

Distributed Systems

Rollback-recovery

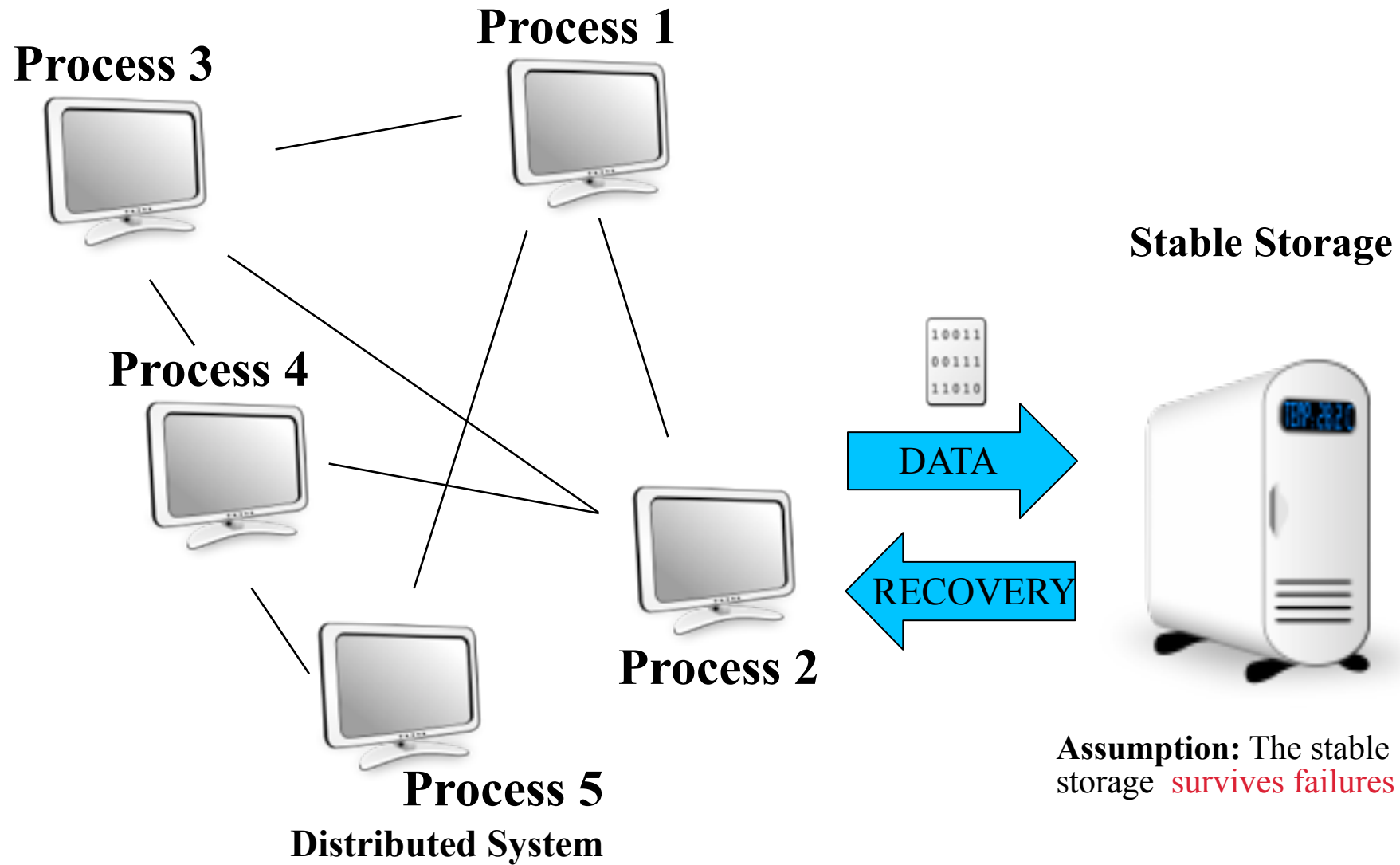
Alberto Montresor
Università di Trento

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Reliability – different techniques, different focus

- ◆ **Fault tolerance – offers the abstraction of a reliable system where faults are tolerated**
 - ◆ Group communication
 - ◆ Offers an abstraction of an ideal communication system that simplifies the development of reliable applications
 - ◆ Transactions
 - ◆ Data-oriented applications
- ◆ **Fault-recovery - Once a fault occurs, we must be able to recover to a previous correct state**
 - ◆ Rollback-recovery
 - ◆ Long-running applications such as scientific computing, where failures are rare but process state is important

The idea



The idea

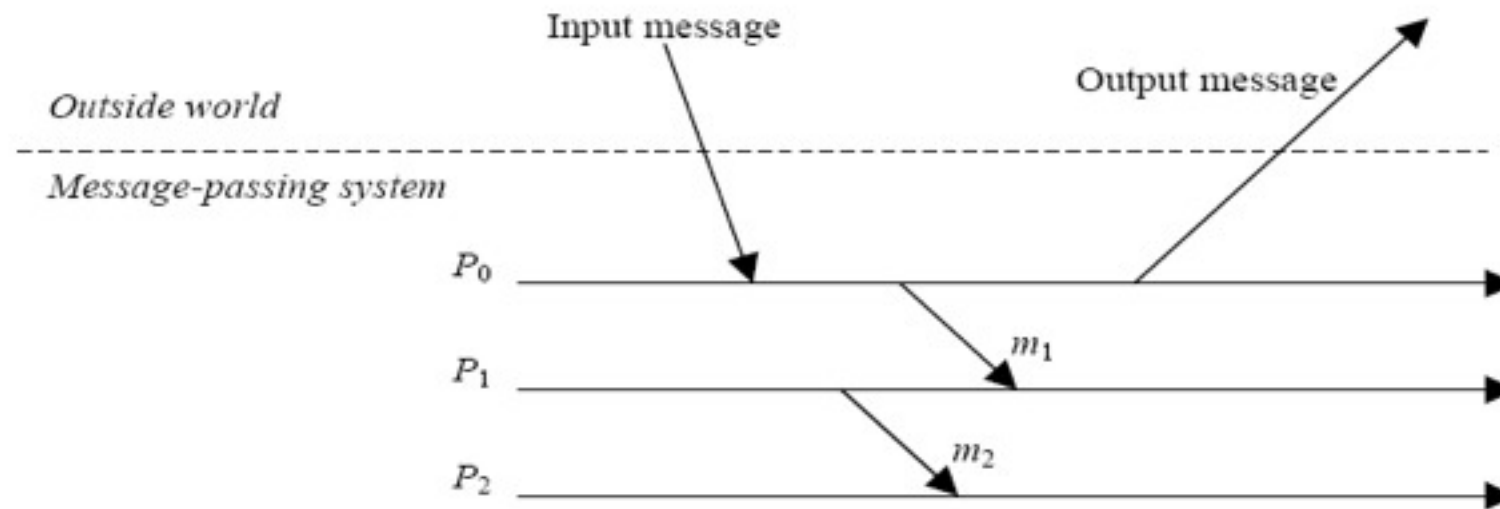
- ◆ **Processes:**
 - ◆ have access to a stable storage that survives all tolerated failures
 - ◆ disks, RAID arrays
- ◆ **Stable storage:**
 - ◆ is used to save recovery information periodically during failure-free periods.
- ◆ **Upon a failure:**
 - ◆ a failed process uses the saved information to restart the computation from an intermediate state, thereby reducing the amount of lost computation

Summary

- ◆ **Background and definitions**
 - ◆ System model
 - ◆ How to save state
 - ◆ Stable storage
 - ◆ Consistent global checkpoints \equiv consistent states
 - ◆ Interactions with the outside world, output commit
 - ◆ Garbage collection
 - ◆ Different type of messages

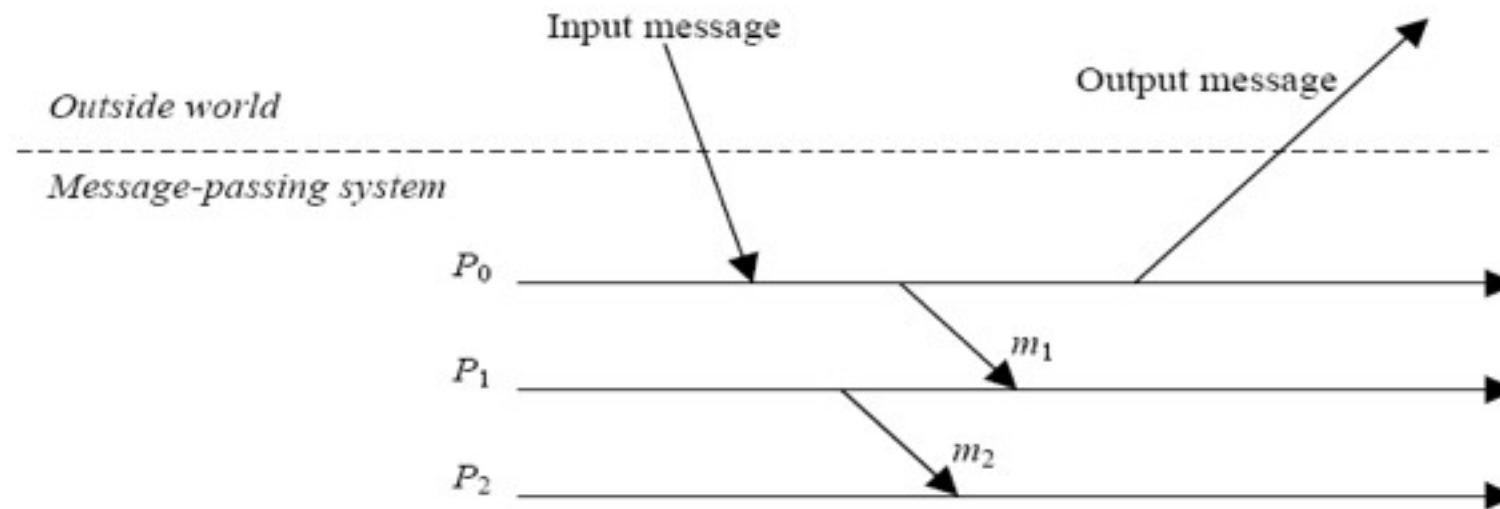
System model

- ◆ Fixed number N of processes
- ◆ Process failures
 - ◆ Fail-stop model
 - ◆ Volatile memory is lost
 - ◆ Access to stable storage that survives failures



