Failures & Recovery Approaches in Distributed Systems

Dr. Rajiv Misra
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

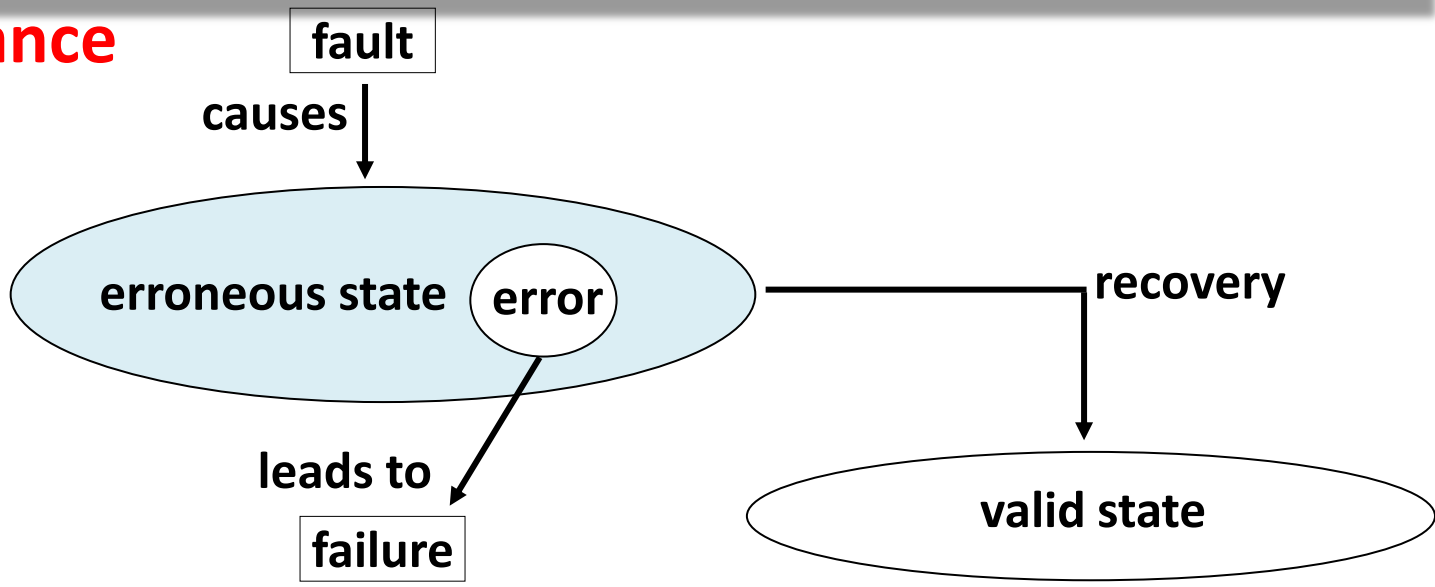
Content of this Lecture:

- In this lecture, we will discuss about basic fundamentals and underlying concepts of '**Failure and Rollback Recovery**' and
- Also discuss **Checkpointing & Rollback Recovery Schemes** in distributed systems i.e.

(i) Checkpoint based and
(ii) Log based

Introduction

- **Fault Tolerance**



- An error is a manifestation of a fault that can lead to a failure.
- **Failure Recovery:**
  - Backward recovery- Restore system state to a previous error-free state
    - **operation-based** (do-undo-redo logs)
    - **state-based** (checkpointing/logging)
  - Forward recovery- Repair the erroneous part of the system state

Introduction

- **Rollback recovery protocols**
  - **Restore the system** back to 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
- **Checkpoints**
  - **The saved states** of a process
- **Why is rollback recovery of distributed systems complicated?**
  - Messages induce inter-process dependencies during failure-free operation
- **Rollback propagation**
  - The dependencies may force some of the processes that did not fail to roll back
  - This phenomenon is called "***domino effect***"

Contd…

- If each process takes its checkpoints independently, then the system **can not avoid the domino effect**
- this scheme is called **independent or uncoordinated checkpointing**
- Techniques **that avoid domino effect**
  - **Coordinated checkpointing rollback recovery**
    - processes coordinate their checkpoints to form a system-wide consistent state
  - **Communication-induced** checkpointing rollback recovery
    - **forces each process to take checkpoints** based on information piggybacked on the application
  - **Log-based rollback recovery**
    - combines checkpointing with logging of non-deterministic events relies on piecewise deterministic (PWD) assumption

Preliminaries

A local checkpoint

- All processes save their **local states** at certain instants of time
- A **local check point** is a **snapshot of the state** of the process at a given instance

Assumptions:

-A process stores all local checkpoints on the stable storage

-A process is able to roll back to any of its existing local checkpoints

- $C_{i,k}$

  -The kth local checkpoint at process P_i

- $C_{i,0}$

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

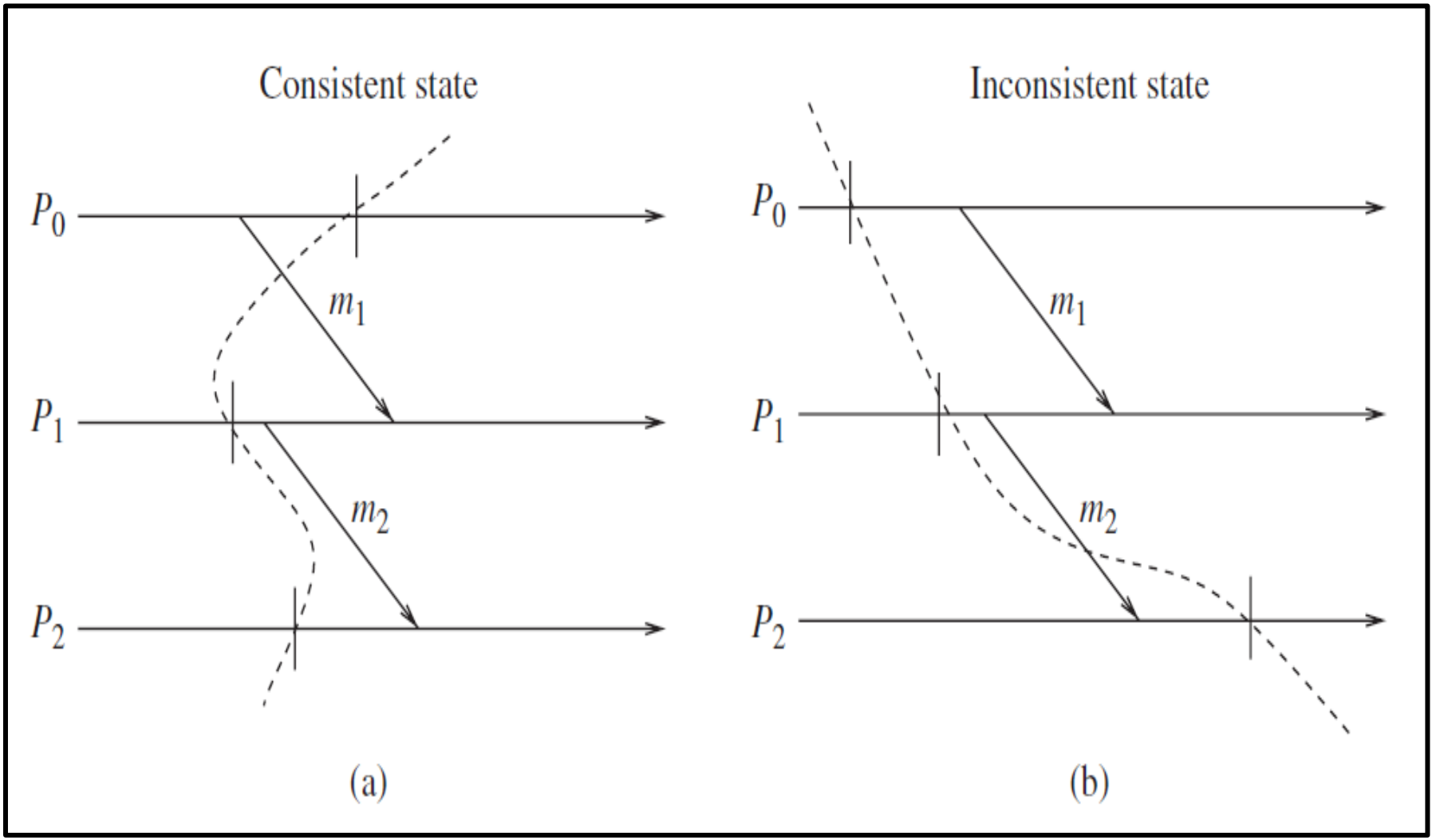
Consistent states

- **A global state of a distributed system**
  - a collection of the individual states of all participating processes and the states of the communication channels

- **Consistent global state**
  - a global state that may occur during a failure-free execution of distribution of distributed computation
  - if a process's state reflects a message receipt, then the state of the corresponding sender must reflect the sending of the message

