

Parallel and Distributed Programming 11/13

コンピュータ科学専攻 M1
大住裕之

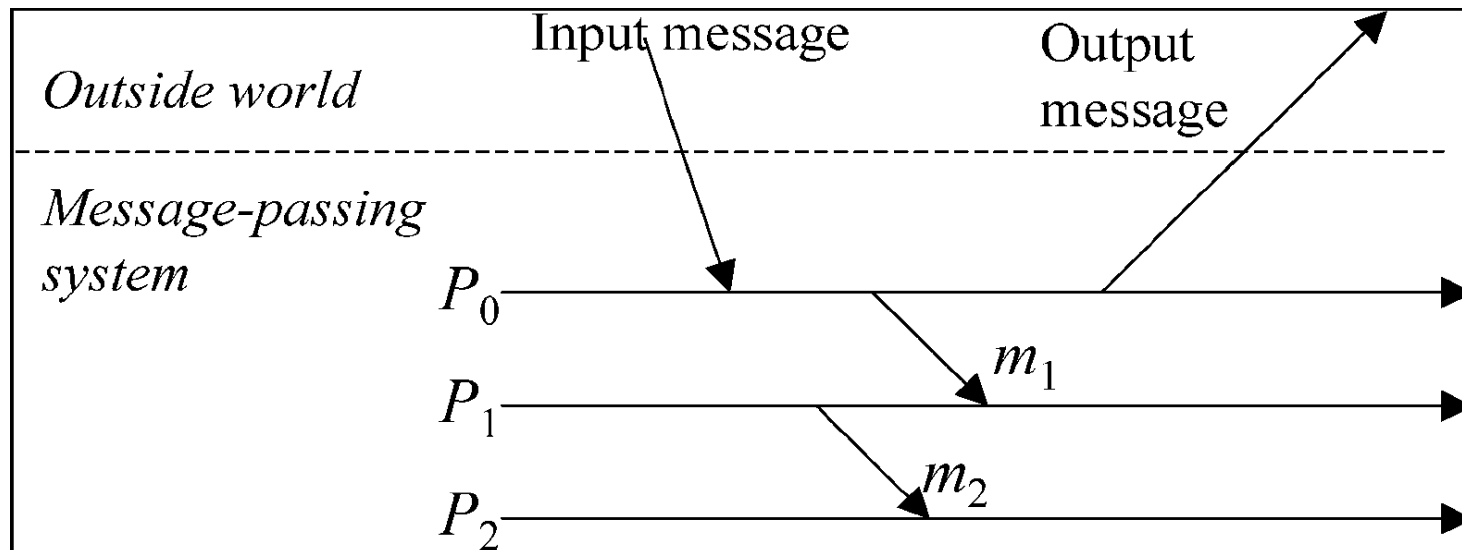
osumi@yl.is.s.u-tokyo.ac.jp

A Survey of Rollback-Recovery Protocols in Message-Passing Systems

E. N. (Mootaz) Elnozahy,
Lorenzo Alvisi, Yi-Min Wang,
David B. Johnson

System Model

- Fixed number(N) of processes
- Processes communicate only through messages



PWD Assumption

- A process execution is a sequence of *state intervals*, each started by a nondeterministic event
- **Piecewise deterministic assumption**
 - The system can detect and capture sufficient information about the nondeterministic event that initiate the state intervals

Recovery

- If a process fails, it loses its volatile state and stops execution
- Processes can use a **stable storage** for recovery
 - State information saved on the storage during failure-free execution

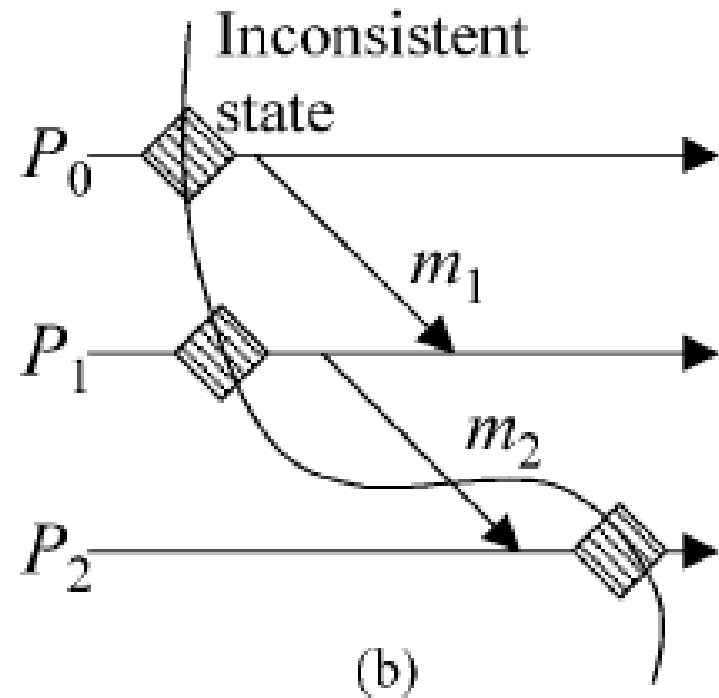
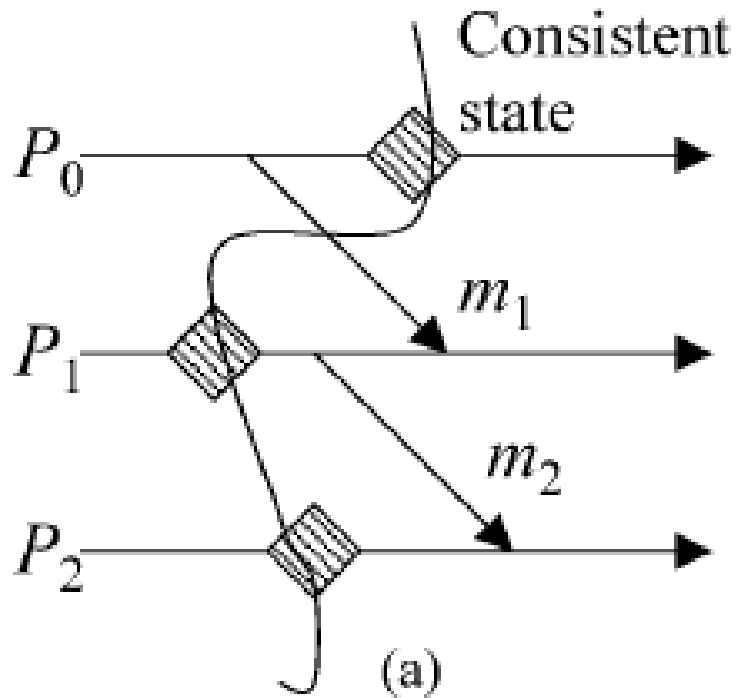
Generic Correctness Condition

A system recovers correctly if its internal state is *consistent* with the observable behavior of the system before the failure

Consistent State

- A consistent system state is one in which, if the state of a process reflects a message receipt, then the state of the corresponding sender reflects sending that message.