System model

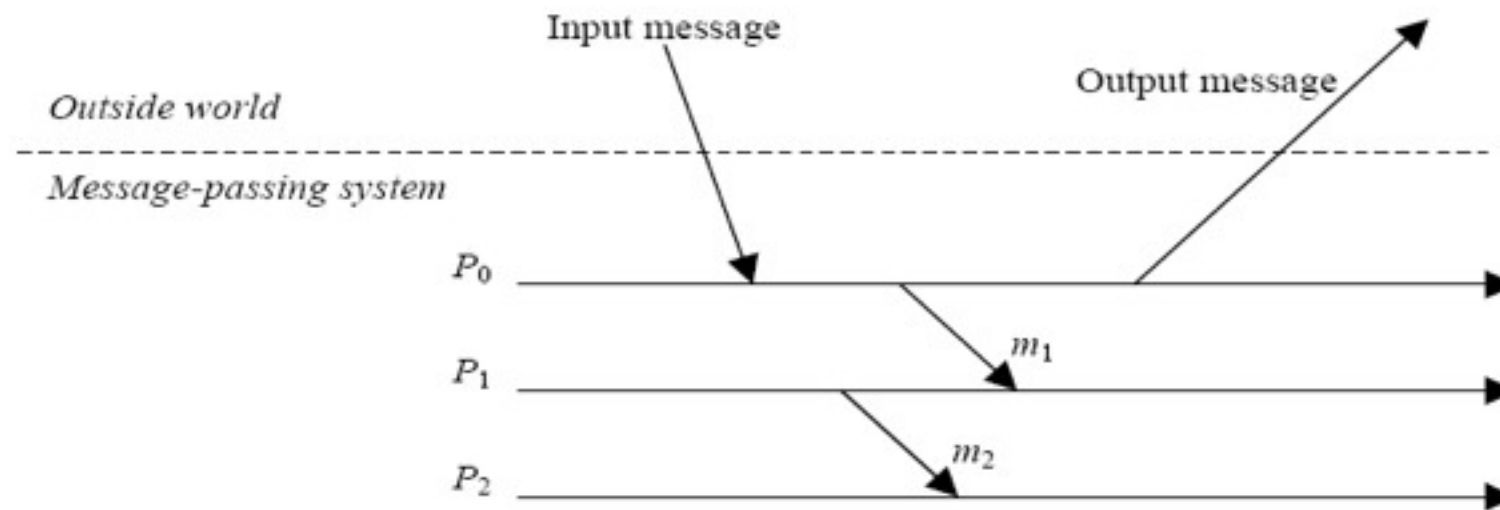
- ◆ **Communication based on message-passing**
 - ◆ No partitioning
 - ◆ Different models:
 - ◆ unreliable channels (lose, duplicate, reorder)
 - ◆ reliable channels (FIFO)
 - ◆ Different protocols for different models



System model

◆ Determinism

- ◆ State-intervals, each started by a non-deterministic event
- ◆ Execution during each state interval is deterministic
- ◆ Piecewise deterministic assumption (PWD)
 - ◆ It is possible to capture sufficient information about non-deterministic events that initiate the state intervals



Correctness condition

- ◆ **A generic correctness condition for rollback-recovery can be defined as follows:**
 - ◆ “a system recovers correctly if its internal state is consistent with the observable behavior of the system before the failure.”
- ◆ **Rollback-recovery protocols therefore must maintain information**
 - ◆ about the internal interactions among processes
 - ◆ the external interactions with the outside world.

Basic approaches

- ◆ **Checkpointing**
 - ◆ Store the states of the participating processes, called **checkpoints**
- ◆ **Message/event logging**
 - ◆ Log messages exchanged among the processes
 - ◆ Log interactions with input and output devices
 - ◆ In general, log events that occur at each process

Message logging combined with checkpointing

- ◆ **Usual approach:**
 - ◆ Make a periodic “big” checkpoint
 - ◆ More frequently, make incremental additions
- ◆ **When recovering**
 - ◆ Restore latest checkpoint
 - ◆ Replay messages/events stored in logs
- ◆ **Result:**
 - ◆ Combining checkpoints with message logging makes it possible to restore a state that lies beyond the most recent checkpoint

How to save state – ask the application

- ◆ **Scientific computing systems might have massive data structures while they run**
 - ◆ ...but maybe not all of them need to be “checkpointed”
- ◆ **If the system uses big but unchanging tables, why write them out?**
 - ◆ If data change slightly, we could save only the increments!
- ◆ **In general, a checkpoint only needs to include data that can't be “regenerated” in any simple, quick way**

How to save state – a transparent approach

- ◆ **A checkpoint could include the entire state of a process/system**
 - ◆ Write out its memory “layout”
 - ◆ Contents of all pages
 - ◆ Contents of registers
- ◆ **This way, we can resume the process by simply reloading its entire state**
 - ◆ Windows XP, GNU/Linux OSs does this for “hibernate” feature
 - ◆ Virtual machines
- ◆ **Potentially, very fast**

How to save state

◆ How (not) to take a checkpoint

- ◆ Block execution, save entire process state to stable storage
 - ◆ very high overhead during failure-free execution
 - ◆ lots of unnecessary data saved on stable storage

◆ How to take a checkpoint

- ◆ Take checkpoints incrementally
 - ◆ save only pages modified since last checkpoint
 - ◆ use “dirty” bit to determine which pages to save
- ◆ Save only “interesting” parts of address space
 - ◆ use application hints or compiler help to avoid saving useless data (e.g. dead variables)
- ◆ Do not block application execution during recovery
 - ◆ copy-on-write

How to save state

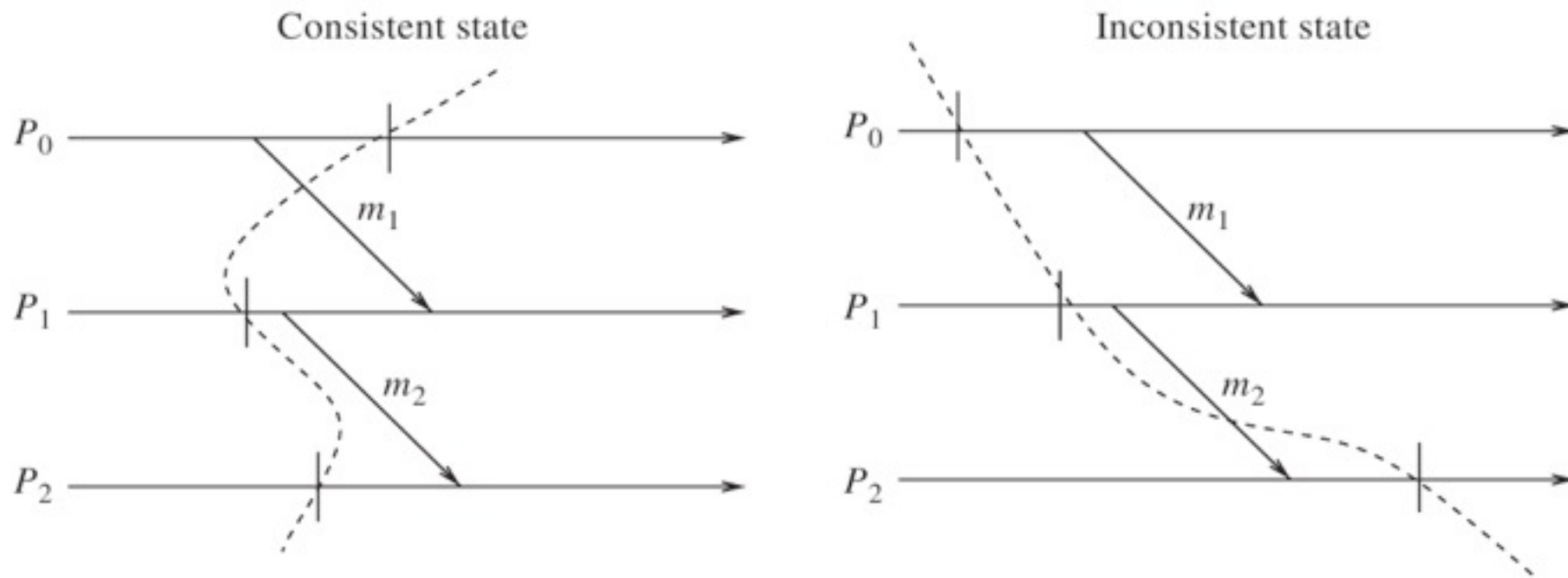
- ◆ **Problem: a program with a bug can...**
 - 1) Crash due to a corrupt data structure
 - 2) Roll back until the last checkpoint
 - 3) Reload the same (still corrupt) structure
 - 4) ... goto 1
- ◆ **One of the advantages of “rebuilding” data structures is that we avoid this risk**

Stable storage

- ◆ **Do not confuse it with the disk storage (frequently) used to implement it.**
- ◆ **Abstraction**
 - ◆ recovery data must persist through the tolerated failures and the their corresponding recoveries
- ◆ **Implementations**
 - ◆ Tolerating a single failure
 - ◆ Volatile memory of another process
 - ◆ Tolerating transient failures
 - ◆ Local disk at each host
 - ◆ Tolerating non-transient failures
 - ◆ Persistent medium outside the host on which a process is running

