UCS2701 DISTRIBUTED SYSTEMS

Unit -4

Consensus and Recovery

Consensus and Agreement Algorithms : Problem Definition - Agreement in a failure-free system (synchronous or asynchronous) - agreement in (message-passing) synchronous systems with failures; Checkpointing and roll back recovery : Introduction - Background & Definition - Issues in failure recovery - (Checkpoint-based recovery - Log-based rollback recovery - Koo-Toueg coordinated checkpointing algorithm - Juang - Venkatesan algorithm for asynchronous checkpointing and recovery)

* Consensus and Agreement

→ Coordination requires processes to exchange information to negotiate with one another & eventually reach an agreement.

e.g. commit decision in database systems where processes collectively decide whether to commit or abort a transaction

Key Assumptions

1. Failure models
   - Fail-stop failure : A process crashes but doesn't act maliciously
   - Omission failure : msgs may be lost

- Byzantine Failure : A faulty process may behave arbitrarily or maliciously

2. Synchronous vs. Asynchronous Communication

- Synchronous - known, fixed upper bound on msg. transmission time
- Asynchronous - no guarantees on msg timing, making failures indistinguishable from delays.

3. Network Connectivity : The system has full logical connectivity,

4. Sender Identification : A process that receives a message always knows the identity of the sender process.

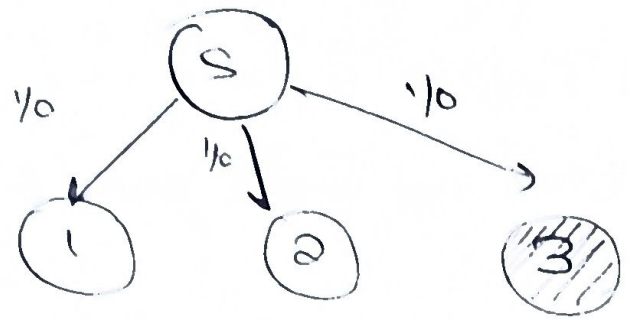
5- Channel Reliability : The channels are reliable, only the processes may fail

- Faulty process can forge and claim that it was received from another process or tamper the contents of the message.
- can be prevented using authentication such as digital signatures

6. Agreement Variable - may be Boolean / multi-valued and need not be an integer.

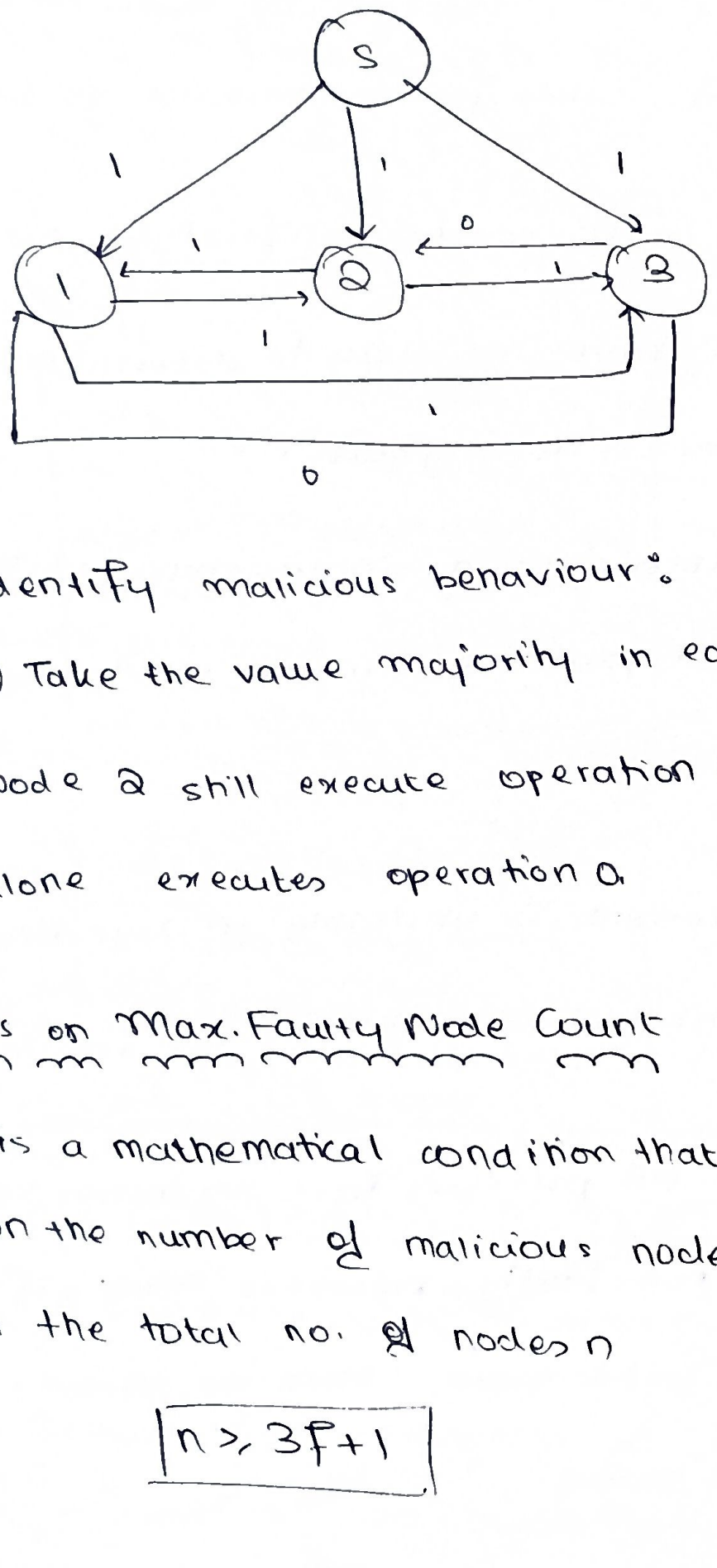
* Byzantine Agreement Problem (Consensus)

→ There is a chief commander & several loyal generals.
→ Loyal generals will always follow the commands.
→ Malicious generals will sometimes be disloyal.



1 → attack
0 → retreat

→ Repeat the order sent by the chief to all the nodes, 3 tries to be malicious and disrupt.



→ To help identify malicious behaviour:

① Take the value majority in each node

Node1 & Node 2 still execute operation 1

Node 3 alone executes operation 0.

Limitations on Max. Faulty Node Count

→ There is a mathematical condition that defines the upper bound on the number of malicious nodes a system can tolerate, based on the total no. of nodes n

$$n \geq 3f+1$$

So, if $n=7$ → at max 2 faulty nodes

if $n=3$ → cannot have any faulty nodes

→ This brings about a crucial limitation of Byzantine agreement in asynchronous distributed systems.

→ In asynchronous systems, there is no guaranteed upper bound on the time it takes for a message to be delivered.

→ For instance, if one node sends conflicting or delayed messages, other nodes have no way to determine if that node is simply slow or if it is fault.

→ The system may interpret a single changed bit as a sign of crash or of a Byzantine failure which disrupts the consensus process.

∴ Byzantine Agreement is unusable in asynchronous distributed algorithms.

* Consensus Problem → all processes have an initial value

→ all non-faulty processes ~~[illegible]~~. If they have the same initial value, then the agreed-upon value must be the same value.

* Interactive Consistency - All non-faulty processes must agree on the same array of values $A[v_1, \ldots, v_n]$

→ If process i is non-faulty, and its initial value is v_i, all non-faulty processes agree on v_i as the i^{th} element of the array A.

→ If process j is faulty, then the non-faulty processes can agree on any value for $A[j]$

* Overall Results

Failure mode	Synchronous System	Asynchronous System
No Failure	agreement attainable common knowledge also attainable	agreement attainable concurrent common knowledge attainable
Crash failure	agreement attainable $f < n$ processes $\Omega(f+1)$ rounds	not attainable
Byzantine failure	agreement attainable $f \leq \lceil (n-1)/3 \rceil$ Byzantine processes $\Omega(f+1)$ rounds	not attainable.

* Agreement in a Failure-Free System

(Synchronous or Asynchronous)

→ In a failure-free system, consensus can be reached by collecting information from the different processes, arriving at a decision, and distributing this decision in the system.

→ A distributed mechanism would have each process broadcast its values to others, and each process computes the same function on the values received.

→ eg. of functions include majority, max & min functions.

Synchronous System - In a synchronous system, this can be done in a constant no. of rounds

- common knowledge of the system can be obtained using an additional round.

Asynchronous System - consensus can be reached using a constant number of message hops.

- Concurrent common knowledge of the consensus value can also be obtained.

∴ Reaching agreement is straightforward in a failure-free system.

* Consensus - Algorithm for fault-free consensus

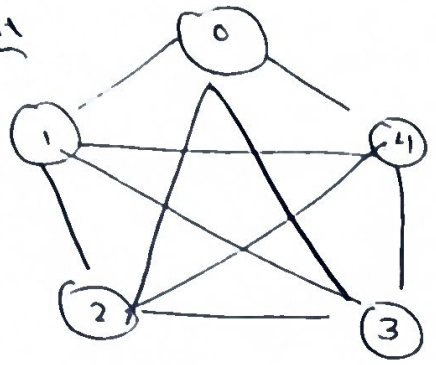
Each processor:

1. broadcasts its input to all processors
2. decide on the minimum

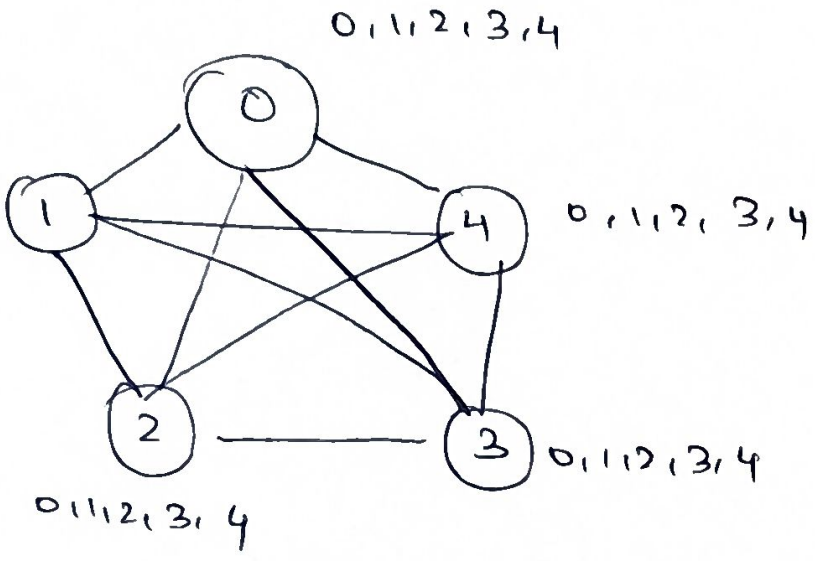
Only 1 round is needed, since the graph is complete.