- **A global checkpoint**
  - a set of local checkpoints, one from each process
- **A consistent global checkpoint**
  - a global checkpoint such that no message is sent by a process after taking its local point that is received by another process before taking its checkpoint

Consistent States: Examples



Interactions with outside world

- A distributed system often interacts with the outside world to receive input data or deliver the outcome of a computation

- **Outside World Process (OWP)**
  - a special process that interacts with the rest of the system through message passing

- **A common approach**
  - save each input message on the stable storage before allowing the application program to process it
- **Symbol "| |"**
  - An interaction with the outside world to deliver the outcome of a computation

Messages

In-transit message

- messages that have been sent but not yet received

Lost messages

- messages whose 'send' is done but 'receive' is undone due to rollback

Delayed messages

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

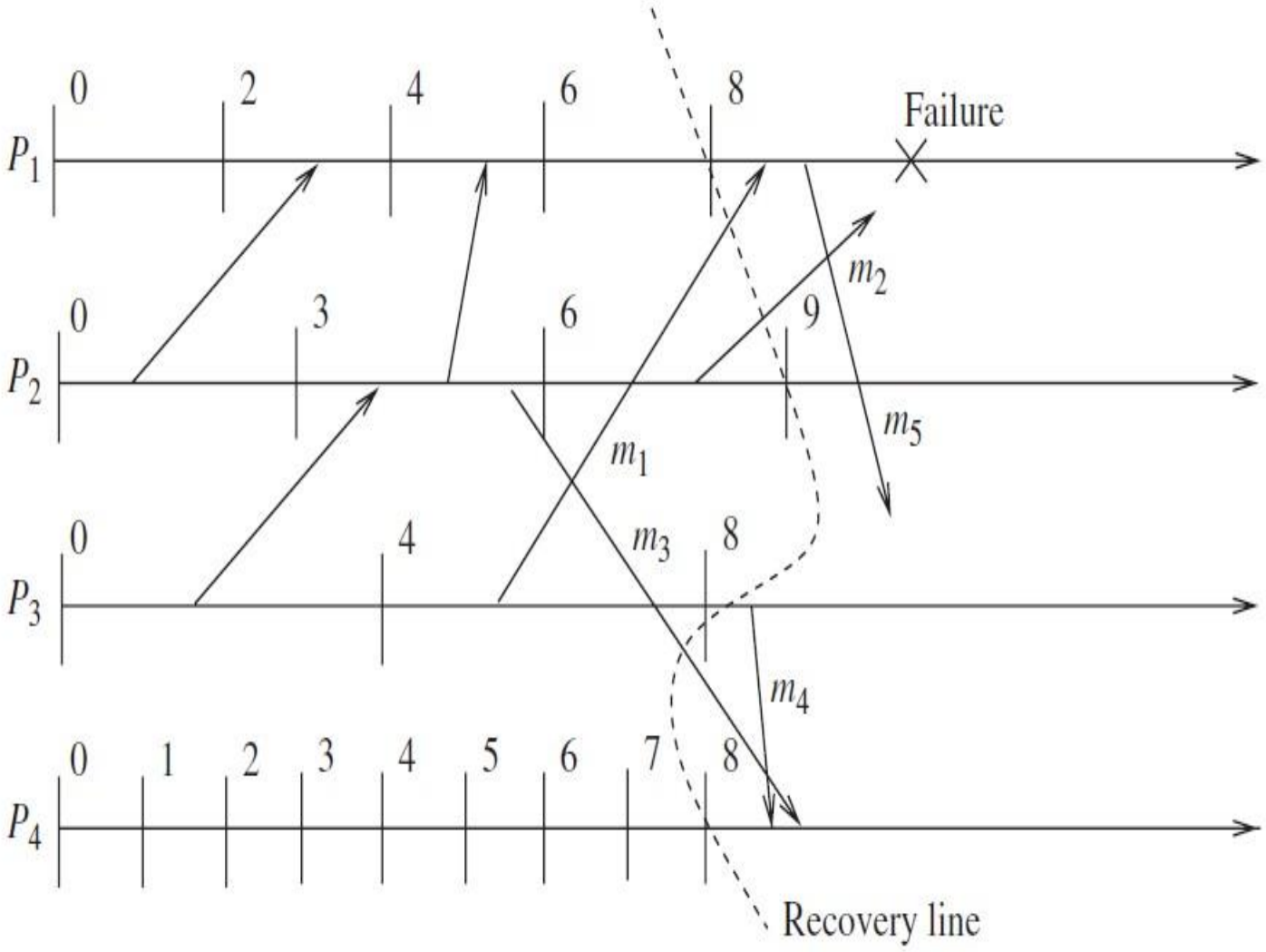
Orphan messages

- messages with 'receive' recorded but message 'send' not recorded
- do not arise if processes roll back to a consistent global state

Duplicate messages

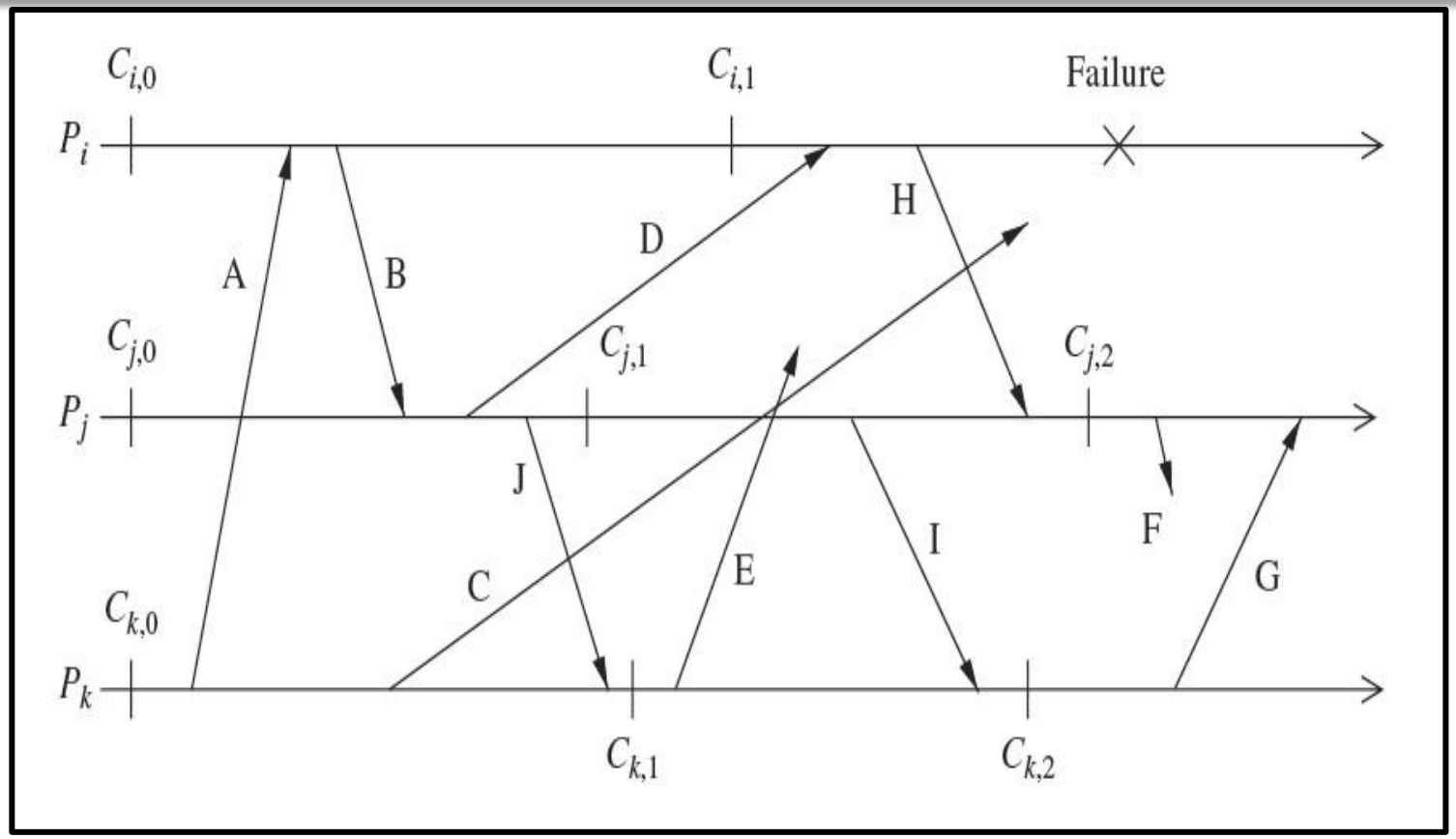
- arise due to message logging and replaying during process recovery

Messages: Example



- **In-transit**
  - m_1, m_2
- **Lost**
  - m_1
- **Delayed**
  - m_1, m_5
- **Orphan**
  - none
- **Duplicated**
  - m_4, m_5

