Distributed-Computing-Series-1-Important-Topics

- 1. Ring-Based Election Algorithm
  - How It Works:
  - Example (Ring-Based):
- 2. Bully Algorithm
  - How It Works:
  - Example (Bully Algorithm):
- Key Differences Between Ring-Based and Bully Algorithm:

- 5. Primitives for distributed communication
  - How Send() and Receive() Work:
  - Buffered vs. Unbuffered Communication:
  - Types of Communication Primitives
    - 1. Synchronous vs. Asynchronous Communication
    - 2. Blocking vs. Non-Blocking Communication
  - How Non-Blocking Communication Works (Handles & Waits)
  - Example Scenarios
- 6. Past and future cones of events.
  - 1. Past Cone of an Event
    - Key Points:
    - Example:
    - Visualizing the Past Cone:
  - 2. Future Cone of an Event
    - Key Points:
    - Example:
    - Visualizing the Future Cone:
- 7. Design issues of distributed system
  - 1. Communication
  - 2. Managing Processes
  - 3. Naming Things
  - 4. Synchronization (Keeping Things in Sync)
  - 5. Storing and Accessing Data
  - 6. Consistency and Replication
  - 7. Handling Failures (Fault Tolerance)
  - 8. Security

1. Difference between distributed systems and distributed programming.

1. Distributed Systems

Definition:
A **Distributed System** is like a group of independent computers working together to appear as *one* single system to the user. Think of it as different stores in a chain acting like one big supermarket online. Even though each store (computer) operates on its own, when you shop, it feels like you're using a single, unified service.

Key Features:

- **No Shared Memory:** Each computer (or *node*) has its own memory. They don't directly share information but talk to each other by sending messages.
- **No Common Clock:** There's no universal clock keeping time for all the computers, meaning they operate at their own pace.
- **Geographical Spread:** These systems can be spread out globally (like Google's servers worldwide) or locally (like a cluster of servers in a data center).
- **Autonomy & Diversity:** Each computer can run different software, have different speeds, and even be used for different purposes, but they all collaborate.

Why Use Distributed Systems?

- **Resource Sharing:** Share data and tools that are too big or expensive to replicate everywhere.
- **Reliability:** If one computer fails, others can keep things running.
- **Scalability:** Easily add more computers to handle more work.
- **Remote Access:** Get data from faraway places, like accessing a cloud server.

2. Distributed Programming: Writing Code for Distributed Systems

Definition:
Distributed Programming is the art of writing code that runs on multiple computers in a distributed system. It's about making sure these computers can *talk* to each other correctly and *work together* efficiently.

Imagine this:
You're organizing a group project. Each team member (computer) works on a different part of the project (task). Distributed programming is like setting up the communication—emails, messages, deadlines—so everyone knows what to do and when.

Key Concepts in Distributed Programming:

- **Message Passing:** Computers don't share memory directly; instead, they send *messages* to share information.
- **Asynchronous Execution:** Computers don't wait around for each other. They keep working independently and process messages when they arrive.

- **Events & State:** Each computer goes through *events* like sending, receiving, or processing data, which changes its internal state.

Challenges in Distributed Programming:

- **Coordination:** Ensuring computers are in sync when needed, like agreeing on the correct data (e.g., banking transactions).
- **Handling Failures:** What if one computer crashes? The program should still work or recover gracefully.
- **Latency & Timing:** Messages might get delayed or arrive out of order, so your program needs to handle that.

Aspect	Distributed Systems	Distributed Programming
What is it?	A group of computers working together to act like one big system.	Writing code that helps those computers talk to each other and work together.
Focus	How computers are connected and how they work as a team.	How to write programs that run on multiple computers at the same time.
Communication	Computers send messages to each other because they don't share memory.	The code handles sending and receiving messages between computers.
Memory	Each computer has its own memory. No sharing.	Programs must send data through messages since memory isn't shared.
Time	No shared clock. Computers don't run in perfect sync.	Code needs to manage delays and make sure events happen in the right order.
Scaling Up	You can easily add more computers to the system.	The code should handle adding or removing computers without breaking.
Handling Failures	If one computer fails, the system keeps running using other computers.	Code must handle failures, like retrying if a message doesn't get through.
Examples	Google Search servers, Netflix's streaming system, cloud storage like Google Drive.	Writing a chat app where messages are sent between servers, or coding for distributed databases.
Goal	To make many computers act like one powerful, reliable system.	To create programs that can run across many computers and keep everything working smoothly.

*2. Transparency requirements***

In distributed systems, **transparency** means hiding the complexity of the system from users and developers. Even though multiple computers are working together behind the scenes, it should feel like you're interacting with a single, simple system.

1. Access Transparency

- **What it means:** Users shouldn’t have to worry about *how* or *where* they access resources.
- **Example:** Whether you open a file from your computer or from Google Drive, it feels the same—you just click and open it.

2. Location Transparency

- **What it means:** You don’t need to know *where* a resource or service is physically located.
- **Example:** When you visit a website, you don’t know (or care) which data center the server is in; the website just works.

3. Concurrency Transparency

- **What it means:** Multiple users can use the system at the same time without interfering with each other.
- **Example:** On Amazon, thousands of people can buy things at once, and no one's orders get mixed up.

4. Replication Transparency

- **What it means:** The system might have multiple copies (replicas) of data to improve speed or reliability, but you only see one version.
- **Example:** When you watch a YouTube video, it might come from a server near you, but you don’t notice—it’s seamless.

5. Failure Transparency

- **What it means:** If a part of the system crashes or fails, you shouldn’t notice any disruption.
- **Example:** If one of Netflix’s servers goes down while you're watching a show, the system switches to another server without interrupting your stream.