Consistent global checkpoint

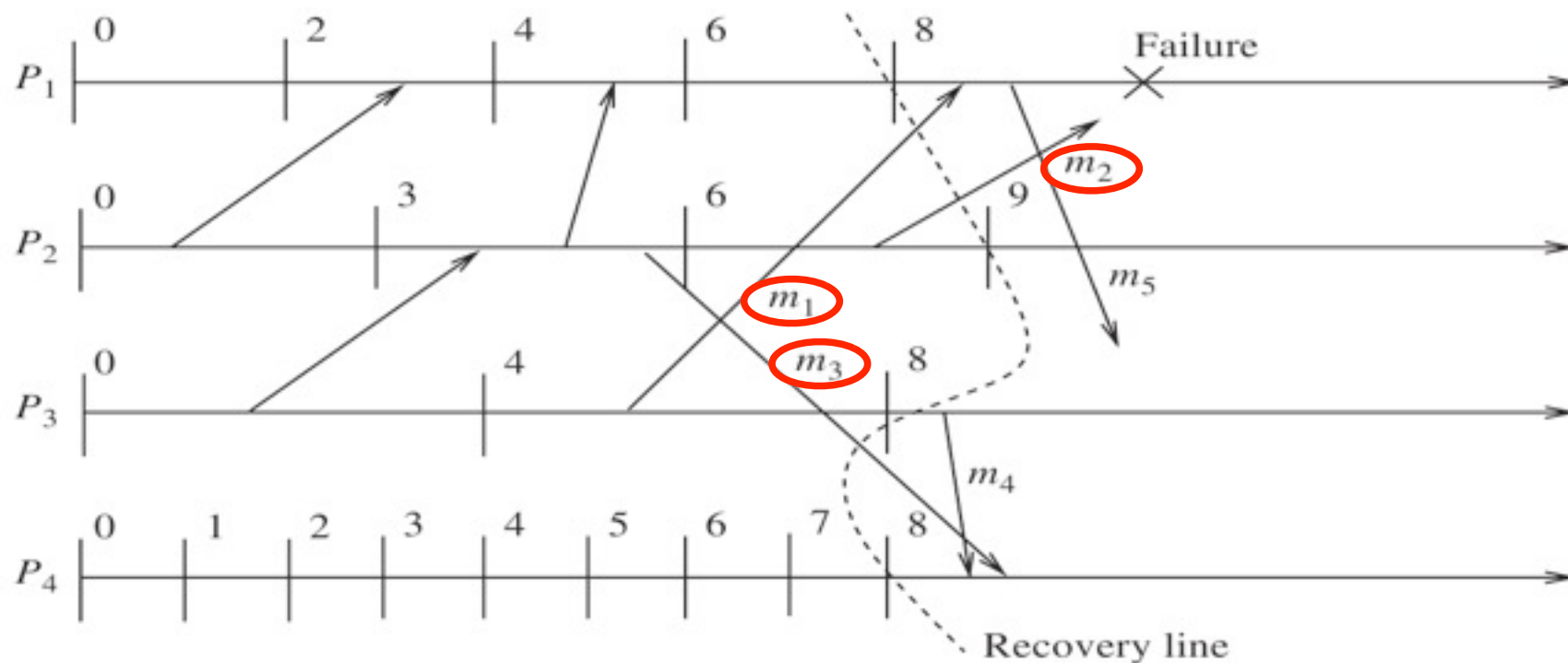
- ◆ **Local checkpoint** - a snapshot of a local state of a process
- ◆ **Global checkpoint** - a set of local checkpoints, one from each process
- ◆ **Consistent global checkpoint** - a global checkpoint such that no message is sent by a process after taking its local checkpoint that is received by another process before taking its local checkpoint.



Different types of messages

◆ In-transit messages

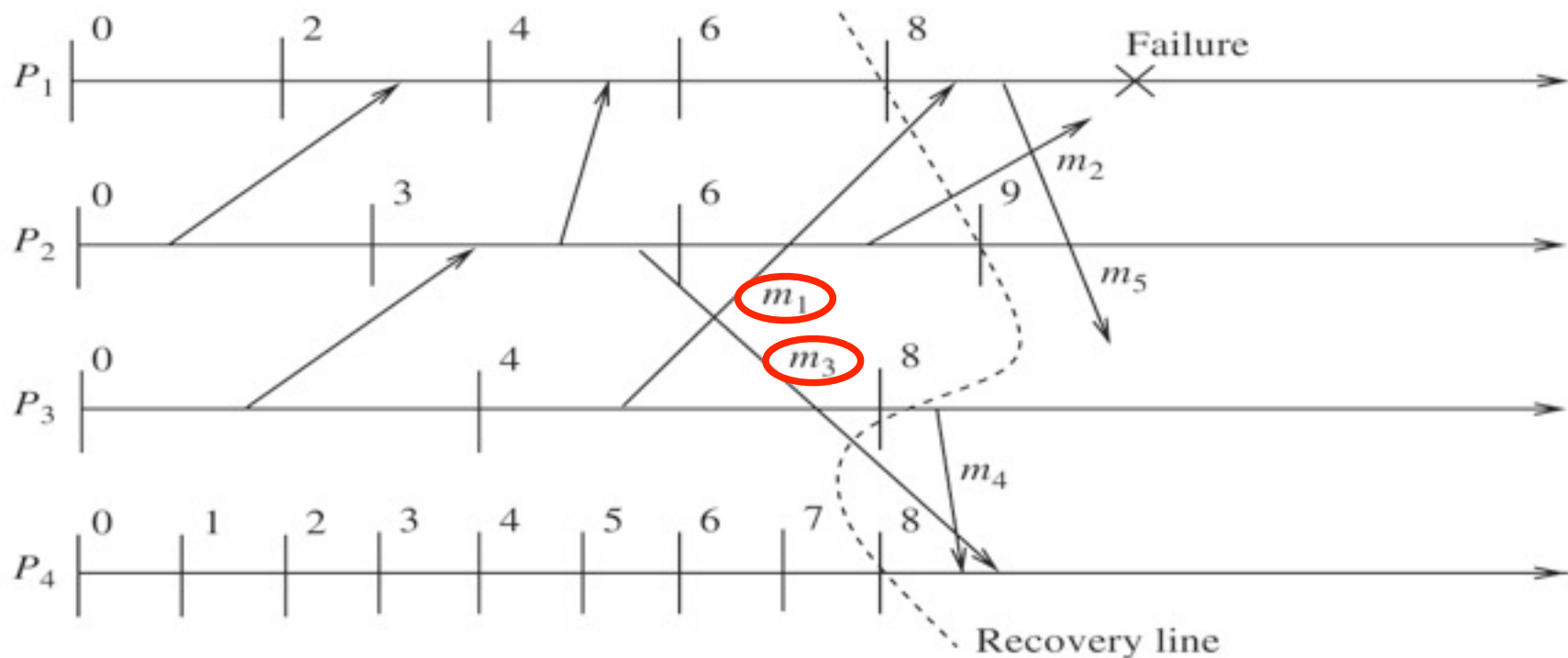
- ◆ Messages sent, not yet received before recovery line
- ◆ No problem with global state consistency
- ◆ Can be lost or delayed



Different types of messages

◆ Lost messages

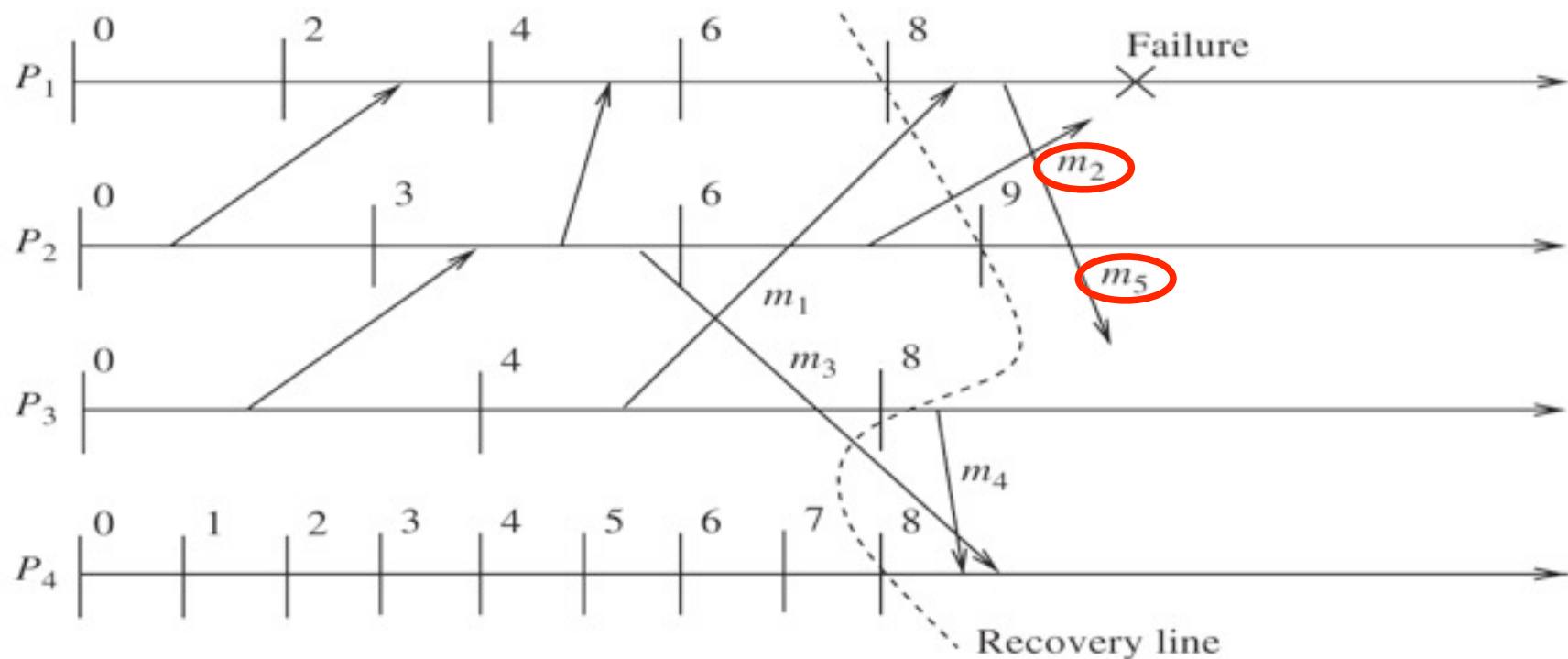
- ◆ Send not undone, receive undone
- ◆ Lost messages must be replied
- ◆ If implemented over
 - ◆ unreliable communication, application is responsible
 - ◆ reliable communication, recovery algorithm is responsible