Issues in failure recovery



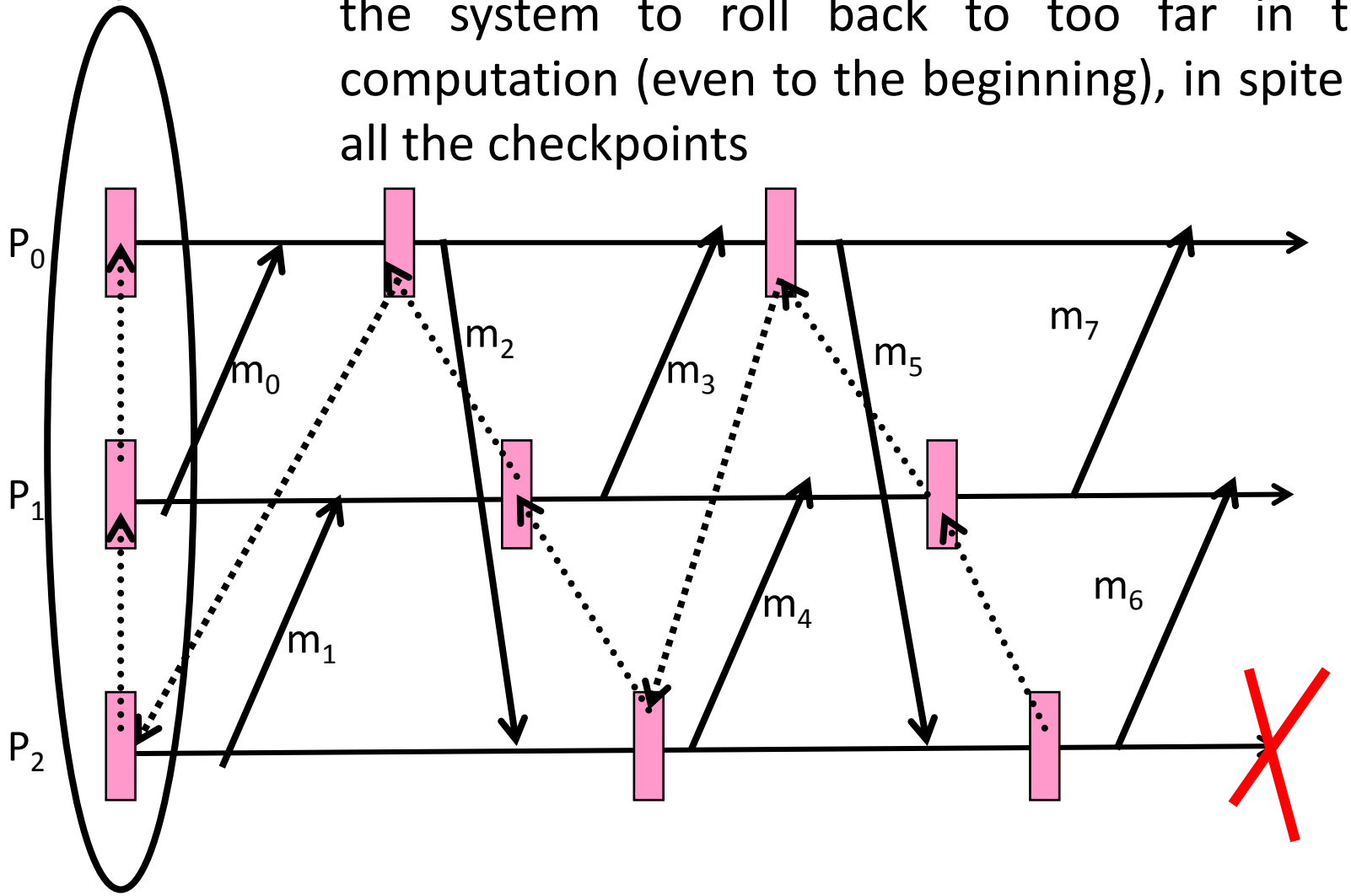
- **Checkpoints :** $\{C_{i,0}, C_{i,1}\}$, $\{C_{j,0}, C_{j,1}, C_{j,2}\}$, and $\{C_{k,0}, C_{k,1}, C_{k,2}\}$
- **Messages :** A -J
- **The restored global consistent state :** $\{C_{i,1}, C_{j,1}, C_{k,1}\}$

Issues in failure recovery

- The rollback of process P_i to checkpoint $C_{i,1}$ created an orphan message H
- Orphan message I is created due to the roll back of process P_j to checkpoint $C_{j,1}$
- Messages C, D, E, and F are potentially problematic
  - Message C: a delayed message
  - Message D: a lost message since the send event for D is recorded in the restored state for P_j, but the receive event has been undone at process P_i
  - Lost messages can be handled by having processes keep a message log of all the sent messages
  - Messages E, F: delayed orphan messages. After resuming execution from their checkpoints, processes will generate both of these messages

Domino effect: example

Domino Effect: Cascaded rollback which causes the system to roll back to too far in the computation (even to the beginning), in spite of all the checkpoints



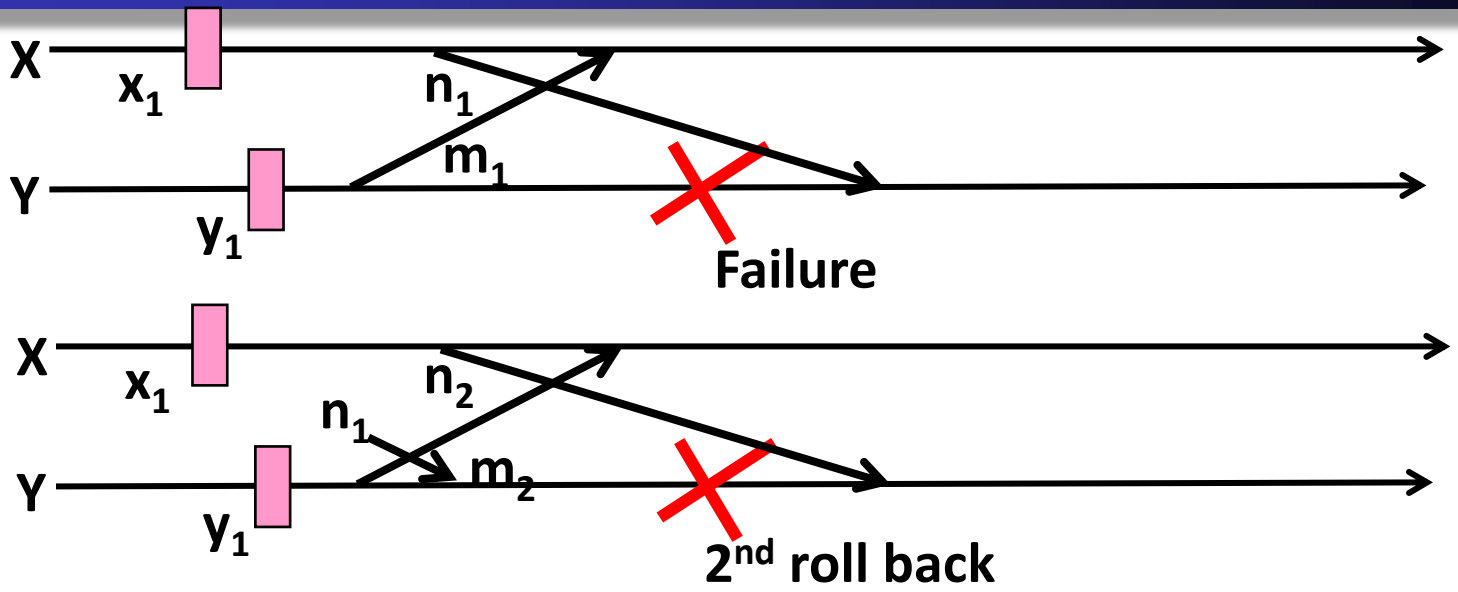
Problem of Livelock

- **Livelock:** case where a single failure can cause an infinite number of rollbacks.

- **The Livelock problem** may arise when a process rolls back to its checkpoint after a failure and requests all the other affected processes also to roll back.

- In such a situation if the roll back mechanism has no synchronization, it may lead to the livelock problem.