6. Mobility Transparency

- **What it means:** You can move around and still access the system as if nothing changed.
- **Example:** Using WhatsApp on your phone while traveling—you still get your messages no matter where you are.

7. Performance Transparency

- **What it means:** The system automatically adjusts to provide the best performance, and you don’t need to manage it.
- **Example:** Google Search feels fast even when millions of people are searching at the same time because it balances the load across servers.

8. Scaling Transparency

- **What it means:** The system can grow (add more resources) or shrink without affecting how it works for users.
- **Example:** Adding more servers to a cloud service like Dropbox doesn’t change how you upload files.

3. Causal precedence relation

Imagine you and your friends are texting in a group chat. Sometimes, messages depend on each other (like when someone replies to a question), and sometimes they don’t (like when two people talk about different things at the same time). **Causal precedence** helps us understand **which events (or messages) are connected** and **which ones are independent**.

What is Causal Precedence?

Causal precedence tells us **which events depend on each other** in a distributed system (like in a group chat with multiple people sending messages).

- If **Event A** happens and **causes Event B**, we say **A → B** (A happens before B).
- If **Event A** and **Event B** have **nothing to do with each other**, they are **concurrent** (they happen separately).

Simple Example:

- **You (Person 1)**: "Hey, what's up?" (**Event A**)
- **Your Friend (Person 2)**: "Not much, you?" (**Event B**)

Here, **Event B** depends on **Event A** because your friend is replying to you. So, we say:

- **A → B** (A happens before B because B is a reply to A)

Now imagine:

- **You (Person 1)**: "Hey, what's up?" (**Event A**)
- **Another Friend (Person 3)** at the same time: "Anyone watched the game last night?" (**Event C**)

These two events have **nothing to do with each other**, so they are **concurrent** (happen independently).

How Does It Work in Distributed Systems?

In distributed systems, computers send messages to each other just like people do in a group chat. These messages (or **events**) can be **connected** or **independent**.

1. **Same Process (Like talking to yourself):**
   - If **Event A** happens before **Event B** on the same computer, we say **A → B**.
2. **Message Between Computers (Like texting a friend):**
   - If **Computer 1** sends a message (**Event A**), and **Computer 2** receives it (**Event B**), then **A → B**.
3. **Chain of Events:**
   - If **A → B** and **B → C**, then **A → C**.
     (If A causes B, and B causes C, then A causes C.)

Logical vs. Physical Concurrency

1. **Logical Concurrency (No Connection):**
   - Two events are **logically concurrent** if **they don’t affect each other**.
   - Example: You send a text, and your friend posts on Instagram at the same time. These actions don’t affect each other.
2. **Physical Concurrency (Same Time in Real Life):**
   - Two events happen **at exactly the same time** in real life.

- Example: You and your friend both press "send" on a message at the exact same second.

Why Does This Matter?

In distributed systems (like cloud servers, online games, or databases), **knowing which events depend on each other** is super important. It helps:

- **Keep data consistent** (no mix-ups in messages or transactions)
- **Avoid errors** when multiple users are doing things at the same time
- **Understand the flow of information** in the system

Quick Recap:

- **A → B** means **A happened before and influenced B**.
- Events with **no connection** are **concurrent**.
- **Logical concurrency** means events don’t affect each other, even if they happen at different times.
- **Physical concurrency** means events happen at the exact same time in real life.

4. Leader election algorithm

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**.

Why Do We Need a Leader Election Algorithm?

- To **choose a unique leader** (only one at a time).
- If the leader **fails or retires**, the system runs the algorithm again to pick a new one.
- **All computers must agree** on who the leader is.

How Does It Work?

- Any computer (called a **process**) can start an election if it thinks the leader is down.
- A process is either:

- **Participant**: Actively involved in the election.
- **Non-participant**: Not involved in the current election.

Types of Leader Election Algorithms

1. **Ring-Based Election Algorithm**
2. **Bully Algorithm**

1. Ring-Based Election Algorithm

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 (Ring-Based):

- Computers in the ring: **3 → 7 → 9 → 4 → 6**

1. **Computer 3** starts the election and sends "3" to **7**.

2. **7** compares IDs. Since **7 > 3**, it forwards "7" to **9**.
3. **9 > 7**, so it forwards "9" to **4**.
4. **4 < 9**, so **4** does nothing.
5. When **9** gets the message back with its **own ID**, it knows **it's the leader**!

2. Bully 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**.

Example (Bully Algorithm):

- Computers with IDs: **1, 2, 3, 4, 5**

1. **Computer 3** notices the leader is down and sends **election messages to 4 and 5**.
2. **Computer 5** responds (because it has a higher ID) and **starts its own election**.
3. Since **5** has the highest ID, **it wins** and sends a **"coordinator" message** to all others.
4. **If 5 crashes**, another process (say **3** or **4**) will start the election again.

Key Differences Between Ring-Based and Bully Algorithm:

Feature	Ring-Based Algorithm	Bully Algorithm
Structure	Processes are arranged in a **ring**.	Any process can talk to any other directly.
Message Direction	Messages move **clockwise** around the ring.	Messages are sent to **higher-ID** processes.
Leader Selection	The process with the **highest ID** in the ring wins.	The process with the **highest ID** always wins (hence "bully").
Message Overhead	**More messages** as they go through every process in the ring.	**Fewer messages** since only higher processes are contacted.
Failure Handling	If a process fails, the ring keeps working, but slower.	If the highest process fails, a **new election** starts quickly.

5. Primitives for distributed communication

In a distributed system (where multiple computers or processes work together), **communication primitives** are the basic building blocks that allow these processes to **send and receive messages**. The two main primitives are:

- `Send()` : Used to **send** data to another process.
- `Receive()` : Used to **receive** data from another process.