Consistent State



Goal

- To bring the system into a consistent state when inconsistencies occur because of failure
- Note
 - The reconstructed consistent state is **not** necessarily one that has occurred before the failure

Checkpoint-Based Rollback-Recovery

Checkpoint-Based Rollback-Recovery

- Restores the system state to the **recovery line**
 - Recovery line: the most recent consistent set of checkpoints
- Does not guarantee that pre-failure execution can be deterministically regenerated after a rollback
- Classified into three categories
 - Uncoordinated checkpointing
 - Coordinated checkpointing
 - Communication-induced checkpointing

Uncoordinated Checkpointing

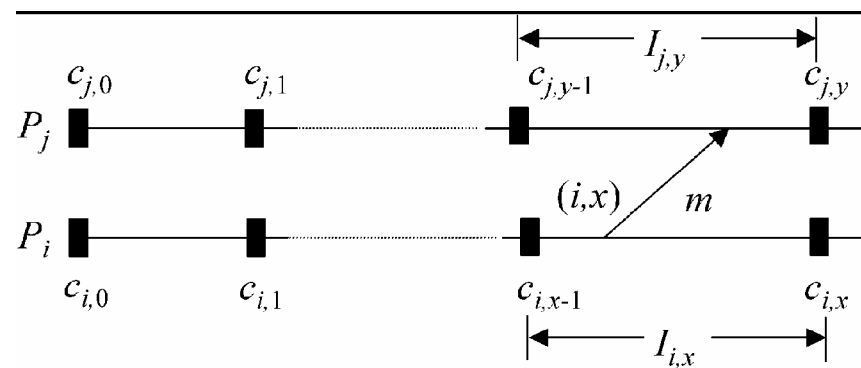
- Allows each process the maximum autonomy in deciding when to take checkpoints

Advantages and Disadvantages

- Advantage
 - Less overhead
 - Each process may take a checkpoint when it is most convenient
- Disadvantages
 - Possibility of the domino effect
 - Useless checkpoints
 - Maintaining multiple checkpoints and garbage collection
 - Not suitable for application with frequent output commit

Determining a Consistent Global Checkpoint during Recovery

- Record the dependencies among the checkpoints during failure-free operation
- $c_{i,x}$: the x^{th} checkpoint of process P_i
- $l_{i,x}$: the *checkpoint interval* between checkpoints $c_{i,x-1}$ and $c_{i,x}$



Rollback

1. Broadcast a *dependency request* message to collect all the dependency information maintained by each process
2. If a process received the message,
 - Stops its execution and replies with the dependency information that is saved on stable storage and, if any, that is associated with current state

Rollback

3. Initiator calculates the recovery line based on the global dependency information
4. Broadcast a rollback request containing the recovery line
5. Each process rolls back to an earlier checkpoints as indicated by the recovery line

Dependency Graphs and Recovery Line Calculation

- Two approach to determine the recovery line in checkpoint-based recovery
 - Rollback-dependency graph
 - Checkpoint graph
- Both approaches are equivalent
 - They always produce the same recovery line

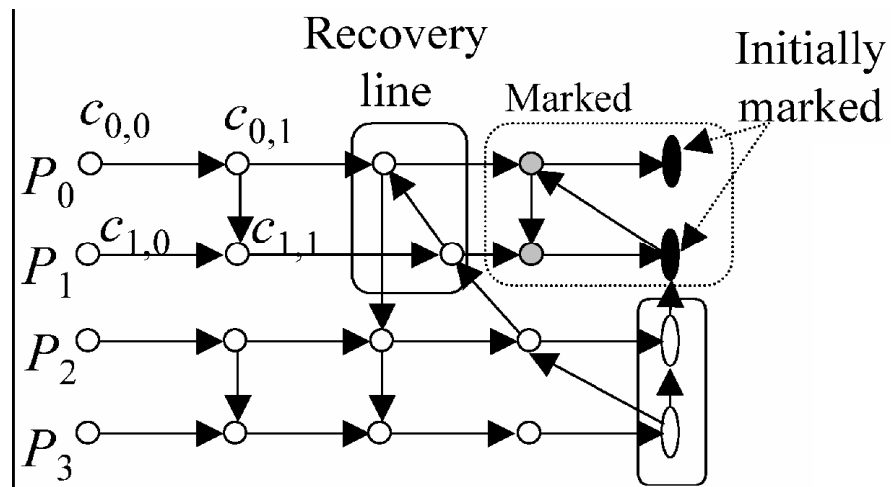
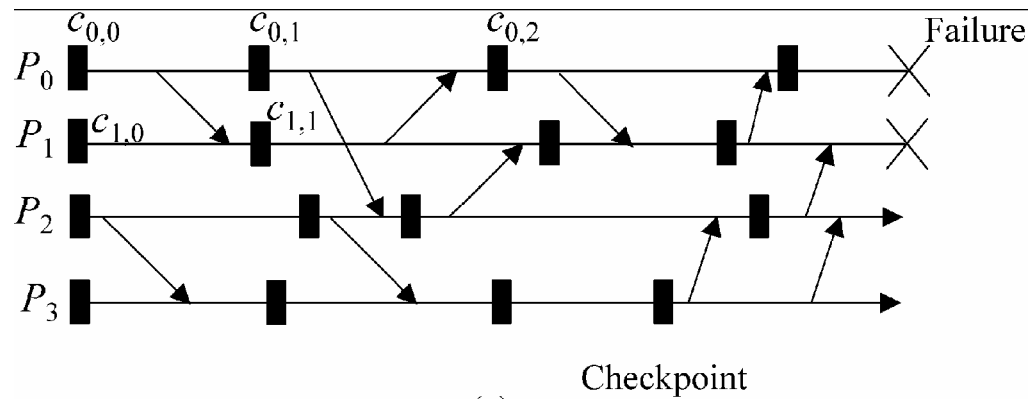
Rollback-Dependency Graph

- Each node represents a checkpoint
- Each node is drawn from $c_{i,x}$ to $c_{j,y}$ if either :
 - $i \neq j$, and a message m is sent from $l_{i,x}$ and received in $l_{j,x}$, or
 - $i = j$, and $y = x + 1$
- If there is an edge from $c_{i,x}$ to $c_{j,y}$ and a failure forces $l_{i,x}$ to be rolled back, then $l_{j,x}$ must also be rolled back

Computing Recovery Line

1. Marks the graph nodes corresponding to the states of processes P_0 and P_1 at the failure point
2. Reachable analysis to mark all reachable nodes from any of the last unmarked nodes over the entire system

Example



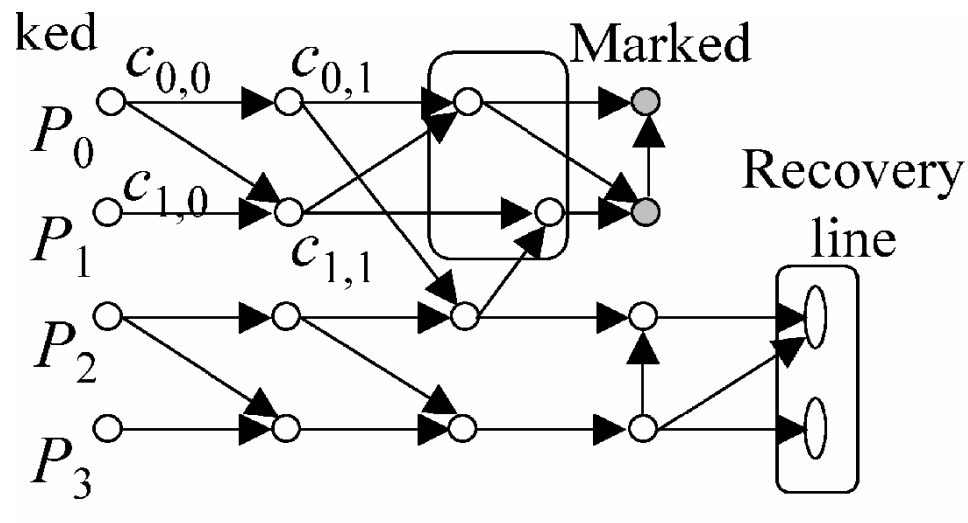
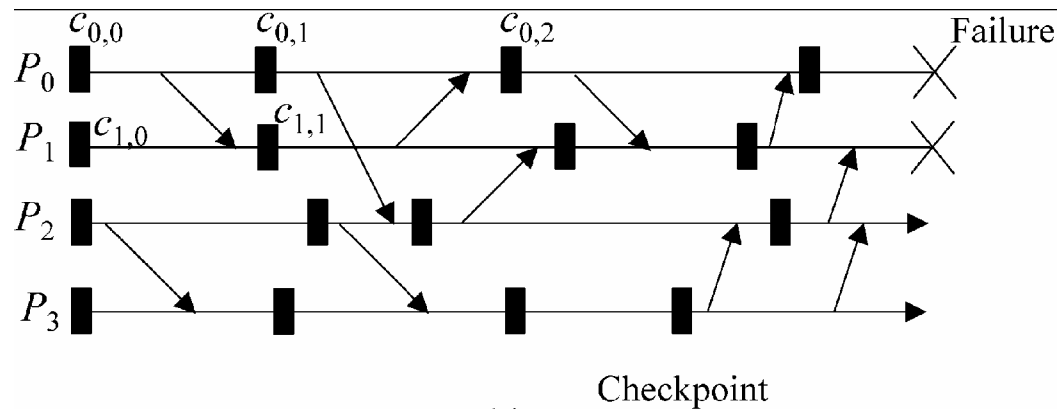
Checkpoint Graph