Livelock: Example



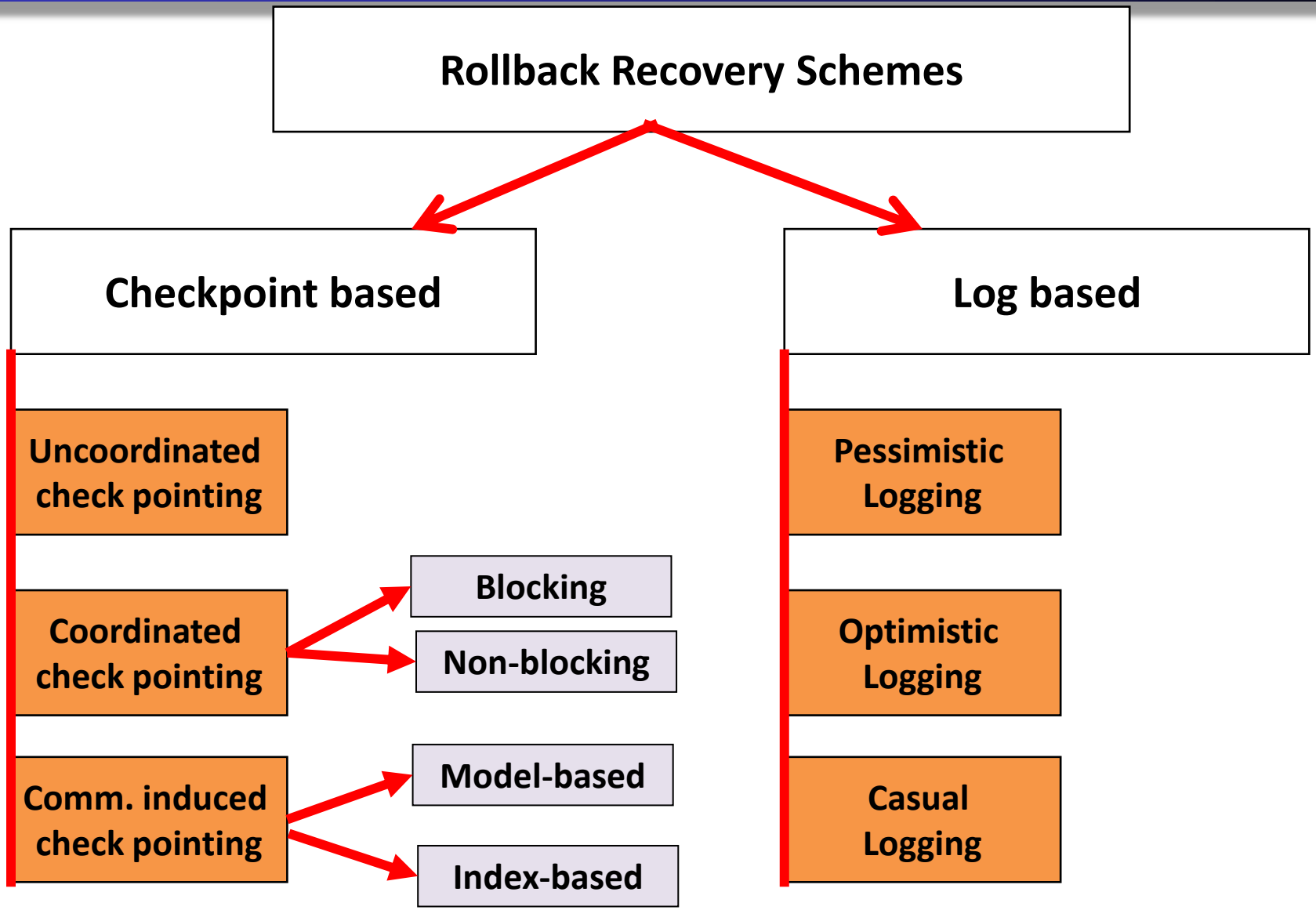
Case-I

- Process Y fails before receiving message 'n1' sent by X
- Y rolled back to y1, no record of sending message 'm1', causing X to rollback to x1

Case-II

- When Y restarts, sends out 'm2' and receives 'n1' (delayed)
- Y has to roll back again, since there is no record of 'n1' being sent
- This cause X to be rolled back again, since it has received 'm2' and there is no record of sending 'm2' in Y
- The above sequence can repeat indefinitely

Different Rollback Recovery Schemes



Checkpoint Based Recovery Schemes

Checkpoint Based Recovery: Overview

1. Uncoordinated Checkpointing: Each process takes its checkpoints independently

2. Coordinated Checkpointing: Process coordinate their checkpoints in order to save a system-wide consistent state. This consistent set of checkpoints can be used to bound the rollback

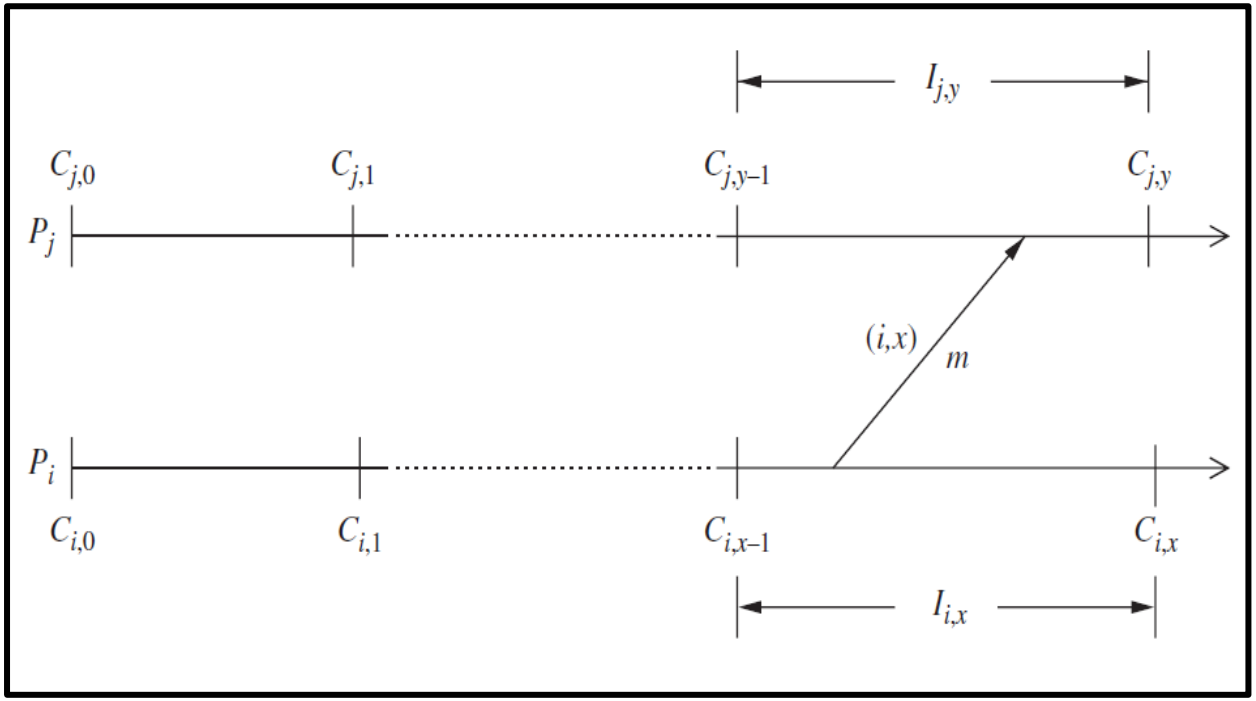
3. Communication-induced Checkpointing: It forces each process to take checkpoints based on information piggybacked on the application messages it receives from other processes.

1. Uncoordinated Checkpointing

- Each process has autonomy in deciding when to take checkpoints
- **Advantages**
  - The **lower runtime overhead** during normal execution
- **Disadvantages**
  - ***Domino effect*** during a recovery
  - **Recovery from a failure is slow** because processes need to iterate to find a consistent set of checkpoints
  - Each process maintains **multiple checkpoints** and periodically invoke a garbage collection algorithm
  - Not suitable for application with frequent output commits
- The processes record the dependencies among their checkpoints caused by message exchange during failure-free operation

Example: Direct dependency tracking technique

- Assume each process P_i starts its execution with an initial
- checkpoint $C_{i,0}$
- $I_{i,x}$: checkpoint interval, interval between $C_{i,x-1}$ and $C_{i,x}$
- When P_j receives a message m during $I_{j,y}$, it records the dependency from $I_{i,x}$ to $I_{j,y}$, which is later saved onto stable storage when P_j takes $C_{j,y}$



2. Coordinated Checkpointing

- **Blocking Checkpointing**
  - After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpointing activity is complete
  - **Disadvantages**
    - The computation is blocked during the checkpointing

- **Non-blocking Checkpointing**
  - The processes need not stop their execution while taking checkpoints
  - A fundamental problem in coordinated checkpointing is to prevent a process from receiving application messages that could make the checkpoint inconsistent.