Example

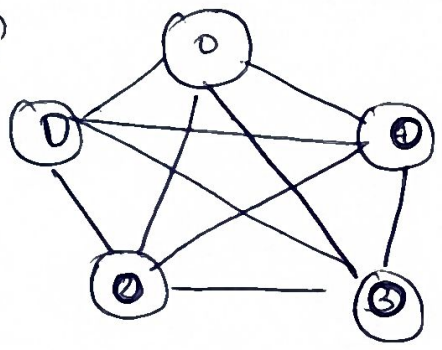
1. Start



2. Broadcast



3. Decide on minimum



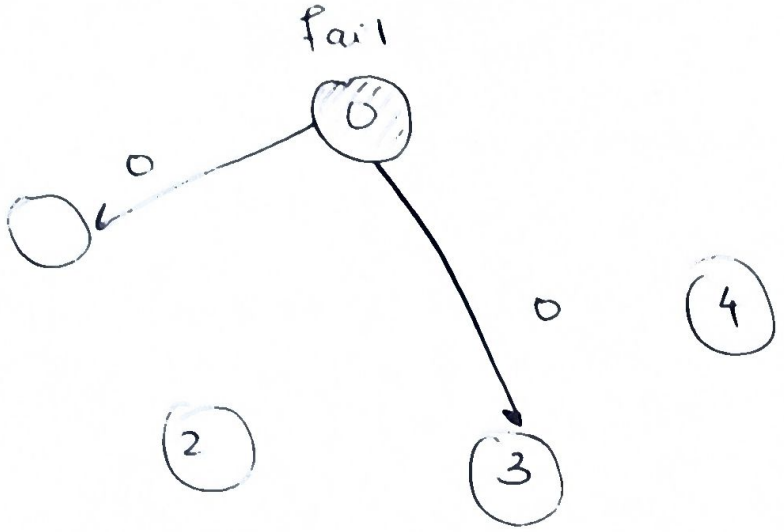
→ Finish

→ **Note**

This algorithm satisfies the validity condition i.e If everybody starts with the same initial value, everybody decides on that failure.

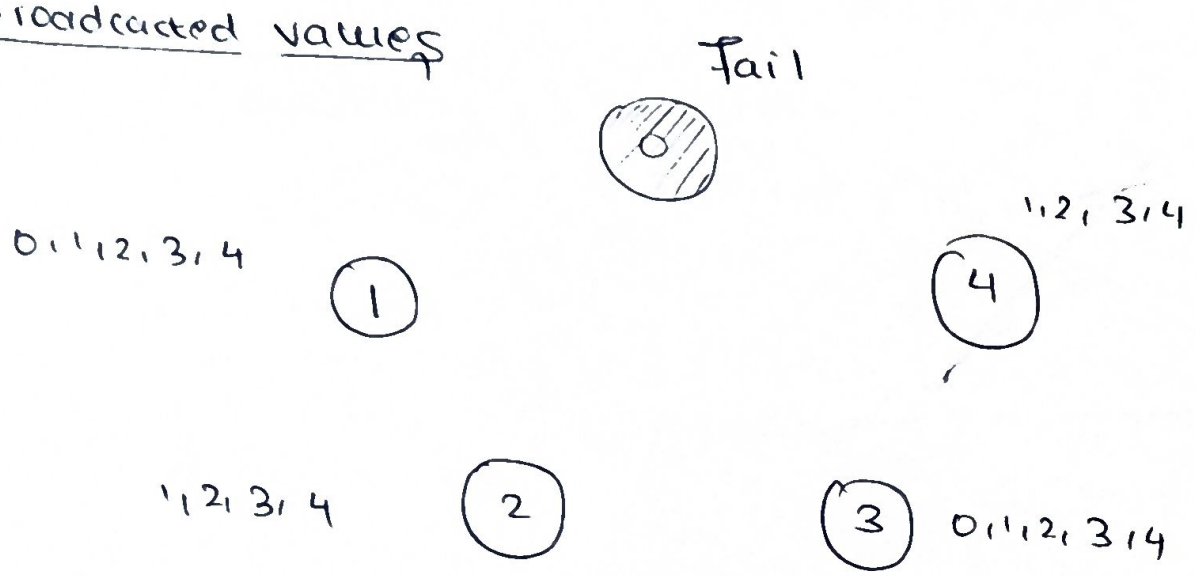
* **Consensus with Crash Failures**

→ This simple algorithm would no longer work consider the following case:

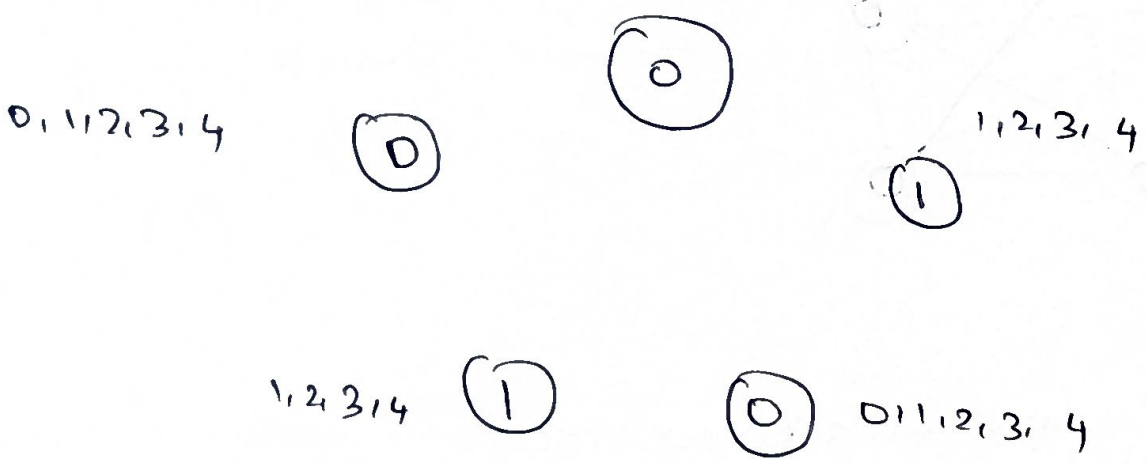


Node 0 fails and doesn't broadcast its value to all processors.

Broadcasted values



Decide on minimum



⇒ No consensus!

* f-resilient consensus algorithm

→ If an algorithm solves consensus for f-failed (crashing) processors, it is called an f-resilient consensus algorithm.

Algorithm

Round 1 : Broadcast own value

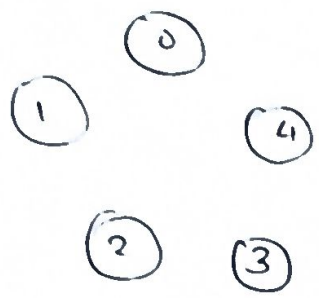
Round 2 - f+1 : Broadcast any new received values

End of round f+1: Decide on the minimum value received.

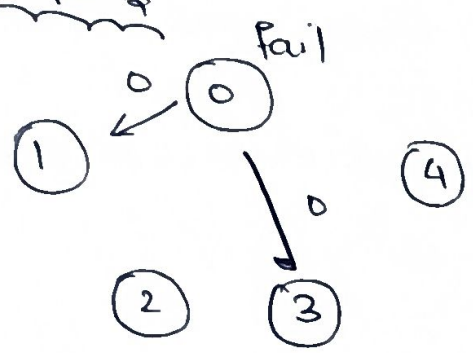
Example $f = 1$ Failure

$\Rightarrow f+1 = 2$ rounds need

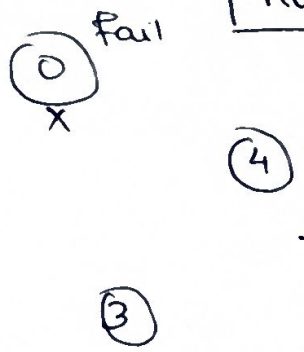
1. Start



2. Node zero fails



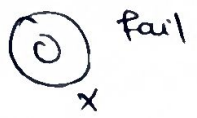
3. Broadcast all values | Round 1 |



0,1,2,3,4 | 1,2,3,4 | 1,2,3,4 | 0,1,2,3,4 | 1,2,3,4

→ The underlined values are new values.