- Similar to rollback-dependency graph
- Each edge is drawn from $c_{i,x-1}$ to $c_{i,x}$ if a message m is sent from $l_{i,x}$ and received in $l_{j,x}$

Computing Recovery Line

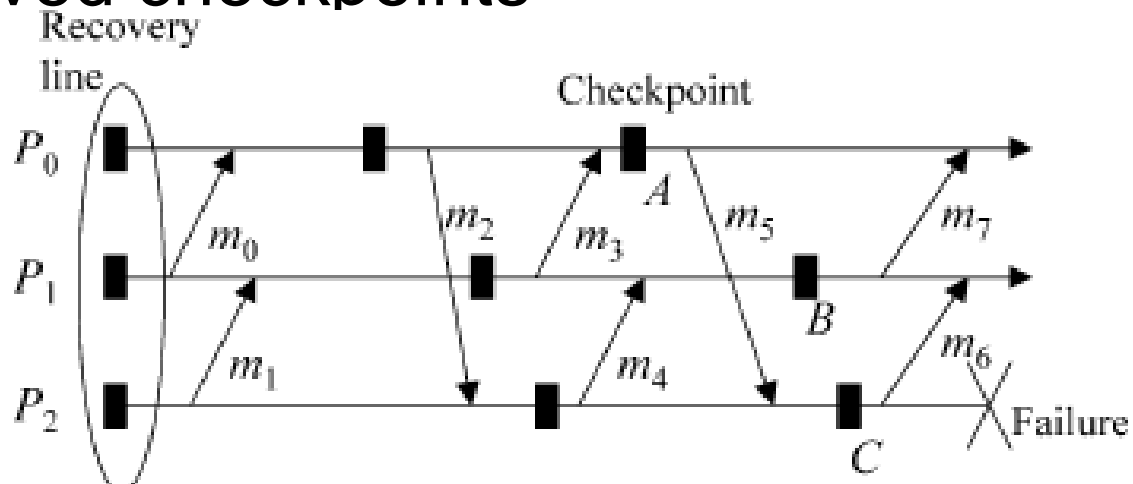
1. Removes both the nodes corresponding to the states of the failed processes at the point of failures and edges incident on them
2. Applies the rollback propagation algorithm

Example



Domino effect

- Uncoordinated checkpointing can lead to the *domino effect*
 - causes the system to roll back to the beginning of the computation in spite of the saved checkpoints



Coordinated checkpointing

- Requires processes to orchestrate their checkpoints in order to form a consistent global state

Coordinated Checkpointing

- Advantages
 - simplifies recovery
 - reduces storage overhead
 - eliminates the need for garbage collection
- Disadvantages
 - not susceptible to the domino effect
 - every process always restarts from its most recent checkpoint
 - the large latency involved in committing output
 - a global checkpoint is needed before messages can be sent to OWP (Outside World Process)

A straightforward approach

- Blocking communications while the checkpointing protocol executes
 1. Coordinator takes a checkpoint and broadcasts a request message to all processes, asking them to take a checkpoint
 2. If a process receives this message,
 1. Stops its execution
 2. Flushes all the communication channels
 3. Takes a tentative checkpoint
 4. Sends an acknowledgement message back to the coordinator

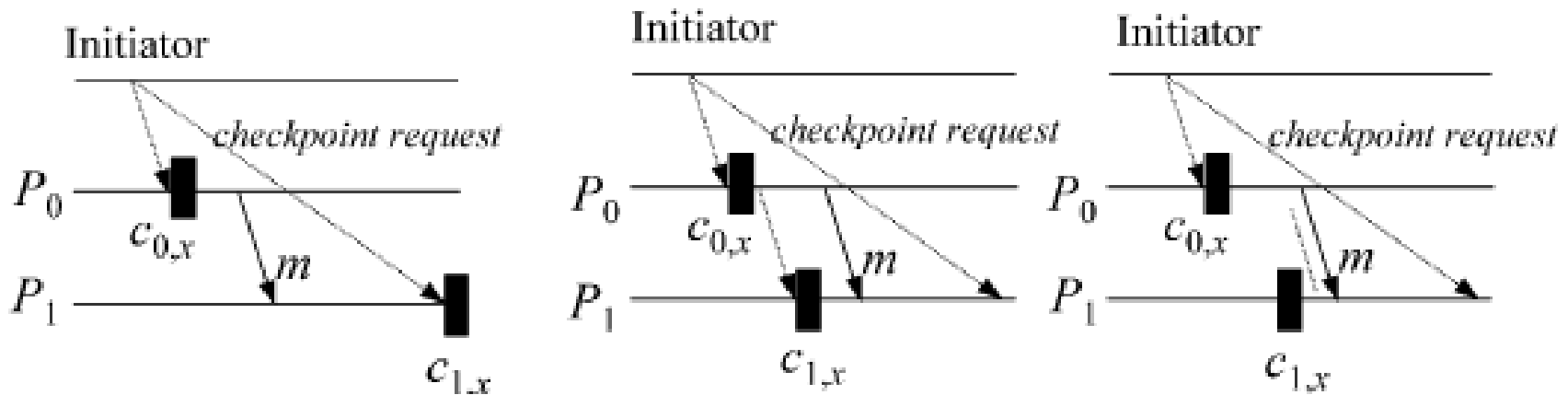
A straightforward approach

3. After the coordinator receives acknowledgements from all processes, it broadcasts a commit message that completes the two-phase checkpointing protocol
4. After receiving the commit message, each process removes the old permanent checkpoint and atomically makes the tentative checkpoint permanent

Large overhead !

Non-Blocking Checkpoint Coordination

- Prevent a process from receiving application messages that could make the checkpoint inconsistent
 - FIFO channel
 - Piggyback



Minimal Checkpoint Coordination

- Coordinated checkpointing requires all processes to participate in every checkpointing
- Can we reduce the number of processes involved in a coordinated checkpointing session ?
 - the processes that needed to take new checkpoints are only those that have communicated with the checkpoint initiator either directly or indirectly since the last checkpoint

Communication-Induced Checkpointing (CIC)

- Avoiding the domino effect without requiring all checkpoints to be coordinated
- Processes take 2 kinds of checkpoints
 - Local
 - can be taken independently
 - Forced
 - must be taken to guarantee the eventual progress of the recovery line
 - prevents the creation of useless checkpoints

Communication-Induced Checkpointing (CIC)

- Piggyback protocol-specific information on each application messages
 - Does not exchange any special coordination messages
- The receiver decide if it should take a forced checkpoint
 - Using Z-path and Z-cycle

Z-path (zigzag path)

