Distributed-Computing-Module-4-Important-Topics-PYQs

1. Explain no orphans consistency condition.

Imagine you and your friends are building a LEGO tower together. One friend, let's say **Alex**, adds a **special block** that changes how the tower should look.

But here's the catch:

- The block is **random** — it's different each time.
- So, you must **remember exactly** what block Alex added in case the tower falls and you need to rebuild it.

Now imagine the tower falls (like a system crash), and you're trying to rebuild it:

- But Alex is **gone** (Alex's process crashed),
- And **no one remembers** what block Alex added,
- Yet your part of the tower was built **based on that block**.

Now you're stuck. You can't rebuild your part correctly. You're like an **orphan**—left behind with no way to move forward.

To avoid this problem, there's a simple rule:

If you're going to depend on a random block someone added, then either:

- **They must write it down safely** (like saving it to stable storage), or
- **You must remember it yourself** (store it in your memory).

That way, even if someone disappears, the group can still rebuild the entire tower correctly. **No one is left behind.**

- **Random block** = Non-deterministic event
- **Remembering it** = Logging the event

- **Notebook** = Stable storage
- **Rebuilding** = System recovery
- **Orphan** = A process depending on something we can't replay

No-Orphans Consistency in Distributed Systems

In distributed systems, **non-deterministic events** are unpredictable things like receiving a message or user input. These events must be **logged** so the system can replay them during recovery.

An **orphan process** is:

- A process that **did not crash**,
- But depends on a non-deterministic event,
- And that event's determinant is **lost** (not saved in stable storage or memory of any surviving process).

Such a process can't be recovered properly and ends up in an inconsistent state.
To prevent this, the **No-Orphans Consistency Condition** says:

If a process depends on a non-deterministic event, then the event's determinant must be:

- **Saved in stable storage**, or
- **Available in the memory of a surviving process**.

This ensures that **no process is left with missing information** after a failure, and the system can always recover to a consistent state.

2. List any three advantages of using Distributed Shared Memory.

Imagine DSM as a **giant shared whiteboard** that multiple computers can read and write on **at the same time**. Instead of sending messages back and forth, they can **directly access shared memory**, making everything faster and simpler.

Advantages of DSM

1. Easy Communication Between Computers

✔ Computers can **share data** just by reading and writing in memory.
✔ No need for **complex message passing** between machines.

2. Single Address Space (No Need to Move Data)

✔ All computers **see the same memory** instead of handling multiple copies.
✔ No need to **move data** back and forth between systems.

3. Faster Access Using Locality of Reference

✔ When a system **needs data**, it **keeps it close** for faster access.
✔ Reduces **network traffic** and improves performance.

🛠️ **Example:** If you **frequently use a book**, you keep it **on your desk** instead of going to the library every time.

4. Cost-Effective (Cheaper Than Multiprocessor Systems)

✔ Uses **regular computers** instead of expensive **special hardware**.
✔ Can be built with **off-the-shelf** components.

🛠️ **Example:** Instead of **buying a supercomputer**, DSM allows you to **connect normal computers** to work together.

5. No Single Memory Bottleneck

✔ Traditional systems can get **slowed down** by a **single memory bus**.
✔ DSM **removes this issue** by distributing memory across multiple computers.

🛠️ **Example:** Instead of **one cashier handling all customers**, multiple cashiers **serve different people at once**.

6. Virtually Unlimited Memory

✔ DSM **combines memory** from multiple computers into **one large memory space**.
✔ Programs can run as if they have **huge memory** available.

🛠️ **Example:** Instead of **one water tank**, DSM **connects multiple tanks** to store more water.

7. Portable Across Different Systems

✔ DSM programs work **on different operating systems** without modification.
✔ The interface remains **the same**, making it easy to develop applications.

🛠️ **Example:** Just like **Google Docs works on Windows, Mac, and Mobile**, DSM works across different computers without changes.

3. Differentiate between coordinated checkpointing and uncoordinated checkpointing

A. Uncoordinated Checkpointing (Independent Checkpoints)

- Each process **saves its own state independently**.
- **Problem:** It may lead to a **Domino Effect** (rolling back too far).
- 🛠️ **Example:**
  - If process **P1 rolls back**, but it has sent data to **P2**, then **P2 must also roll back**, and so on. This can cause the system to **restart from a very old state**.