4. Broadcast new values | Round 2 |



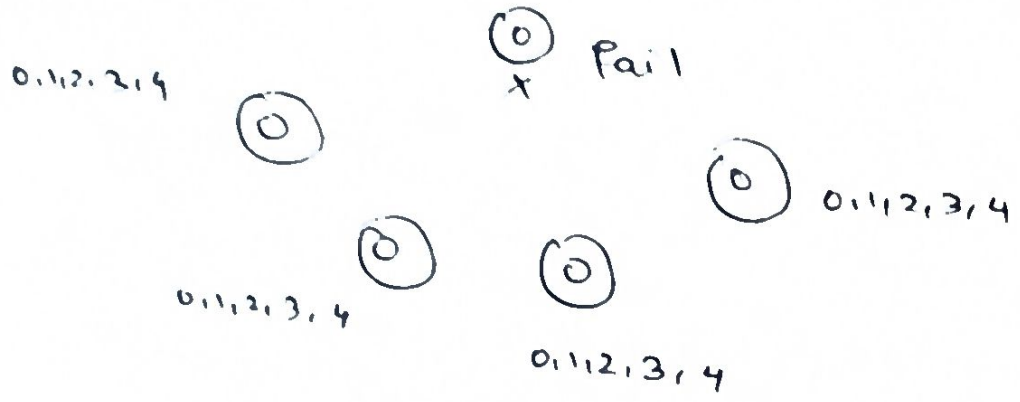
1: 0,1,2,3,4

4: 0,1,2,3,4

2: 0,1,2,3,4

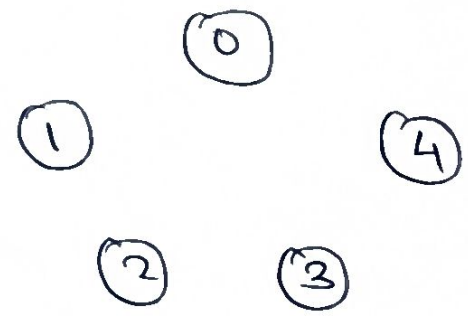
3: 0,1,2,3,4

5. Decide on a minimum



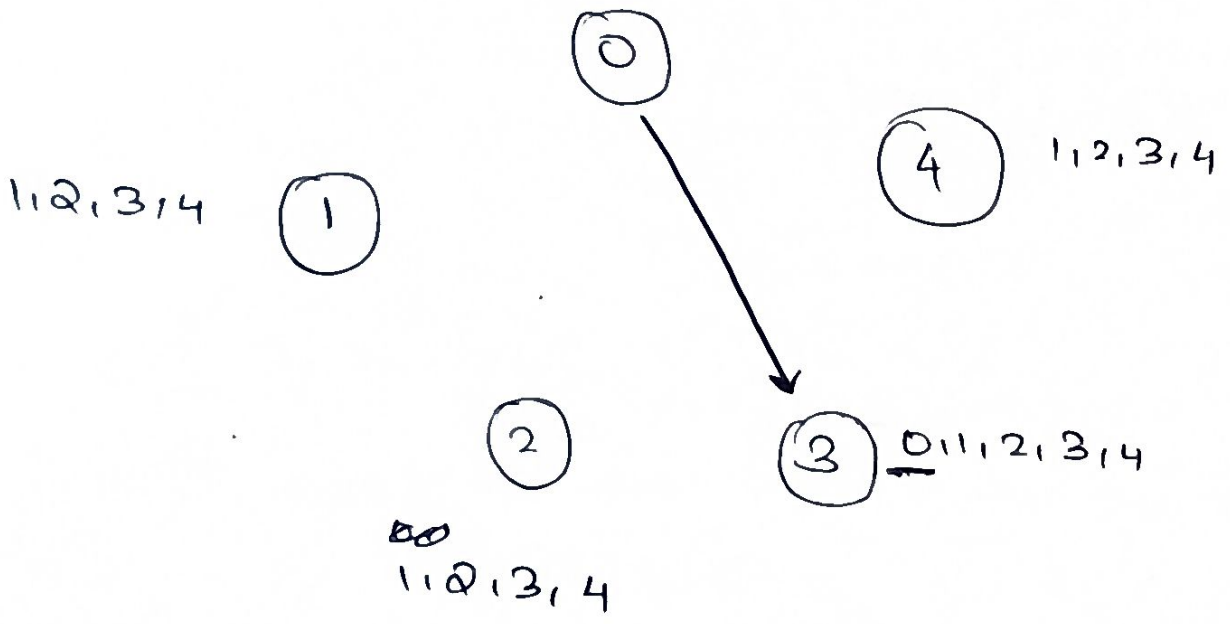
Example2 - 2 node failures, Node 0 fails, sends msg only to Node 3, Then node 3 fails, it sends a message only to Node 1.

Ans 1. Start



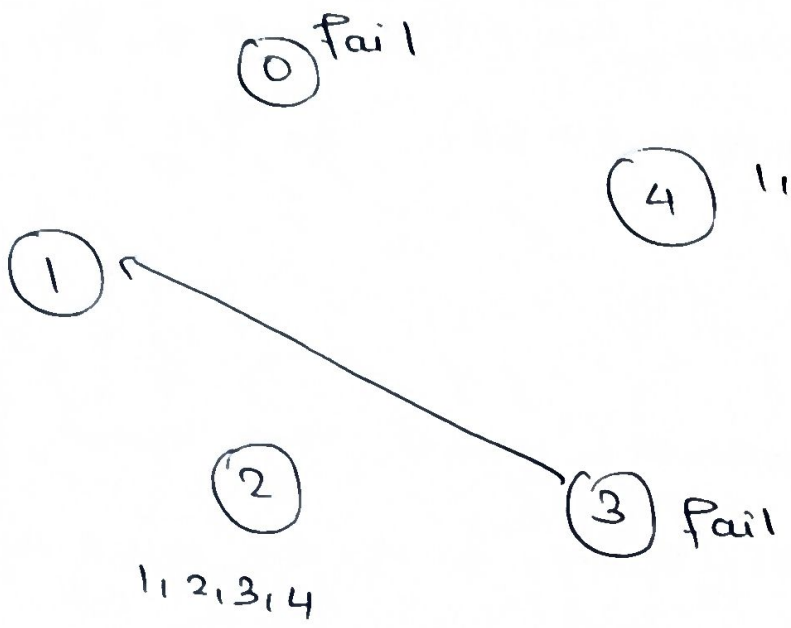
2. Broadcast 1 - Node 0 fails

ROUND 1

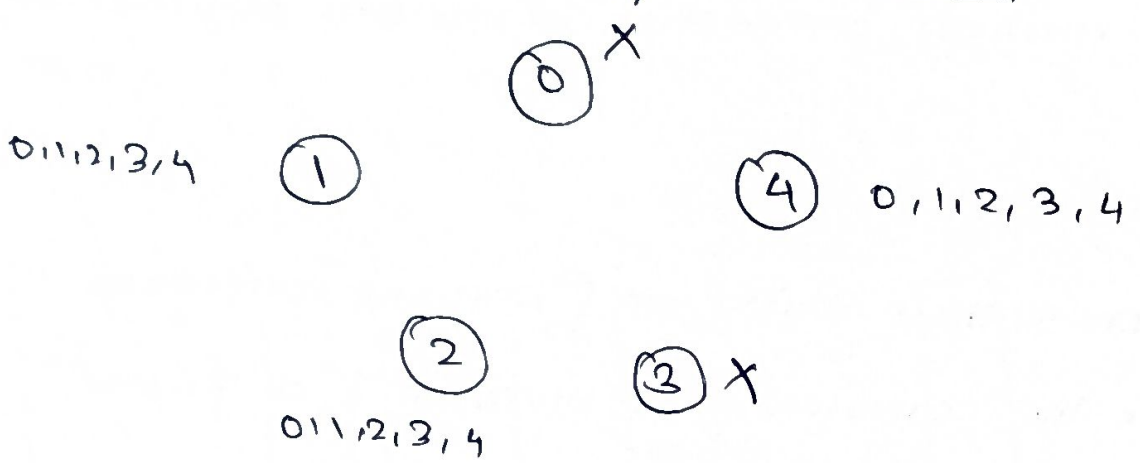


3. Broadcast new values to everybody - Node 3 fails

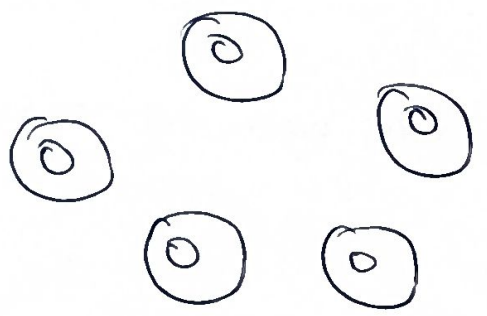
Round 2



4. Broadcast new values **Round 3**



5. Decide upon minimum



⇒ consensus reached

* Why f+1 rounds?

→ At the end of the algorithm, at the end of the round with no failure, every non-faulty process knows about all the values of all other participating processes

→ This means at the end of the round w/ no failure, everybody would decide the same value.

→ However, we don't know the exact position of this round, so we have to let the algorithm execute for f+1 rounds.

* Validity of the Algorithm

→ When all processes start with the same input value, then the consensus is that value. This holds, since the value decided from each process is some input value.

Theorem

Lower-Bound : Any f-resilient consensus algorithm requires at least f+1 rounds.

Proof by contradiction : Suppose there exists an f-resilient consensus algorithm that can solve the consensus problem in f or fewer rounds.

→ Consider the scenario in which there are only f rounds of communication.

→ In the fth round, faulty nodes can behave maliciously & disrupt communication

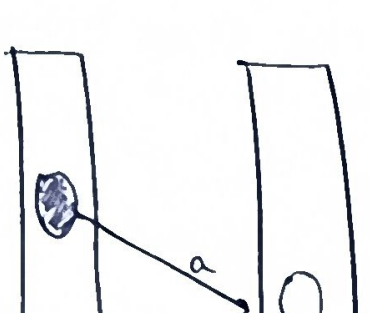
→ Since there are no further rounds, there is no opportunity for the nodes to detect this malicious behavior.

→ This causes different nodes to assume diff. values for consensus.

→ This is a contradiction, since all nodes must agree on the same value.

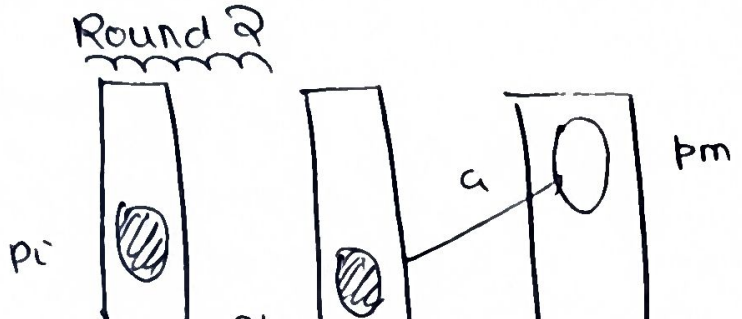
Consider the worst-case scenario - There is a process that fails in each round.