Example

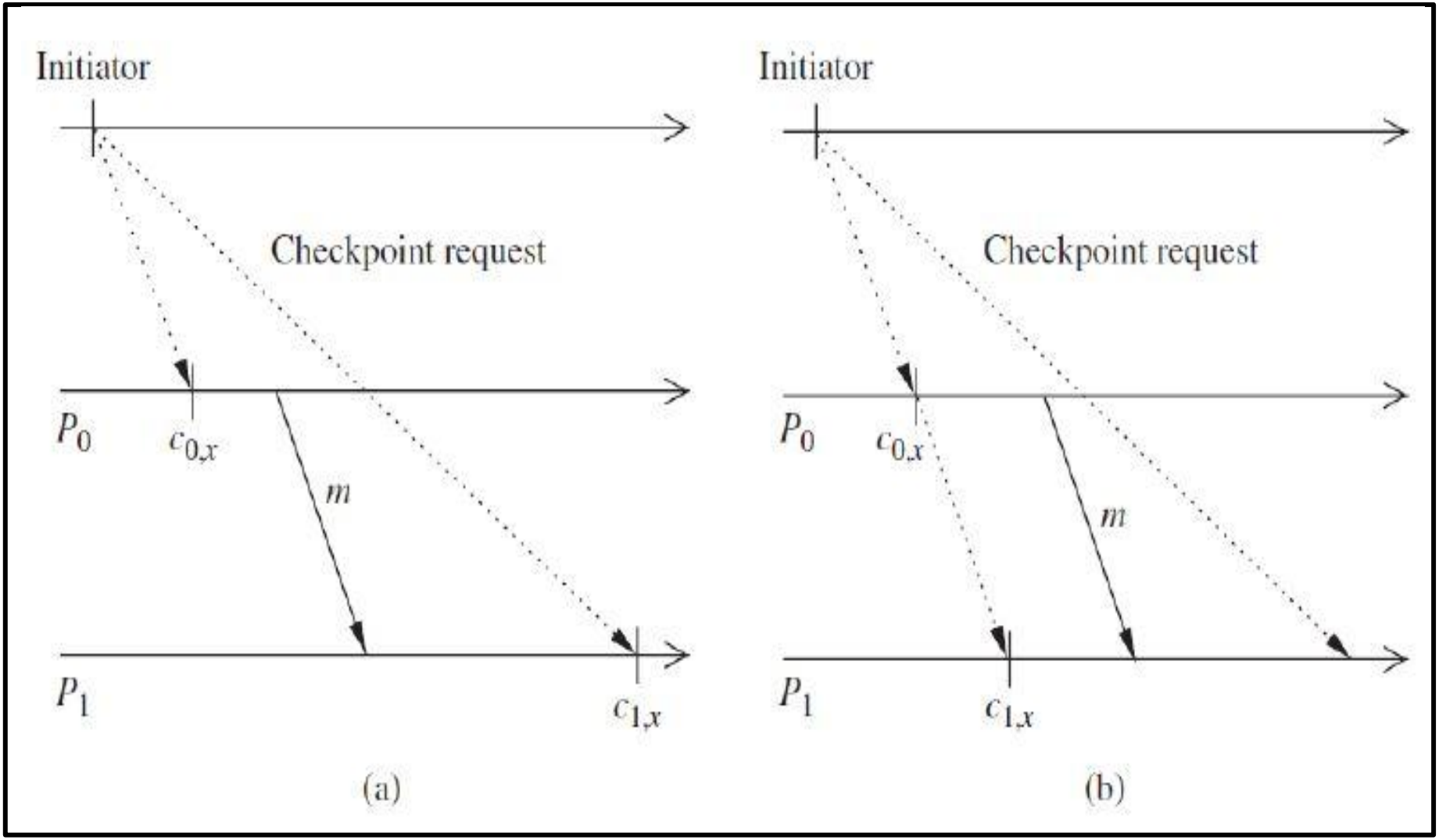
Example (a) : checkpoint inconsistency

- message m is sent by P_0 after receiving a checkpoint request from the checkpoint coordinator
- Assume m reaches P_1 before the checkpoint request
- This situation results in an inconsistent checkpoint since checkpoint
- $C_{1,x}$ shows the receipt of message m from P_0, while checkpoint $C_{0,x}$ does not show m being sent from P_0

Example (b) : a solution with FIFO channels

- If channels are FIFO, this problem can be avoided by preceding the first post-checkpoint message on each channel by a checkpoint request, forcing each process to take a checkpoint before receiving the first post-checkpoint message.

Contd…


Initiator
Checkpoint request
P_0
$c_{0,x}$
m
P_1
$c_{1,x}$
(a)
Initiator
Checkpoint request
P_0
$c_{0,x}$
m
P_1
$c_{1,x}$
(b)

3. Communication-induced Checkpointing

- **Two types of checkpoints**
  - autonomous and forced checkpoints

- 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
- In contrast with coordinated checkpointing, no special coordination messages are exchanged

Contd…

- **Two types of communication-induced checkpointing**
  - model-based checkpointing and
  - index-based checkpointing.

In '***model-based checkpointing',*** the system **maintains checkpoints and communication structures** that prevent the domino effect or achieve some even stronger properties.

In '***index-based checkpointing',*** *the system* ***uses an indexing scheme*** *for the local* and forced checkpoints, such that the checkpoints of the same index at all processes form a consistent state.

Log-based Rollback Recovery Schemes

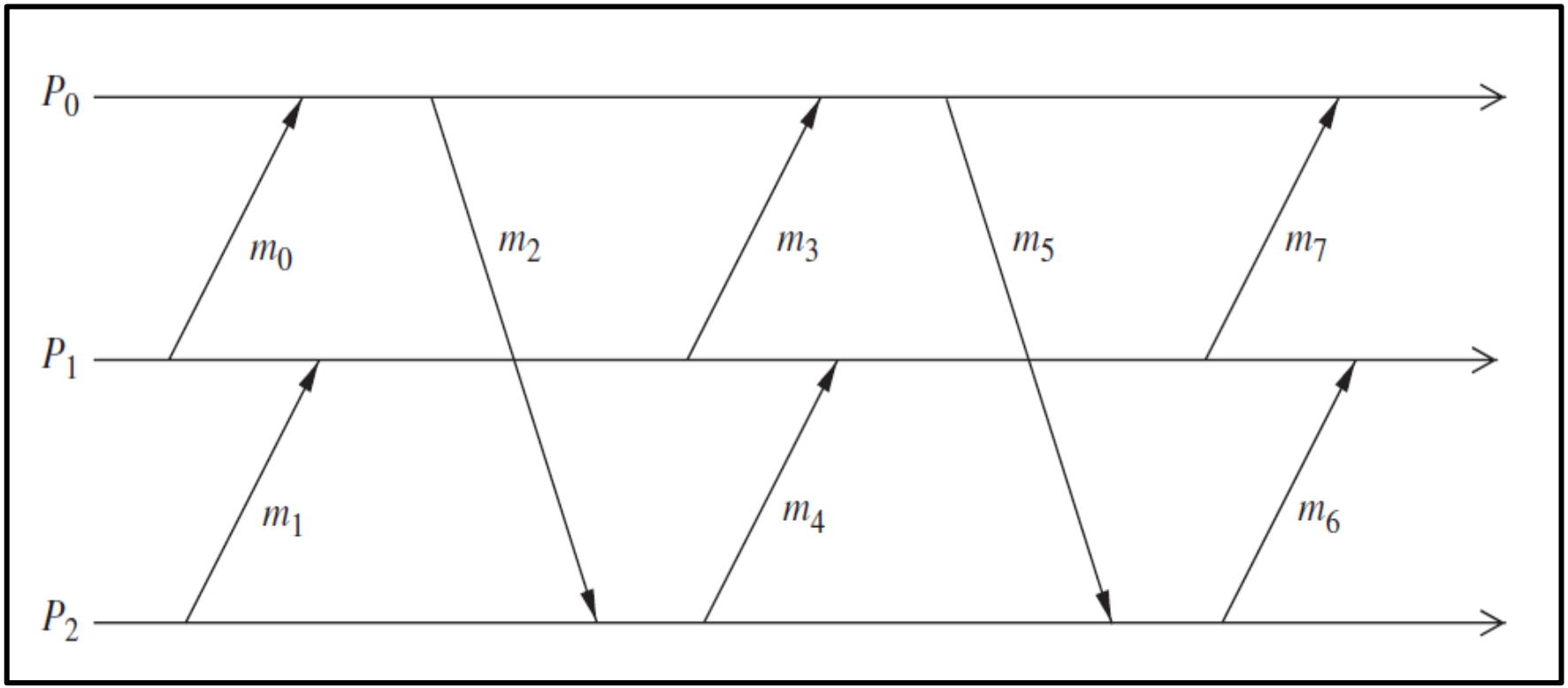
Log-based Rollback Recovery: Overview

- **It combines checkpointing with logging of nondeterministic events**.
- It relies on the **piecewise deterministic (PWD) assumption**, which postulates that all nondeterministic events that a process executes can be identified and that the information necessary to replay each event during recovery can be logged in the event's determinant (all info. necessary to replay the event).
- By logging and replaying the nondeterministic events in their exact original order, a process can deterministically recreate its pre-failure state even if this state has not been checkpointed.
- Log-based rollback recovery is in general attractive for applications that frequently interact with the outside world which consists of input and output logged to stable storage.