B. Coordinated Checkpointing (Planned Checkpoints)

- All processes **coordinate** and save a **consistent global state**.
- Prevents **the Domino Effect** and ensures all processes are in sync.
- Uses a **coordinator process** to signal all processes to take checkpoints together.
- 🛠️ **Example:**
  - An online shopping system **saves a checkpoint every hour** across all servers. If one server crashes, it restores **all servers to the last saved state** to maintain consistency.

4. List the different types of Messages in Rollback Recovery.

What is Rollback Recovery?

Rollback Recovery is the process of **restoring a system to a previous consistent checkpoint** after a failure.

Steps in Rollback Recovery:

1. **Detect Failure**
   - The system notices that a process has failed.
2. **Restore Checkpoint**
   - The system **reloads the last saved checkpoint** for that process.
3. **Re-execute**
   - The process **continues from where it left off**, avoiding major data loss.

Example:
If an airline booking system crashes **after booking 50 tickets**, it restores the **last saved checkpoint** (e.g., after booking 40 tickets) and **reprocesses the last 10 tickets**.

Message Handling in Rollback Recovery

When recovering from a failure, messages between processes can be affected:

Types of Messages:

1. **In-Transit Messages**
   - These messages were **sent but not yet received** at the time of failure.
   - Think of them as messages "on the way" when the system crashed.
   - They **don't cause inconsistency** and are expected to be delivered eventually.
   - Example: Message `m1` is sent but not yet received.
2. **Lost Messages**
   - These were **sent by a process**, but the **receiving process rolled back** to a state **before receiving them**.
   - The sender remembers the message, but the receiver doesn't — like someone saying something, but you forgot it after fainting.
   - Example: Message `m1` after rollback.
3. **Delayed Messages**
   - These messages were **sent**, but **arrived too late** — either the receiver was down or already rolled back.
   - The message shows up **after** the rollback point, so the system can't use it directly.
   - Example: Messages `m2` and `m5`.
4. **Orphan Messages**
   - These are **received messages** for which the **send event was rolled back**.

- It looks like a message was received **without anyone sending it**, which breaks consistency.
- Example: Message `H` becomes orphan when sender Pi rolls back, but receiver Pj doesn't.

5. **Duplicate Messages**
    - These happen when messages are **logged and then replayed** after rollback, causing the receiver to **receive them again**.
    - Re-sending is fine, but re-receiving without checking can cause **duplicates**.

5. Discuss about the issues in implementing distributed shared memory software.

Imagine a group of computers working together, sharing data like they're using a common notebook. That's what DSM tries to do — it makes different computers feel like they are using one big memory together. But doing this isn't easy. Let's look at the main problems

1. How should sharing work?

When many computers access the same data, we need clear **rules**.
For example:

- If two computers try to change the same data at the same time, **what should happen?**
- Should one wait? Should both be allowed?
- If the rules aren't clear, programs might behave **weirdly or incorrectly**. So the first challenge is just deciding **how sharing will actually work**.

2. How to make sharing happen?

Let's say you and 3 friends are working on a group project. You have **one shared notebook**. If every time someone wants to read or write something, they have to come all the way to *you* to look at the notebook — it's going to be:

- **Very slow**
- **Annoying if you're busy**
- Everyone has to wait in line!

A solution would be

- Make **copies of the notebook** and give one to each friend. That way:
  - Everyone can **read** from their own copy whenever they want.
  - Things become **faster** and more **convenient**.
    That's exactly what **replication (copying)** does in DSM.

But this brings questions like:

- Should we keep a copy on every computer or just a few?
- Should we make copies only when someone wants to **read** the data?
- Or also when they want to **change** it?
  The system has to figure out **what kind of copying** is the best.

3. Where to keep the data copies?

If we're not copying everything everywhere (because that's slow), we need to decide:

- **Which computers should hold the copies?**
  This matters because it can affect **how fast** we get the data.

For example:
If you always use data that's stored far away, everything becomes slow. So we want to keep data **closer to where it's needed most**.

4. How to find the data?

Sometimes, a computer needs data that it doesn't have.
Now the system must **figure out where the data is** and **fetch it**.

If there are multiple copies in different places, it gets tricky to know which one is **correct or up-to-date**.

