagine you have several computers (processes) working together in a distributed system. These computers send messages, perform actions, and we want to know:

- Which event happened **before** another?
- Did two events happen **independently**, or were they **causally connected**?
  That's where **Vector Clocks** come in.

A **vector clock** is like a special timestamp, but instead of just one number (like in scalar clocks), it's a **list of numbers** — one for **each process** in the system.
If there are 3 processes (P1, P2, P3), each process has a vector clock like this:

```
[clock_P1, clock_P2, clock_P3]
```

Each process keeps track of:

- **Its own actions**
- **What it knows** about the others

How Does It Work?

Let's say we have **3 processes (P1, P2, P3)**.
Each one starts with a clock: `[0, 0, 0]`

Local event at P1:

- P1 does something.
- It updates its own entry: `[1, 0, 0]`

P1 sends a message to P2:

- It sends the message **with its vector clock** `[1, 0, 0]`

P2 receives the message:

- It compares the received clock `[1, 0, 0]` with its own clock `[0, 1, 0]`
- It **takes the max of each position** and then adds 1 to its own position:

```
max([1, 0, 0], [0, 1, 0]) = [1, 1, 0]
Then increment P2's own value:
```

```
[1, 2, 0]
```

Key Properties of Vector Time

1. Isomorphism (Partial Ordering)

If **event A → event B** (A happened before B), then **A's vector clock < B's vector clock** (we compare them element-by-element).

This helps us create a **partial order** of events in the system.

2. Strong Consistency

Unlike scalar clocks, **vector clocks can tell us exactly if two events are related**:

- If A's vector clock < B's, then A happened before B
- If A and B are not comparable, they happened **concurrently**

3. Event Counting

Just like scalar clocks, we can count events because vector entries increase with every action.

Applications of Vector Time

Because vector clocks **track causality perfectly**, they're super useful:

- **Distributed debugging** – trace what happened and in what order
- **Causal message ordering** – ensure messages are delivered respecting cause-and-effect
- **Causal shared memory** – maintain consistent views of shared data
- **Global breakpoints** – pause systems in a stable state
- **Checkpoint consistency** – useful in failure recovery

6. Illustrate bully algorithm for electing a new leader. Does the algorithm meet liveness and safety conditions?

In distributed systems (like a group of computers working together), it's important to have one computer in charge—this is called the **leader** or **coordinator**. The **Leader Election Algorithm** helps decide **which computer becomes the leader**.

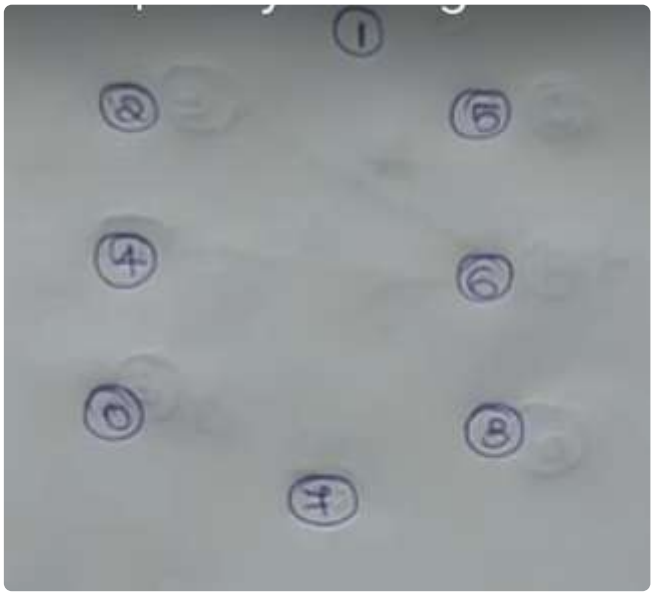
Bully Algorihtm is a type of leader election algorithm

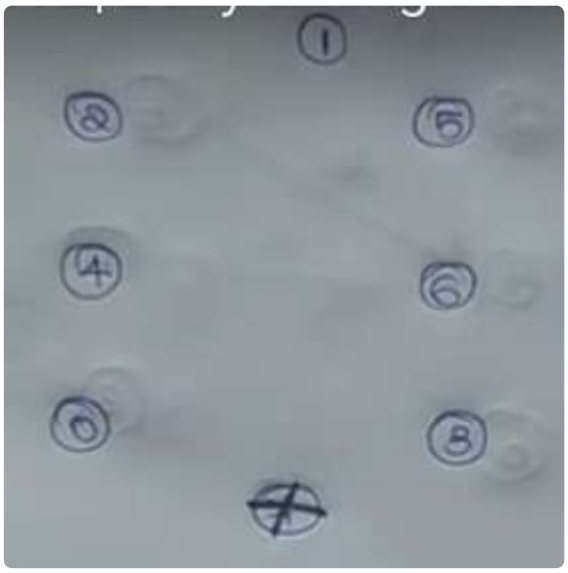
In this algorithm, the **computer with the highest ID always wins**—that's why it's called the **Bully Algorithm**

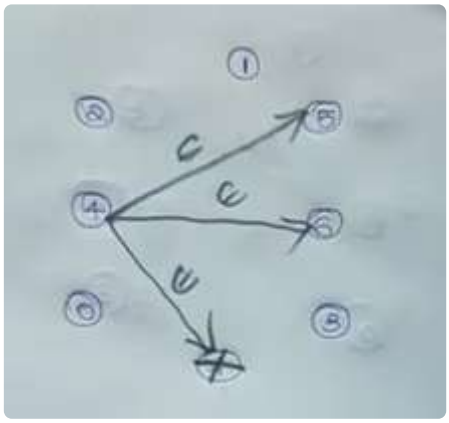
How It Works:

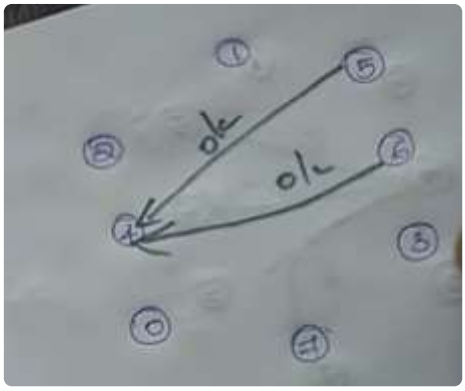
1. **Starting the Election:**
   - A process that thinks the leader has failed **sends an election message to all processes with higher IDs**.
2. **Responses:**
   - If **no higher process responds**, the process **declares itself the leader**.
   - If a **higher process responds**, that process **takes over** the election.
3. **Choosing the Leader:**
   - The process with the **highest ID** becomes the leader and sends a **"coordinator" message** to all others.
4. **If the Leader Fails Again:**
   - The process that notices the failure **starts a new election**.