How Send() and Receive() Work:

- **Send(destination, data)**:
  - `destination` : Who you are sending the data to.
  - `data` : The actual message or information you want to send.
- **Receive(source, buffer)**:
  - `source` : Who you are expecting data from (can be anyone or a specific process).
  - `buffer` : The space where the received data will be stored.

Buffered vs. Unbuffered Communication:

1. **Buffered Communication:**
   - Data is first copied from the **user's buffer** to a **temporary system buffer** before being sent over the network.
   - Safer because if the receiver isn't ready, the data is still stored temporarily.
2. **Unbuffered Communication:**
   - Data goes **directly** from the **user's buffer to the network**.
   - Faster but riskier—if the receiver isn't ready, the data could be lost.

Types of Communication Primitives

Communication primitives can be classified based on how they handle **synchronization** and **blocking**.

1. Synchronous vs. Asynchronous Communication

- **Synchronous Primitives**:
  - The **sender and receiver must "handshake"**—both must be ready for the message to be sent and received.
  - The `Send()` only finishes when the `Receive()` is also called and completed.
  - Good for ensuring messages are properly received but can slow things down.
- **Asynchronous Primitives**:
  - The `Send()` returns control immediately **after copying the data out of the user buffer**, even if the receiver hasn't received it yet.
  - **Receiver doesn't need to be ready immediately**.
  - Faster, but there's a risk the message might not be delivered right away.

2. Blocking vs. Non-Blocking Communication

- **Blocking Primitives**:
  - The process **waits (or blocks)** until the operation (sending or receiving) is fully done.
  - Example: In a **blocking** `Send()`, the process won't continue until it knows the data has been sent.
- **Non-Blocking Primitives**:
  - The process **immediately continues** after starting the send or receive operation, even if it's not finished.

- It **gets a handle (like a ticket)** that it can use later to check if the message was successfully sent or received.
- Useful for **doing other work while waiting** for communication to finish.

How Non-Blocking Communication Works (Handles & Waits)

When you use **non-blocking communication**, the system gives you a **handle** (like a reference number) to check if the operation is complete.

1. **Polling**: You can **keep checking** in a loop to see if the operation is done.
2. **Wait Operation**: You can use a `Wait()` function with the handle, and it will block until the communication is complete.

Example Scenarios

1. **Blocking Synchronous Send Example:**
   - You send a file and **wait** until the receiver confirms they've received it before doing anything else.
2. **Non-Blocking Asynchronous Send Example:**
   - You send an email and **immediately start working on something else**, trusting that the system will handle sending it in the background.
3. **Blocking Receive Example:**
   - You wait by the phone until your friend calls—you won't do anything else until you get the call.
4. **Non-Blocking Receive Example:**
   - You keep your phone nearby while you do other tasks, checking occasionally to see if you've missed a call.

6. Past and future cones of events.

In **distributed systems** and **relativity theory**, the concept of **past** and **future cones** helps explain how events are connected over time and space. Here's a breakdown:

1. Past Cone of an Event

- The **past cone** of an event **ej** includes **all events that could have influenced it**.
- In other words, any event **ei** that happened **before ej** and could have sent information or had an effect on **ej** is part of its **past cone**.

Key Points:

- **Causal Relationship (`ei → ej`)**:
  If `ei → ej`, it means **event ei happened before event ej** and could have influenced it.
- **Information Accessibility**:
  All the information present at **ei** could have potentially been **accessible** to **ej**.

Example:

Imagine you're sending a message in a chat.

- **Event ei**: You **type** the message.
- **Event ej**: Your friend **receives** the message.

Since your friend receiving the message (**ej**) could only happen **after** you typed it (**ei**), **ei** is in the **past cone** of **ej**.

Visualizing the Past Cone:

If you plot events on a **timeline**, the **past cone** of an event looks like this:

```
    |
    |    ej (event)
   / \
  /   \
 ei   ei' (events that could influence ej)
```

Everything inside the **cone** represents events that could have influenced **ej**.

2. Future Cone of an Event

Just like the **past cone** shows events that could have influenced a particular event, the **future cone** represents **all events that could be influenced** by a given event.

- The **future cone** of an event **ei** includes **all events that can be affected by it**.
- In other words, if **event ei** happens, any **event ej** that occurs **after** it and can be influenced by **ei** belongs to the **future cone** of **ei**.

Key Points:

- **Causal Relationship (`ei → ej`)**:
  If `ei → ej`, it means **event ej** happens after **ei** and could be influenced by **ei**.
- **Information Flow**:
  All the information available at **ei** could potentially **reach ej**.

Example:

Imagine you're in a virtual meeting.

- **Event ei**: You **speak** into your microphone.
- **Event ej**: Other participants **hear** your voice.

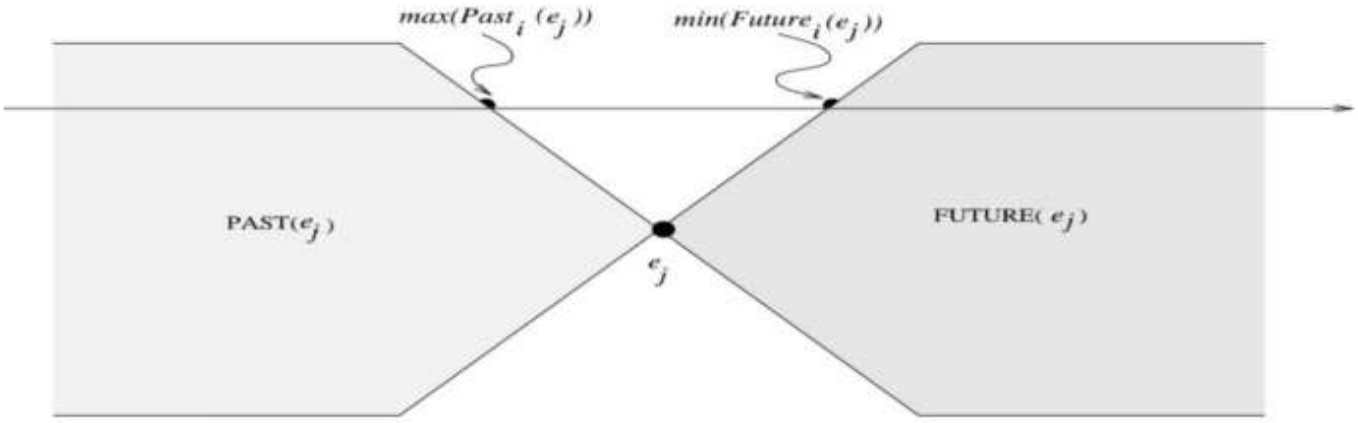
Since your speaking (**ei**) can directly influence others hearing you (**ej**), **ej** is in the **future cone** of **ei**.