Different types of messages

◆ Delayed messages

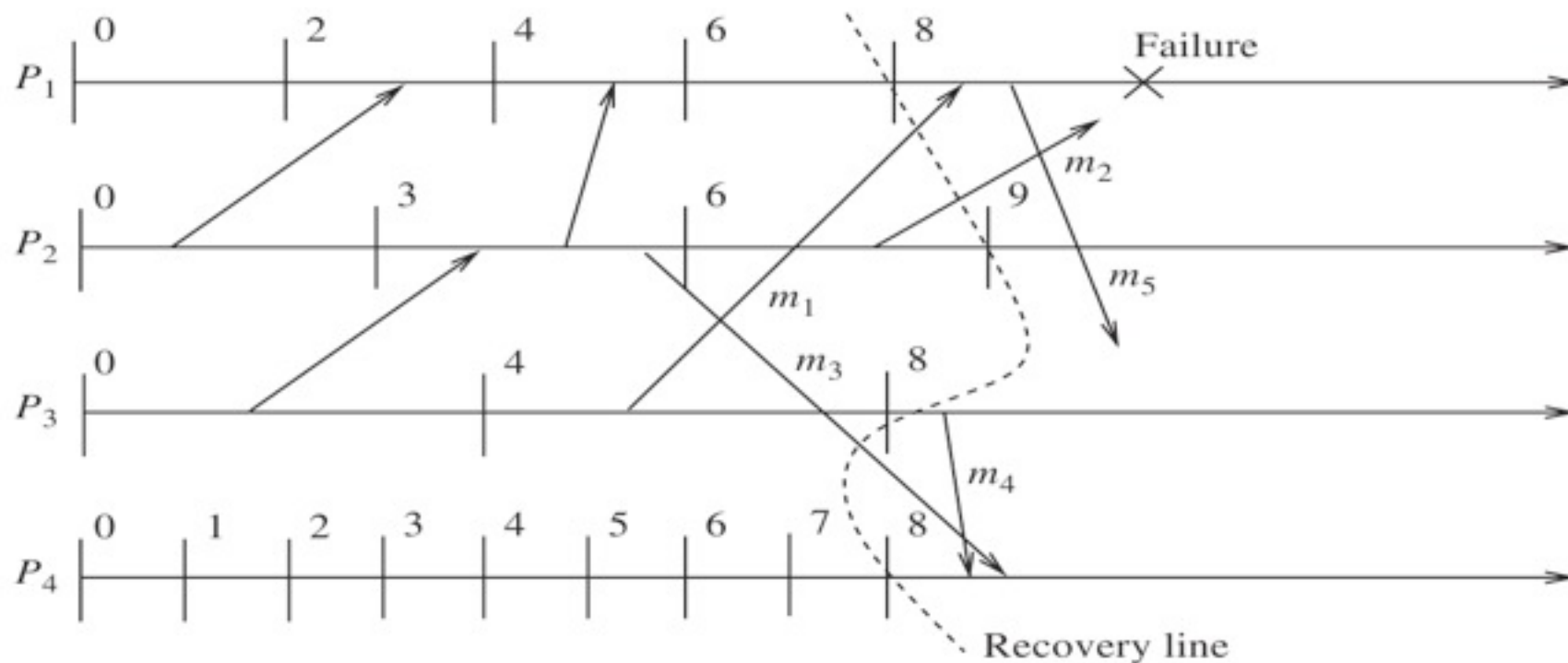
- ◆ Messages whose receive is not recorded because the receiving process was either down or the message arrived after the rollback of the receiving process



Different types of messages

◆ Orphan messages

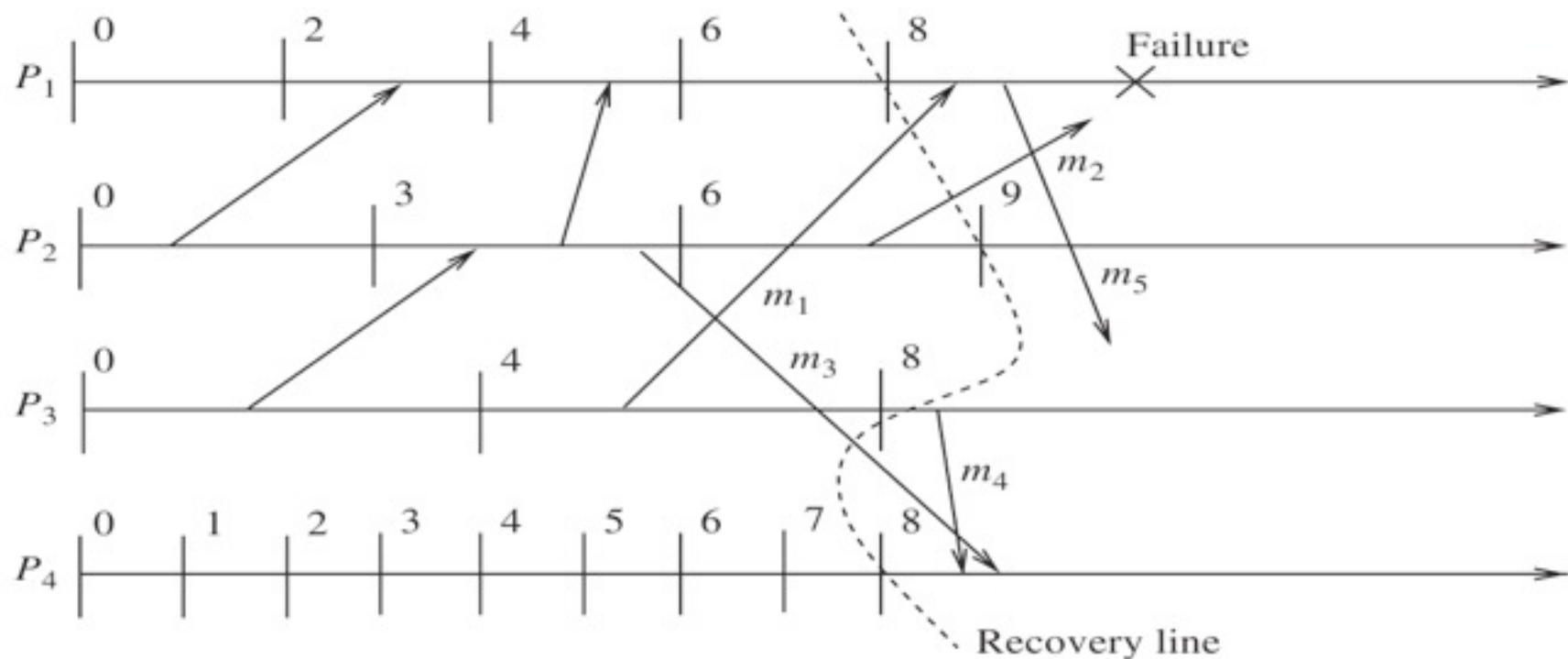
- ◆ Send undone, receive not undone
- ◆ To be avoided if we want consistent global states
- ◆ No examples here



Different types of messages

◆ Duplicate messages

- ◆ Duplicate messages arise due to message logging and replaying during process recovery
- ◆ Example: m_5 will be replayed, so it can arrive twice at P_3



Interaction with outside world

- ◆ **The outside world cannot be “rolled back”**
 - ◆ A printer cannot “unprint”
 - ◆ A bancomat cannot ask money back....
- ◆ **Special “Outside World Process” (OWP)**
 - ◆ cannot fail
 - ◆ cannot maintain state, cannot participate in the recovery protocol
- ◆ **Input**
 - ◆ Input cannot be replied by OWP, should be replied by R-R
 - ◆ External messages must be logged