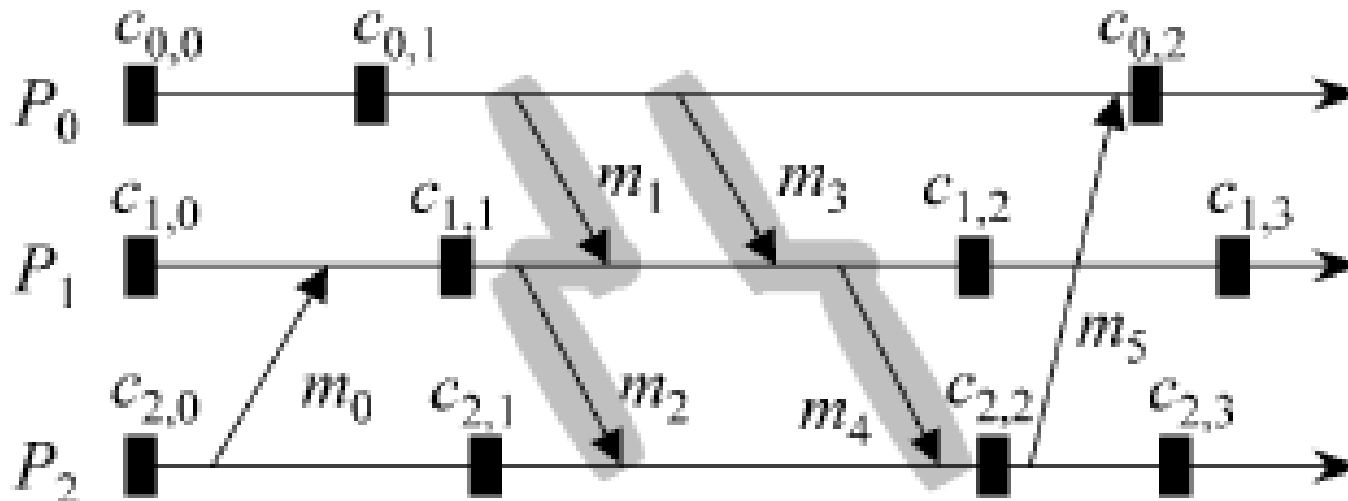
- A special sequence of messages that connect two checkpoints
- \mapsto : Lamport's happened-before relation
- $C_{i,x}$: the x^{th} checkpoint of process P_i
- *Execution portion between 2 consecutive checkpoints on the same process*: the checkpoint interval starting with the earlier checkpoint

Z-path (zigzag path)

- Given two checkpoints $c_{i,x}$ and $c_{j,y}$, a Z-path exists between $c_{i,x}$ and $c_{j,y}$ iff
 1. $x < y$ and $i = j$; or
 2. There exists a sequence of messages $[m_0, m_1, \dots, m_n]$, $n > 0$ such that:
 - $c_{i,x} \mapsto \text{send}_i(m_0)$;
 - $\forall l < n$, either $\text{deliver}_k(m_l)$ and $\text{send}_k(m_{l+1})$ are in the same checkpoint interval,
or $\text{deliver}_k(m_l) \mapsto \text{send}_k(m_{l+1})$; and
 - $\text{deliver}_j(m_n) \mapsto c_{j,y}$

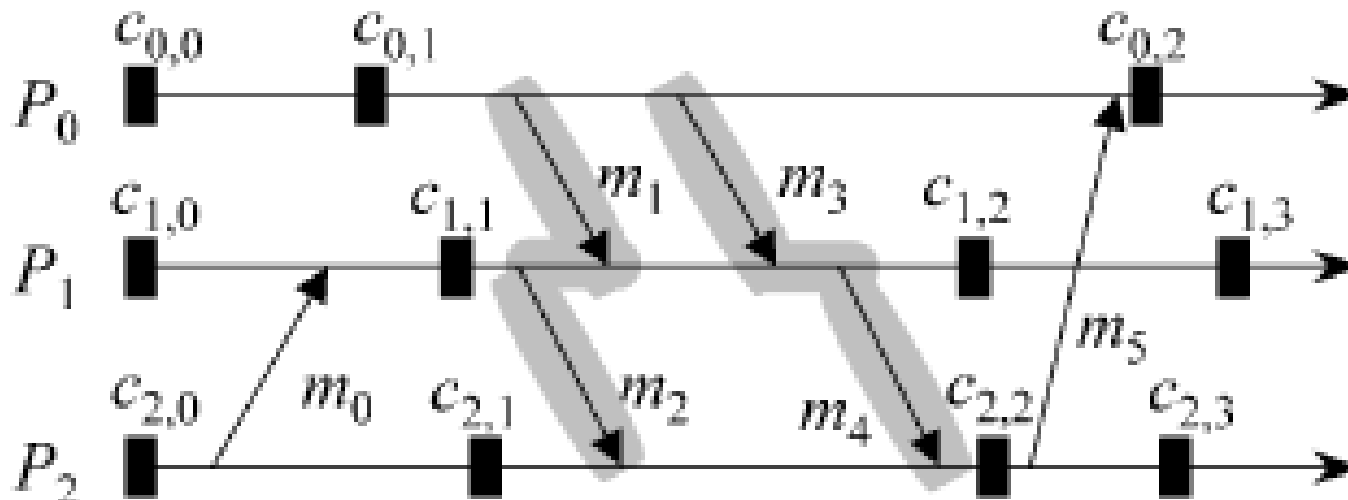
Z-path (zigzag path)

- $[m_1, m_2]$ and $[m_3, m_4]$ are Z-paths between checkpoints $c_{0,1}$ and $c_{2,2}$



Z-cycle

- A Z-path that begins and ends with the same checkpoint
 - $[m_5, m_3, m_4]$ is a Z-path that starts and ends at checkpoint $c_{2,2}$



Useless checkpoints

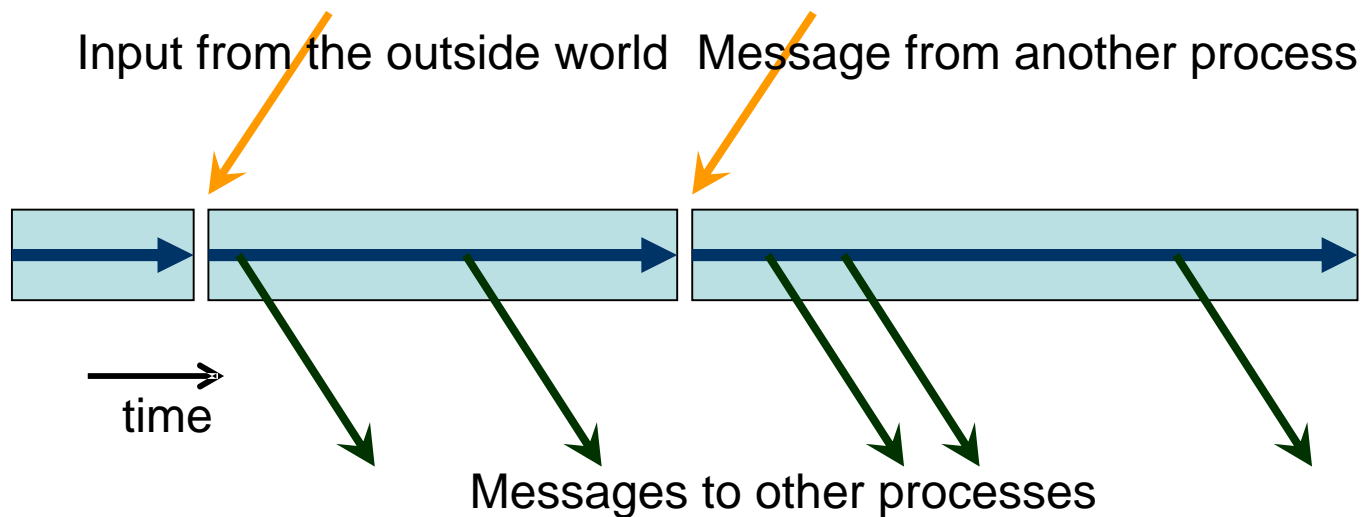
- A checkpoint is useless if and only if it is part of a Z-cycle
- We can avoid useless checkpoints by examining that no Z-path becomes a Z-cycle

- CIC protocols is classified in one of two types
 - Model-based checkpointing
 - Index-based checkpointing
- It is proved that the two types are fundamentally equivalent

Log-Based Rollback-Recovery

Model of process execution

- Process execution: sequence of deterministic state intervals, each starting with the execution of a nondeterministic event