Contd…

- A log-based rollback recovery makes **use of deterministic and nondeterministic events in a computation.**
- **Deterministic and Non-deterministic events**
  - Non-deterministic events can be the receipt of a message from another process or an event internal to the process
  - a message send event is ***not*** a non-deterministic event.
  - the execution of process P_0 is a sequence of four deterministic intervals
  - Log-based rollback recovery assumes that all non-deterministic events can be identified and their corresponding determinants can be logged into the stable storage
  - During failure-free operation, each process logs the determinants of all non-deterministic events that it observes onto the stable storage

Deterministic and Non-deterministic events: Example



The execution of process P_0 is a sequence of four deterministic intervals. The first one starts with the creation of the process, while the remaining three start with the receipt of messages m_0, m_3, and m_7, respectively. Send event of message m_2 is uniquely determined by the initial state of P_0 and by the receipt of message m_0, and is therefore not a non-deterministic event.

No-orphans consistency condition

Let *e* be a non-deterministic event that occurs at process *p*

***Depend*(*e*)**

- the set of processes that are affected by a non-deterministic event *e*. This set consists of *p*, and any process whose state depends on the event *e* according to **Lamport's *happened before* relation**

***Log*(*e*)**

- the set of processes that have logged a copy of *e*'s determinant in their volatile memory

***Stable*(*e*)**

- a predicate that is true if *e*'s determinant is logged on the stable storage

***always-no-orphans* condition**

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

Log-based recovery schemes

- Schemes differ in the way the determinants are logged into the stable storage.

1. **Pessimistic Logging:** The application has to block waiting for the determinant of each nondeterministic event to be stored on stable storage before the effects of that event can be seen by other processes or the outside world. It simplifies recovery but hurts the failure-free performance.
2. **Optimistic Logging:** The application does not block, and the determinants are spooled to stable storage asynchronously. It reduces failure free overhead, but complicates recovery.
3. **Casual Logging:** Low failure free overhead and simpler recovery are combined by striking a balance between optimistic and pessimistic logging.

1. Pessimistic Logging

- Pessimistic logging protocols assume that a failure can occur after any non-deterministic event in the computation

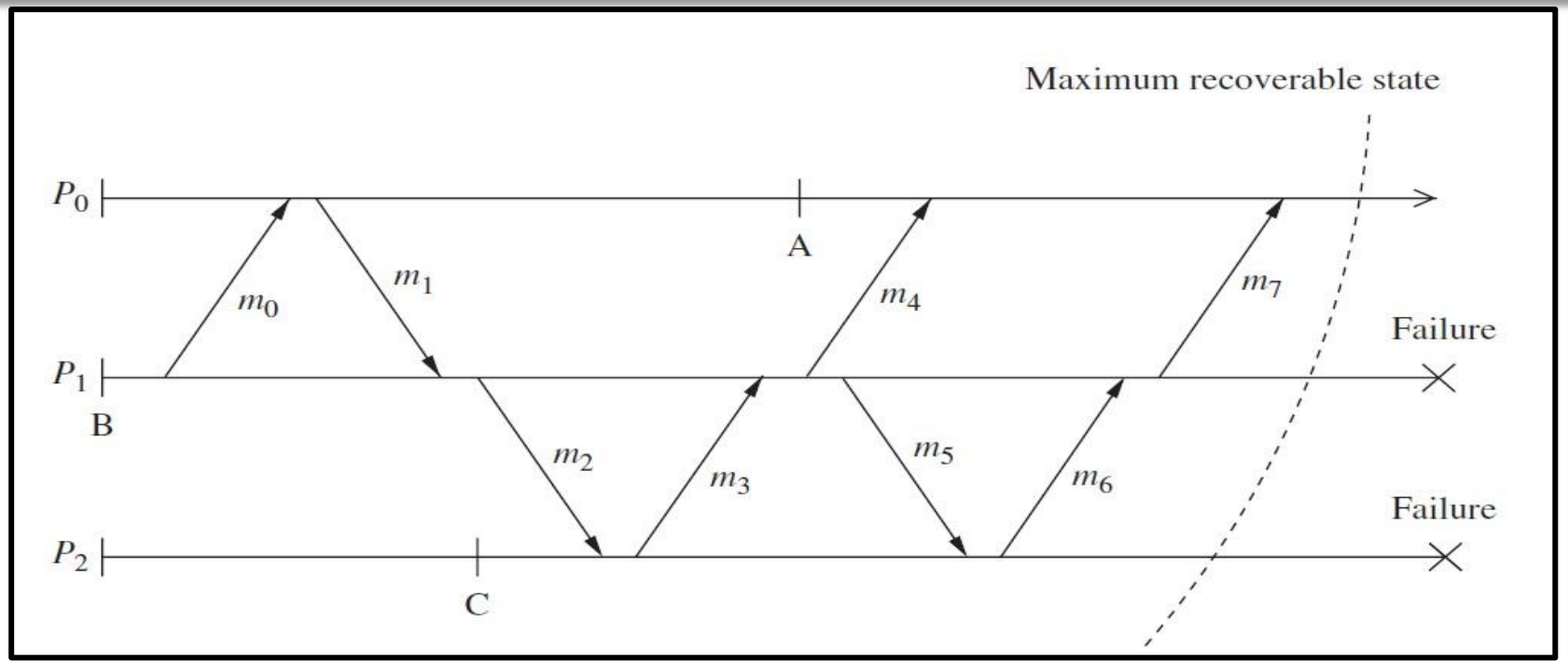
- However, in reality failures are rare

Synchronous logging:

  - $\forall e: \neg Stable(e) \Rightarrow |Depend(e)| = 0$
  - if an event has not been logged on the stable storage, then no process can depend on it.

  - stronger than the always-no-orphans condition

Pessimistic Logging: Example



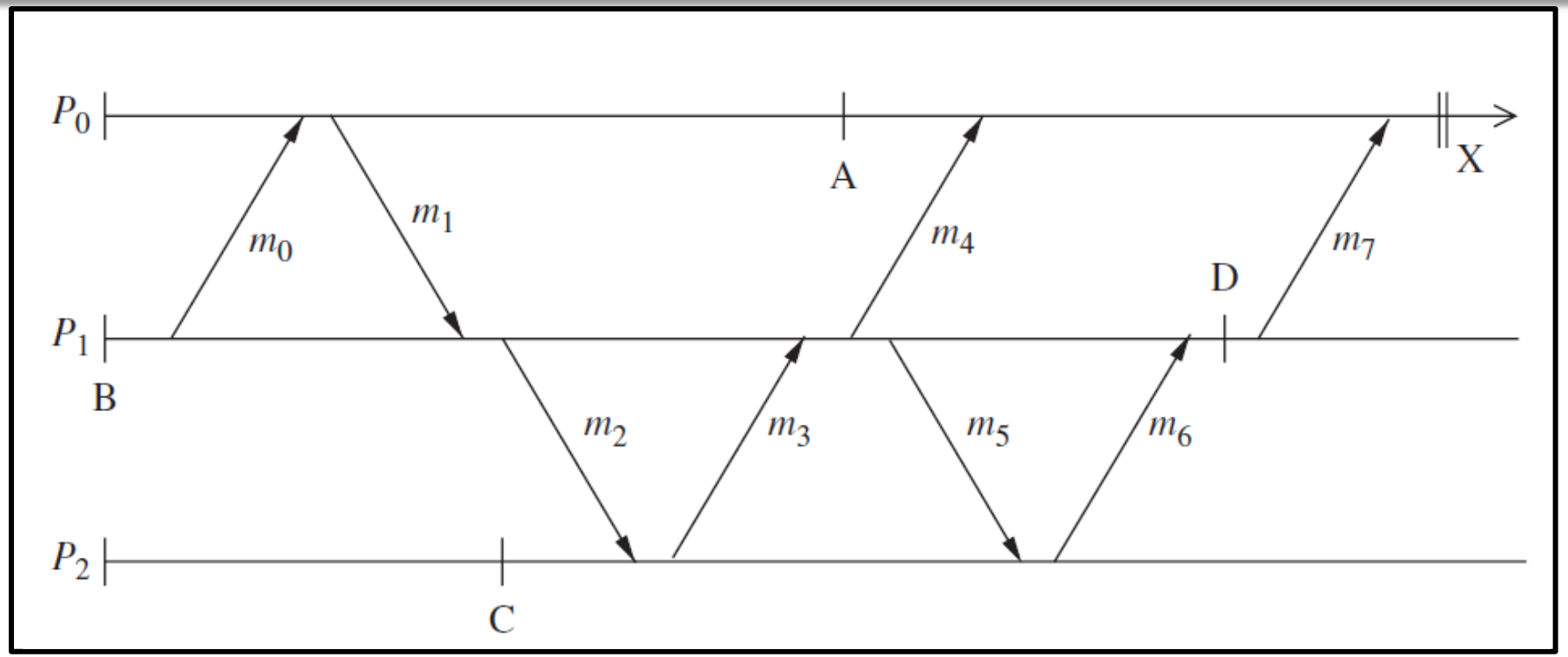
- Suppose processes P_1 and P_2 fail as shown, restart from checkpoints B and C, and roll forward using their determinant logs to deliver again the same sequence of messages as in the pre-failure execution