Bully Algorithm Example

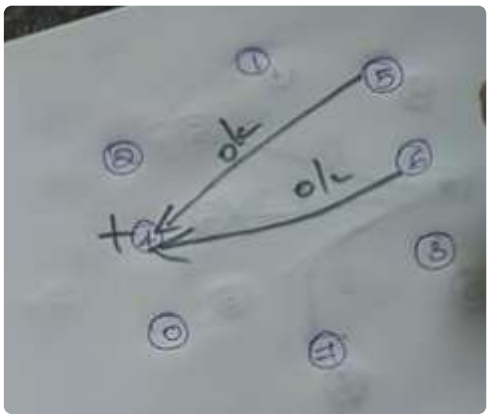
- There are processes 0-7
  - 
- Assume 7th process is down

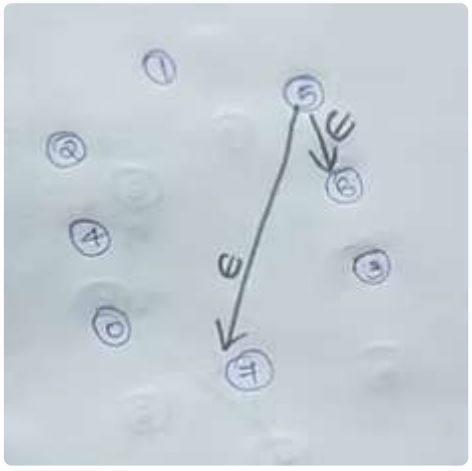
- 

- Process 4 reports that 7th process is down
- 4 sends an election message
  - 4 Sends the election message to the larger numbers
  - Here the numbers larger than 4 are: 5,6,7
  - 

  - An Election message's response will be an OK Message
  - Since 7 is down, it wont return OK message
- But 5 and 6 are active, so it will return OK message
  - 

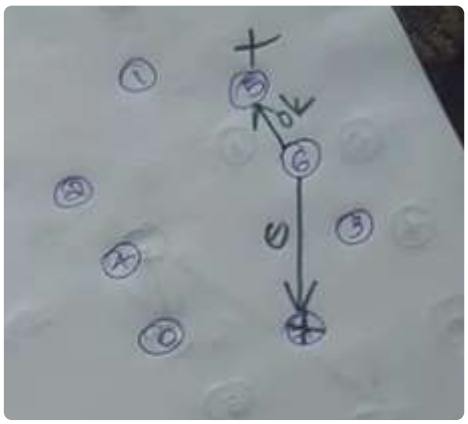
  - Since 4 gets OK message, it understands that it cant be coordinator, because there are larger numbers

- 

- The ones that gave OK message will now conduct election
  - 5 and 6 will now Conduct election
- When 5 conducts the election
  - 

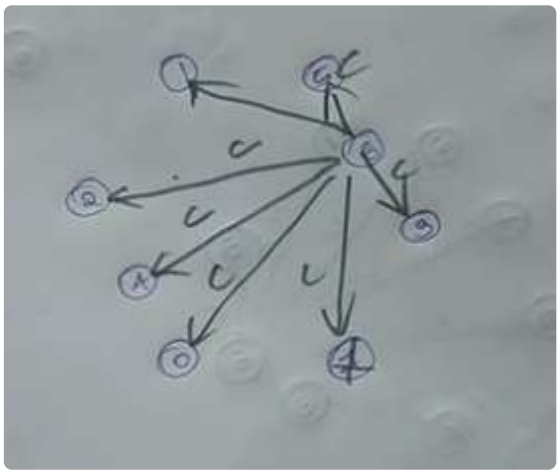
  - It gives election message to 6 and 7
  - Since 7 is down. it wont send OK message
  - 6 sends an OK message
  - 

  - 5 understands, theres someone larger so it cant be coordinator
  - Now 6 will conduct election
- When 6 conducts election
  - The only number greater than 6 is 7, so it sends an election message to it



- But since 7 is down, there wont be any OK messages
- Since there are no OK messages, 6 declares itself as the coordinator

- 6 Sends the coordinator message
  - 6 will now send a coordinator message to all other processes to let them know that 6 is the new coordinator



  - Note
    - If 7 comes back up, it notices 6 is the coordinator, and it retakes back its position as the coordinator, Making it a bully algorithm

Safety and Liveness

Safety

- **Safety** means: *Nothing bad happens.*
- In the context of the Bully Algorithm:
  - **Only one process becomes the coordinator (leader)** after the election.

    - The elected coordinator is always the **highest-priority (i.e., highest-ID) active process**.
    - No two processes declare themselves as coordinators at the same time **after** the algorithm finishes.
- **Conclusion**: The Bully algorithm **meets the safety condition**.

Liveness

- **Liveness** means: *Something good eventually happens.*
- In this case:
    - The algorithm should **eventually elect a leader**, even if there are failures.
    - The election eventually completes, and a process becomes the coordinator.
- The Bully Algorithm can suffer **delays** if many messages are lost or delayed.
- But under the assumption that:
    - All messages are eventually delivered (i.e., **reliable communication**), and
    - Crashed processes remain crashed or eventually recover,

Conclusion: The Bully algorithm **meets the liveness condition**, **under standard assumptions** (finite time message delivery, failure detection eventually succeeds).

7. Discuss the method of termination detection by weight throwing in detail.

We want to **know when a distributed computation has ended**. This can be tricky because different processes are working independently and might still be sending messages to each other. So we use a technique called **Termination Detection by Weight Throwing** to figure this out.

Key Characters:

1. **Controlling Agent** – like the manager, it monitors everything.
2. **Processes (P1, P2, etc.)** – the workers who actually do the job.
3. **Messages** – ways to communicate between the controller and processes, and between processes.

Weights (W):

- Everything starts with a **total weight of 1**.

- Initially, the **controlling agent has all the weight (W = 1)**.
- All other processes start **idle** with weight 0.

How it works – Rules and Flow:

1. Starting the Work (Rule 1):

- The **controlling agent** wants a process to do some work.
- It **splits its weight** into two parts, say W1 and W2 (W1 + W2 = current W).
- It **keeps W1**, and **sends W2** in a **basic message** `B(W2)` to a process `P`.
- That process now has **W2 weight** and is considered **active**.

2. When a Process Gets a Message (Rule 2):

- Suppose a process `P` receives a **basic message** `B(DW)`.
- It **adds DW** to its own weight.
- If it was idle, it becomes **active**.

3. Becoming Idle (Rule 3):

- If a process finishes its work and wants to stop, it:
  - Sends a **control message** `C(W)` to the **controlling agent**.
  - Sets its own weight to **0** (it's now passive).

4. When the Manager Gets Weight Back (Rule 4):

- The **controlling agent** receives a **control message** `C(DW)`.
- It **adds DW** to its own weight.
- If its total weight becomes **1 again**, it means:
  - No processes are active.
  - No messages are in transit.
  - **Computation has ended!**

Why this works?

- **Weight Conservation**: The total weight (across all processes + all messages in transit + the controller) is always **exactly 1**.
- So if the controller ends up with all of it back, it means:

- No weight is left anywhere else → No ongoing work → **Safe to terminate**.