Log-Based Rollback-Recovery

- Assumption
 - All nondeterministic events can be identified and their corresponding determinants can be logged to stable storage

Log-Based Rollback-Recovery

1. Each process logs all the nondeterministic events onto the stable storage
2. Each process also takes checkpoints
3. If some processes fail, they recover the system to its consistent state using the checkpoints and the log
 - Replaying nondeterministic events precisely as they occurred during the pre-failure execution

Log-Based Rollback-Recovery

- Guarantees that upon recovery of all failed processes, the system does not contain any **orphan process**
- Orphan process
 - A process whose state depends on a nondeterministic event that cannot be reproduced during recovery

No-Orphans Consistency Condition

- e : a nondeterministic event that occurs at process p
- $Depend(e)$: the set of processes that are affected by a nondeterministic event e
- $Log(e)$: 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 stable storage

No-Orphans Consistency Condition

- *Always-no-orphans condition :*

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

Three Flavors of the Protocols

- Pessimistic log-based
- Optimistic log-based
- Causal log-based

Pessimistic Logging

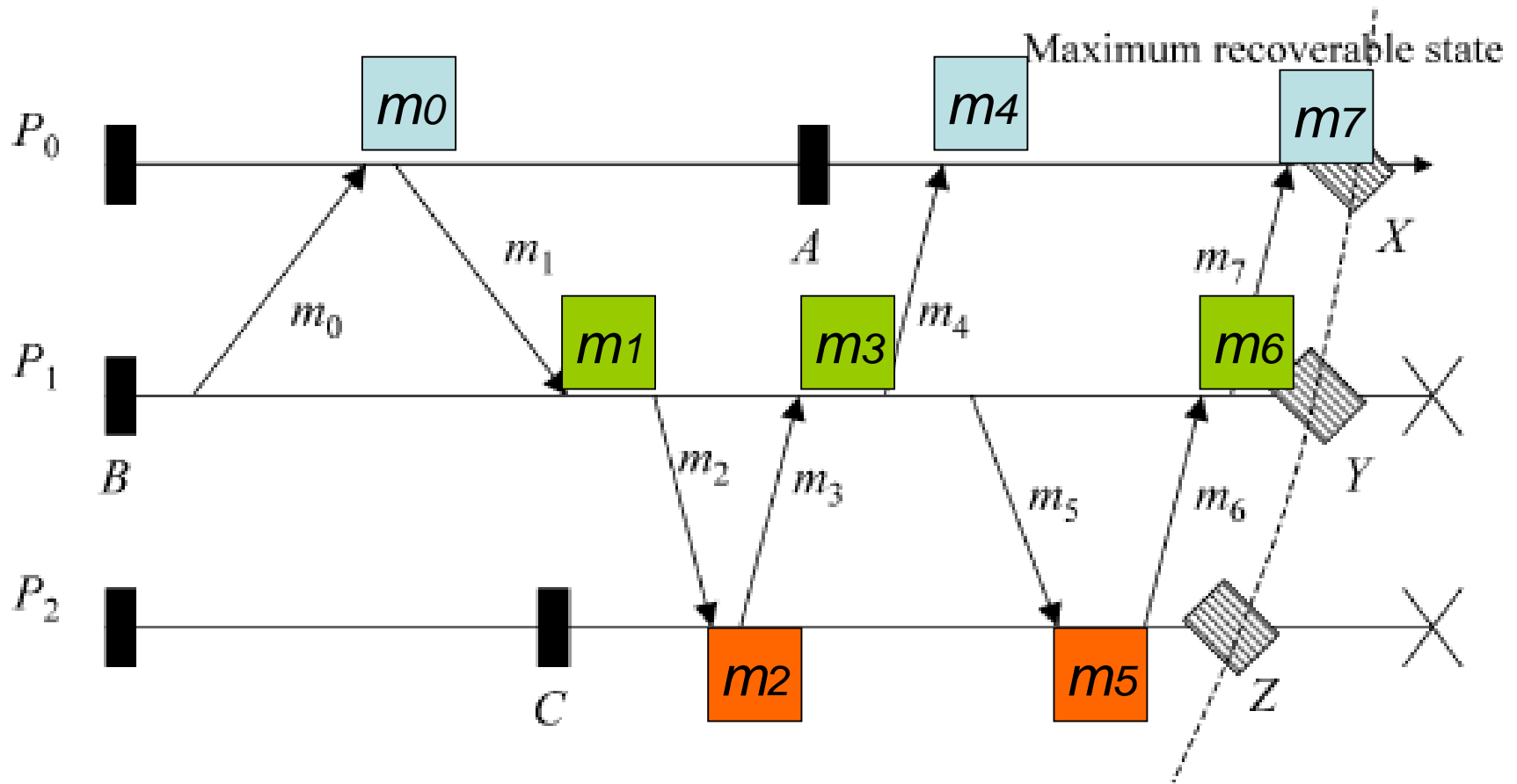
- Assumes that a failure can occur after any nondeterministic event (**Pessimistic assumption**)
- Logs the determinant of each nondeterministic event before the event affects the computation
- Implements the following property:

$$\forall e : \neg \text{Stable}(e) \Rightarrow |\text{Depend}(e)| = 0$$

Pessimistic Logging

- Also takes periodic checkpoints to limit the amount of the work to replay
- If a failure occurs,
 1. Restart the program from the most recent checkpoint
 2. Recreate the pre-failure execution using the logged determinants

Example



Advantages of Pessimistic Logging

- the frequency of checkpoints can be determined by trading off
 - *the runtime performance* with
 - *the potential protection of the ongoing execution*
- Recovery is simplified
- Garbage collection is simple

Disadvantage of Pessimistic Logging

- A performance penalty incurred by **synchronous logging**
- Special techniques to reduce the overhead of synchronous logging are required