Round 1



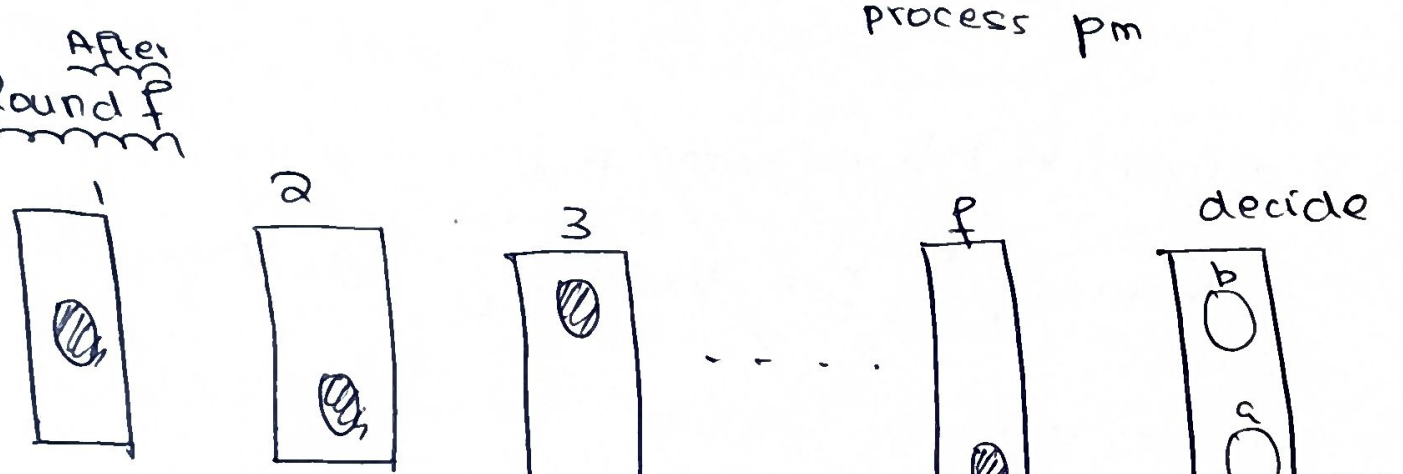
Before p_i fails, it sends its value a to only one process p_k

Round 2



Before process p_k fails, it sends a value a to only process p_m

After Round f



Process p_n may decide a, and all other processes may decide another value.

∴ f rounds are not enough. At least $f+1$ rounds are needed.

* Formal Representation of the Algorithm for Consensus in Synchronous Systems

On initialization

$$Values_i^1 = \{v_i\} \quad values_i^0 = \{\}$$

In round r $(1 \le r \le f+1)$

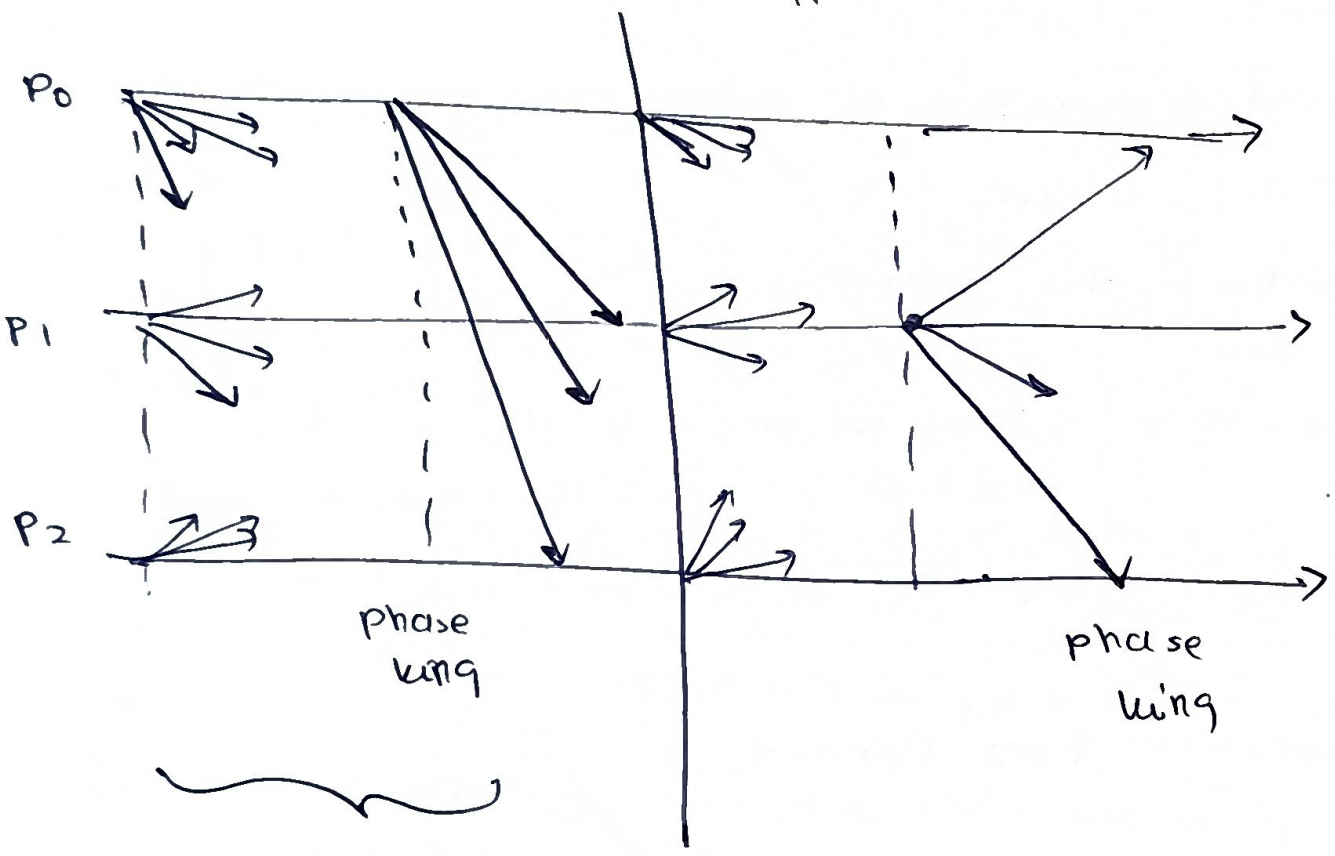
* Phase King Algorithm

→ Each phase has a unique 'phase king' derived from PID

→ Each phase has 2 rounds:

① In the 1st round, each process sends its estimate to all other processes.

② In the 2nd round, the phase king process arrives at an estimate based on the values it recieved in the 1st round, and broadcasts its new estimate to all others.



Algorithm

boolean v ← initial value

Integer f ← max. no. of malicious processes, $f < \lceil n/4 \rceil$

(1) Each process executes the following f+1 phases, where $f < n/4$

* Checkpointing and Rollback Recovery

A. Rollback Recovery Protocols

→ restore the system back into a consistent state after a failure

→ achieve fault tolerance by periodically saving the state of a process during the failure-free execution

→ treats a distributed system application as a collection of processes that communicate over a network

B. **Checkpoints** – the saved states of a process

C. **Complications in Rollback Recovery**

→ Due to message passing between processes, dependencies arise.

→ This complexity can lead to the domino effect, where multiple processes, that didn't fail may also have to roll back because of these dependencies.

* Techniques to avoid the Domino Effect

1. Independent (Uncoordinated) Checkpointing → If each process takes checkpoints independently, the system can still from the domino effect

2. Coordinated Checkpointing → In this method, processes coordinate their checkpoints to ensure a system-wide consistent state, preventing domino effects.

3. Communication-induced checkpointing : Checkpoints are forced based on information exchanged during communication between processes, thus avoiding domino effects, without full coordination.

4. Log-based rollback recovery : This combines checkpointing with logging of non-deterministic events (those that vary in executions) to achieve fault tolerance, relying on a model called the piecewise deterministic (PWD) assumption

* Local Checkpoints

→ These are snapshots of the state of a process at certain instances.

→ A process stores these snapshots on stable storage and can roll back to any of these checkpoints.

$C_{i,k}$ - k^{th} local checkpoint at Process P_i

$C_{i,0}$ - A process P_i takes a checkpoint before starting execution.

* Consistent Global States

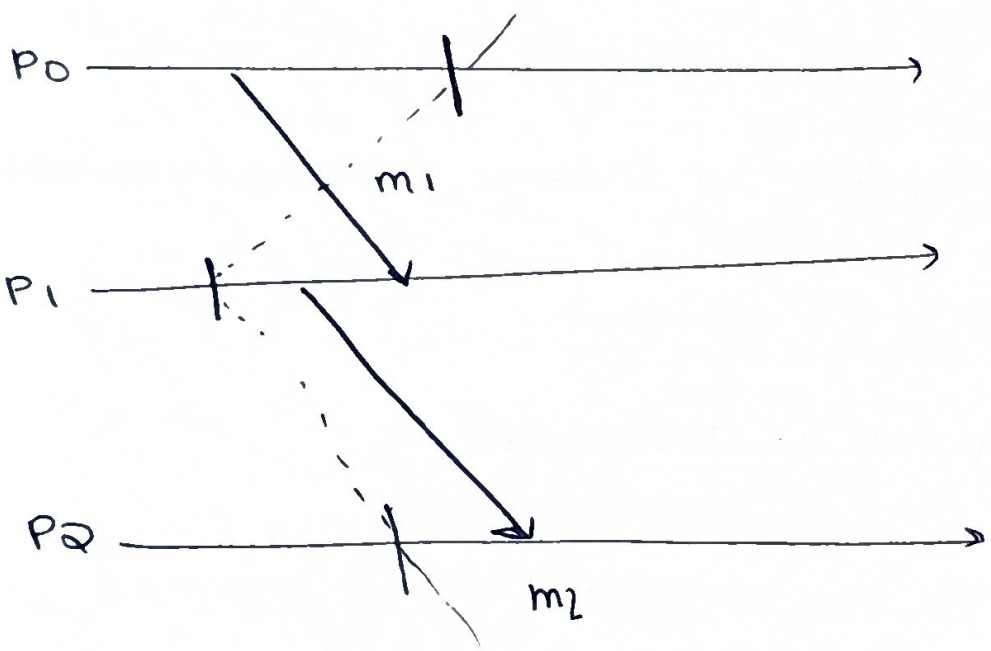
Global State - A collection of the individual states of all processes and states of communication channels in the distributed system.