Visualizing the Future Cone:

When plotted on a timeline, the **future cone** looks like this:

```
    ei (event)
    / \
   /   \
ej    ej' (events influenced by ei)
   |
```

Everything inside the **cone** represents events that **can** be affected by **ei**.



7. Design issues of distributed system

When building a distributed system (a system where different computers work together over a network), you face several important challenges.

1. Communication

- **What's the problem?**
  Computers in different places need to talk to each other clearly and quickly.
- **What to think about:**
  - **Remote Procedure Call (RPC):** Calling a function on another computer as if it's on your own.
  - **Remote Object Invocation (ROI):** Using objects from another computer like they're local.
  - **Messages vs. Streams:** Should you send single messages or a continuous flow of data?

2. Managing Processes

- **What's the problem?**
  You need to handle running programs (processes) across many machines.
- **What to think about:**
  - **Starting and Stopping Processes:** How do you manage programs running on different computers?

- **Moving Code Around:** Sometimes it’s better to move the program to where the data is.
- **Smart Agents:** Programs that can move and act on their own across different systems.

3. Naming Things

- **What’s the problem?**
  You need a simple way to find computers, files, or services on the network.
- **What to think about:**
  - **User-Friendly Names:** Easy-to-use names instead of complicated addresses.
  - **Moving Devices:** How to keep track of mobile devices that change location.

4. Synchronization (Keeping Things in Sync)

- **What’s the problem?**
  Making sure computers work together in a coordinated way.
- **What to think about:**
  - **Preventing Conflicts:** Stopping two computers from accessing the same resource at the same time.
  - **Choosing a Leader:** Picking one computer to be in charge when needed.
  - **Matching Clocks:** Making sure all computers agree on the time.

5. Storing and Accessing Data

- **What’s the problem?**
  You need to store data in a way that’s fast and easy to access from anywhere.
- **What to think about:**
  - **Distributed File Systems:** Storing files across multiple machines.
  - **Fast Access:** Making sure data can be accessed quickly, even as the system grows.

6. Consistency and Replication

- **What’s the problem?**
  When you copy data to multiple places (replication), you need to keep it consistent.
- **What to think about:**
  - **Keeping Data in Sync:** Making sure all copies of the data are up-to-date.

- **When to Update:** Do you need instant updates or is it okay if they happen later?

7. Handling Failures (Fault Tolerance)

- **What's the problem?**
  Systems need to keep working even when something goes wrong.
- **What to think about:**
  - **Reliable Messaging:** Making sure messages don't get lost.
  - **Recovering from Crashes:** Saving progress so you can pick up where you left off if a computer fails.
  - **Detecting Failures:** Knowing when a computer or connection goes down.

8. Security

- **What's the problem?**
  Keeping data safe from hackers and unauthorized access.
- **What to think about:**
  - **Encryption:** Protecting data by turning it into unreadable code unless you have the key.
  - **Access Control:** Making sure only the right people or systems can access resources.
  - **Secure Connections:** Ensuring data sent over the network is safe from spying.

9. Scalability and Modularity

- **What's the problem?**
  The system should handle more users and data as it grows.
- **What to think about:**
  - **Spreading Workload:** Distribute tasks across multiple machines.
  - **Using Caches:** Storing frequently accessed data temporarily to speed things up.
  - **Breaking Things into Parts:** Designing the system in small, manageable pieces that can be updated separately.

8. Models of communication network

When computers in a distributed system talk to each other, they send messages over networks. The way these messages are sent and received can follow different models:

1. **FIFO (First-In, First-Out)**
   - **What it means:** Messages are delivered in the same order they were sent.
   - **Example:** Imagine you're standing in a line at a coffee shop. The first person to order is the first to get their coffee.
   - **Why it’s useful:** It’s predictable—messages don’t get mixed up.
2. **Non-FIFO (Non-First-In, First-Out)**
   - **What it means:** Messages can arrive in any order, not necessarily the order they were sent.
   - **Example:** Think of tossing several letters into a mailbox. When the mailman delivers them, they might come out in a different order than you sent them.
   - **Why it’s useful:** It can be faster in some situations, but it’s harder to manage since the order isn’t guaranteed.
3. **Causal Ordering (CO)**
   - **What it means:** Messages are delivered in an order that respects cause-and-effect relationships. If one message depends on another, it will be delivered afterward.
   - **Example:** If you ask a question in an email and someone replies, causal ordering makes sure you receive the reply *after* your question.
   - **Why it’s useful:** It helps keep the logic of conversations or processes intact, which simplifies how distributed systems work.

How These Models Relate to Each Other:

- **Causal Ordering ⊂ FIFO ⊂ Non-FIFO**
  - **Causal Ordering** is the strictest—it always respects cause and effect.
  - **FIFO** ensures the order is correct but doesn’t always track cause and effect.
  - **Non-FIFO** is the most flexible—messages can arrive in any order.