Interaction with the outside world

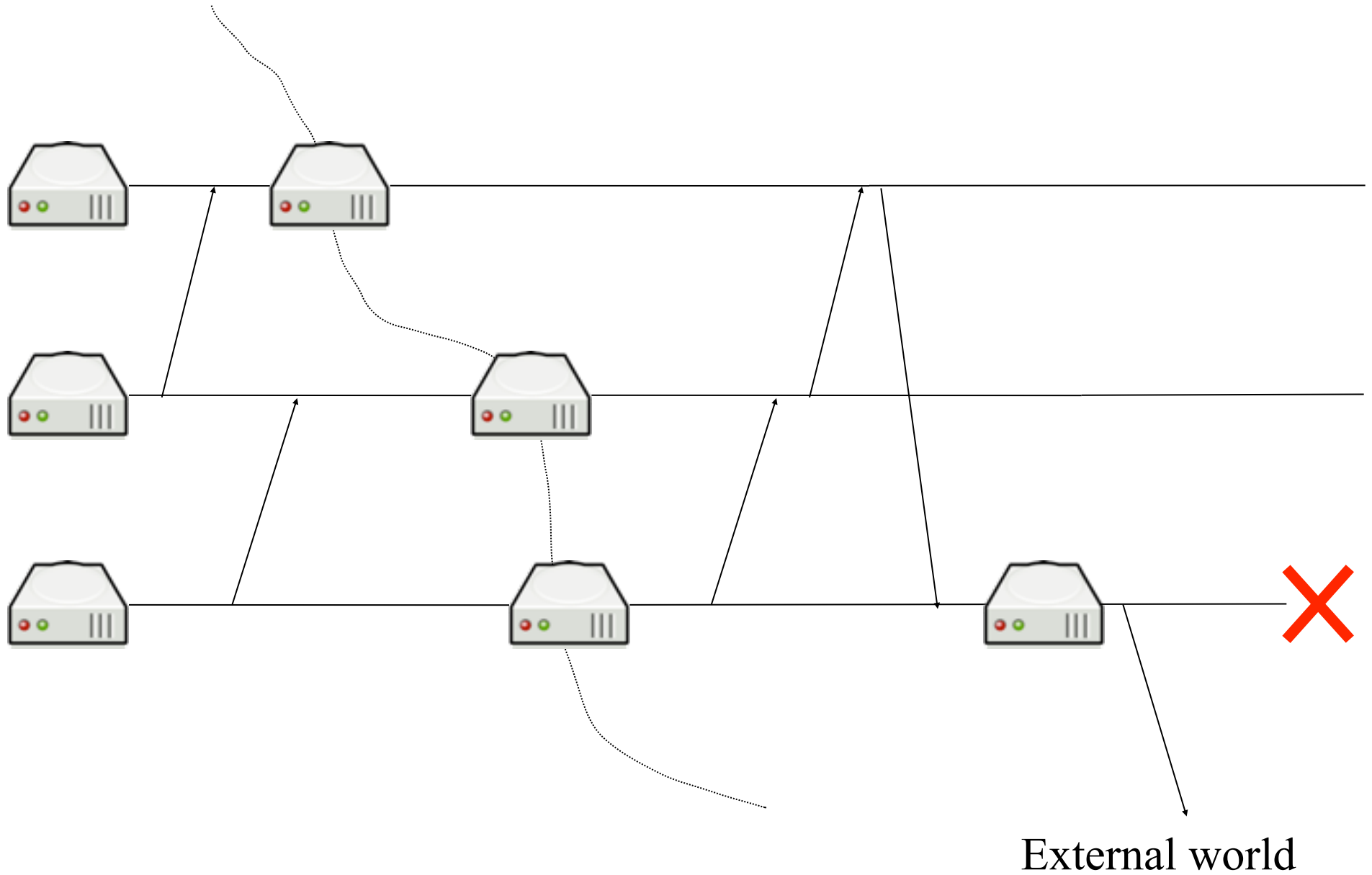
◆ Output commit problem

- ◆ The externally visible behavior of a rollback-recovery system must be equivalent to some failure-free execution
- ◆ Equivalent \equiv same messages to OWP, same order

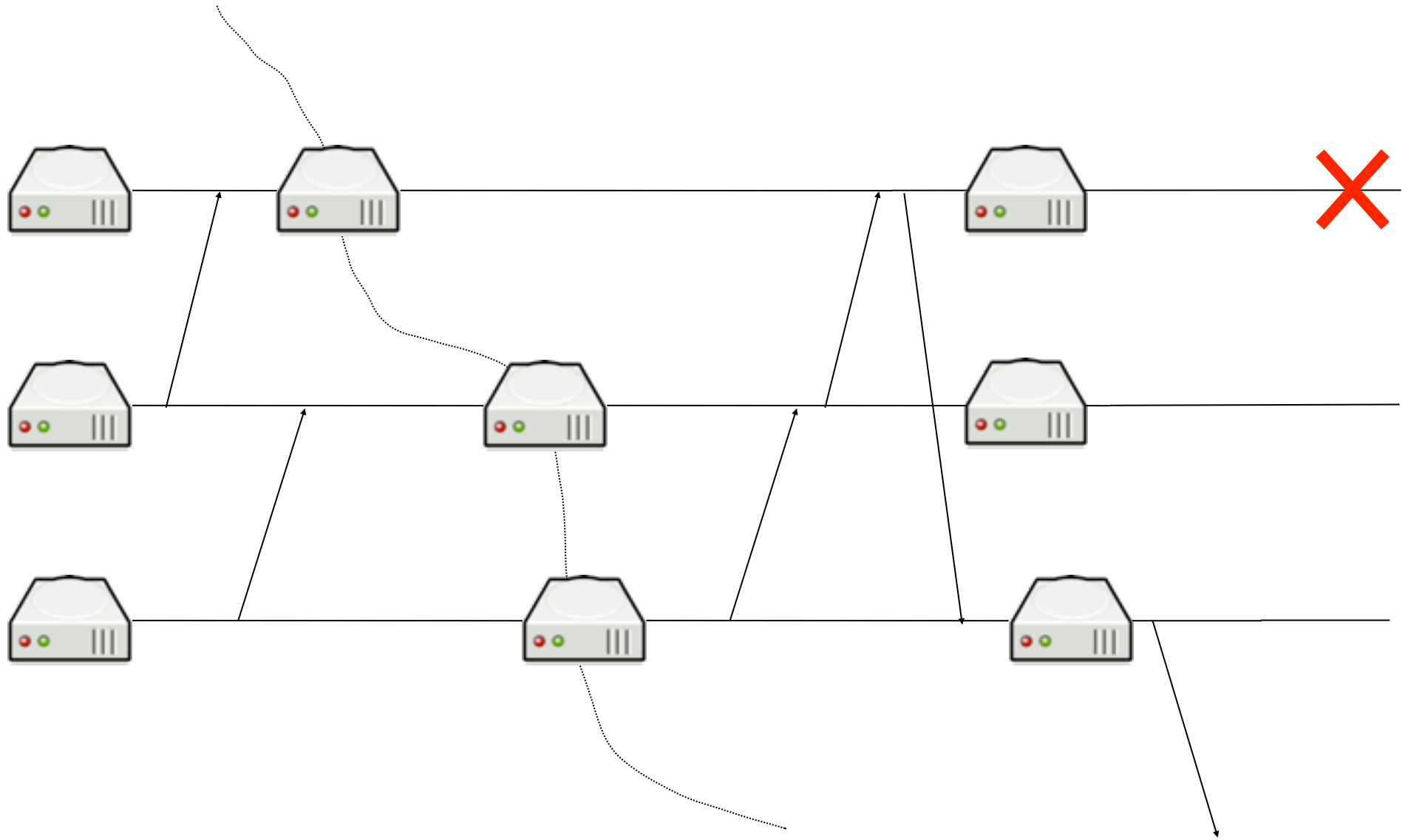
◆ How to do it

- ◆ Commit a msg to the outside world as soon is determined that the state that generated the message will never need to be rolled back

Output commit problem



Output commit problem

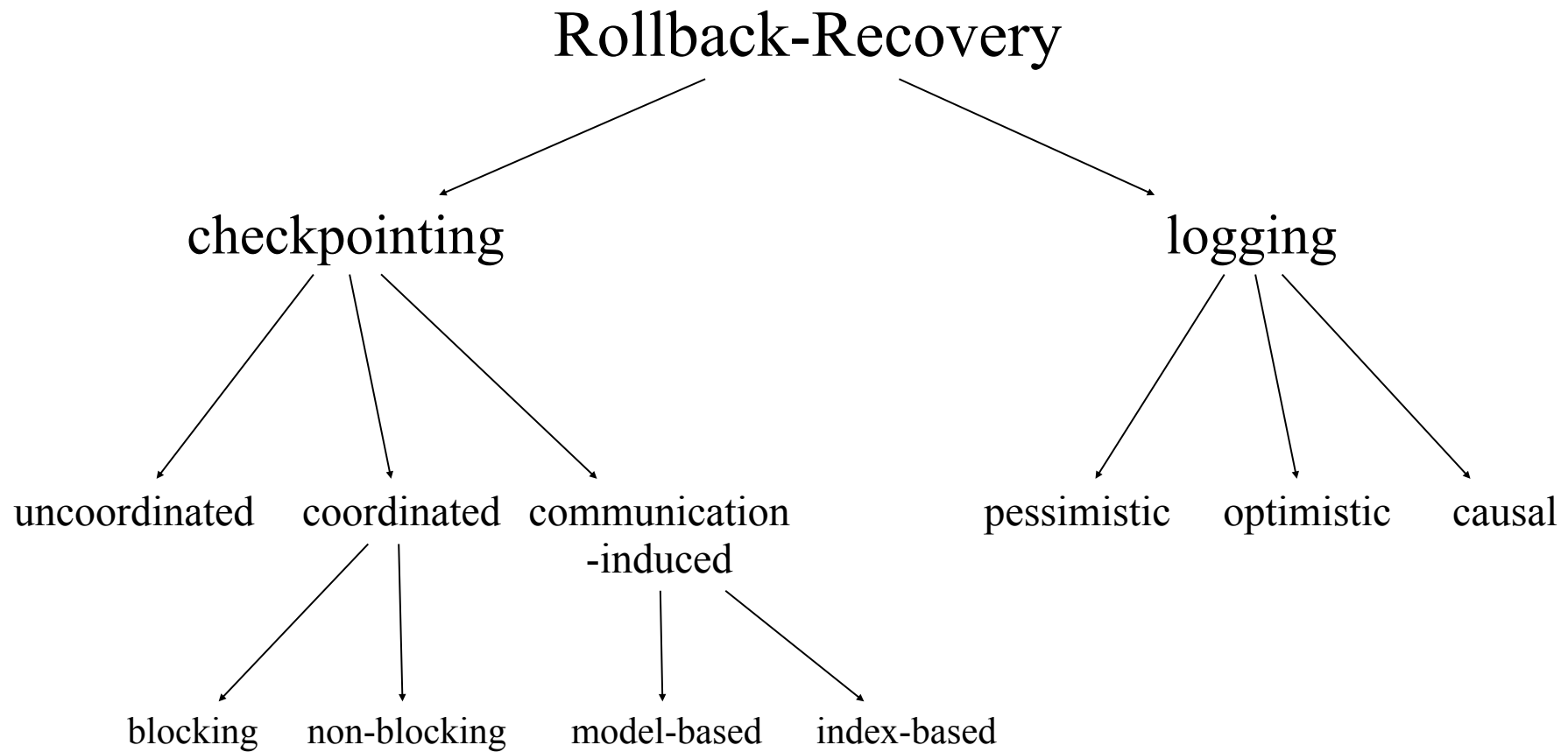


External world

Garbage collection

- ◆ **Checkpoints and event logs consume storage resources**
 - ◆ Can grow in an unbounded way
 - ◆ Subset of the stored information becomes useless
- ◆ **Garbage collection – common approach**
 - ◆ Identify the most recent consistent set of checkpoints, which is called the recovery line
 - ◆ Discard all information relating to events that occurred before the line
- ◆ **Pragmatic problem – but an important one**