5. Too much back-and-forth!

If computers keep asking each other for data all the time, it leads to:

- **Slower performance**
- **A lot of messages flying around**

So the system needs to be smart and try to **reduce how much they talk to each other**. It should make sharing smooth, without everyone shouting over the network all the time.

6. Differentiate between deterministic and non-deterministic events in log based rollback recovery.

In **log-based rollback recovery**, **deterministic** and **non-deterministic events** are essential to understanding how processes recover from failures by using logs.

1. Deterministic Events

- **Definition**: A deterministic event is an event whose outcome can be fully predicted or reproduced given the prior state of the system.
- **Characteristics**:
    - Deterministic events do not involve any uncertainty or external factors beyond the initial state.
    - The sequence and result of these events are solely dependent on prior state information, and they are predictable as long as you know the state of the system before the event.
- **Role in Recovery**
    - During recovery, deterministic events can be replayed because they are based on the state and sequence of prior events. They are simply a result of previous deterministic logic.

2. Non-Deterministic Events

- **Definition**: A non-deterministic event is an event whose outcome is not fully determined by the system's prior state alone and may depend on external factors, like inputs from other processes or sources outside the system.
- **Characteristics**:
    - Non-deterministic events can't be predicted or exactly replicated solely from the state of the system before the event occurred.
    - These events introduce uncertainty, and the exact outcome can vary each time it happens, depending on external inputs.
- **Role in Recovery**:

- Non-deterministic events must be logged with their **determinants** (information necessary to replay the event). For example, the arrival of message **m0** may require logging the sender and timestamp to correctly replay the event after a failure.
- During failure recovery, processes rely on the logged determinants of non-deterministic events to correctly **replay** or **reconstruct** the system's state and events that occurred before the failure. The process cannot proceed from a failed state without knowing the exact sequence and conditions of the non-deterministic events.

7. Show that Lamport's Bakery algorithm for shared memory mutual exclusion, satisfy the three requirements of critical section problem.

What is lamports bakery algorithm?

Lamport's Bakery Algorithm is a **mutual exclusion algorithm** designed to solve the **critical section problem** in distributed systems, particularly in systems with shared memory. It works by ensuring that only one process can access the critical section (CS) at a time while respecting the rules of mutual exclusion, bounded waiting, and progress.

The name **"Bakery"** comes from the analogy of a bakery where customers take a number when they enter and are served in the order of the numbers.

Steps in the Bakery Algorithm

In this algorithm, each process follows a set of steps to enter the critical section.

Step 1: Choose a ticket number

- When a process wants to enter the critical section, it picks a **ticket number**.
- This ticket number is assigned based on the maximum ticket number from all other processes, plus one.
- Essentially, the ticket number for a process `i` is the **highest number of tickets currently assigned** + 1.
- **Choosing a ticket number ensures an orderly sequence**, meaning the process with the smallest number gets to enter the CS(Critical Section) first.

Step 2: Wait for your turn

- Once a process has its ticket number, it waits for all other processes with smaller ticket numbers (or those with the same ticket number but a smaller process ID) to finish their work in the critical section.
- If another process has a **smaller ticket number**, it is given priority to enter the CS first.

Step 3: Enter the critical section

- Once a process has the smallest ticket number, it enters the critical section and performs its work.

Step 4: Leave the critical section

- After completing its work in the critical section, the process **resets its ticket number to 0**. This step is critical because it signals that the process is no longer requesting access to the CS.

The three requirements of critical section problem

1. Mutual Exclusion

Mutual exclusion means that **only one process** can be inside the critical section at a time.
How the Bakery Algorithm Ensures Mutual Exclusion:

- Each process **picks a unique ticket number**.
- A process can only enter the critical section once its ticket number is the smallest among all the processes.
- When multiple processes pick the same ticket, the algorithm **uses lexicographic order** (i.e., a combination of ticket number and process ID) to decide which process goes first.
- The algorithm ensures that no two processes can be in the critical section at the same time because one process will always have a smaller ticket number or lexicographically smaller combination of ticket number and process ID.

Steps in the Bakery Algorithm Ensuring Mutual Exclusion:

- A process picks a ticket number greater than the current maximum ticket number.
- If two processes pick the same number, the one with the smaller process ID will proceed first. This guarantees no conflicts in entry to the critical section.