9. Global state and snapshot recording algorithm

In distributed systems, multiple computers (or processes) work together but don’t share memory or a common clock. This makes it tricky to understand the **whole system's current**

state at any given time. That’s where **global state** and **snapshot recording** come in.

What is a Global State?

- **Global State** = The combined information about what’s happening in *all* processes and communication channels in the system at a specific time.
  - **Local State:** Each process (computer) has its own local state, which includes its memory, tasks it's working on, and the messages it has sent/received.
  - **Channel State:** Each communication channel (the connection between processes) has its state, which includes messages that have been sent but not yet received.

Why Record the Global State?

Recording the global state is important for:

1. **Detecting Problems:** Like finding deadlocks (when processes are stuck waiting for each other) or checking if tasks have finished.
2. **Failure Recovery:** Saving the system’s state (called a **checkpoint**) helps restore it after a crash.
3. **System Analysis:** Understanding how the system behaves for testing and verifying correctness.

How Does Snapshot Recording Work?

To get a **meaningful global snapshot**, processes need to record their local states in a way that makes sense together. Here’s how it's modeled:

1. **Processes and Channels:**
   - **Processes (p1, p2, ... pn):** These are the computers or tasks in the system.
   - **Channels (Cij):** The connections where messages are passed from process **pi** to process **pj**.
2. **Events in the System:**
   - **Internal Events:** Actions happening inside a process (like updating a variable).
   - **Send Events:** When a process sends a message.
   - **Receive Events:** When a process receives a message.
3. **Recording a Global Snapshot:**

- The goal is to capture the states of all processes and channels **consistently**, even though messages are flying around at different times.

What is 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!).

Challenges in Recording Snapshots:

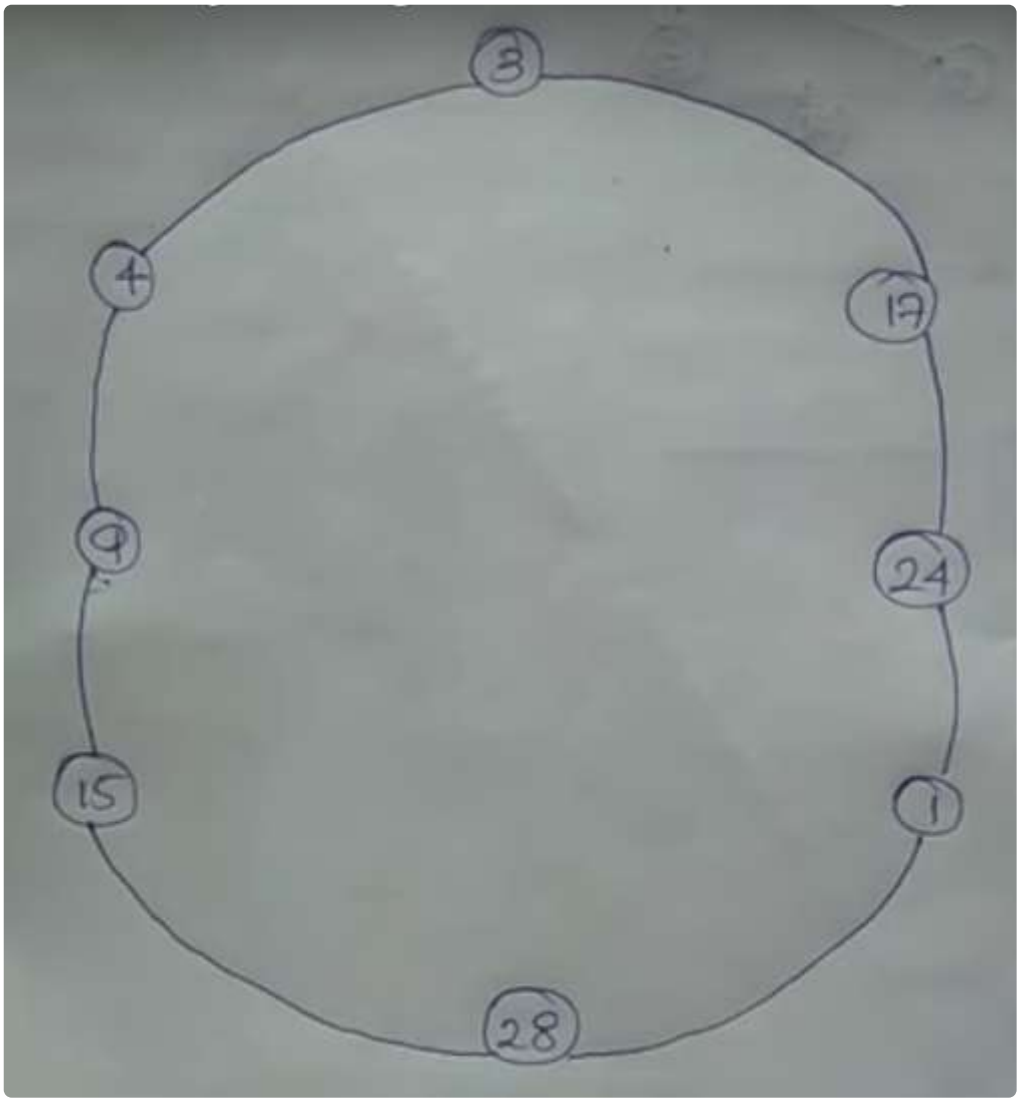
1. **No Global Clock:** You can't tell all processes to take a snapshot at the exact same time.
2. **Message Confusion:** It's hard to decide which messages should be included in the snapshot—messages that are in transit can cause inconsistency.