Example

Setup:

- 1 **Controlling Agent** (let's call it **C**)
- 2 **Processes** (let's call them **P1** and **P2**)
  Initially:
- C has **weight = 1**
- P1 and P2 are **idle**, weight = 0

Step 1: Controlling agent starts the computation

C splits its weight:

- Keeps **0.4**
- Sends **0.6** to **P1** via `B(0.6)`
- So now:
  - C = 0.4
  - P1 = 0.6 (active)
  - P2 = 0

Step 2: P1 does some work and sends a message to P2

P1 splits its 0.6 weight:

- Keeps **0.3**
- Sends **0.3** to P2 via `B(0.3)`
- Now:
  - P1 = 0.3
  - P2 = 0.3 (active)
  - C = 0.4

Step 3: P2 finishes its work

P2 has weight = 0.3

- Sends a control message `C(0.3)` to C

- Becomes idle (weight = 0)
  Now:
- C = 0.4 + 0.3 = 0.7
- P1 = 0.3
- P2 = 0

Step 4: P1 also finishes

P1 has weight = 0.3

- Sends `C(0.3)` to C
- Becomes idle
  Now:
- C = 0.7 + 0.3 = **1**
- P1 = 0
- P2 = 0

The controlling agent has **all the weight (1)** again, and everyone else is idle. So it **safely concludes** that the computation is complete.

8. Illustrate the working of spanning tree-based termination detection algorithm.

Basic Idea:

- There's a **root process** (P0) and **child processes** forming a tree.
- Processes do some work and communicate.
- When they finish, they **send tokens** back to the root to indicate they're done.
- **Black = sent a message**, **White = did not send any message**.
- If a process is done and hasn’t sent messages, it sends a **white token**.
- If it sent messages, it sends a **black token**.
- **If the root gets only white tokens and is also idle**, then the whole system is done.

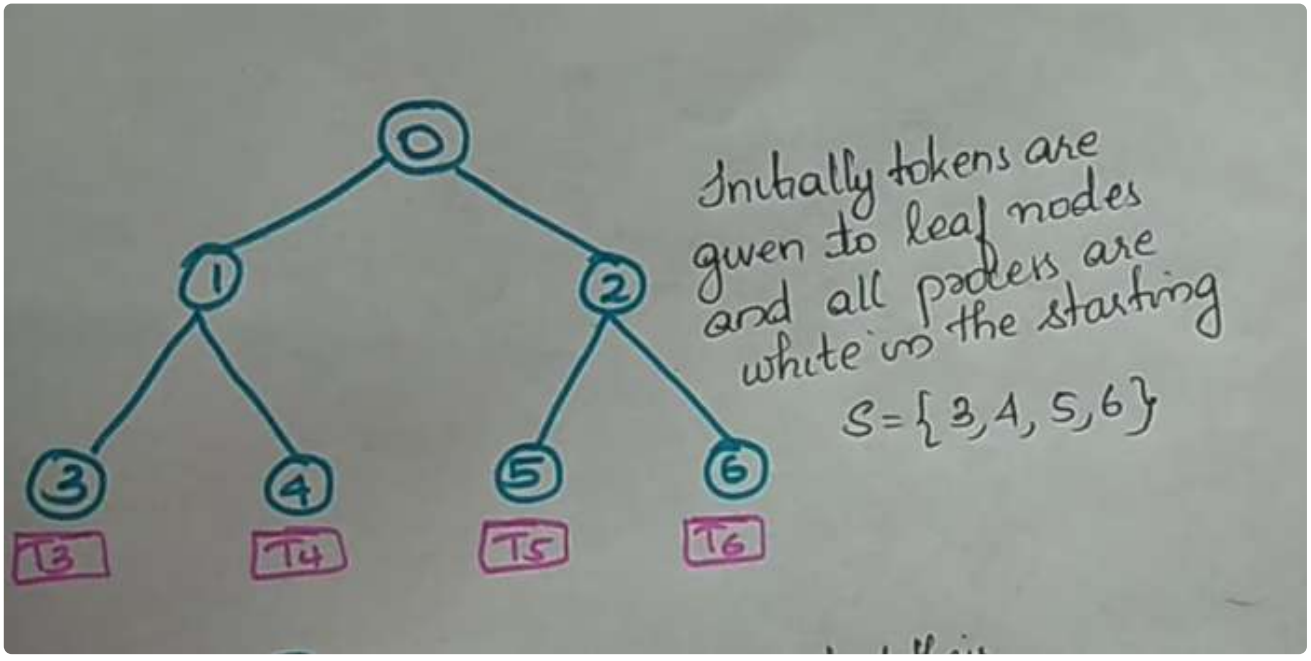
Rules Recap in Simple Terms:

1. **Leaf processes** start with tokens.

2. Everyone is **white** at the start (no messages sent).
3. If a process **sends a message**, it turns **black**.
4. When a process finishes:
    - If it's **white**, it sends a **white token** to parent.
    - If it's **black**, it sends a **black token**.
5. Parent waits for tokens from all its children.
6. If a parent gets even **one black token**, it must send **black token to its parent** (because someone in its subtree sent a message).
7. If root gets a **black token**, it knows someone was still active → sends **Repeat** signal to restart the check.
8. Root concludes **termination** if:
    - It is **idle**.
    - It is **white**.
    - It has received **white tokens from all its children**.

Example of the algorithm

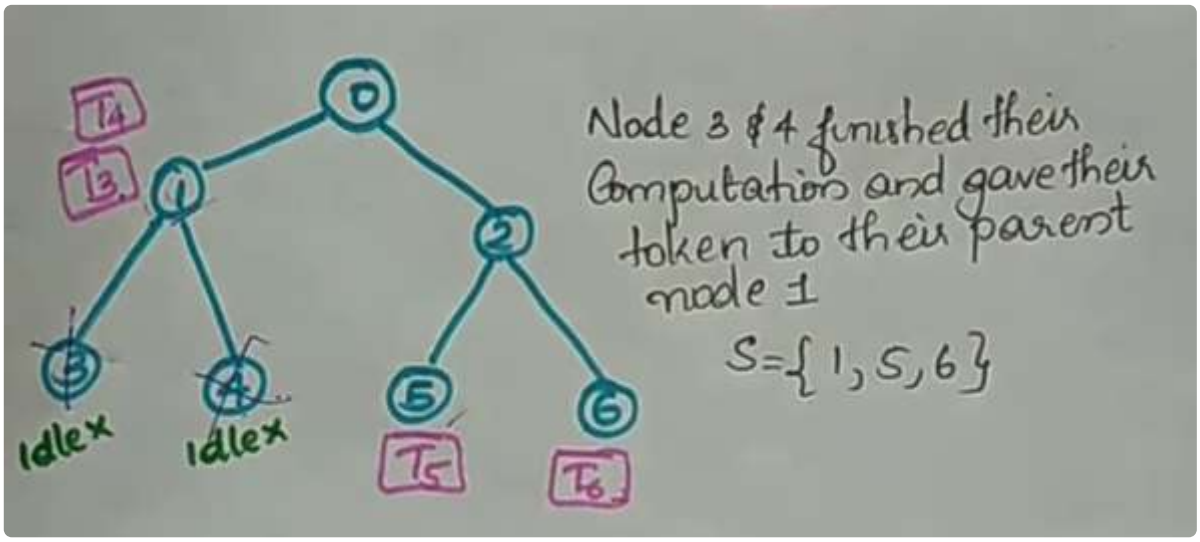
Step 1

- 

- Initially the tokens are given to leaf node
- All processes are white
- We also maintain a set called S = {3,4,5,6} which contains the leaf nodes
- Leaf nodes are: 3,4,5,6