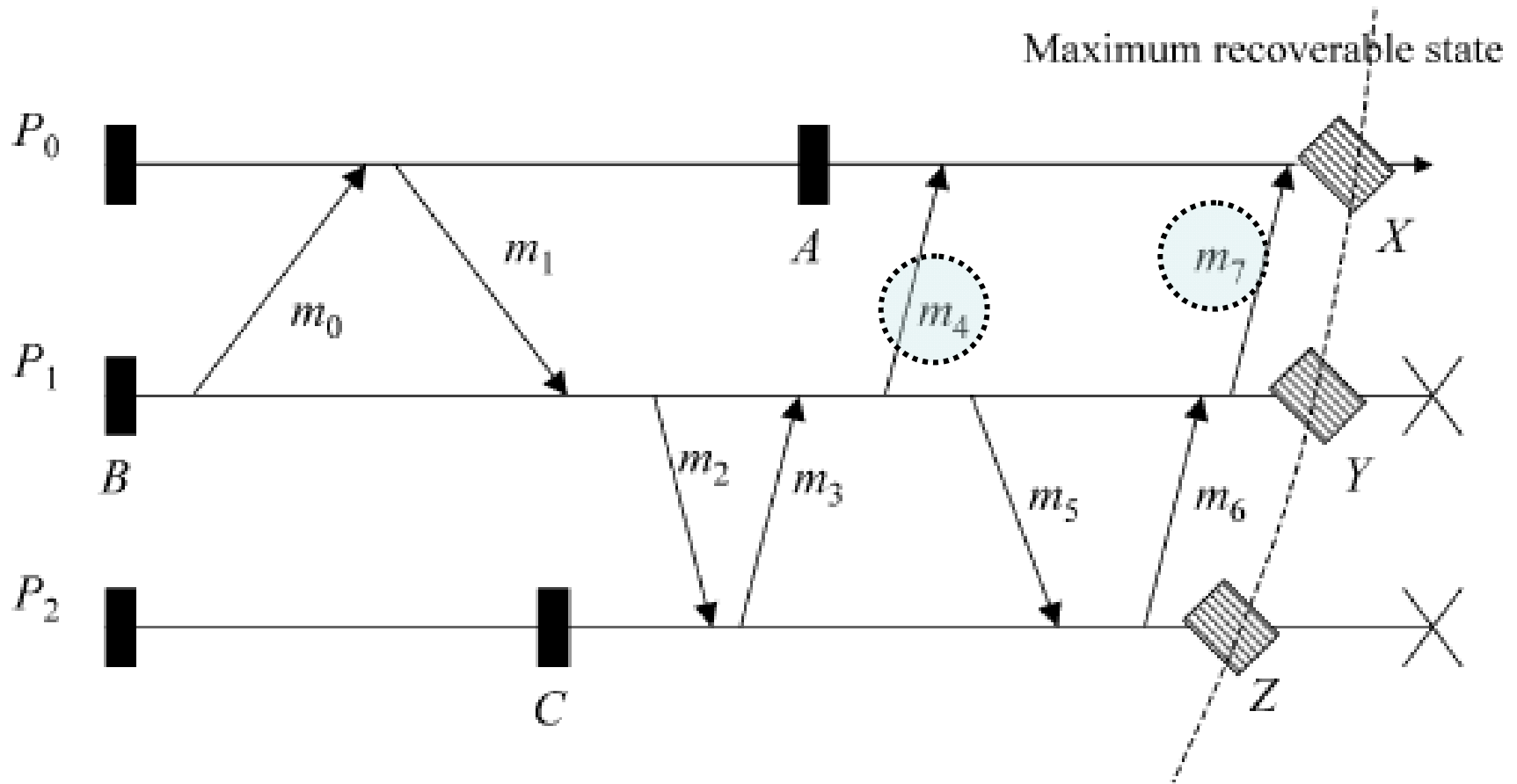
Relaxing Logging Atomicity

- Delivering a message or an event and deferring its logging until the receiver communicates with any other processes
- Processes implement the following weaker property:

$$\forall e : \neg \text{Stable}(e) \Rightarrow |\text{Depend}(e)| \leq 1$$

- This property still guarantees the *always-no-orphans* condition

Example



Optimistic Logging

- Processes log determinants **asynchronously** to stable storage
 - Determinants are kept in a volatile log, which is flushed to stable storage periodically
- The protocol is based on the optimistic assumption
 - Logging will complete before a failure occurs

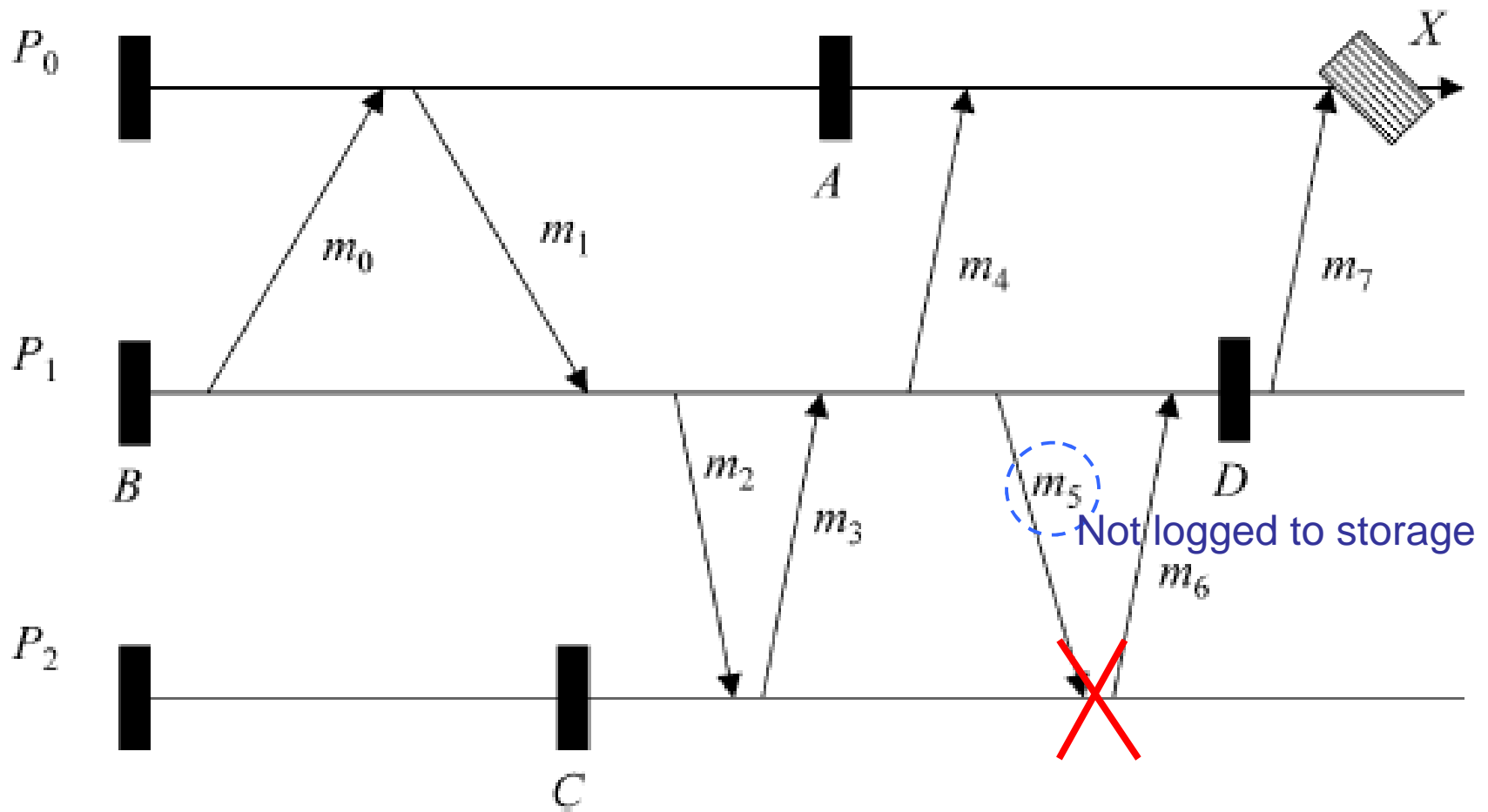
Advantages and Disadvantages

- Advantage
 - Little overhead during failure-free execution
 - The application does not block waiting for the determinants to be written to stable storage
- Disadvantages
 - More complicated recovery and garbage collection
 - Slower output commit

Temporary Orphan Processes

1. The determinants in the volatile log will be lost if the process failures
2. State intervals starting by the events in the log can't be recovered
3. If the process set a message during any of the intervals, the receiver of the message becomes an **orphan process**
4. **The orphan process must roll back** to undo the effect of receiving the message

Example



Tracking Causal Dependencies

- In this protocol, however, *always-no-orphan* condition holds by the time recovery is complete
- Using causal dependencies, the protocol calculate and recover the latest global state of the pre-failure execution
- Recovery can be **synchronous** or **asynchronous**

Synchronous recovery

- All processes compute the maximum recoverable system state
 - Based on dependency and logged information
- During failure-free execution, each process increments a state interval index of each state interval
 - This dependency tracking can be either *direct* or *transitive*

Direct Dependency Tracking

- The state interval index of the sender is piggybacked on each outgoing message
- The receiver records the dependency directly caused by the message
- Lamport's logical clock