10.Problem using Leader election algorithm

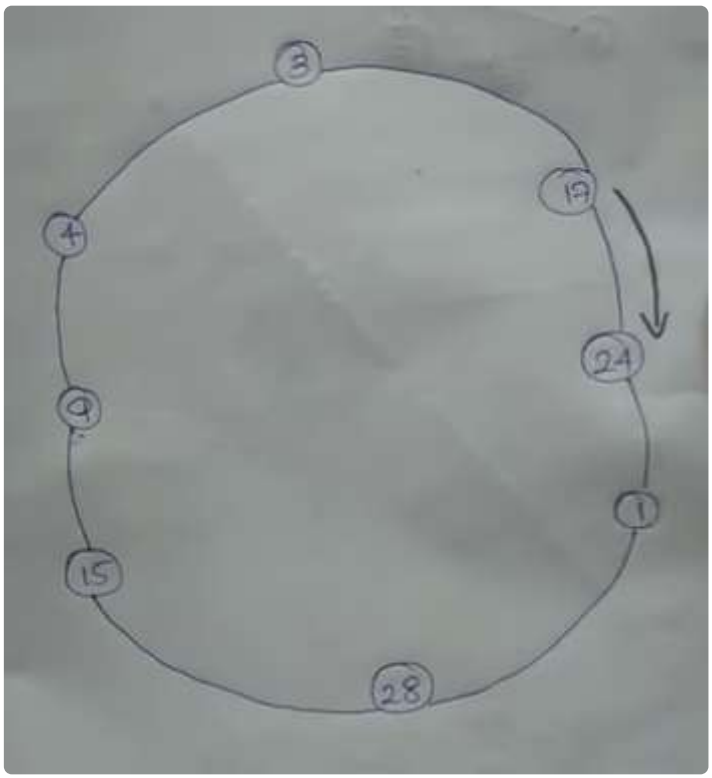
1. Ring Based Election Algorithm Example

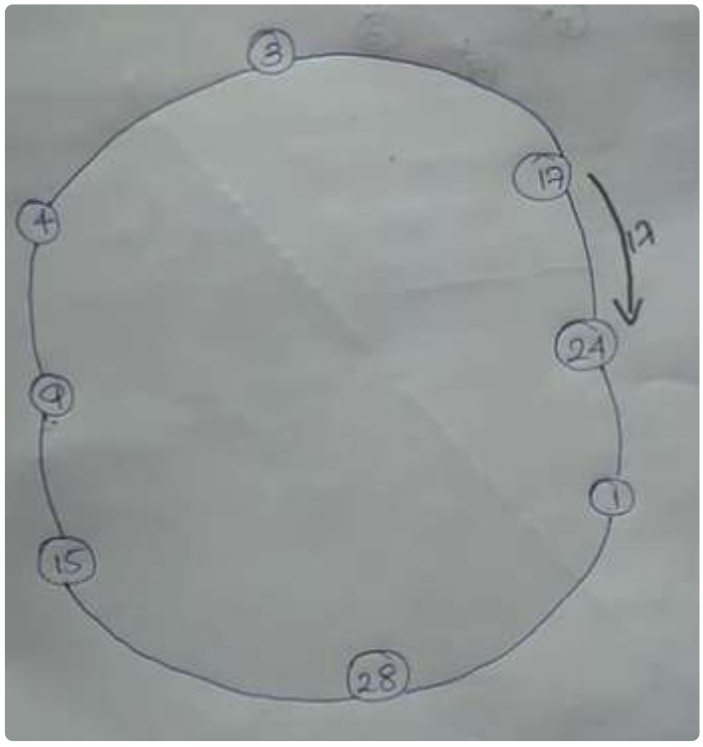
Consider a ring of numbers (8 numbers)