2. Bounded Waiting

Bounded waiting means that **there is a limit on the number of times** other processes can enter the critical section before the requesting process.

How the Bakery Algorithm Ensures Bounded Waiting:

- When a process picks a ticket, it waits until its ticket number is the smallest. If multiple processes have the same ticket, the one with the lower process ID will proceed first.
- **After a process picks a ticket**, the only way a process can get ahead of it is if another process (with a larger ticket number) goes first.
- Since each process can only overtake another process **at most once** (if two processes pick the same ticket number), the waiting time is bounded.

Explanation of Bounded Waiting in Action:

- Suppose process `i` picks a ticket, and process `j` picks the same ticket. The **next time** `j` picks a ticket, its value will definitely be **greater** than `i`'s ticket, thus `i` can eventually enter the critical section.

3. Progress

Progress means that **if no process is in the critical section**, then a process wishing to enter the critical section should be able to do so, provided it follows the rules.

How the Bakery Algorithm Ensures Progress:

- The algorithm guarantees that, at any point in time, the process with the smallest ticket number will eventually enter the critical section.
- This is because the ticket numbers are assigned in a way that no two processes can have conflicting orders.
- **Lexicographic ordering** (first by ticket number and then by process ID) ensures that the process with the smallest ticket or lexicographically smallest combination of ticket number and process ID will always proceed first.

Steps Ensuring Progress:

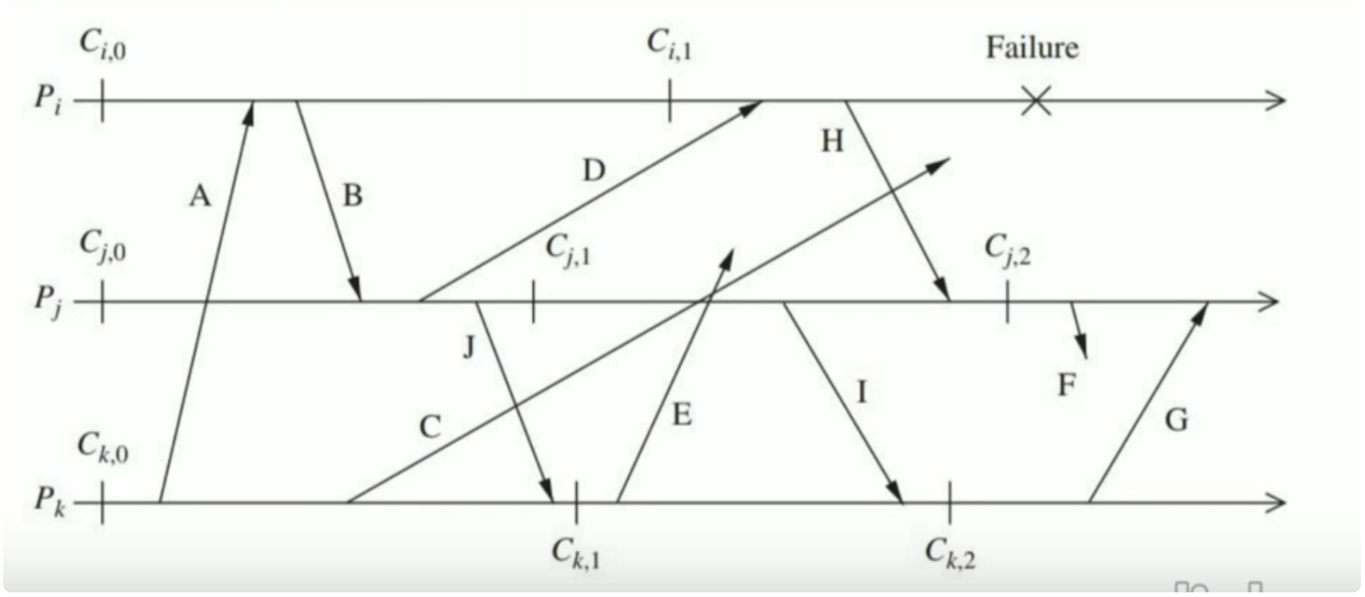
- As soon as a process's ticket number becomes the smallest, it enters the critical section.
- Since the ticket number is **always updated** based on the max ticket number across all processes, the system ensures that **the next process in line** will always be able to proceed to the CS.