Rollback-recovery - Taxonomy



Uncoordinated checkpointing

- ◆ **How it works**

- ◆ Each process autonomously decide when a checkpoint is needed

- ◆ **Advantages**

- ◆ Autonomy – take a checkpoint when it is most convenient
- ◆ e.g., when the amount of information to save is small

- ◆ **Disadvantages**

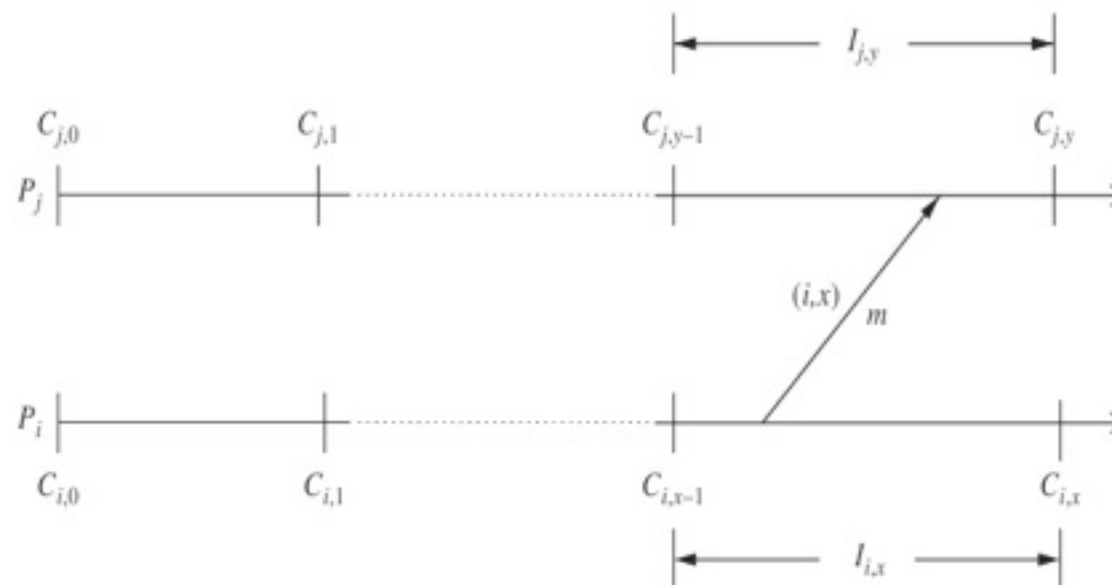
- ◆ Susceptible to domino effect
- ◆ Can generate useless checkpoints
- ◆ Complicates storage / garbage collection
- ◆ Not suitable for frequent output commits
 - ◆ Coordination among all processes to compute recovery lines, thus neglecting autonomy



Uncoordinated checkpointing

◆ Definitions

- ◆ let $C_{i,x}$ be the x checkpoint of process P_i
- ◆ let $I_{i,x}$ be the *interval* between checkpoints $C_{i,x-1}$ and $C_{i,x}$
- ◆ if P_i sends a msg m to P_j during $I_{i,x}$ and P_j receives m during $I_{j,y}$
 - ◆ P_i piggybacks (i,x) on m
 - ◆ P_j records the dependency from $I_{i,x}$ to $I_{j,y}$
 - ◆ ... which is later stored on stable storage when P_j takes $C_{j,y}$

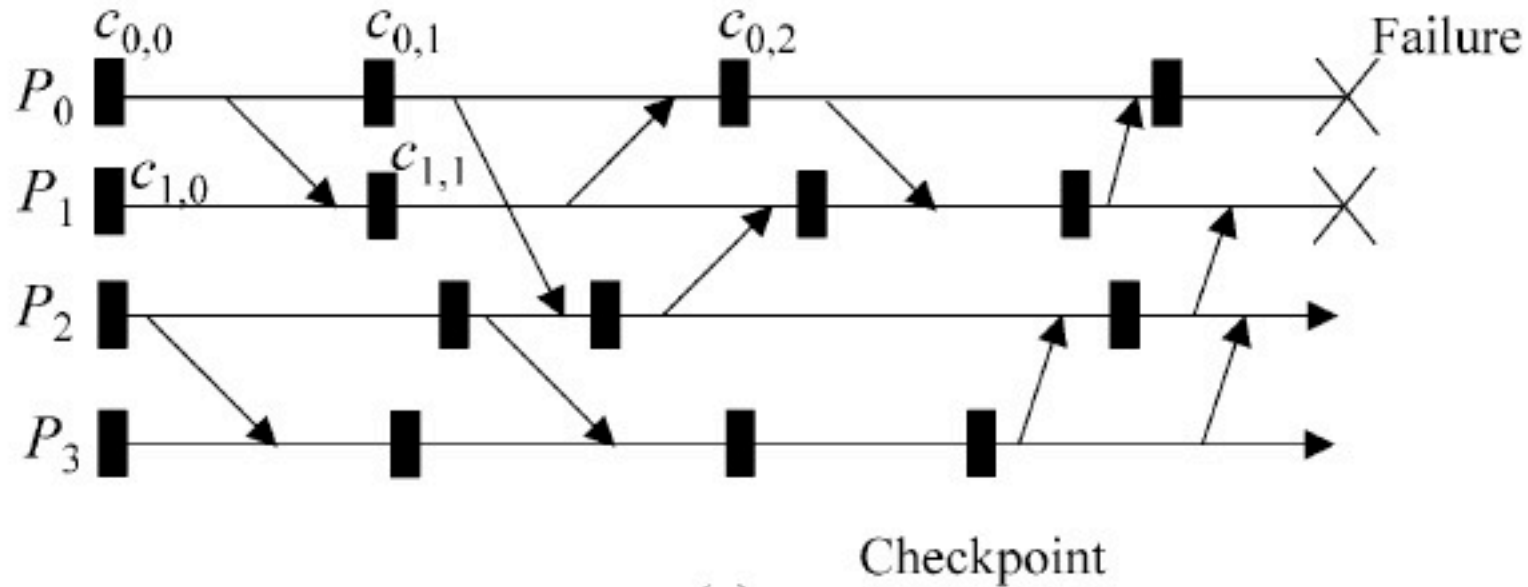


Uncoordinated checkpointing

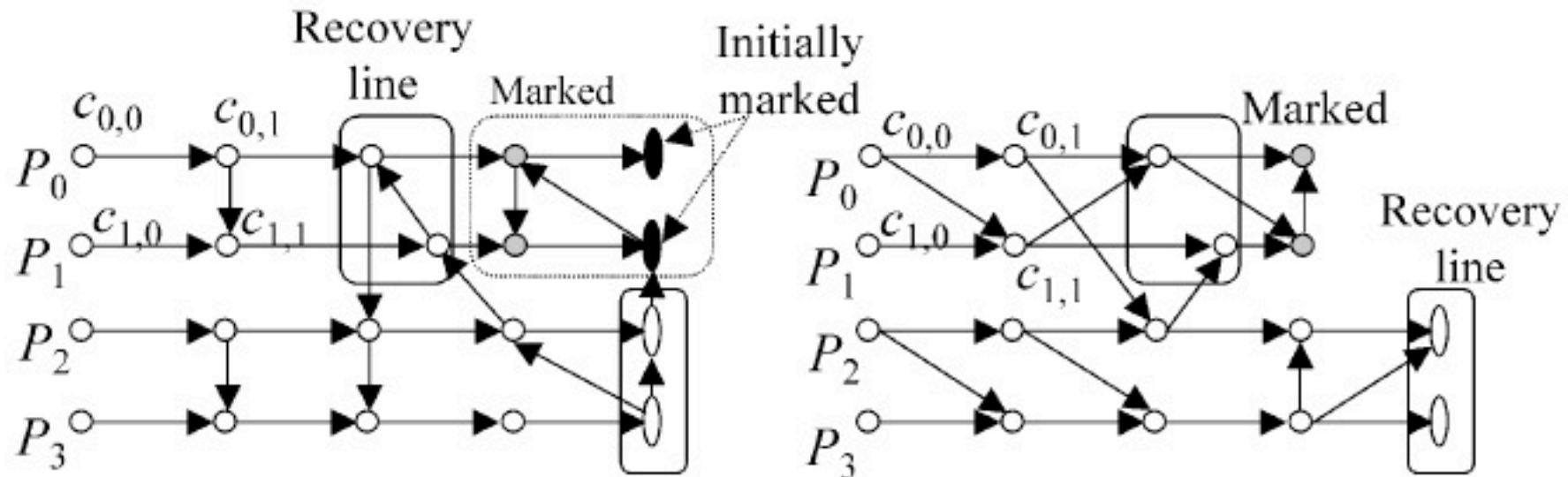
- ◆ **The recovering process**
 - ◆ Broadcast a dependency request to collect all dependency information maintained at each process
- ◆ **When a process receives this message**
 - ◆ it stops its execution
 - ◆ replies with dependencies stored on stable storage and those associated to current interval
- ◆ **The recovering process**
 - ◆ Computes the recovery line
 - ◆ Sends a rollback request containing the recovery line
- ◆ **When a process receives the rollback request**
 - ◆ Resume execution from specified checkpoint