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

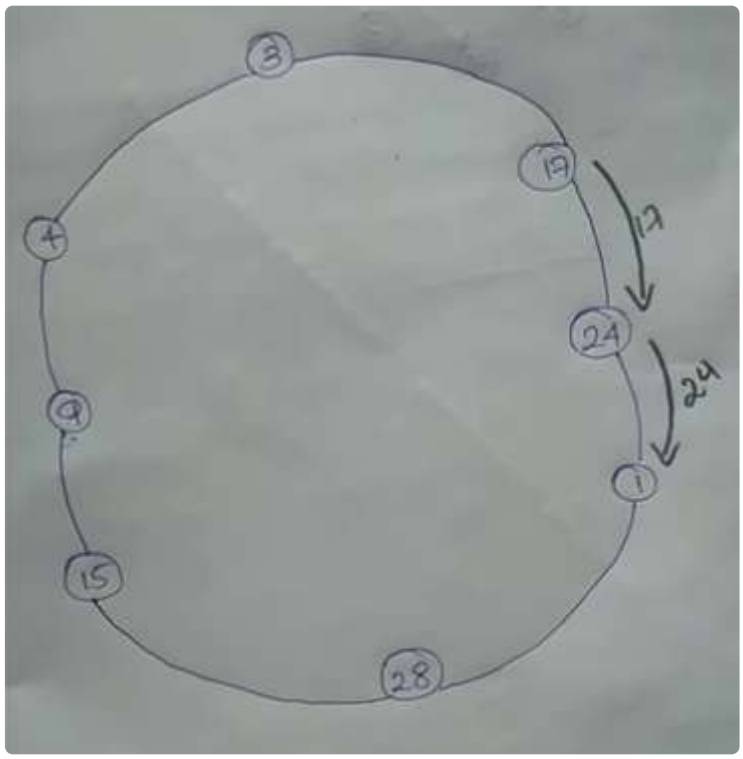


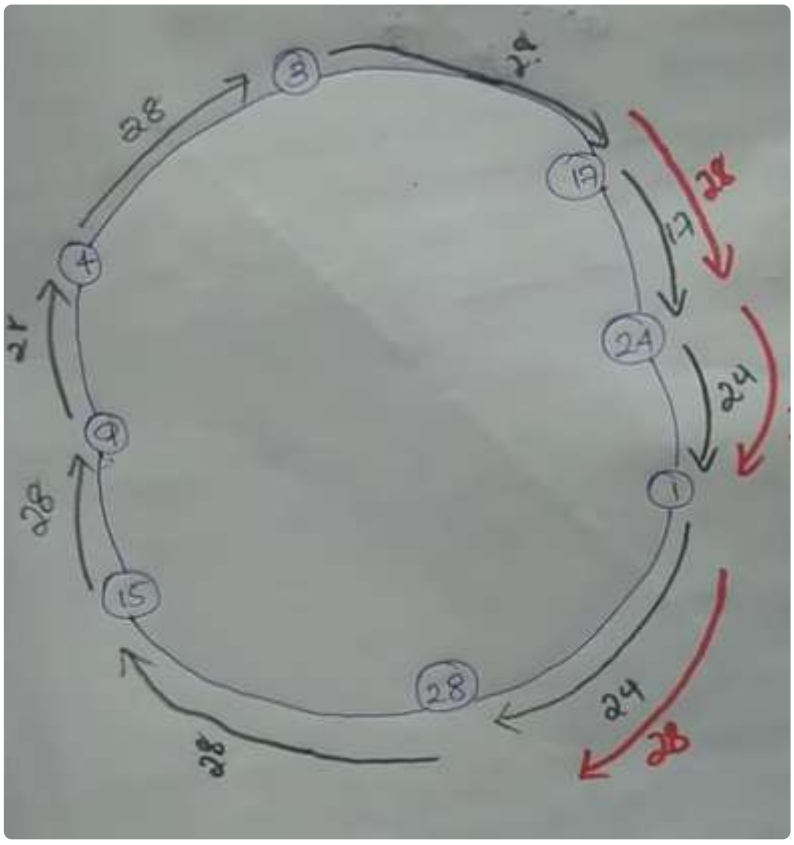
- Suppose 17 notices the leader is down and starts the election
- **17 sends an election message to 24**