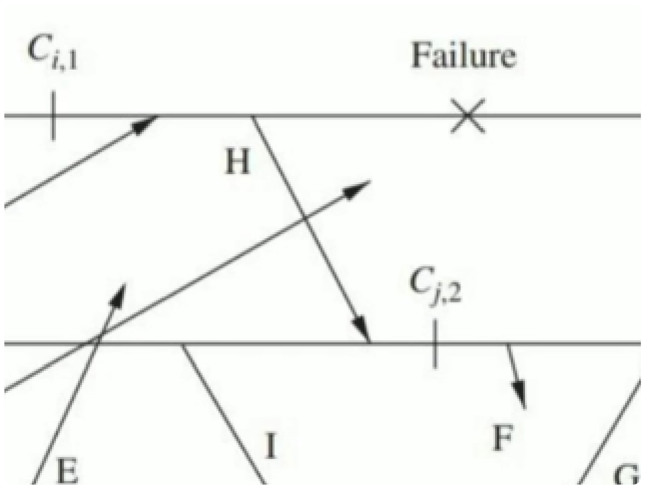
8. What are the issues in failure recovery? Illustrate with suitable examples.

Failure recovery involves restoring the system to a consistent state after a failure. The following issues arise during this process:

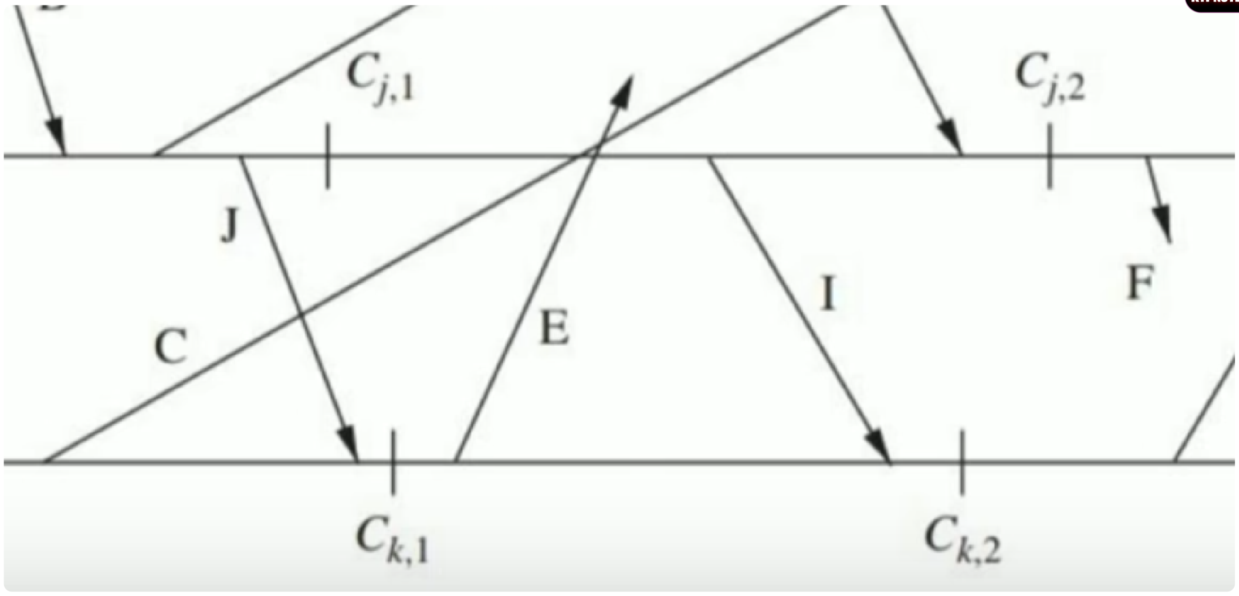
1. **Orphan Messages**: These occur when a process receives a message, but the sender's state is rolled back and it no longer has a record of sending that message.
2. **Cascading Rollbacks**: To remove orphan messages, other processes may also need to roll back to earlier checkpoints, leading to a chain reaction of rollbacks.
3. **Lost Messages**: A message is considered lost if the sender remembers sending it (after recovery), but the receiver has no record of receiving it due to rollback.
4. **Delayed Messages**: Messages that are still in transit during failure can arrive at unpredictable times—before, during, or after recovery—causing inconsistency.
5. **Delayed Orphan Messages**: These messages arrive at the receiver even though their send event has been undone due to rollback. Such messages must be discarded.
6. **Duplicate Messages**: If message logging is used, messages may be replayed during recovery, which can result in duplicates if not handled properly.
7. **Overlapping Failures**: If multiple processes fail around the same time, it leads to complications like amnesia and makes consistent recovery more difficult.