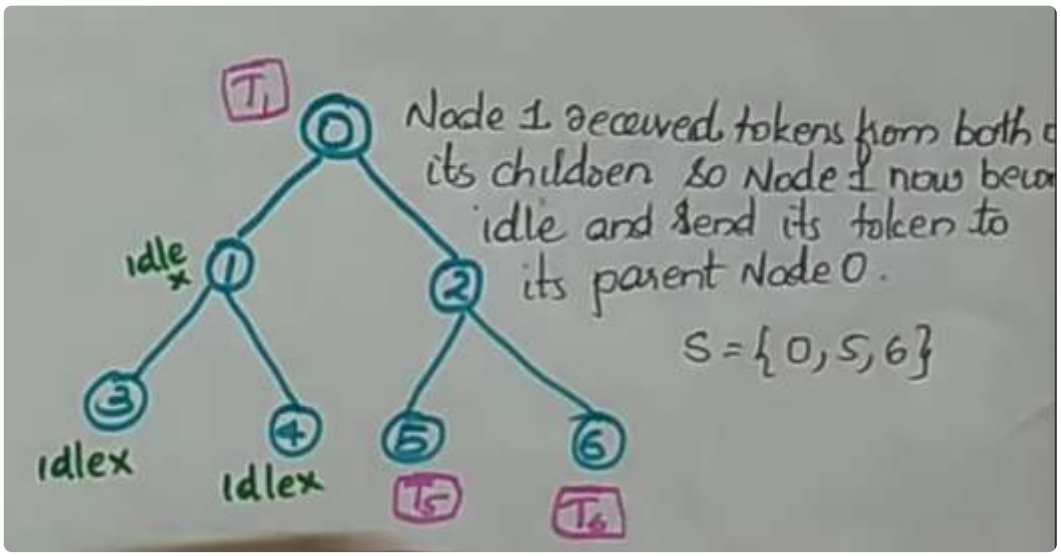
- Tokens are t3,t4,t5,t6

Step 2



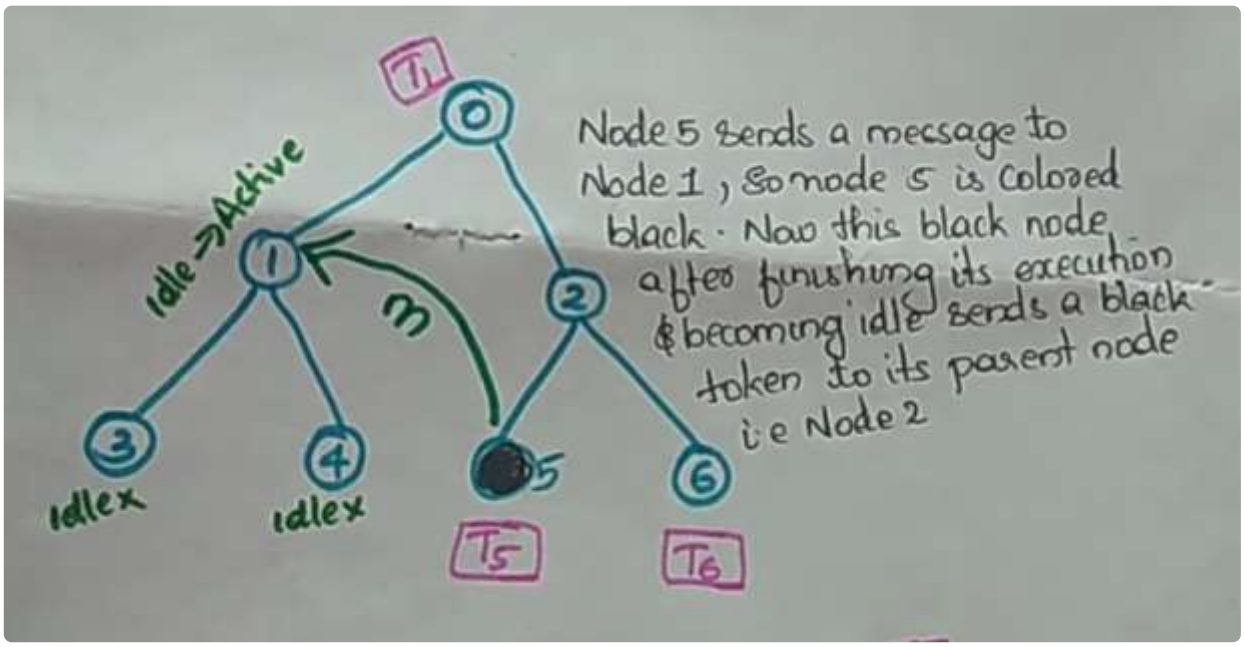
- Now Assume that Node 3 and 4 finished their computation and gave their token to their parent node 1
- 3 and 4 now become idle
- Now the set is S = {1,5,6}

Step 3



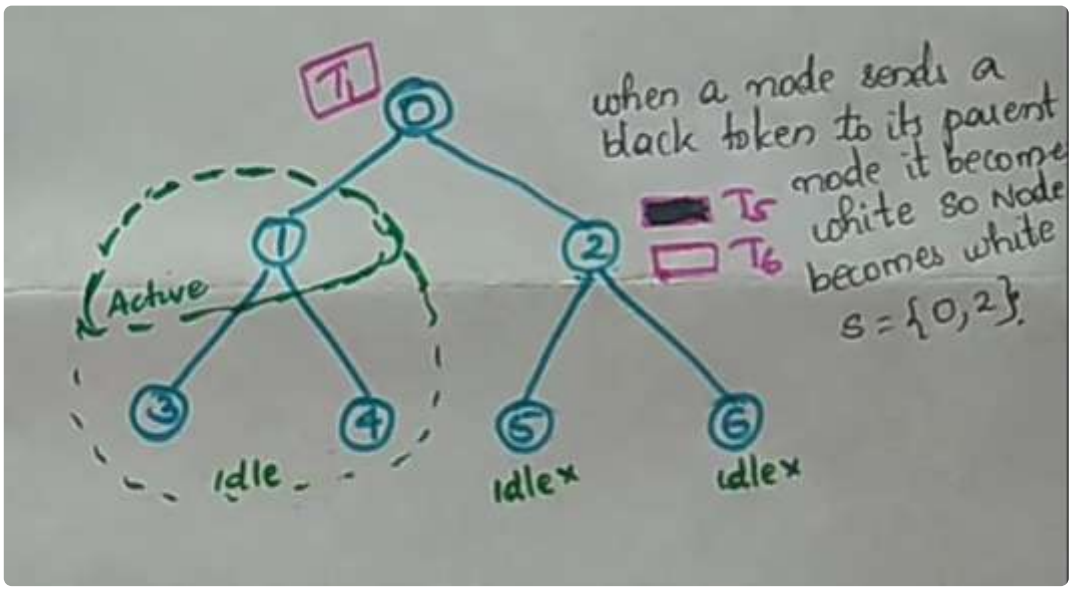
- Node 1 received tokens from both the children
- Now node 1 is marked as idle
- Now the token is passed to 0
- S = {0,5,6}