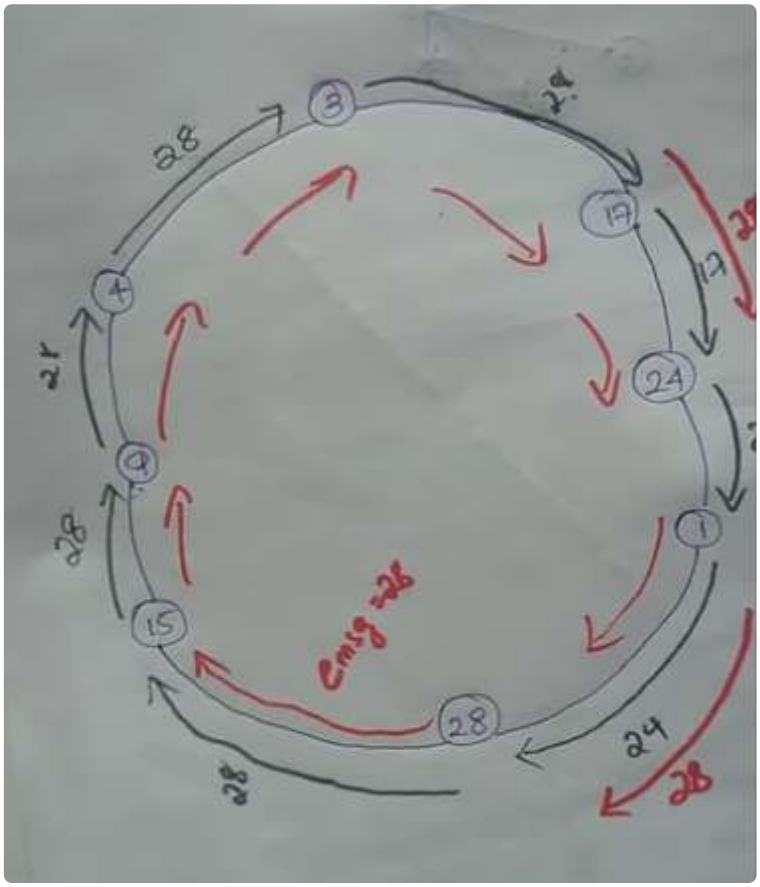
- 

- 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

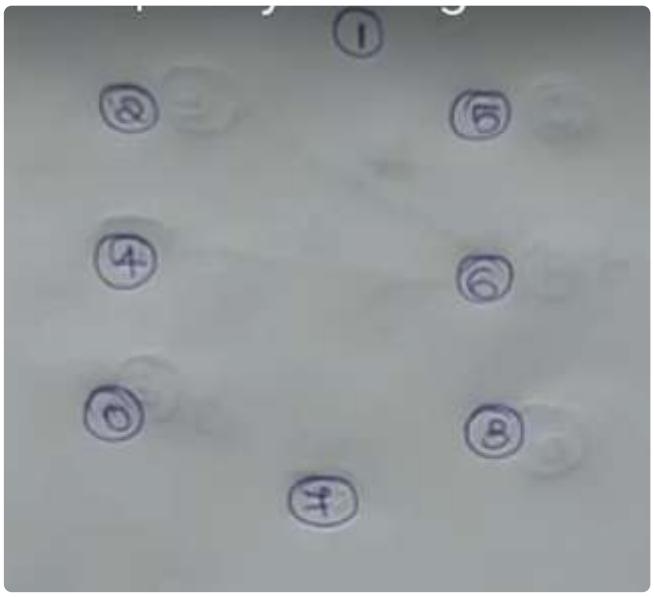
- 

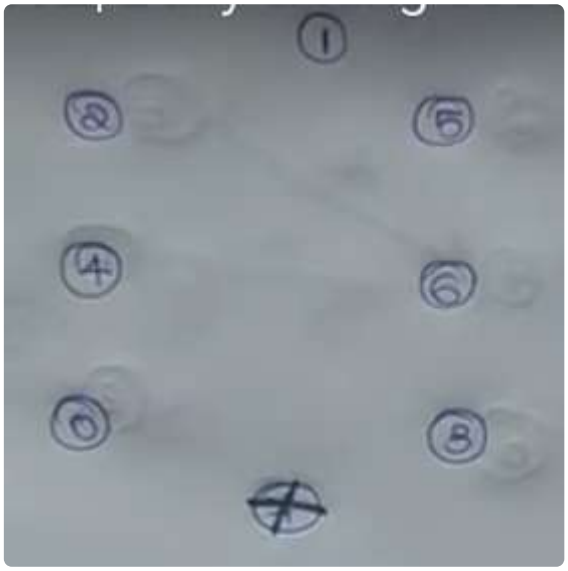
- **The election message reaches 1**
  - We compare `1` and the number inside the election message (`24`)
  - The greater number is `24`
- **Similarly we can repeat until identifier and the election message number matches**
  - 

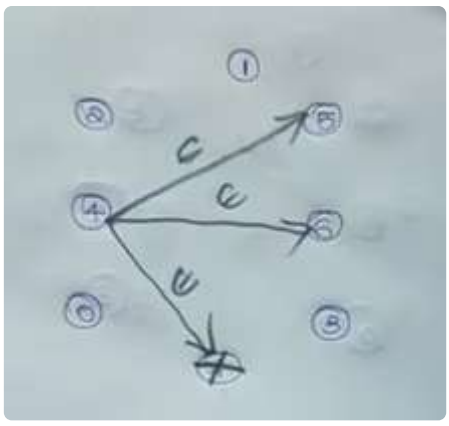
    - This we repeat until the process number (Identifier) and the message inside the election message matches
    - This happens at `28`
        - The process is `28`
        - and the incoming election message (Marked in red) is also `28`
    - We will stop the election in such condition
- **Start sending 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)
    - 

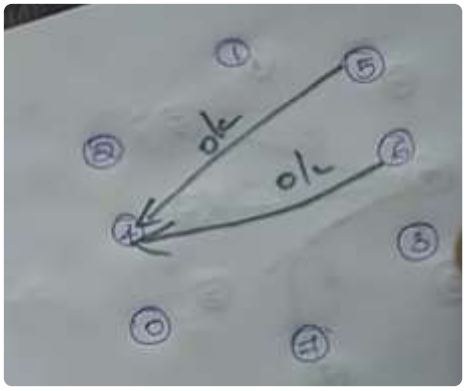
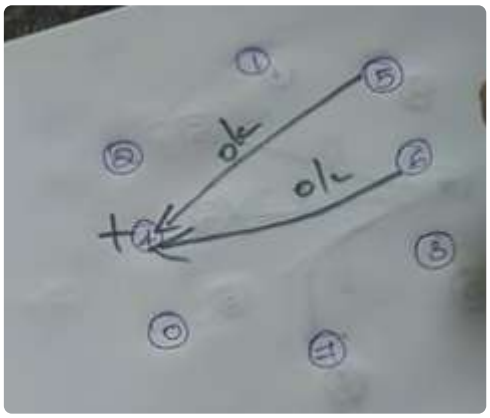
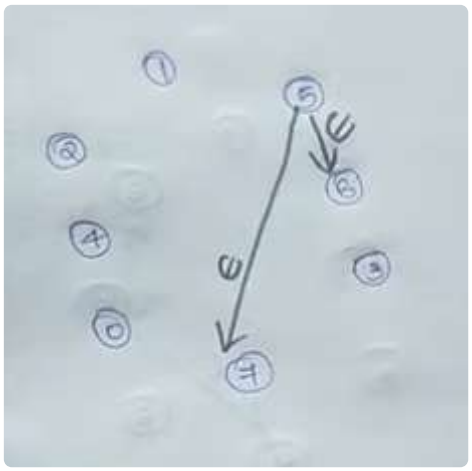
2. Bully Algorithm Example

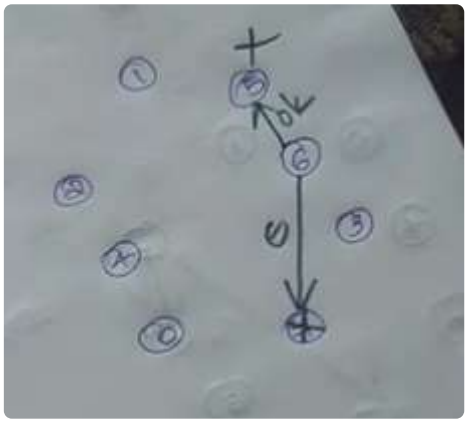
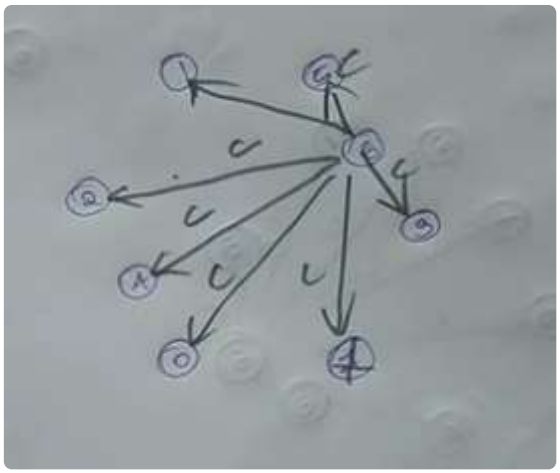
- There are processes 0-7

- 

- 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

11. Chandy Lambore 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.

12. Rules for 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.