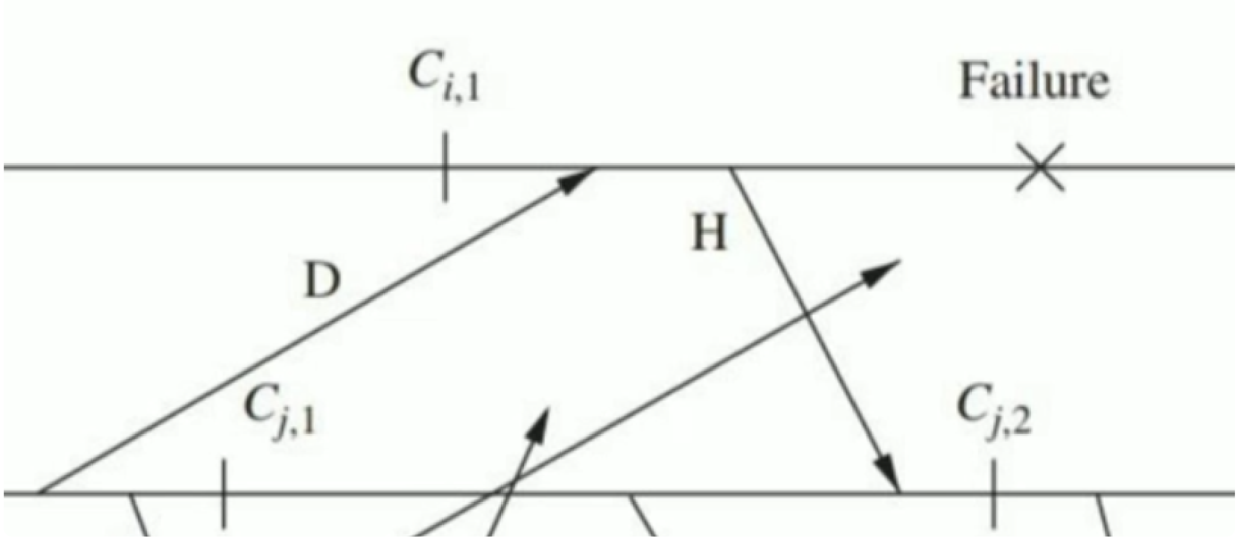
Example



- Imagine 3 processes: **Pi**, **Pj**, and **Pk**.
- They talk to each other by **sending messages** over a **FIFO (First In First Out)** network.
- Each process takes **checkpoints** to save their progress:
  - Pi: {Ci,0, Ci,1}
  - Pj: {Cj,0, Cj,1, Cj,2}
  - Pk: {Ck,0, Ck,1}
- Let’s say they exchange messages A to J.
- Now, **Pi fails** suddenly.
- When the process fails, it rollbacks to the last checkpoint, which is Ci,1
- Pj also needs to be rolled back, this is because
  - 

  - After rollback to Ci,1 there is an **orphan message** called H going to P2
    - Orphan message means
    - Receive is recorded, but send is not recorded
  - So we rollback Pj to Cj,1
- Pk also needs to be rolled back

- 

  - Orphan message I is there
    - Send is not recorded, only receive recorded
  - Rolled back to Ck,1
  - So this is an example of **cascading rollback**
- Message C is a **delayed message**
  - It can reach Pi before, during, or after recovery
- Message D is a **Lost Message**
  - 

  - This is because send is done
  - But the receive has been rolled back to Ci,1
- Message E and F are **Delayed orphan messages**

9. Differentiate consistent and inconsistent state with example.

1. Consistent State

- A **consistent state** is a state where:
  - If a process has **received a message**, then the corresponding **send event** must have happened.
  - The system is in a state that **could have naturally occurred during failure-free execution**

Example of a Consistent State

- Process A sends **message m1** to Process B.
- Process B **receives m1** before a failure occurs.
- After recovery, the system still shows that **A sent m1 and B received it**.