- Once the recovery is complete, both processes will be consistent with the state of P_0 that includes the receipt of message m_7 from P_1

2. Optimistic Logging

- Processes log determinants asynchronously to the stable storage
- Optimistically assume that logging will be complete before a failure occurs
- Do not implement the ***always-no-orphans*** condition
- To perform rollbacks correctly, optimistic logging protocols track causal dependencies during failure free execution
- Optimistic logging protocols require a non-trivial garbage collect ion scheme
- Pessimistic protocols need only keep the most recent checkpoint of each process, whereas optimistic protocols may need to keep multiple checkpoints for each process

Optimistic Logging: Example

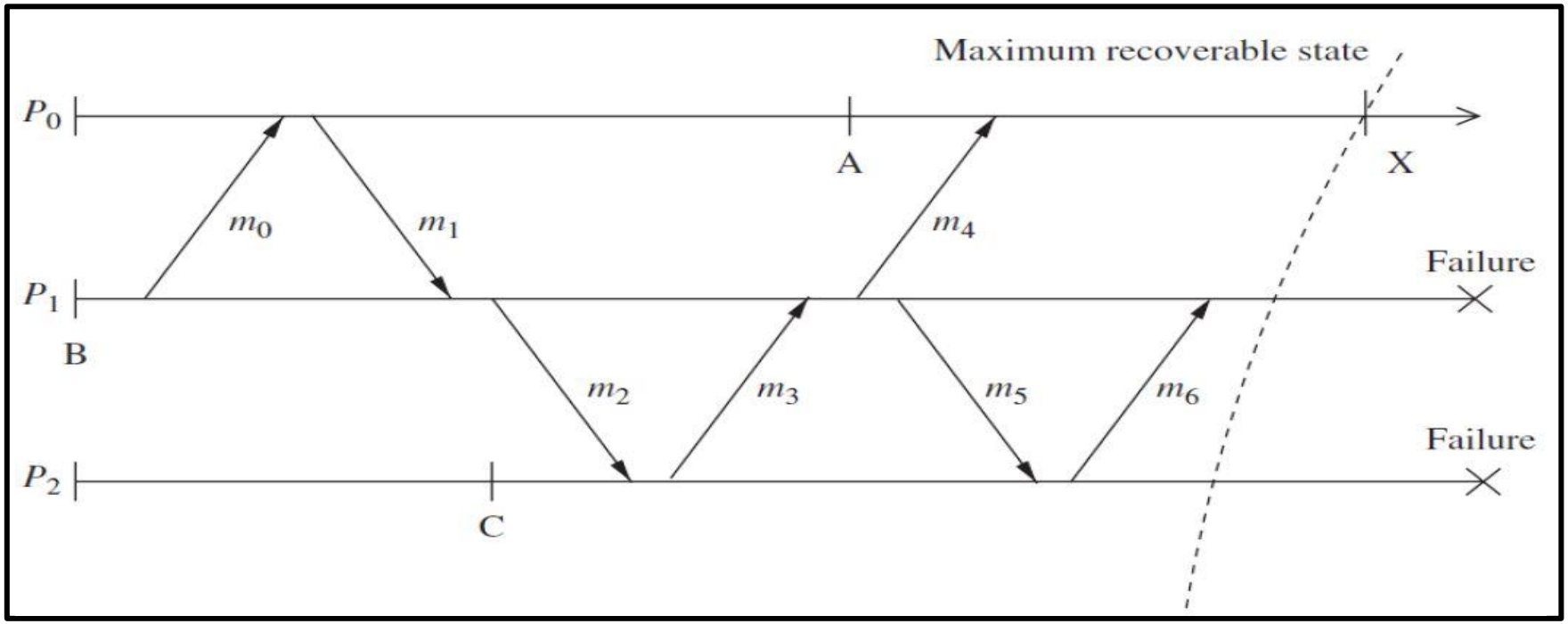


Consider the example shown in figure. Suppose process P_2 fails before the determinant for m_5 is logged to the stable storage. Process P_1 then becomes an orphan process and must roll back to undo the effects of receiving the orphan message m_6. The rollback of P_1 further forces P_0 to roll back to undo the effects of receiving message m_7.

3. Causal Logging

- **Combines the advantages of both pessimistic and optimistic logging** at the expense of a more complex recovery protocol
- **Like optimistic logging**, it does not require synchronous access to the stable storage except during output commit
- **Like pessimistic logging**, it allows each process to commit output independently and never creates orphans, thus isolating processes from the effects of failures at other processes
- Make sure that the always-no-orphans property holds
- Each process maintains information about all the events that have causally affected its state

Causal Logging: Example



Messages m_5 and m_6 are likely to be lost on the failures of P_1 and P_2 at the indicated instants. Process P_0 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 m_0, m_1, m_2, m_3, and m_4. 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 P_0.

Checkpointing and Recovery Algorithms

Koo-Toueg Coordinated Checkpointing Algorithm

- **Koo and Toueg (1987)** proposed a coordinated checkpointing and recovery technique that takes a consistent set of checkpointing and **avoids 'domino effect' and 'livelock problems'** during the recovery

- It Includes 2 parts:

(i) The checkpointing algorithm and

(ii) The recovery algorithm

Contd…

(i) The Checkpointing Algorithm

- **Assumptions:** FIFO channel, end-to-end protocols, communication failures do not partition the network, single process initiation, no process fails during the execution of the algorithm

- Two kinds of checkpoints: permanent and tentative
  - **Permanent checkpoint:** local checkpoint, part of a consistent global checkpoint
  - **Tentative checkpoint:** temporary checkpoint, become permanent checkpoint when the algorithm terminates successfully

Contd…

Checkpointing Algorithm:

- **2 phases**
  1. The initiating process takes a tentative checkpoint and requests all other processes to take tentative checkpoints. Every process can not send messages after taking tentative checkpoint. All processes will finally have the single same decision: do or discard
  2. All processes will receive the final decision from initiating process and act accordingly
- **Correctness: for 2 reasons**
  - Either all or none of the processes take permanent checkpoint
  - No process sends message after taking permanent checkpoint
- **Optimization:** maybe not all of the processes need to take checkpoints (if not change since the last checkpoint)

Contd…

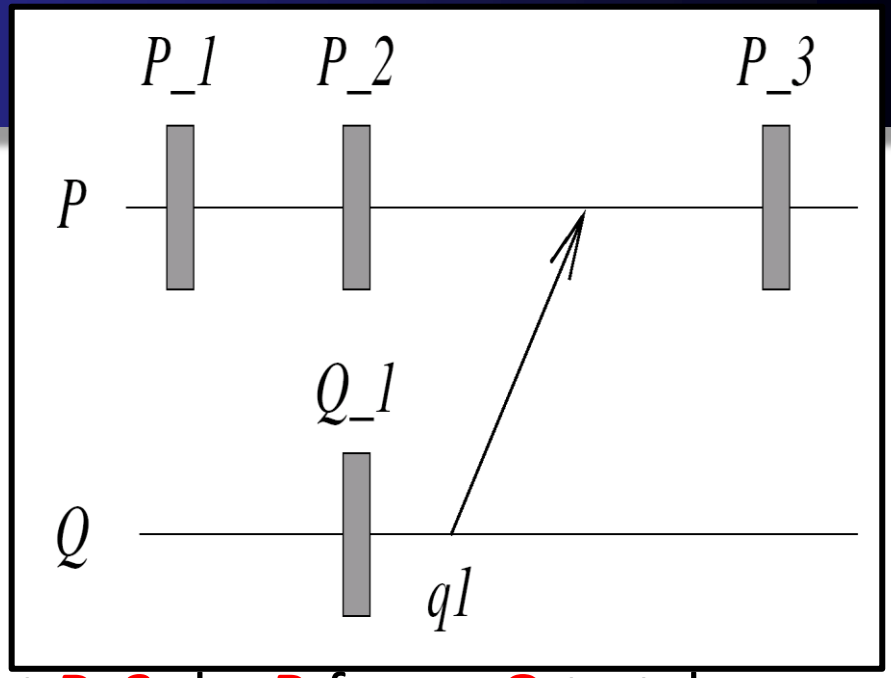
(ii) The Rollback Recovery Algorithm:

- Restore the system state to a consistent state after a failure with assumptions: single initiator, checkpoint and rollback recovery algorithms are not invoked concurrently

- **2 phases**
  1. The initiating process send a message to all other processes and ask for the preferences – restarting to the previous checkpoints. All need to agree about either do or not.
  2. The initiating process send the final decision to all processes, all the processes act accordingly after receiving the final decision.