Consistent Global States - A global state that could occur during failure-free execution. If a process' state reflects receiving a msg, the sender's state should reflect sending that message

Global Checkpoints - A collection of local checkpoints from each process

Consistent Global Checkpoint - This occurs when no message is received by an process after it has taken its local checkpoint, but before the sender has also taken its checkpoint

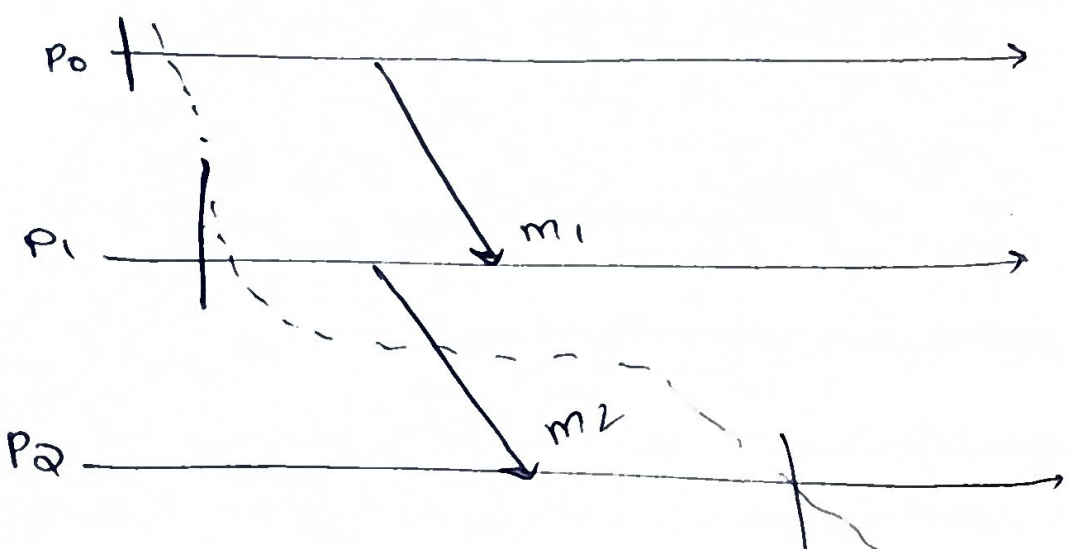
Example 1 - Consistent States



| ⇒ checkpoint taken

(⇒ represent when msg is sent & received

Example 2 - Inconsistent States



checkpoint before sending msg

checkpoint before sending msg

* Interactions with the outside world

Outside World Process – A distributed system often interacts with the outside world to receive input data or deliver the outcome of a computation. OWP is a special process that interacts with the rest of the system through message passing.

→ A common approach is that before allowing the system to process an input, each input message from the outside world is saved on stable storage to ensure recovery in case of failure.

* Types of Messages

1. In-transit messages - messages that have been sent but not yet received

2. Lost messages - messages whose send is done but receive is undone due to rollback.

3. Delayed messages - message whose 'receive' is not recorded because the receiving process was either down or the message arrived after rollback

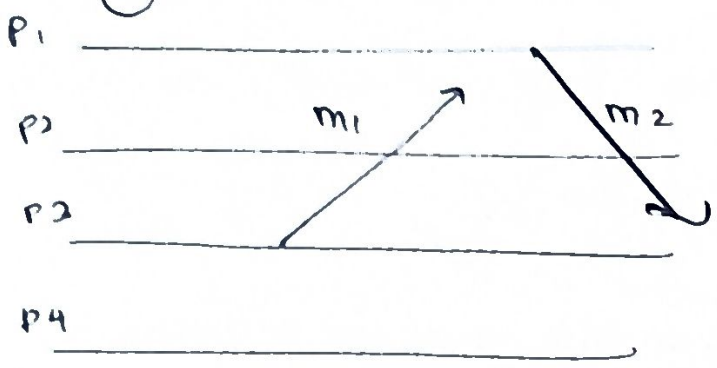
4. Orphan messages - messages with 'receive' recorded but message 'send' not recorded

   - do not arise if processes roll back to a consistent global state

5. Duplicate messages : arise due to message logging and replaying during process recovery

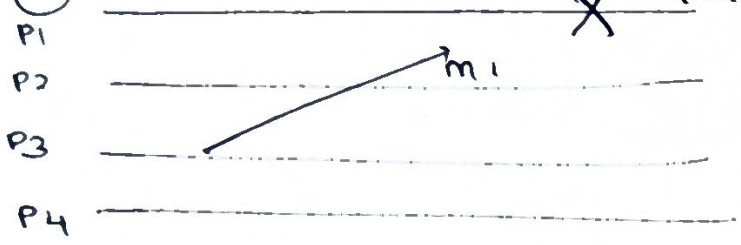
Examples

①



messages in-transit

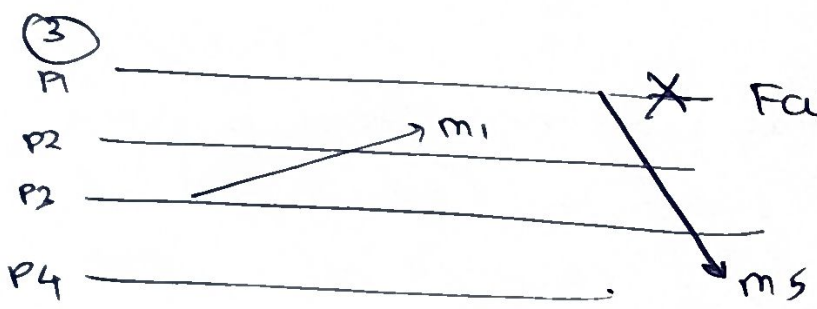
②



lost = m_1

not received and hence not recorded before failure

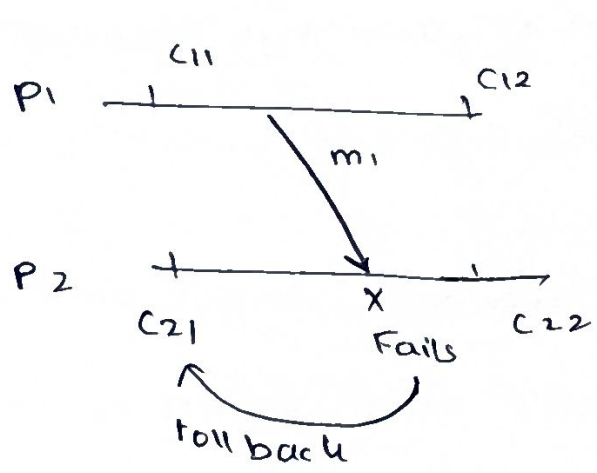
③



delayed

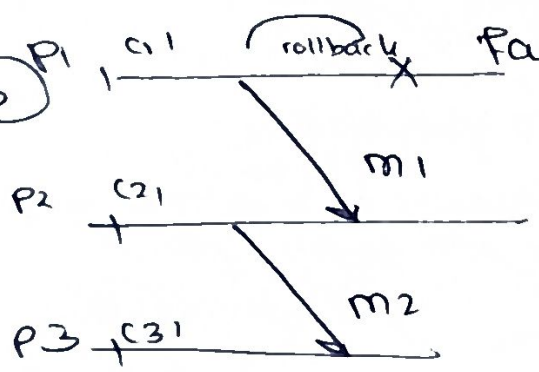
m_1, m_5 - need to be resent and are delayed because they need to be re-processed before rollback

④



m_1 is an orphan

⑤



m_1 send is erased

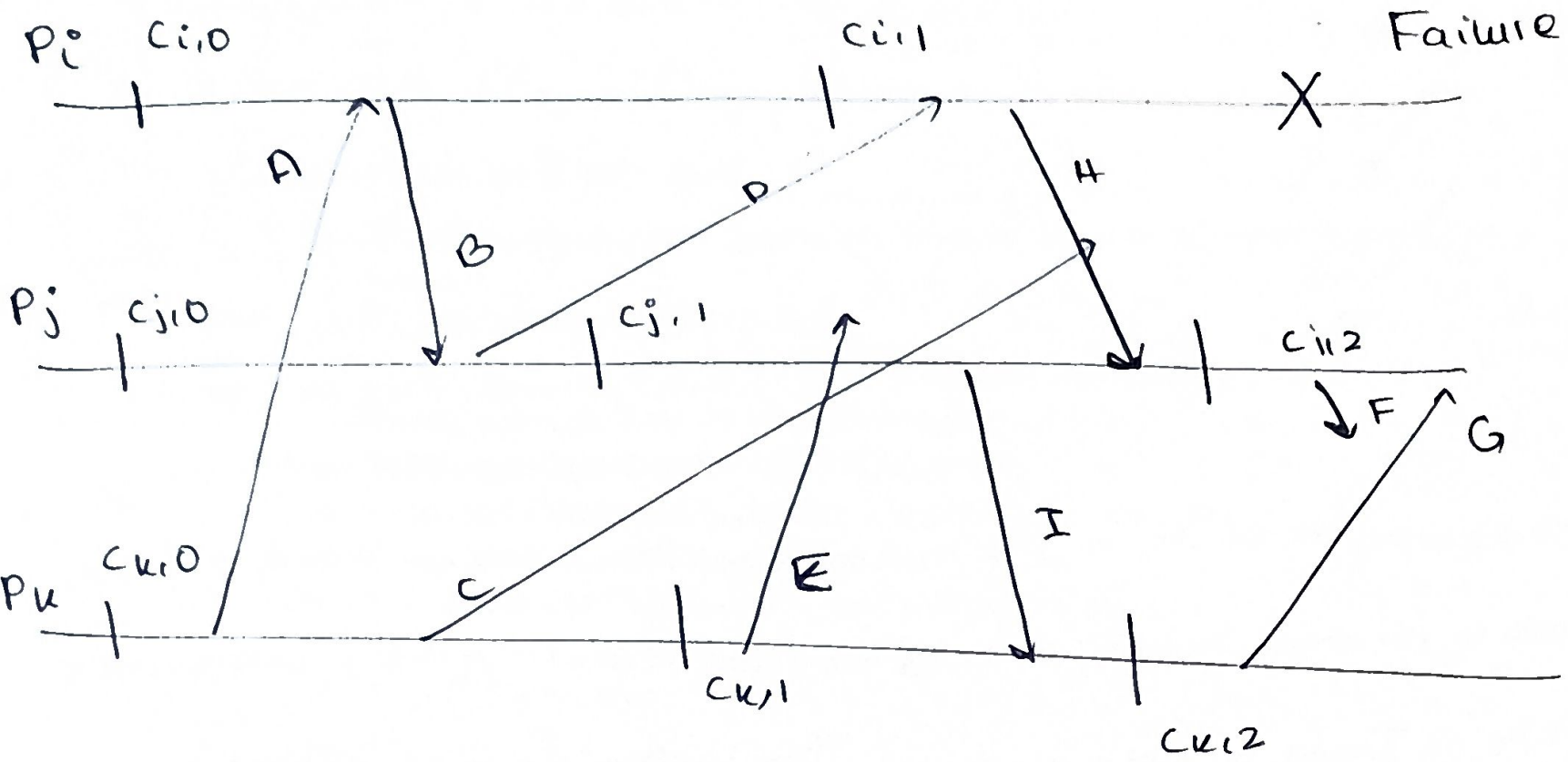
P2 realizes that 2 rolls back to C_{21}

m_2 send is erased

domino effect

* Issues in Failure Recovery

Consider the following example:



→ The checkpoints are: $\{C_{i,0}, C_{i,1}\}$

$\{C_{j,0}, C_{j,1}, C_{j,2}\}$

$\{C_{k,0}, C_{k,1}, C_{k,2}\}$

→ Messages: A-J

→ The restored consistent global state: $\{C_{i,1}, C_{j,1}, C_{k,1}\}$

Rollback to $C_{i,1}$ causes the following problems:

(i) H becomes an orphan (receive recorded, send undone)

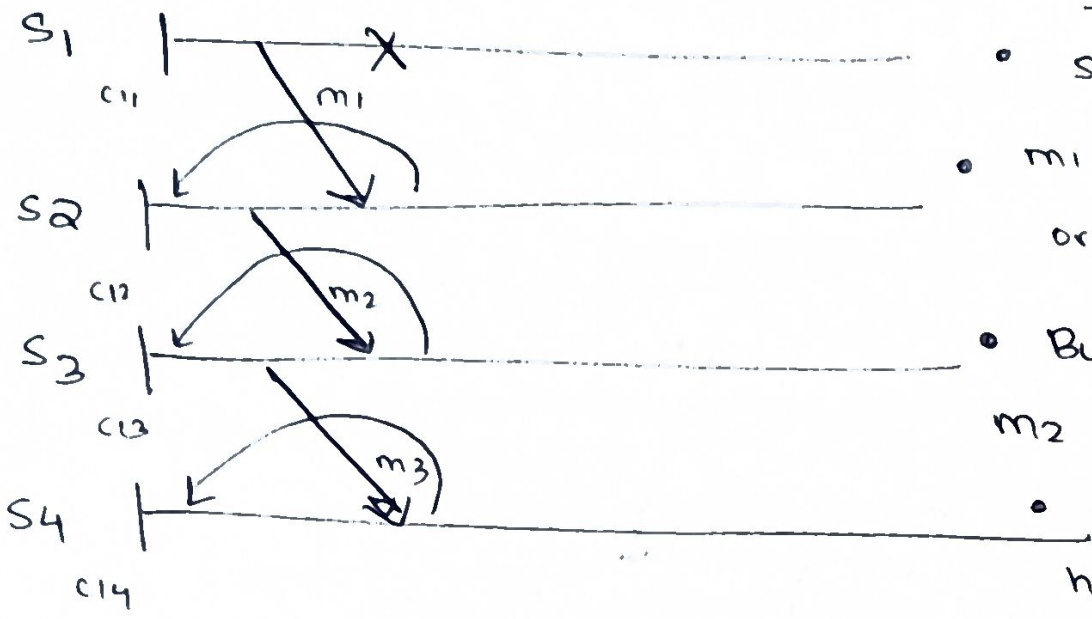
(ii) I becomes an orphan (" ")

(iii) C - delayed msg

(iv) D - lost msg (received but, the receive has been undone)

(v) E, F - delayed orphans (no receive, send also undone, needs to be done again after roll back)

* Dominoes Effect



Process : • S_1 sends m_1 to m_2

• S_1 fails. It has to rollback

• m_1 reaches S_2. However, it is an orphan, S_2 rolls back

• But before rollback S_2 sends m_2 to S_3

• m_2 reaches S_3, but S_2 has no record of m_2 being sent

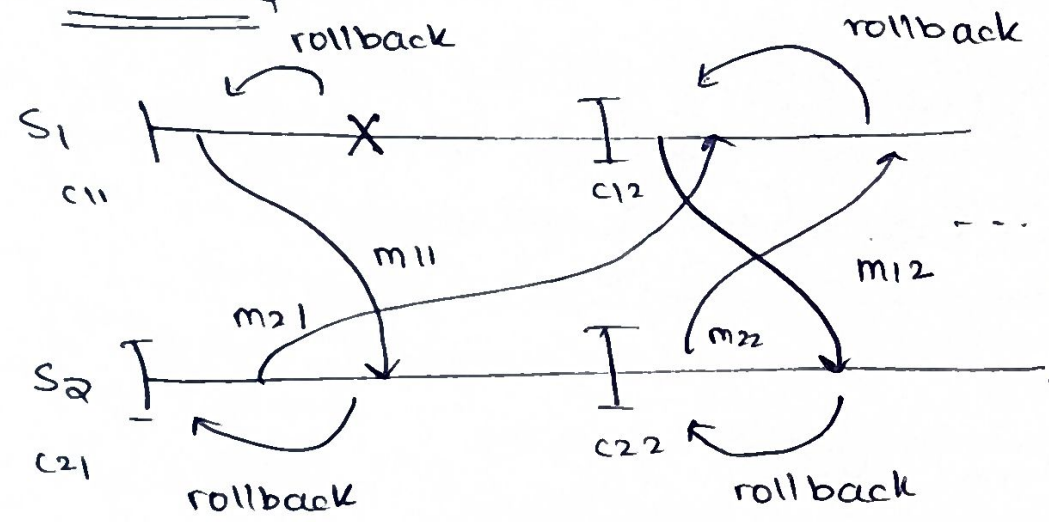
• S_3 too has to rollback

• However, it sends m_3 to S_4, before the rollback occurs.

• m_3 is received by S_4, but S_3 has no record of m_3 being sent

- Thus S_4 is forced to rollback too. This is the domino effect.

* Livelocks



Process : - Initial checkpoints by S_1 and S_2 are C_{11} and C_{21}.

- S_1 sends m_{11} to S_2
- S_1 then fails, and has to rollback
- m_{11} becomes an orphan
- m_{11} reaches S_2, however it has no sender, S_2 has to roll back, however m_{21} has already been sent to S_1
- m_{21} becomes an orphan
- S_1 receives m_{21} but has no record of the sender, S_1 has to rollback again.
- But S_1 has already sent m_{22} to S_{22}. m_{22} becomes an orphan
- Now S_2 has to rollback because of this.

∴ $S_1 \rightarrow S_2$ keeps sending msgs and rolling back because of orphan processes stuck in between. This is called a livelock.

UCS2701 Distributed Systems

Unit 4

Consensus and Recovery Models

Topics

1. What is consensus and agreement, what are the assumptions made in these models?
2. Byzantine Consensus Algorithm
3. Comparison of failure types on synchronous and asynchronous systems
4. Consensus algorithm for fault free consensus
5. Consensus algorithm for crash failures
6. f-resilient Consensus Algorithm and proof of its validity
7. Phase King Algorithm (just what it is)
8. Introduction to Checkpointing and rollback recovery
9. Techniques to avoid the domino effect
10. Types of Messages
11. Domino Effect
12. Livelocks

Checkpoint-based Recovery Methods

1. Types of Checkpoint-based Recovery Methods

- **Uncoordinated Checkpointing**
  - Direct Dependency Tracking Technique
- **Coordinated Checkpointing**
  - Blocking Coordinated Checkpointing
  - Non-Blocking Coordinated Checkpointing
- **Communication-induced Checkpointing**
  - Model-based Checkpointing
  - Index-based Checkpointing

2. What is Checkpoint-Based Recovery?

Checkpointing involves saving the state of a process or system at regular intervals. If a failure occurs, the system can restart from the most recent checkpoint, avoiding the need to start from scratch. It is a simpler approach compared to log-based recovery, but it does have limitations:

- It may not fully replicate pre-failure execution.
- It is not well-suited for applications that frequently interact with the outside world.

3. Uncoordinated Checkpointing

Uncoordinated checkpointing allows processes to save their states independently without coordination with others.

Advantages:

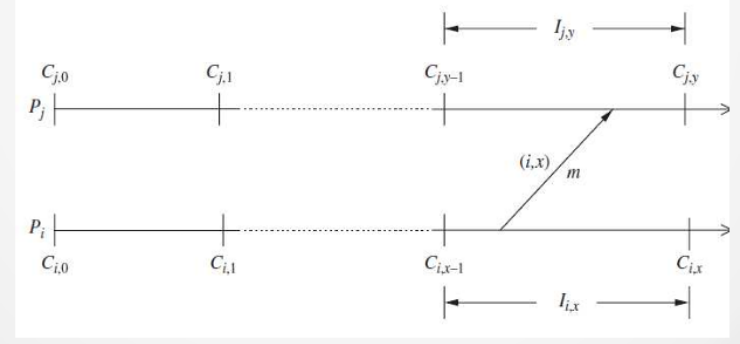
1. **Lower Runtime Overhead**:
    - Processes do not need to pause or communicate to take checkpoints, reducing delays during normal operation.
2. **Autonomy**:
    - Each process chooses the best time to take its checkpoints, optimizing performance.

Disadvantages:

1. **Domino Effect**:
    - A failure in one process can cause cascading rollbacks in others, leading to the loss of a significant amount of work.
2. **Slow Recovery**:
    - Processes must iterate through their checkpoints to find a consistent global state, increasing recovery time.
3. **Unnecessary Checkpoints**:
    - Processes might save states that are not useful for recovery, wasting memory and computation.
4. **Not Suitable for Certain Applications**:
    - Applications that frequently produce output require global coordination, making uncoordinated checkpointing inefficient.