Uncoordinated checkpointing

- Computing the recovery line – rollback dependency graph



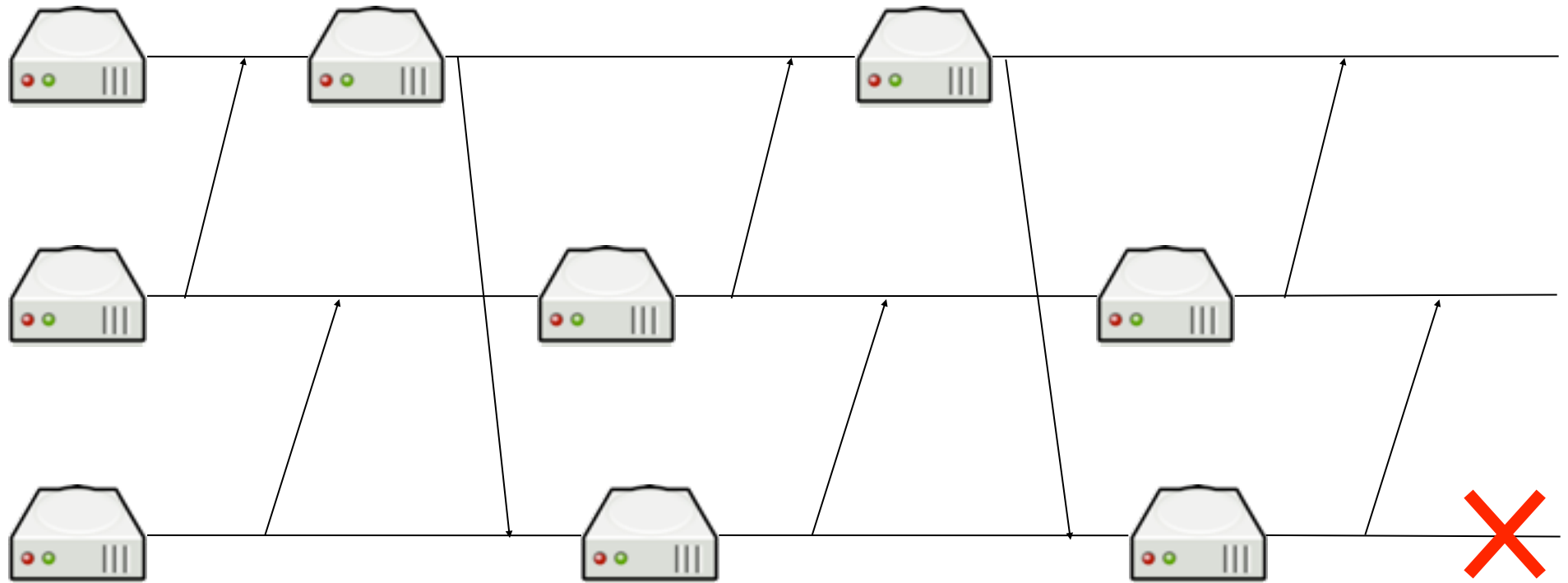
(a)



Uncoordinated checkpointing

- ◆ **Garbage collection**
 - ◆ can be based on periodically computing the recovery line and removing all the checkpoints prior to it
- ◆ **Removing “useless checkpoints”**
 - ◆ can be done in the same way
- ◆ **The big problem**
 - ◆ The domino effect

The Domino effect



Coordinated checkpointing

- ◆ **Coordinated checkpointing requires processes to orchestrate their checkpoints in order to form a consistent global state**
- ◆ **Advantages**
 - ◆ No domino effect
 - ◆ Each process is required to maintain only one checkpoint (the last)
 - ◆ No computation of recovery line is required
- ◆ **Disadvantages**
 - ◆ Large latency in committing output
- ◆ **Two approaches**
 - ◆ Blocking
 - ◆ Non-blocking

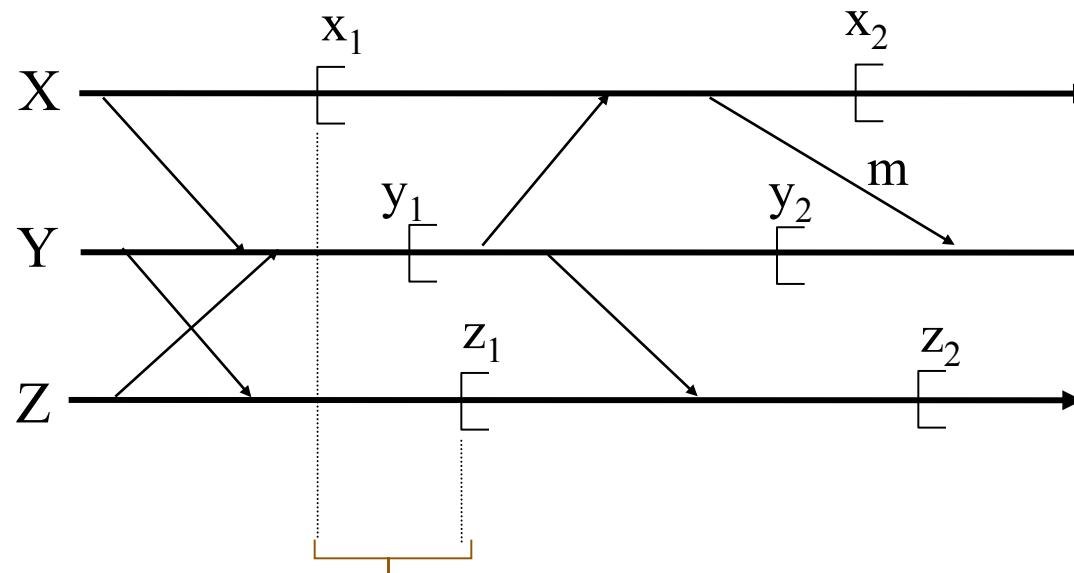
Coordinated checkpointing - blocking

◆ Coordinator

- ◆ sends “take checkpoint” to all →
- ◆ collects acks ←
- ◆ sends “commit” message →

◆ Processes

- ◆ suspend execution
- ◆ flush all messages
- ◆ take “tentative” checkpoints
- ◆ send ack
- ◆ transform “tentative” in permanent



Coordinate checkpointing – non blocking

- ◆ Goal – prevent orphan messages (avoid inconsistencies)
- ◆ Does anyone recall anything similar?

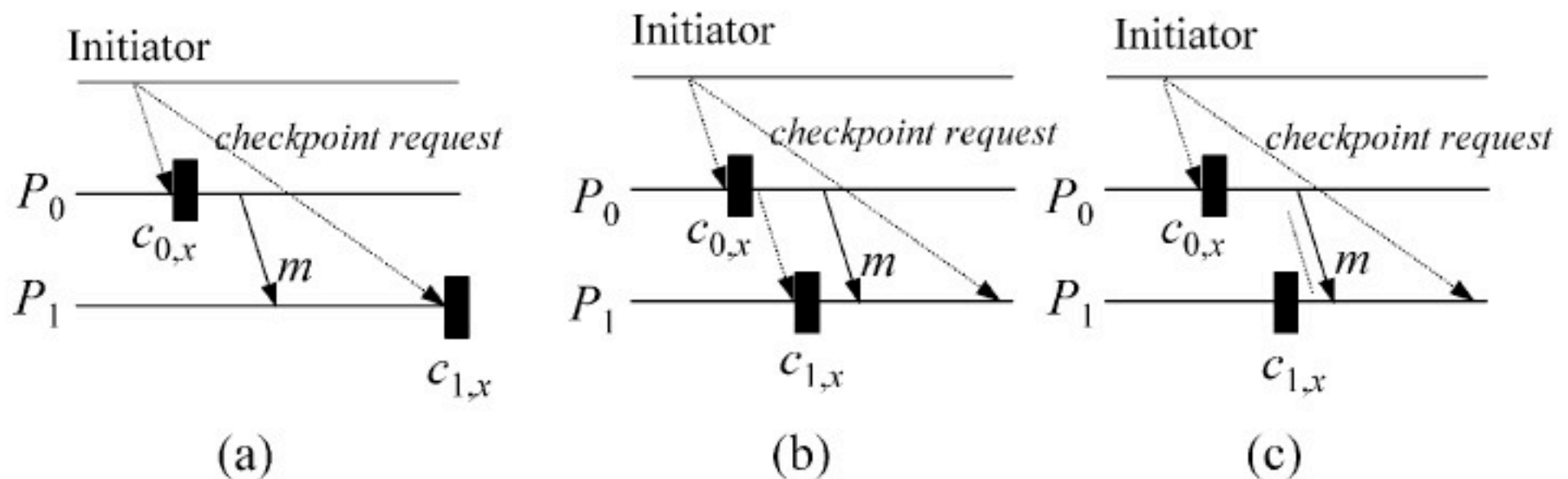
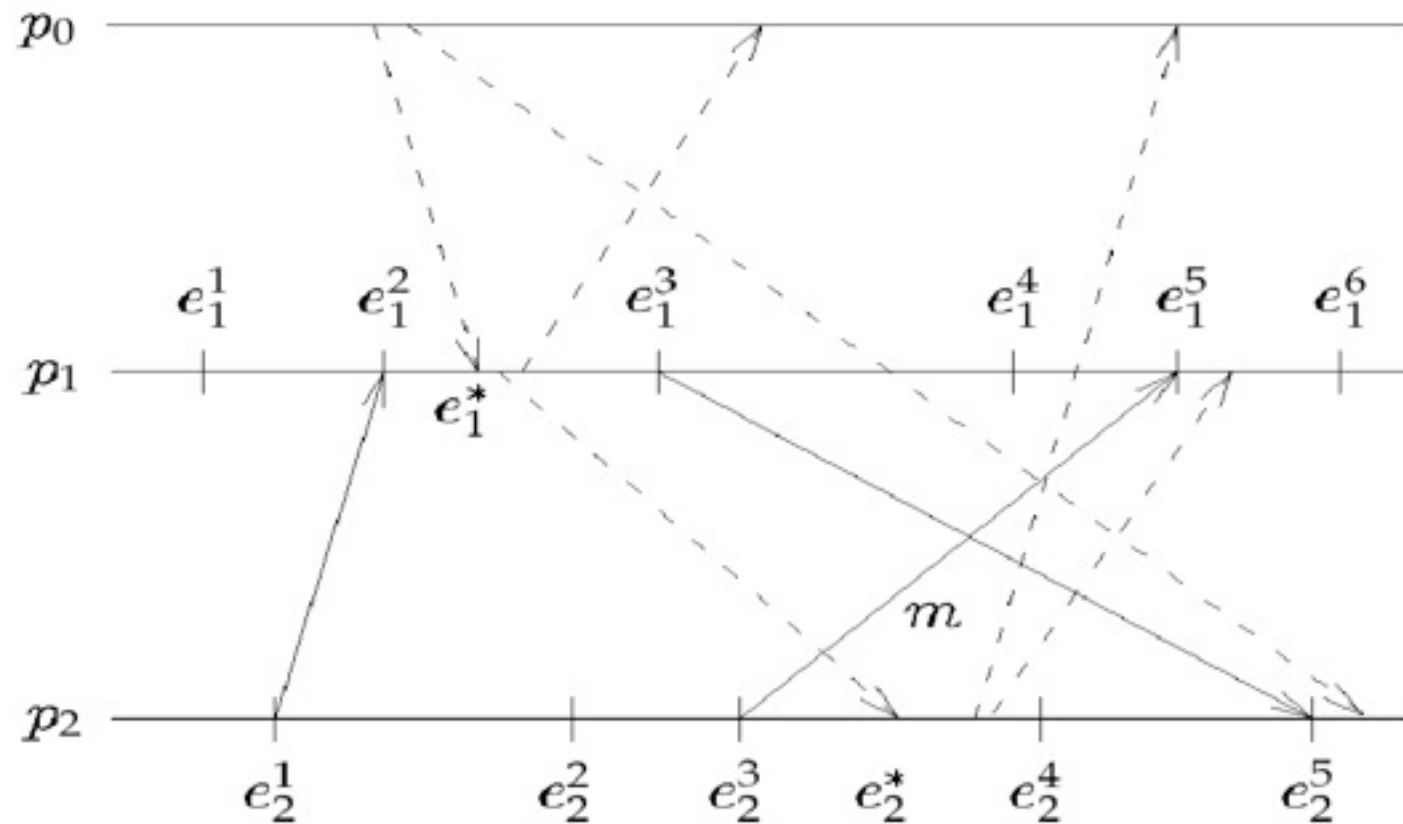