Example

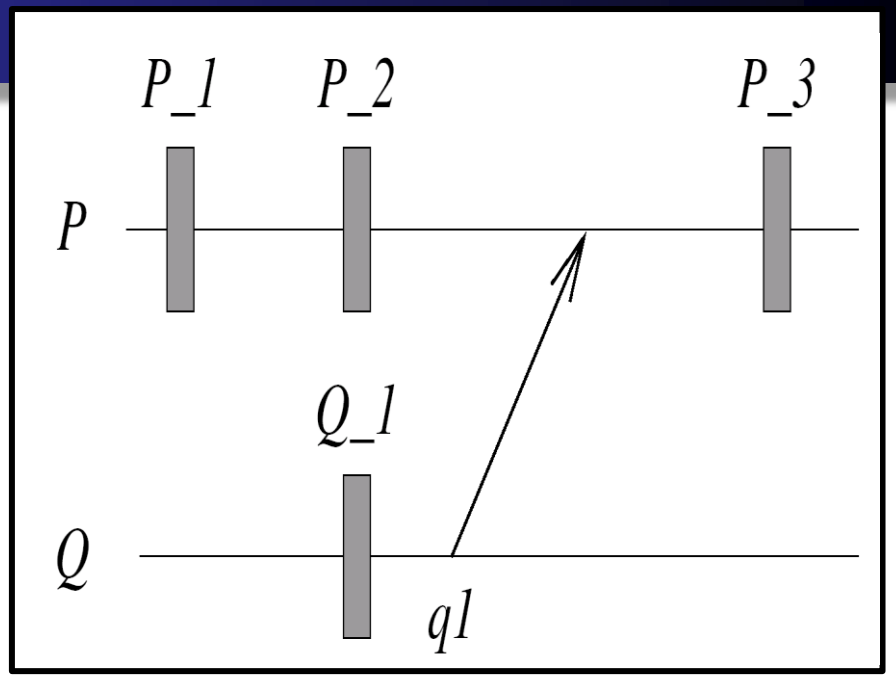
Suppose **P** wants to establish a checkpoint at **P_3** This will record that **q1** was received from **Q** - to prevent q1 from being orphaned, Q must checkpoint as well



- Thus, establishing a checkpoint at **P_3** by **P** forces **Q** to take a checkpoint to record that **q1** was sent
- An algorithm for such coordinated checkpointing has two types of checkpoints - tentative and permanent
- **P** first records its current state in a tentative checkpoint, then sends a message to all other processes from whom it has received a message since taking its last checkpoint
- Call the set of such processes Π

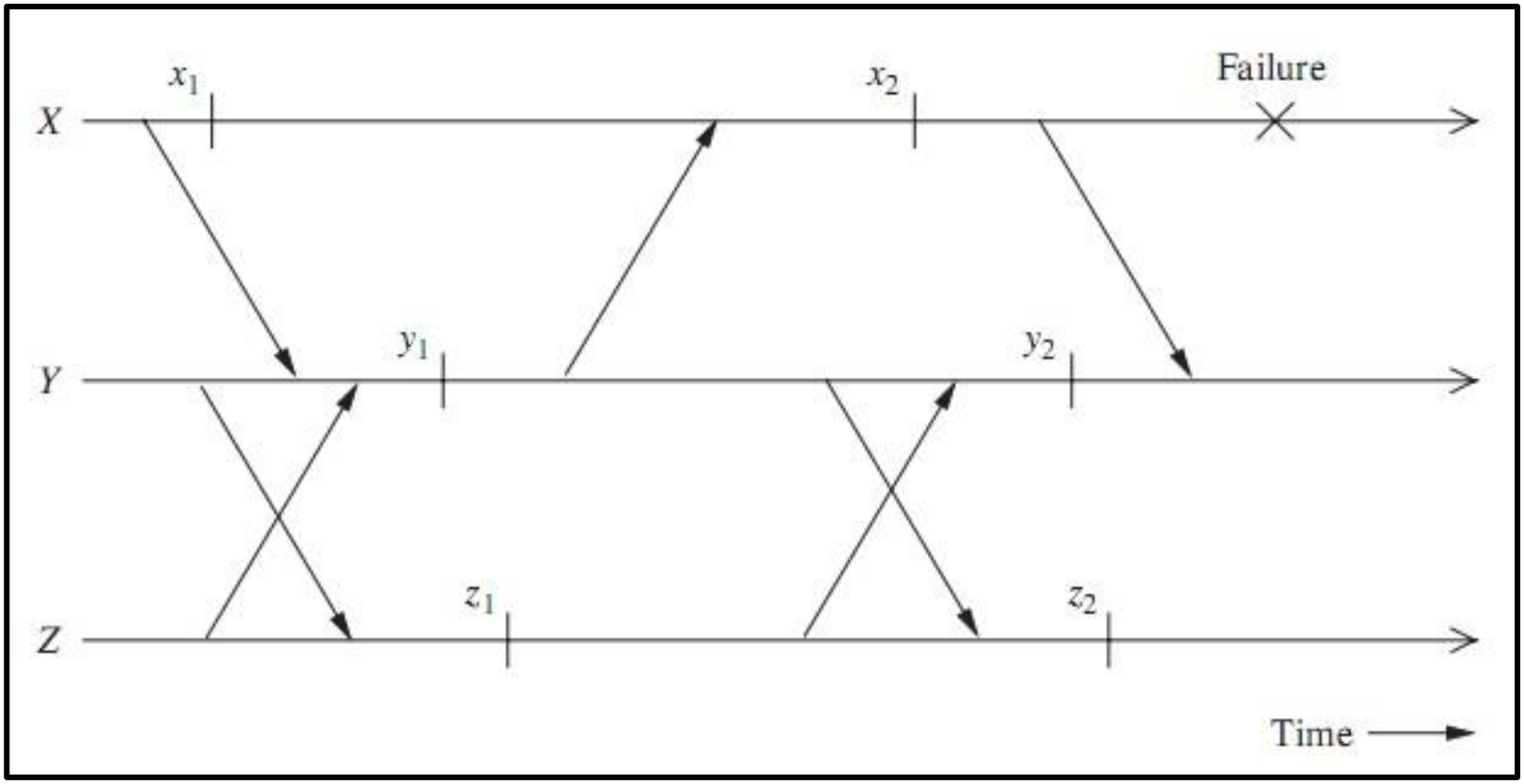
Contd…

The message tells each process in Π (e.g., Q), the last message, **m_qp**, that **P** has received from it before the tentative checkpoint was taken If **m_qp** was not recorded in a checkpoint by **Q**: to prevent **m_qp** from being orphaned, **Q** is asked to take a tentative checkpoint to record sending **m_qp**



- If all processes in Π, that need to, confirm taking a checkpoint as requested, then all tentative checkpoints can be converted to permanent
- If some members of Π, are unable to checkpoint as requested, **P** and all members of Π abandon the tentative checkpoints, and none are made permanent
- This may set off a chain reaction of checkpoints
- Each member of Π can potentially spawn a set of checkpoints among processes in its corresponding set

Contd…



- **Correctness:** resume from a consistent state
- **Optimization:** may not to recover all, since some of the processes did not change anything

Other Algorithms for Checkpointing and Recovery

Algorithm	Basic Idea
Juang–Venkatesan (1991) algorithm for asynchronous checkpointing and recovery	Since the algorithm is based on asynchronous checkpointing, the main issue in the recovery is to find a consistent set of checkpoints to which the system can be restored. The recovery algorithm achieves this by making each processor keep track of both the number of messages it has sent to other processors as well as the number of messages it has received from other processors.
Manivannan–Singhal (1996) quasi-synchronous checkpointing algorithm	The Manivannan–Singhal quasi-synchronous checkpointing algorithm Improves the performance by eliminating useless checkpoints. The algorithm is based on communication-induced checkpointing, where each process takes basic checkpoints asynchronously and independently, and in addition, to prevent useless checkpoints, processes take forced checkpoints upon the reception of messages with a control variable.
Peterson–Kearns (1993) algorithm based on vector time	The Peterson–Kearns checkpointing and recovery protocol is based on the optimistic rollback. Vector time is used to capture causality to identify events and messages that become orphans when a failed process rolls back.
Helary–Mostefaoui–Netzer–Raynal (2000, 1997) communication-induced protocol	The Helary–Mostefaoui–Netzer–Raynal communication-induced checkpointing protocol prevents useless checkpoints and does it efficiently. To prevent useless checkpoints, some coordination is required in taking local checkpoints.

Conclusion

- **Rollback recovery achieves fault tolerance** by periodically saving the state of a process during the failure-free execution, and restarting from a saved state on a failure to reduce the amount of lost computation.

- There are three basic approaches for checkpointing and failure recovery: (**i) uncoordinated, (ii) coordinated, and (iii) communication induced checkpointing** and for **log based: (i) Pessimistic, (ii) Optimistic and (iii) Casual Logging**

- Over the last two decades, **checkpointing and failure recovery** has been a very active area of research and several checkpointing and failure recovery algorithms have been proposed. In this lecture, we described '**Koo-Toueg Coordinated Checkpointing Algorithm'** and given an overview of other algorithms.