4. Direct Dependency Tracking Technique (for Uncoordinated Checkpointing)

To handle dependencies during recovery, processes record message dependencies during execution.



Steps:

1. **Dependency Recording**:
    - When one process sends a message to another, it notes the dependency between the sender's checkpoint and the receiver's state.
2. **Failure Handling**:
    - In the event of failure, the recovering process broadcasts a request to other processes, asking for their dependency information.
3. **Recovery Line Calculation**:
    - Using the collected dependency information, the system calculates a "recovery line" — the latest consistent state across all processes.
4. **Rollback**:
    - Processes roll back to their respective checkpoints as determined by the recovery line.

5. Coordinated Checkpointing

What It Is:

- Involves all processes **orchestrating their checkpointing activities** to ensure that all checkpoints form a globally consistent state.

Advantages:

1. **Simplifies Recovery**:
    - Avoids the **domino effect**, as processes always restart from their most recent checkpoint.
2. **Reduces Storage Overhead**:
    - Each process only maintains **one checkpoint**, eliminating the need for garbage collection.

Disadvantages:

- **High Latency**:
    - Large delays in committing output, as a global checkpoint is required before sending a message to the outside world.

6. How Coordinated Checkpointing Works

Synchronized Clocks (Ideal Scenario):

- If clocks were perfectly synchronized, processes could trigger checkpointing at the same time.

In Reality:

- Processes either:
    1. Block communication during the protocol.
    2. Use piggybacking (attaching checkpoint indices to messages) to avoid blocking.

7. Blocking Coordinated Checkpointing

Steps:

1. **Communication Blocking**:
    - All communication stops while the protocol executes.
2. **Checkpoint Coordinator**:
    - Broadcasts a request for all processes to take a checkpoint.
3. **Tentative Checkpoints**:
    - Each process:
        - Stops execution.
        - Takes a tentative checkpoint.

        - Sends an acknowledgment back to the coordinator.
4. **Commit Phase**:
    - After acknowledgments, the coordinator sends a commit message.
    - Processes replace old checkpoints with the tentative one and resume execution.

Disadvantage:

- **Blocked Computation**:
    - The system halts during checkpointing, causing delays.

8. Non-Blocking Coordinated Checkpointing

How It Works:

1. Processes continue execution while taking checkpoints.
2. Prevents application messages from making checkpoints inconsistent:
    - Example: A message sent before the checkpoint request might still reach the destination afterward, causing inconsistency.

Challenges:

- Processes must ensure all messages are accounted for to create a consistent checkpoint.

There can be two cases of non-blocking coordinated checkpointing.

a. **Communication Channels are Reliable & FIFO**

- Chandy and Lamport in which markers play the role of the checkpoint request messages.
- The initiator takes a checkpoint and sends a marker (a checkpoint request) on all outgoing channels.
- Each process takes a checkpoint upon receiving the first marker and sends the marker on all outgoing channels before sending any application message.

b. **Communication Channels are Non-FIFO**

- The marker can be piggybacked on every post-checkpoint message.

- When a process receives an application message with a marker, it treats it as if it has received a marker message, followed by the application message.

9. Scalability in Coordinated Checkpointing

Key Points:

1. All processes must participate in each checkpoint, affecting scalability.
2. To improve efficiency:
    - Limit checkpointing to only those processes that communicated with the initiator since the last checkpoint.

Minimal Coordination:

- **Koo and Toueg Protocol**:
    - Reduces coordination to the minimum necessary for consistent checkpoints using a two-phase protocol

10. Communication-Induced Checkpointing

Overview:

- Combines features of uncoordinated and coordinated checkpointing.
- **Purpose**: Avoids the domino effect while still allowing processes to take checkpoints independently.
- **How it Works**: Processes may take extra **forced checkpoints** (over and above autonomous ones) to ensure recovery progress.

Types of Checkpoints:

1. **Autonomous Checkpoints**:
    - Checkpoints taken independently by processes, also called **local checkpoints**.
2. **Forced Checkpoints**:
    - Processes are forced to take these checkpoints to ensure the global recovery line advances.

Key Features:

- Communication-induced checkpointing piggybacks protocol related information on each application message.
- The receiver of each application message uses the piggybacked information to determine if it has to take a forced checkpoint to advance the global recovery line.
- The forced checkpoint must be taken before the application may process the contents of the message, possibly incurring some latency and overhead.
- No special coordination messages are exchanged unlike coordinated checkpointing

11. Communication Inducted Checkpoint Types:

Model-Based Checkpointing:

- Maintains checkpoints and communication patterns to prevent inconsistencies or domino effects.
- **Key Features**:
    1. **Piggybacking**: All protocol information is piggybacked onto application messages.
    2. **MRS Model (Mark, Send, Receive)**:
        - Ensures no message-receiving event happens before message-sending events in a checkpoint interval.
    3. **Efficient Checkpoints**:
        - Additional checkpoints may be taken before every message-receiving event to maintain consistency.
        - Recent focus ensures checkpoints belong to a consistent global checkpoint to avoid redundancy.

Index-Based Checkpointing:

- Uses **monotonically increasing indexes** for checkpoints to ensure consistent recovery.
- **Key Features**:
    1. **Consistency by Index**:

- Checkpoints with the same index across all processes form a consistent state.

2. **Piggybacking**:
   - Indexes are piggybacked on application messages to help processes decide when to take forced checkpoints.
3. **Briatico et al. Protocol**:
   - Forces a process to take a checkpoint when it receives a message with an index greater than its local one.

Log-Based Rollback Recovery

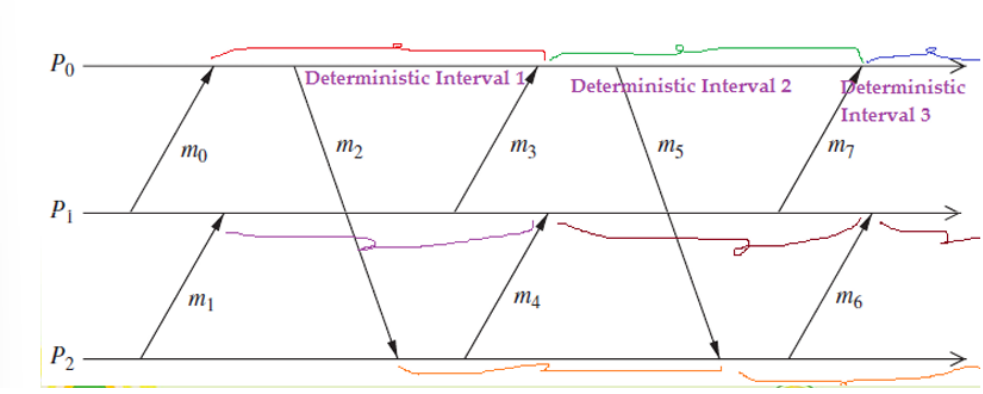
1. Log-Based Rollback Recovery

- This method uses the idea that a process's execution can be divided into two types of intervals:

Deterministic State Intervals: Parts of execution where the outcome is predictable.

Non-Deterministic Events: Parts where the outcome depends on external factors, making them unpredictable.

- **Key Points**:
  - **Send events** are deterministic: The process behaves the same way every time the event is repeated.
  - **Receive and internal events** are non-deterministic: The behavior can vary depending on external factors (like the order of incoming messages).



- **Non-Determinism**:

- Means the same input might lead to different outcomes depending on the context.
- For example, receiving a message at a different time could affect the result.
- **Deterministic execution**, in contrast, ensures the same result for the same input and context.

2. How Log-Based Rollback Recovery Works

Before Failure (Normal Operation):

- Processes keep a log of non-deterministic events in **stable storage** (permanent memory).
- These logs include all the necessary information (determinants) to replay the events later.
- Processes also take periodic **checkpoints** (saved states) to reduce the amount of rollback needed during recovery.

After Failure (Recovery Process):

a. When a process fails, it restarts from its last **checkpoint**.

b. It uses the **log of determinants** to re-execute the non-deterministic events **exactly as they happened before the failure**.

Limitation: If the determinant for a non-deterministic event is missing, the recovery can only proceed up to the point **before that event**.

2. No-Orphans Consistency Condition

What It Means:

- "Orphan processes" are processes that depend on events that **cannot be regenerated** during recovery (because their information is lost).
- To avoid orphan processes, this condition ensures:
  - Any process that depends on a non-deterministic event **e** must either:

1. Have a copy of **e's determinant** in its memory, OR
2. Ensure that **e's determinant** is stored in **stable storage**.

Key Terms:

1. **Non-Deterministic Event (e)**:
    - An event whose outcome depends on external factors, like receiving a message.
2. **Depend(e)**:
    - The set of processes that are affected by event **e**. This includes:
        - The process where **e** occurred.
        - Other processes that depend on the result of **e**.
3. **Log(e)**:
    - The set of processes that have stored a copy of **e's determinant** in their memory.
4. **Stable(e)**:
    - A condition that is true if **e's determinant** is saved in **stable storage**.

3. Always-No-Orphans Condition

- If **Stable(e)** is false (the determinant isn't stored permanently), then all processes that depend on **e** must have a copy of its determinant in their memory (**Depend(e) ⊆ Log(e)**).
- This ensures that recovery can still proceed without creating orphan processes.

$$\forall(e) : \neg Stable(e) \Rightarrow Depend(e) \subseteq Log(e)$$

Explanation:

- If a process depends on a non-deterministic event, it must be able to reproduce it during recovery. This is only possible if the event is either:
    1. Stored permanently in stable storage, OR
    2. Still accessible in the memory of dependent processes.

4. Types of Log-Based Rollback Recovery

1. **Pessimistic Logging**

- Assumes failures can happen at any time.
- Logs determinants synchronously (before the event completes).

2. **Optimistic Logging**
   - Assumes failures are rare.
   - Logs determinants asynchronously (after the event completes).
3. **Causal Logging**
   - Ensures that logs capture the causal relationships between events.

5. Differences Between the Types of Log-Based Rollback Recovery

- They differ in:
  - **Failure-free performance** (how efficient they are when there's no failure).
  - **Output latency** (time taken to commit results).
  - **Recovery complexity** and **garbage collection overhead**.
  - **Impact on surviving processes** (rolling them back).