Transitive dependency tracking

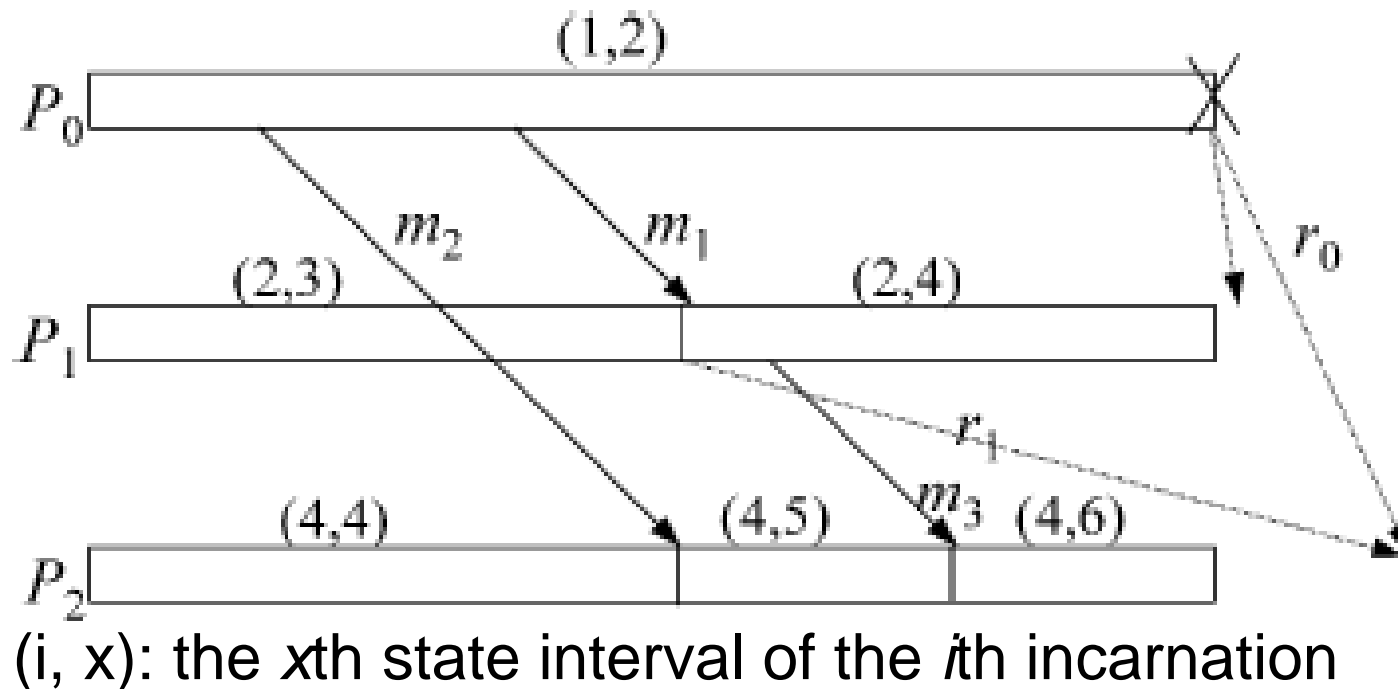
- Each process P_i maintains a size N vector TD_i
 - $TD_i[i]$: P_i 's current state interval index
 - $TD_i[j]$: the highest index of any state interval of P_j on which P_i depends ($i \neq j$)
- Vector clock

Asynchronous Recovery

- A failed process restarts by sending
 - a rollback announcement broadcast *or*
 - a recovery message broadcast
- When a process receives the rollback announcement,
 1. It rolls back if it detects that it has become an orphan
 2. It broadcasts its own rollback announcement

Exponential Rollbacks

- A simple failure causes a process to roll back an exponential number of times



Eliminating Exponential Rollbacks

- To ensure that any process roll back at most once in response to a single failure
- Distinguish failure announcements from rollback announcements and broadcast only the former
- Piggyback the original rollback announcement from the failed process on every subsequent rollback announcement that it triggers

Causal Logging

- Failure-free performance advantage of optimistic logging
 - Avoids synchronous access to stable storage
- Most of the advantages of pessimistic logging
 - Allows each process to commit independently and creates no orphans
 - Limits the rollback to the most recent checkpoint

always-no-orphans property

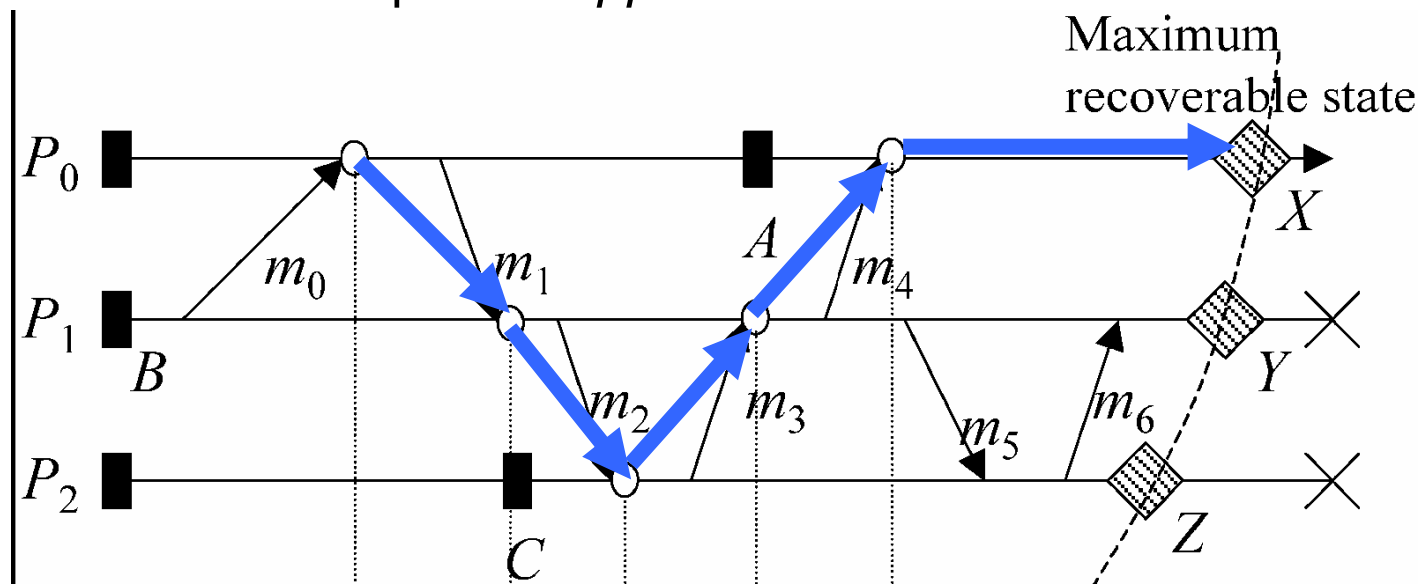
- To implement the *always-no-orphans* property,

the determinant of each nondeterministic event that causally precedes the state of a process is either stable or it is available locally to that process