Step 4

- 

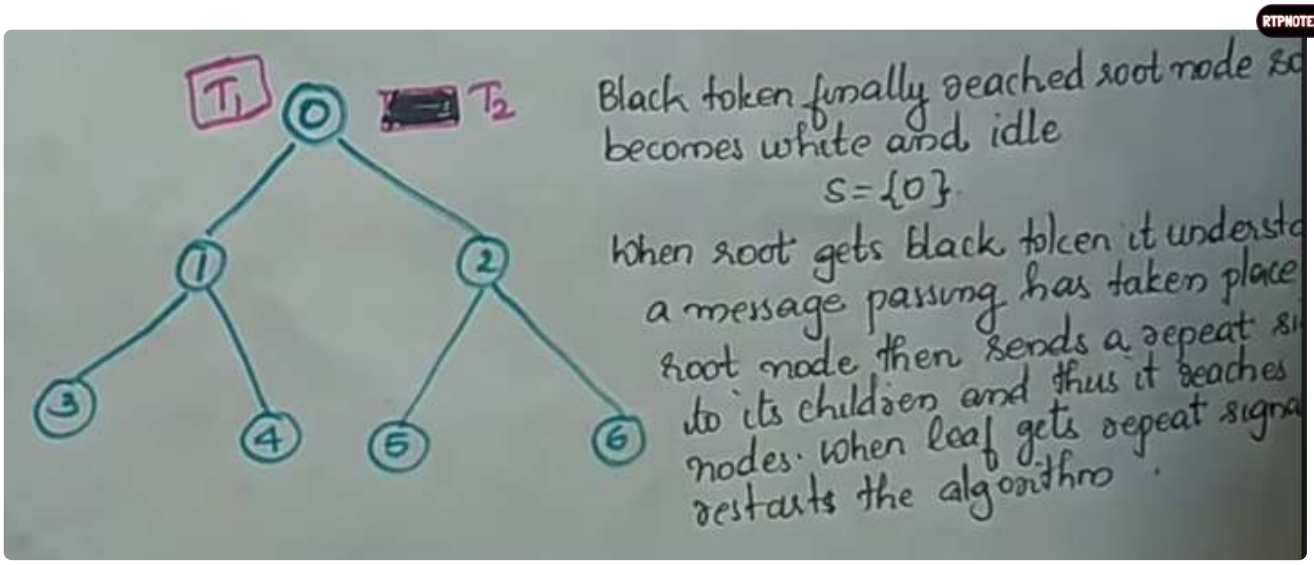
- Now suppose, node 5 sends a message to node 1
- Node 5 is colored black
- Since node 1 is receiving the message, it changes from idle to active state (But 0 doesnt know it)
- When node 5 finishes execution, it will send a black token to parent(Node 2)

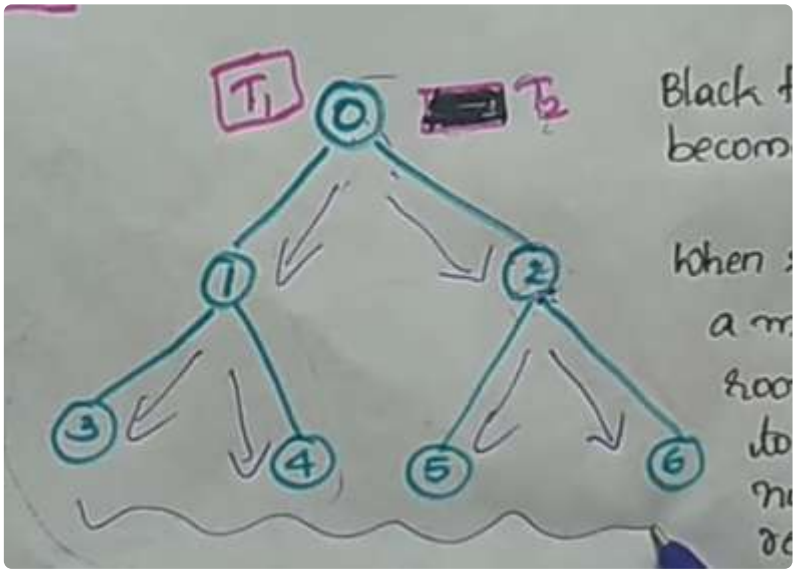
Step 5

- 

- When a node sends a black token to its parent node, it becomes white so node becomes white
- S = {0,2}

Step 6

- 

- Suppose the black token now reaches the root node and Node 2 is now idle
- S = {0}
- When black token reaches the root, it knows that message passing has been done
  - Root will now give a repeat signal
  - 

  - The signal will reach leaf nodes
- When it reaches the leaf, It will restart the algorithm.
- This process repeats until **black token no longer exists in root node**

Time complexitiy

- Best Case O(N)
- Worst case O(N * M)
  - N = No of processes
  - M = No of messages exchanged

9. Explain in detail about Chandy Lamport algorithm.

The **Chandy-Lamport algorithm** is a method to capture a **consistent global snapshot** of a distributed system. It ensures that all processes and communication channels are recorded in a way that reflects the true state of the system, even though processes run independently and messages arrive at different times.

How It Works (Step-by-Step):

1. **Starting the Snapshot (Marker Sending Rule):**
    - A process (let's say **Process A**) decides to start the snapshot.
    - **Process A** immediately:
        1. **Records its own state** (its local memory, tasks, etc.).
        2. **Sends a special control message** called a **marker** through all its outgoing channels *before* sending any other messages.
2. **Receiving the Marker (Marker Receiving Rule):**
    - When another process (say **Process B**) receives this **marker**:
        - **If Process B hasn't recorded its state yet:**
            1. It **records the incoming channel as empty** (no messages counted).
            2. It then **records its own state**.
            3. Sends markers along all its outgoing channels (just like Process A did).
        - **If Process B *has* already recorded its state:**
            - It records the messages received on that channel **between** recording its state and receiving the marker.
3. **Ending the Snapshot:**
    - The algorithm ends when **all processes have received a marker on every incoming channel**.
    - Once this happens, the system has enough information to assemble a **global snapshot**.