Since both the **send and receive** events are recorded, this state is **consistent**.

2. Inconsistent State

An **inconsistent state** is a state where:

- A process **shows that it received a message**, but the corresponding **send event is missing**.
- This situation is **impossible in a correct failure-free execution**.

Example of an Inconsistent State

- Process A sends **message m2** to Process B.
- Process B **records that it received m2**, but Process A **rolls back to a checkpoint before sending m2**.
- Now, the system shows **B received m2, but A never sent it**.
  Since this state **cannot occur in normal execution**, it is **inconsistent**

---

10. Explain log based roll back recovery

Log-based rollback recovery is a technique used to recover a system after a failure by leveraging logs that track the non-deterministic events that happen during the system's execution. This method helps bring back the system to its last known consistent state, ensuring that processes can continue their execution as smoothly as possible after a failure.

Key Concepts

1. **Deterministic and Non-Deterministic Events**
   - **Deterministic events** are those that always occur in the same way and are predictable. For example, the calculation of a number or the execution of a function where the output is always the same for the same input.
   - **Non-deterministic events** are those that may vary depending on external factors or interactions, like receiving a message from another process or user input. These events can't always be predicted, and the system must handle them carefully.
2. **Modeling Execution with Intervals**
   - A process's execution can be divided into intervals, where each interval starts with a non-deterministic event (like receiving a message). The rest of the interval depends on the events that occurred before it and is deterministic. This means that once a process has passed through a non-deterministic event, the sequence of operations that follow it can be predicted and repeated exactly in case of a failure.
3. **Role of Logs**
   - The key idea in log-based recovery is to **log the determinants** (the factors or conditions that determine the outcome) of non-deterministic events. These logs are stored in a safe, stable storage so that, in case of a failure, the process can roll back to the state it was in before the failure and "replay" the non-deterministic events exactly as they happened before.

How Log-based Rollback Recovery Works

1. **Logging Non-Deterministic Events**
   - During normal operation (i.e., when there are no failures), the system logs every non-deterministic event it observes. For example, if a process receives a message, it will log the details of that message so that it can be recreated if needed. The logs are saved in stable storage, meaning they are protected and won't be lost even if the system crashes.
2. **Checkpoints**

- To reduce the work needed for recovery, processes take **checkpoints** at regular intervals. A checkpoint records the current state of a process. If a failure occurs, the system can recover from the most recent checkpoint and then use the logs to replay the events that happened after that checkpoint.

3. **Recovery After Failure**
   - When a failure occurs, the process will recover by:
     - Returning to its last checkpoint.
     - Using the logged events to re-execute the non-deterministic events (like receiving messages).
     - This ensures that the process's execution will be identical to the pre-failure execution, avoiding any inconsistencies.
4. **No-Orphans Consistency**
   - A critical condition in rollback recovery is the **no-orphans consistency** condition. This condition ensures that after a failure, no process is left in an inconsistent state, also known as an "orphan."
   - Every process must have the correct log information or a stable checkpoint to avoid such inconsistencies.

Types of Log-based Rollback Recovery

1. **Pessimistic Logging**
   - **Pessimistic logging** assumes that failures can happen at any time. Therefore, it logs each non-deterministic event before it happens. This ensures that if a failure occurs, the system can always roll back to a known consistent state.
   - **Synchronous logging** is a feature of pessimistic logging, where logs are immediately written to stable storage before proceeding. While safe, this can introduce overhead since the system must wait for the log to be saved before continuing.
   - **Checkpointing** is also used to minimize recovery time.
2. **Optimistic Logging**
   - **Optimistic logging** is based on the assumption that failures are rare. It logs events asynchronously, meaning the system doesn't immediately save the logs but does so periodically.
   - This reduces overhead but allows temporary inconsistencies (orphans) to appear, which will eventually be fixed during recovery.

    - The system tracks dependencies between processes to ensure proper recovery in case of failure.
3. **Causal Logging**
    - **Causal logging** combines the strengths of both pessimistic and optimistic logging. It allows processes to log asynchronously but still ensures no orphan processes are created. It tracks causal dependencies (the relationship between events) to ensure that recovery can be done efficiently without losing consistency.
    - This method provides a balanced approach, reducing overhead while maintaining consistency.