Example

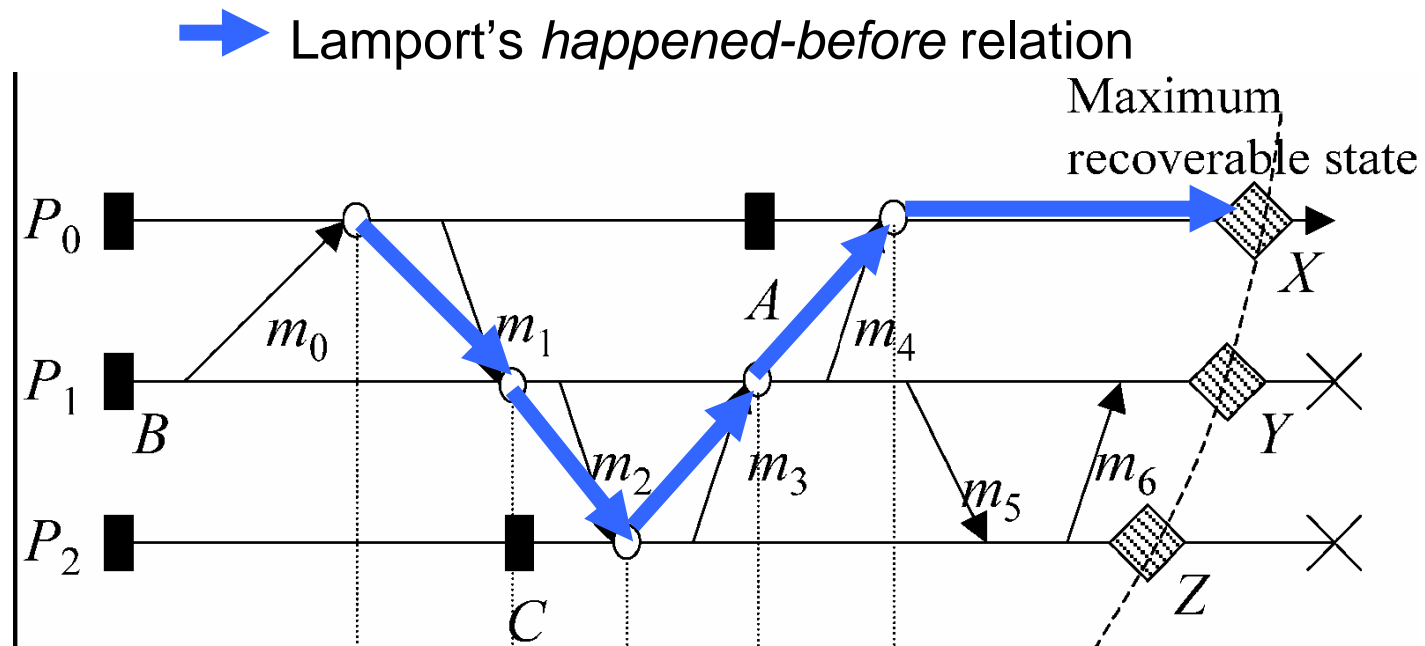
- Message m_5 and m_6 may be lost
- P_0 have logged the determinants of the delivery of m_0 , m_1 , m_2 , m_3 and m_4

➔ Lamport's *happened-before* relation



Example

- P_0 knows the order in which P_1 should replay messages m_1 and m_3



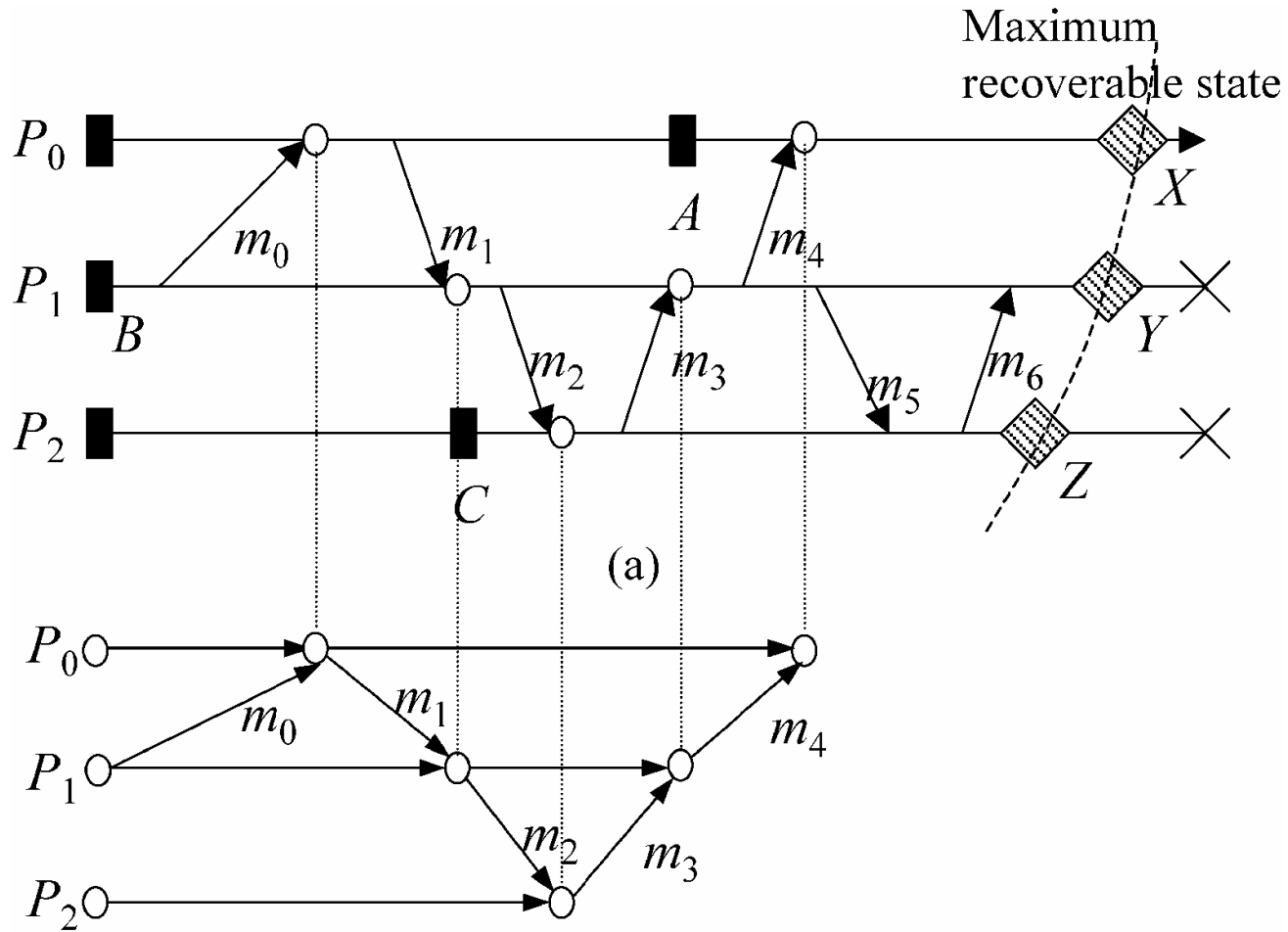
Tracking Causality

- Sender process
 1. Piggybacks the non-stable determinants in its volatile log on the message they send to other processes
- Receiver process
 1. Adds any piggybacked determinants to its volatile log
 2. Delivers the message to the application

Antecedence Graph

- Provides every process with a complete history of the nondeterministic event that have causal effect on its state
- **Nodes:** represent nondeterministic events that precedes the state of a process
- **Edges:** correspond to the *happened-before* relation

Antecedence graph



Reducing the Overhead

- A graph carried by each message is a **superset** of the one piggybacked on the previous message sent from the same process
- Any message between processes carries only the difference of the graphs

Further Reduction: Family Based Logging Protocols (FBL)

- To tolerate f process failure, it's sufficient to log each nondeterministic event in the volatile store of $f + 1$ different hosts
- $Stable(e)$ and $|Log(2)| > f$
- For $f < N$, FBL protocol don't access stable storage except for checkpointing

Comparison

| | Uncoordinated Checkpointing | Coordinated Checkpointing | Comm. Induced Checkpointing | Pessimistic Logging | Optimistic Logging | Causal Logging |
|--------------------|-----------------------------|------------------------------|------------------------------|----------------------|------------------------------|-----------------|
| PWD assumed? | No | No | No | Yes | Yes | Yes |
| Checkpoint/process | Several | 1 | Several | 1 | Several | 1 |
| Domino effect | Possible | No | No | No | No | No |
| Orphan processes | Possible | No | Possible | No | Possible | No |
| Rollback extent | Unbounded | Last global checkpoint | Possibly several checkpoints | Last checkpoint | Possibly several checkpoints | Last checkpoint |
| Recovery data | Distributed | Distributed | Distributed | Distributed or local | Distributed or local | Distributed |
| Recovery protocol | Distributed | Distributed | Distributed | Local | Distributed | Distributed |
| Output commit | Not possible | Global coordination required | Global coordination required | Local decision | Global coordination required | Local decision |

Rollback-Recovery in Practice

- 商用のシステムではほとんど使われていない
 - 実装の難しさ
 - 適用してメリットのあるアプリケーションが少ない
 - Long-running scientific programs
 - メリットのあるものはsuper computerや専用機
- Log-based は、さらに少ない
 - OSへのoverheadを伴う改造の必要
 - 存在している数少ない例も、pessimisticを使っている