Example Scenario:

1. **Process A** starts the snapshot:
    - Records its own state.

- Sends markers to **Process B** and **Process C**.

2. **Process B** receives the marker:
   - It hasn't recorded its state yet, so:
     - It records its state.
     - Marks the channel from **A to B** as empty.
     - Sends markers to its outgoing channels.
3. **Process C** receives the marker *after* it had already recorded its state:
   - It records any messages it received between its snapshot and the marker.

When is This Useful?

- **Detecting Deadlocks:** Check if processes are stuck waiting on each other.
- **Failure Recovery:** Save system states (checkpoints) to recover after crashes.
- **Debugging & Analysis:** Understand how processes interact in real-time.

In simple terms, the Chandy-Lamport algorithm is like taking a group photo where everyone pauses for a second so the picture accurately reflects who was doing what, even though they usually work independently.

10. What is meant by a consistent global state?

A **consistent global state** means:

- If a message has been *received*, the snapshot should also show that it was *sent*.
- No message should appear as being received before it was sent (that wouldn't make sense!).

11. Explain ring based election algorithm with an example. Does the algorithm meet liveness and safety conditions?

Imagine computers arranged in a **circle (ring)**. Each computer can only talk to its **next neighbor** in the circle.

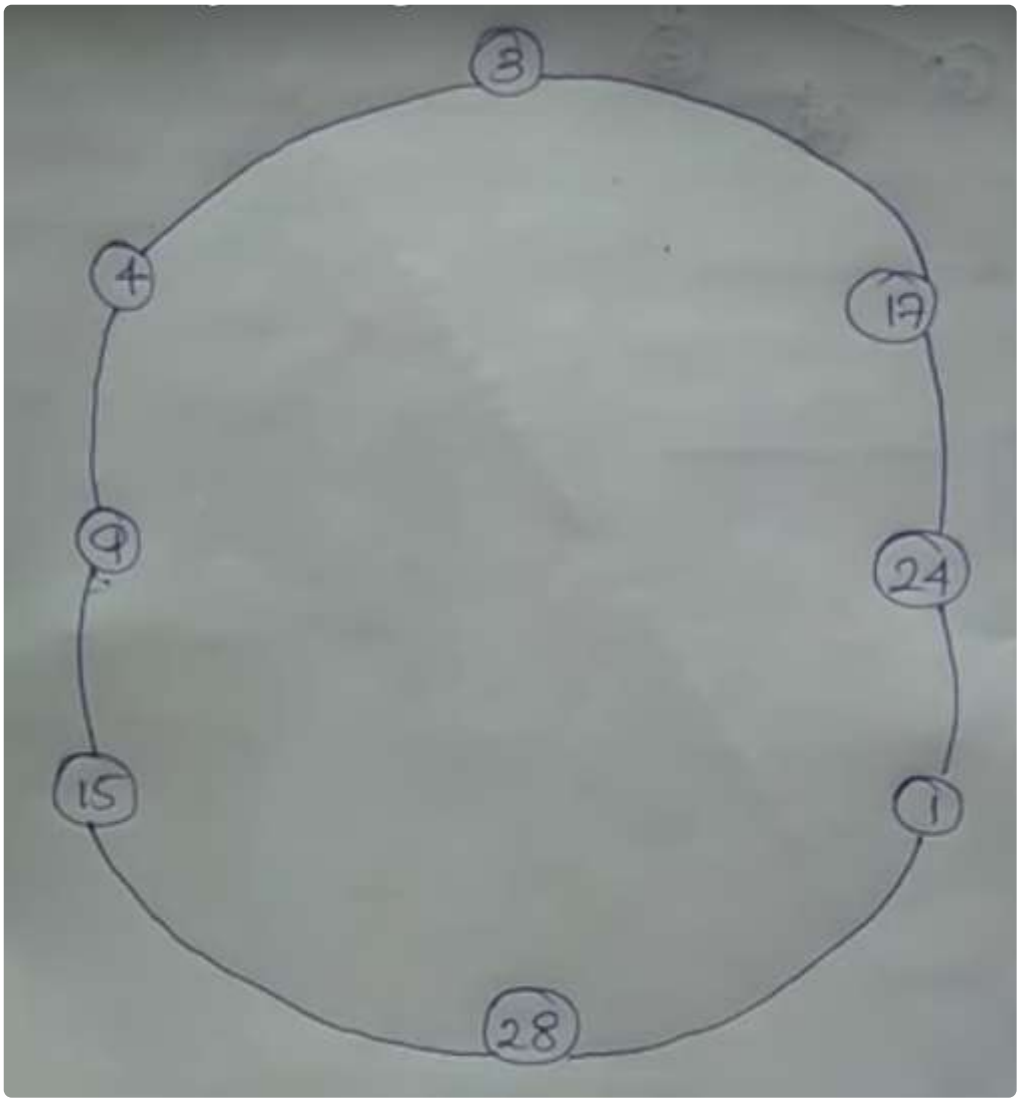
How It Works:

1. **Starting the Election:**
   - Any process can start the election by **sending its ID** to its neighbor.
   - It marks itself as a **participant**.
2. **Passing the Message:**
   - When a process gets an election message, it compares the ID in the message with its own:
     - **If the incoming ID is bigger**: Forward the message.
     - **If the incoming ID is smaller** and the receiver is **not a participant**: Replace the message with its **own ID** and forward it.
     - **If already a participant**, it just ignores smaller IDs.
3. **Choosing the Leader:**
   - When a process receives a message with **its own ID**, it means **it has the highest ID**.
   - This process **becomes the leader** (coordinator).
4. **Announcing the Leader:**
   - The new leader sends an **"elected" message** around the ring to inform everyone.