11. Explain Checkpointing and Rollback Recovery in Detail.

Imagine you are playing a video game, and you reach an important level. To avoid losing progress if something goes wrong, you **save the game** at certain points. If you fail later, you can **reload** the saved checkpoint instead of starting from scratch.

This is exactly how **Checkpointing and Rollback Recovery** work in **distributed systems** to handle failures!

What is Checkpointing?

Checkpointing is the process of **saving the state of a process at a certain point** so that if a failure occurs, it can restart from that point instead of starting over.

Why Use Checkpoints?

- Reduces the amount of lost work after a failure.
- Helps systems **recover quickly** instead of re-executing everything.
- Saves **CPU time and network resources**.

🛠️ **Example:**
A bank transaction system saves its progress after every 100 transactions. If the system crashes after 150 transactions, it **restores the last saved state (100 transactions) and reprocesses the last 50** instead of all 150.

What is Rollback Recovery?

Rollback Recovery is the process of **restoring a system to a previous consistent checkpoint** after a failure.

Steps in Rollback Recovery:

1. **Detect Failure** 🚨
   - The system notices that a process has failed.
2. **Restore Checkpoint** 🔄
   - The system **reloads the last saved checkpoint** for that process.
3. **Re-execute** ⏩
   - The process **continues from where it left off**, avoiding major data loss.

🛠️ **Example:**
If an airline booking system crashes **after booking 50 tickets**, it restores the **last saved checkpoint** (e.g., after booking 40 tickets) and **reprocesses the last 10 tickets**.

Types of Checkpointing (How to Save Progress)

There are **three main ways** to implement checkpointing:

A. Uncoordinated Checkpointing (Independent Checkpoints)

- Each process **saves its own state independently**.
- **Problem:** It may lead to a **Domino Effect** (rolling back too far).

🛠️ **Example:**
If process **P1 rolls back**, but it has sent data to **P2**, then **P2 must also roll back**, and so on. This can cause the system to **restart from a very old state**.

B. Coordinated Checkpointing (Planned Checkpoints)

- All processes **coordinate** and save a **consistent global state**.
- Prevents **the Domino Effect** and ensures all processes are in sync.
- Uses a **coordinator process** to signal all processes to take checkpoints together.

🛠️ **Example:**
An online shopping system **saves a checkpoint every hour** across all servers. If one server crashes, it restores **all servers to the last saved state** to maintain consistency.

C. Communication-Induced Checkpointing (Smart Checkpointing)

- Checkpoints are **automatically triggered** when needed.
- Prevents unnecessary rollbacks and reduces overhead.
- Uses **extra information** in messages to decide when to save a checkpoint.

🛠️ **Example:**
A cloud storage service **automatically saves checkpoints** when many files are being uploaded, ensuring smooth recovery if a failure occurs.

Message Handling in Rollback Recovery

When recovering from a failure, messages between processes can be affected:

Types of Messages:

1. **In-Transit Messages** – Sent but not yet received.
2. **Lost Messages** – Sent before a crash but not delivered.
3. **Orphan Messages** – Received but the sender's checkpoint doesn't show that it was sent.

---

12. Explain the disadvantages of distributed shared memory.

Imagine Distributed Shared Memory (DSM) as **a giant shared notebook** that many people (computers) can write and read from. It **makes sharing data easy**, but it has some problems too.

1. Programmers Need to Learn Special Rules

- Just like **different schools have different rules** for writing assignments, DSM has **different consistency rules** for how data is shared.
- Programmers must **understand these rules** to avoid errors.

🛠️ **Example:** If two people write on the same notebook at the same time, how do we decide whose writing is correct? DSM needs rules for such cases.

2. DSM Uses a Lot of Resources (Slow Communication)

- DSM works by **sending messages between computers** to keep data updated.
- These messages **take time** and **slow things down**.

🛠️ **Example:** Imagine you and your friend are writing in a shared online document, but updates take **5 seconds to appear**. It would be frustrating!

3. Not as Fast as Custom Solutions

- DSM is **one-size-fits-all**, but some applications need **faster, custom solutions**.
- Custom-built systems can be **optimized** for speed and efficiency.

🛠️ **Example:** Buying a **ready-made suit** vs. getting a **tailor-made suit**. DSM is like the ready-made suit—it works, but **not always a perfect fit**.