6. Pessimistic Logging

- **Key Idea**:
  - Logs all determinants to stable storage **before** events can affect computation, ensuring safe recovery even if failure occurs immediately.
- **Protocol**:
  - Implements **Synchronous Logging**, which ensures:
    - If a determinant is not logged (**¬Stable(e)**), then no process depends on it (**|Depend(e)| = 0**).
- **Steps**:

1. Determinants are logged and checkpoints are taken periodically.
2. In case of failure, the process restarts from its most recent checkpoint.

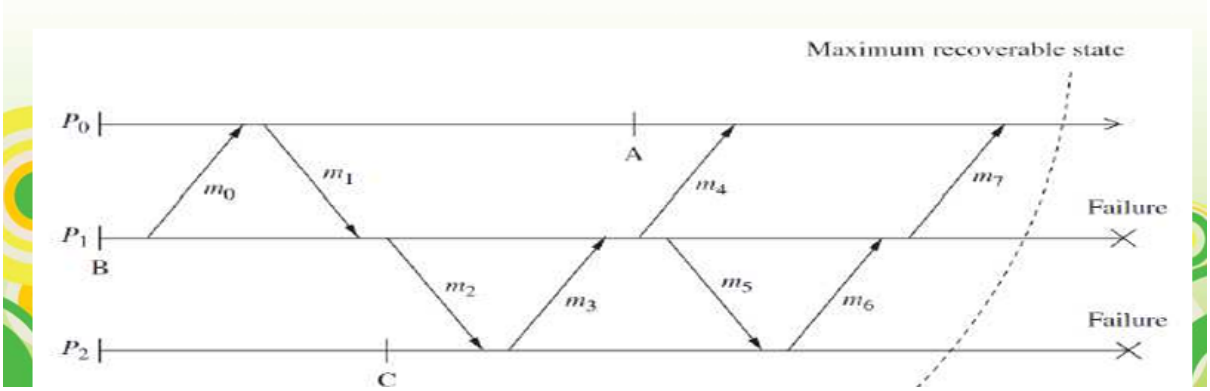
3. Determinants are replayed to recreate pre-failure execution.

Challenges of Pessimistic Logging:

1. **Performance Overhead**:
    - Synchronous logging can slow down the system.
2. **Solutions**:
    - Use faster memory (non-volatile).
    - Minimize failures requiring rollback.
    - Use **Sender-Based Message Logging (SBML)**:
        - Keeps determinants in the sender's volatile memory to reduce overhead.

Example

- During failure-free operation the logs of processes ***P0, P1, and P2 contain the determinants*** *needed* to **replay messages *m0, m4, m7, m1, m3, m6, and m2, m5***, *respectively.*
- *Suppose* processes *P1 and P2 fail as shown, restart from* ***checkpoints B and C***, *and* **roll forward using their determinant logs** to deliver again the same sequence of messages as in the pre-failure execution.
- This guarantees that *P1 and P2* will repeat exactly their pre-failure execution and re-send the same messages.



7. Optimistic Logging

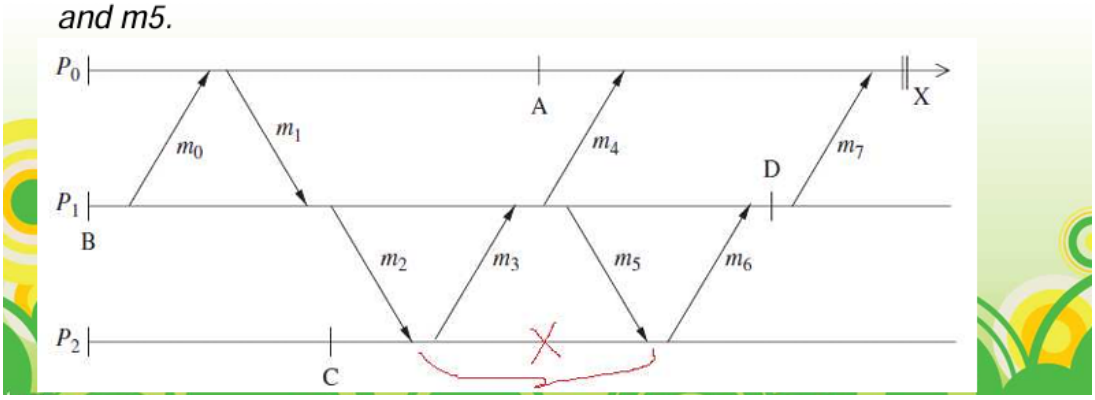
- **Key Idea**:
  - Logs determinants asynchronously (not immediately) to stable storage.
  - Optimistic protocols do not implement the always-no-orphans condition.
- **Steps**:

1. Determinants are stored in a **volatile log** and periodically flushed to stable storage.
2. Assumes logging will complete before failure occurs.
3. If failure happens:
   - Determinants in the volatile log are lost, and affected processes must rollback.

- **Advantages**:
  - **Less Overhead** during normal execution.
- **Disadvantages**:
  - Recovery is more complicated.
  - Slower output commit and higher garbage collection costs.

Example

- Suppose process *P2 fails* before the determinant for *m5 is logged to the stable storage.*
- *Process P1 then* becomes an orphan process and must roll back to undo the effects of receiving the orphan message *m6.*
- *The rollback of P1 further forces P0 to roll back to* undo the effects of receiving message *m7.*
- For example, if process *P0 needs to commit output at state X, it must* log messages *m4 and m7 to the stable storage and ask P2 to log m2 and m5.*



8. Differences between Optimistic and Pessimistic Logging

- To perform rollbacks correctly, **optimistic logging protocols** track **causal dependencies** during failure-free execution.
- Upon a failure, the **dependency information** is used to calculate and recover the latest global state of the pre-failure execution in which **no process is in an orphan**.
- **Pessimistic protocols** need only keep the **most recent checkpoint** of each process.
- **Optimistic protocols** may need to keep **multiple checkpoints** for each process.

9. Causal Logging

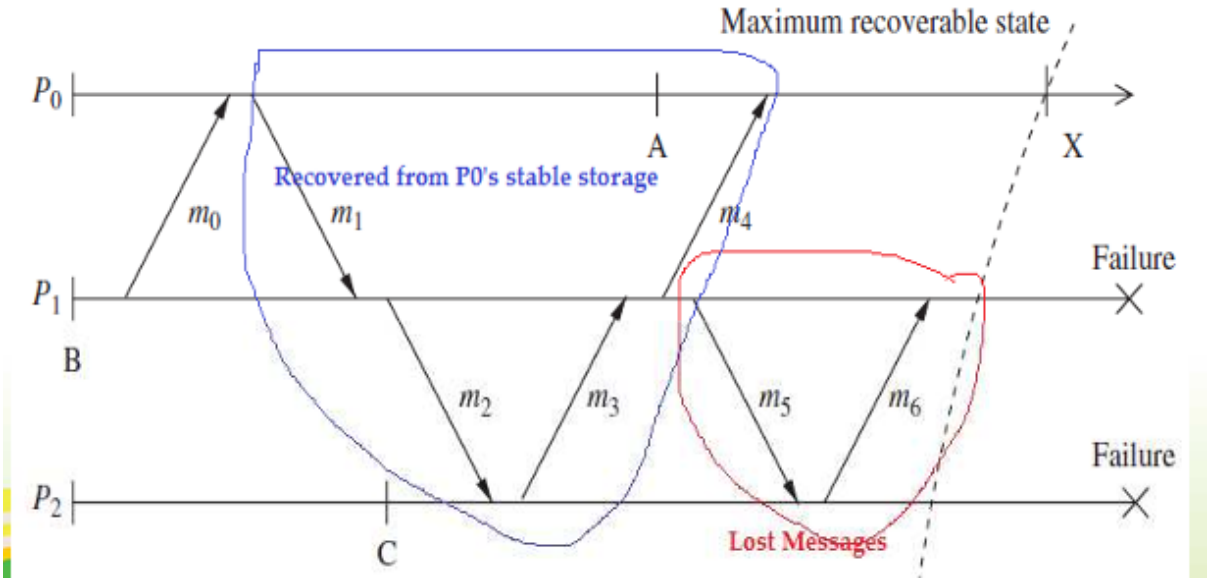
1. **Definition**:
    - Combines the **advantages of pessimistic and optimistic logging**.
    - Trades simplicity for a more **complex recovery protocol**.
2. **Key Features**:
    - **Like optimistic logging**:
        - Does not require **synchronous access** to stable storage except during output commit.
    - **Like pessimistic logging**:
        - Allows processes to **commit output independently** and ensures there are **no orphans**.
3. **Rollback Limitation**:
    - Limits the rollback of failed processes to the **most recent checkpoint on stable storage**.
    - This reduces:
        - **Storage overhead**.
        - The **amount of lost work**.

4. **Guarantee**:
    - Ensures that the **always-no-orphans property** holds.

How Causal Logging Works

1. **Event Tracking**:
    - Each process maintains information about all events that have **causally affected its state**.
2. **Failure Protection**:
    - The recorded information protects processes from failures of other processes.
    - Makes the process's state **recoverable** by logging all necessary information locally.
3. **Output Commit**:
    - Processes can **commit output independently** without requiring global synchronization.

Example



- Messages m5 and m6 are likely to be lost on the failures of P1 and P2 at the indicated instants.
- Process P0 at state X will have logged the determinants of the nondeterministic events that causally precede its state according to Lamport's happened-before relation.
- These events consist of the delivery of messages m0, m1, m2, m3, and m4. The determinant of each of these non-deterministic events is either logged on the stable storage or is available in the volatile log of process P0.
- The determinant of each of these events contains the order in which its original receiver delivered the corresponding message.
- The message sender, as in sender-based message logging, logs the message content.
- Process P0 will be able to "guide" the recovery of P1 and P2 since it knows the order in which P1 should replay messages m1 and m3 to reach the state from which P1 sent message m4.
- Similarly, P0 has the order in which P2 should replay message m2 to be consistent with both P0 and P1. The content of these messages is obtained from the sender log of P0 or regenerated deterministically during the recovery of P1 and P2. Note that information about messages m5 and m6 is lost due to failures.