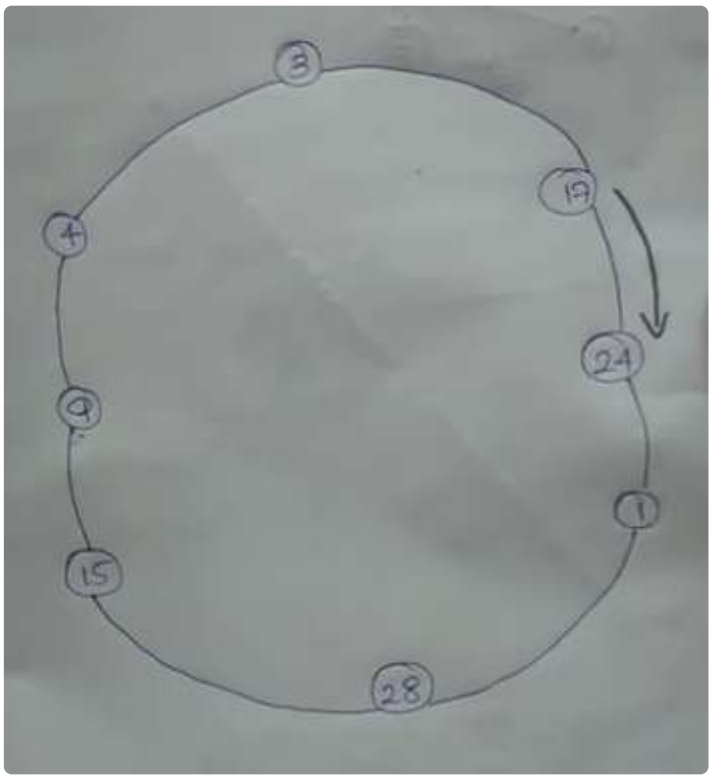
Example

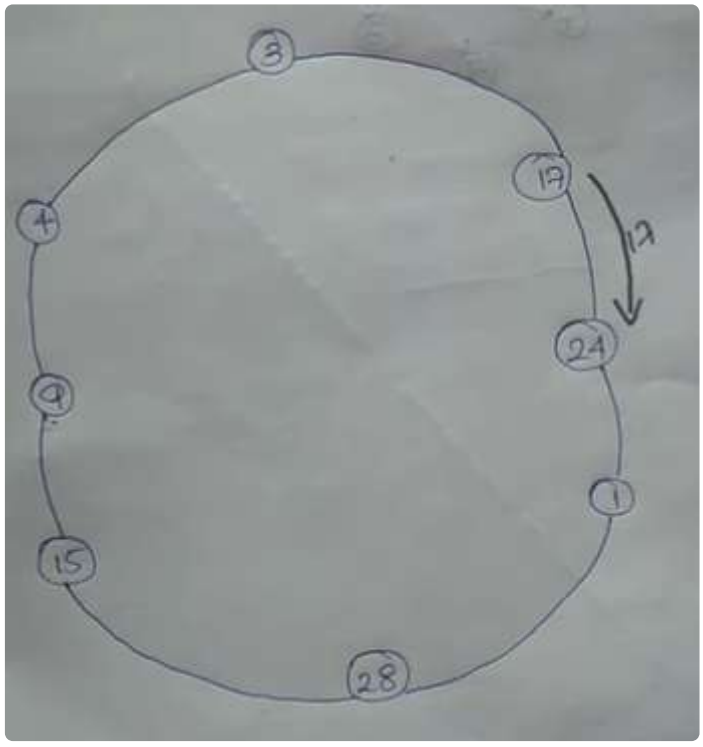
Consider a ring of numbers (8 numbers)

3->17->24->1->28->15->9->4

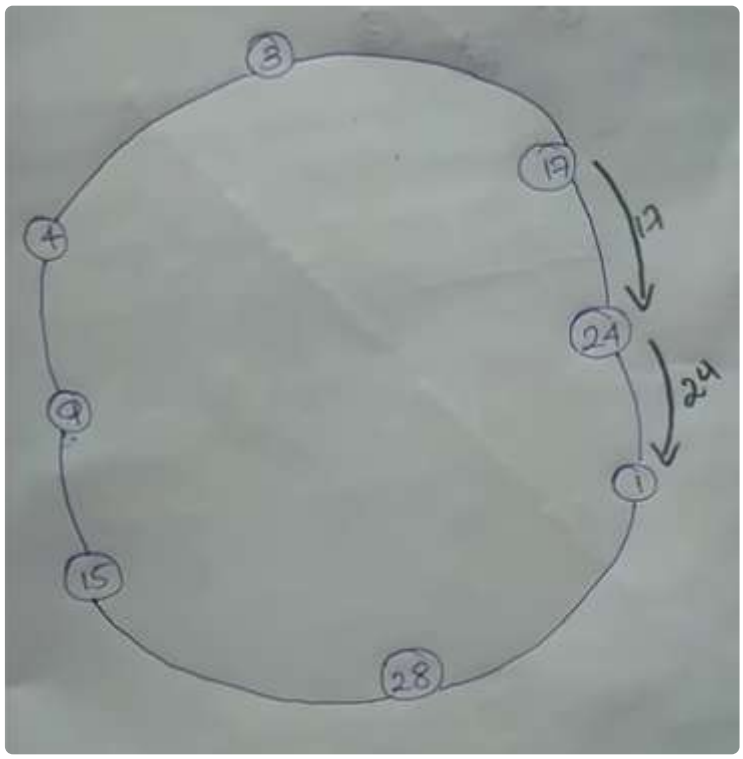


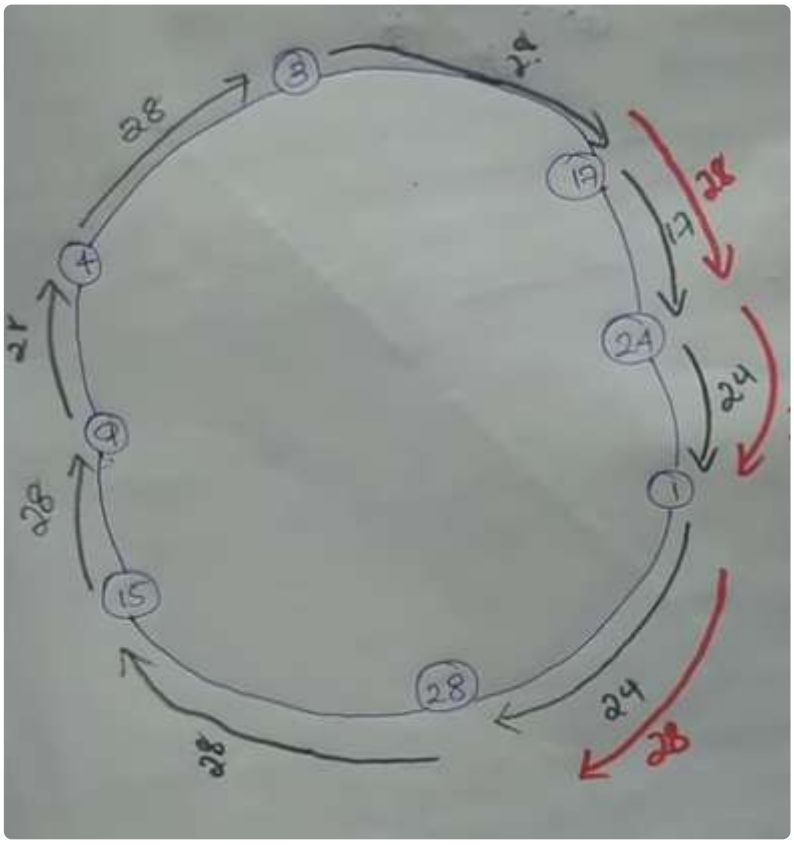
- Suppose 17 notices the leader is down and starts the election
- **17 sends an election message to 24**

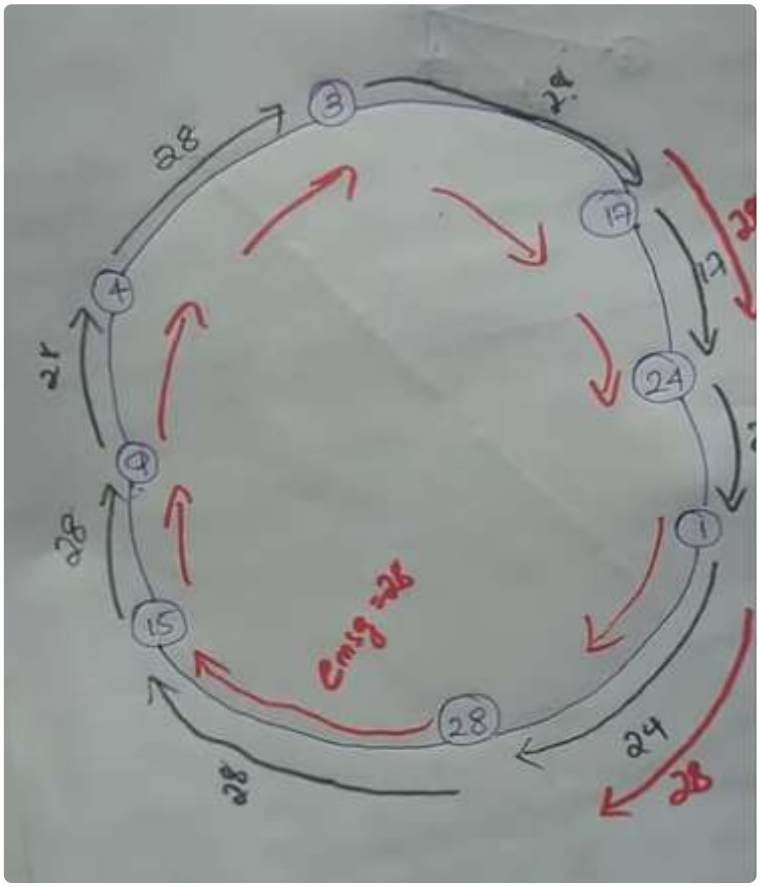
- 

- Here the identifier will be `17` so that will be also sent
  - 

- **The election message reaches 24**
  - Now we compare `24` and the number inside the election message (which is `17`)
  - The greater number is `24`
  - So we use 24 for the next election message

- 

- **The election message reaches 1**
  - We compare `1` and the number inside the election message (`24`)
  - The greater number is `24`
- **Similarly we can repeat until identifier and the election message number matches**
  - 

    - This we repeat until the process number (Identifier) and the message inside the election message matches
    - This happens at `28`
        - The process is `28`
        - and the incoming election message (Marked in red) is also `28`
    - We will stop the election in such condition
- **Start sending coordinator messages**
    - The election stopped at `28`
    - So this means `28` is the coordinator (leader)
    - This `28` will now send coordinator messages with content as `28` to everyone (The red arrows marked inside)
    - 

Liveness and safety conditions

Safety Condition

- **Definition**: *Nothing bad happens — e.g., no two coordinators at the same time.*

- In the ring algorithm:
  - **Only the process with the highest ID** is selected as the coordinator.
  - The election result is **consistent** across the system.
  - Even if multiple elections are started, **only one coordinator** will be chosen once the elections finish.
- **Conclusion**: **Ring-Based Election satisfies safety** — at most one correct leader is elected.

Liveness Condition

Definition: *Something good eventually happens — the system eventually chooses a coordinator.*
In this algorithm:

- If at least one non-failed process starts the election,
- And messages eventually get delivered (reliable channels),
- Then the election **always completes**, and a coordinator is elected.
- If the ring is broken (e.g., some processes crash and cut off the ring), or message loss is permanent, then **liveness can break**.
- But under the **assumption of reliable delivery and non-permanent failures**, it **does** meet liveness.
- **Conclusion**: **Ring-Based Election satisfies liveness** under standard distributed systems assumptions.

12. Explain how logical clock is implemented.

In distributed systems, **logical clocks** are used to **track the order of events** across different processes — even when there's **no synchronized global clock**.

1. Data Structures:

Each process has:

- **Local Logical Clock** (lc_i) → keeps track of that process's internal time.
- **Global Logical Clock** (gc_i) → process p_i 's view of the overall logical time in the system.
  - lc_i is part of gc_i .

2. Protocol (Rules) for Updating the Clock:

The protocol ensures that all clocks progress in a **consistent** and **causally correct** way.

R1 – Local Event Rule

When a process performs an event (send, receive, or internal), it updates its local clock.
How?

- Simply increment lc_i by 1 for every event.
- Example:
  - If $lc_i = 5$ and it sends a message or does some internal operation, then → $lc_i = 6$.

R2 – Message Rule (Global View Update)

When a process receives a message, it updates its clock based on the timestamp attached to the message.

How?

- The sender **attaches its clock value** with the message.
- The receiver sets:
  $lc_i = max(lc_i, \text{received timestamp}) + 1$
  This ensures **causality is preserved** — the receiver's clock is always ahead of any message it receives.
  Example:
- If P1 sends a message with $timestamp = 10$ and P2 has $lc = 8$,
  P2 sets $lc = max(8, 10) + 1 = 11$.

13. Clearly mentioning assumptions, explain the rules of termination detection using distributed snapshots.

Assumptions

- **Stable Property**:
  - Termination is a *stable property*, meaning once it becomes true (computation terminated), it stays true forever.

    - A **consistent snapshot** can capture stable properties like termination.
- **Communication Assumptions**:
    - Logical **bidirectional communication channels** exist between all pairs of processes.
    - Channels are **reliable** (no messages lost) but **non-FIFO** (message order may be arbitrary).
    - **Message delay is arbitrary but finite**.

Rules of Termination Detection (Using Distributed Snapshots)

Rule 1: Unique Last Idle Process

- When a distributed computation terminates, **there is one process** that becomes **idle last**.
- This process takes the **initiative** to check for termination.

Rule 2: Snapshot Request Trigger

- When any process becomes **idle**, it:
    - Sends a **snapshot request** to **all other processes**.
    - Also triggers a **snapshot of itself**.

Rule 3: Granting the Snapshot Request

- When a process receives a snapshot request:
    - It checks: *"Did the requester become idle before I did?"*
    - If **yes**, it **grants the request** by taking a **local snapshot** for that request.

Rule 4: Successful Snapshot

- A snapshot request is called **successful** if:
    - **All processes** agree (and take local snapshots) for that request.
- This means a **consistent global snapshot** has been formed for that specific request.

Rule 5: Termination Detected via Snapshot

- The **requester or an external agent** collects all the local snapshots.
- If the **collected global snapshot** shows:
    - All processes are **idle**, and
    - **No messages** are in transit,

- Then, it confirms that the **termination has occurred**.