Fig. 8. Non-blocking coordinated checkpointing: (a) checkpoint inconsistency; (b) with FIFO channels; (c) non-FIFO channels (short dashed line represents piggybacked *checkpoint request*).

Coordinate checkpointing – non blocking

- ◆ Snapshot protocol (Chandy – Lamport)
 - ◆ In-transit messages must be recorded



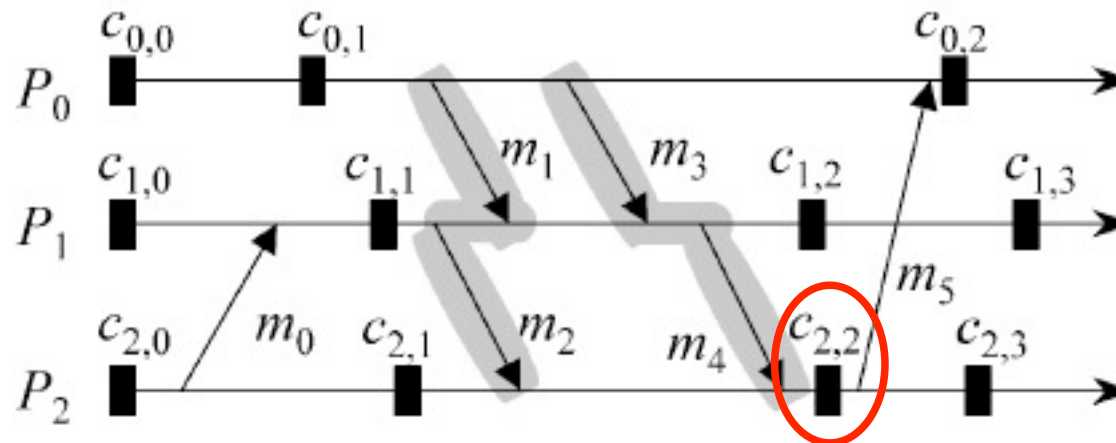
Coordinated checkpointing

- ◆ **Minimal checkpointing coordination**
 - ◆ Coordinated checkpointing requires all processes to participate
 - ◆ Scalability problem
- ◆ **Kuo and Toueg, 1987**
 - ◆ Only processes that communicated (directly or indirectly) with the coordinator need to take a checkpoint
 - ◆ Can be identified through recursive message sending

Communication-induced checkpointing (CIC)

◆ CIC Protocols

- ◆ avoid the domino effect without all checkpoints be coordinated
- ◆ two types of checkpoints
 - ◆ local taken independently
 - ◆ forced needed to guarantee progress of recovery line
avoid the creation of *useless checkpoints*



Communication-induced checkpointing (CIC)

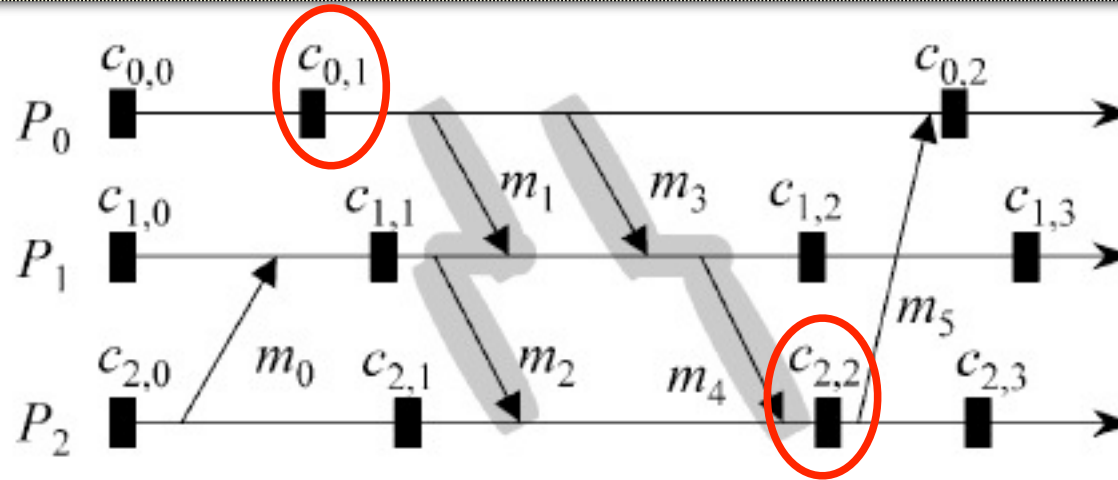
- ◆ **How CIC works?**
 - ◆ No special coordination messages
 - ◆ Information piggybacked on application messages
- ◆ **Informally**
 - ◆ Check whether past communication and checkpoints pattern can lead to the creation of a useless checkpoint
 - ◆ A forced checkpoint is taken to break these patterns
- ◆ **Elegant theory based on z-paths and z-cycles**

Z-path

◆ Definitions

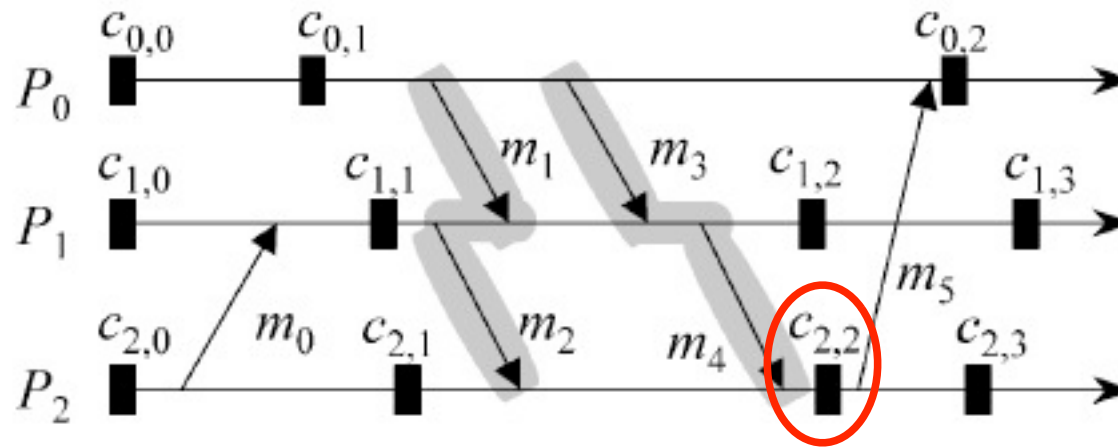
- ◆ let \rightarrow denote the happen-before relation of Lamport
 - ◆ let $C_{i,x}$ denote the x -th checkpoint of process P_i
 - ◆ let $I_{i,x}$ denote the interval between $C_{i-1,x}$ and $C_{i,x}$
- ◆ **Given two checkpoints $C_{i,x}$ and $C_{j,y}$, a Z-path exists between them if and only if one of the following conditions holds:**
- ◆ $x < y$ and $i = j$, or
 - ◆ there exists a sequence $[m_0, m_1, \dots, m_n]$, $n > 0$ such that
 - ◆ $C_{i,x} \rightarrow \text{send}_i(m_0)$
 - ◆ for all $b < n$, either $\text{receive}_k(m_b)$ and $\text{send}_k(m_{b+1})$ belongs to the same checkpoint interval, or $\text{receive}_k(m_b) \rightarrow \text{send}_k(m_{b+1})$
 - ◆ $\text{receive}_j(m_n) \rightarrow C_{j,y}$

Z-path



- ◆ Given two checkpoints $C_{i,x}$ and $C_{j,y}$, a Z-path exists between them if and only if one of the following conditions holds:
 - ◆ $x < y$ and $i = j$, or
 - ◆ there exists a sequence $[m_0, m_1, \dots, m_n]$ such that
 - ◆ $C_{i,x} \rightarrow \text{send}_i(m_0)$
 - ◆ for all $b < n$, either $\text{receive}_k(m_b)$ and $\text{send}_k(m_{b+1})$ belongs to the same checkpoint interval, or $\text{receive}_k(m_b) \rightarrow \text{send}_k(m_{b+1})$
 - ◆ $\text{receive}_j(m_n) \rightarrow C_{j,y}$

Z-cycle



- ◆ A Z-cycle is a Z-path that begins and ends with the same checkpoint
 - ◆ $[m_5, m_3, m_4]$
- ◆ It can be proved (Netzer and Xu 1995) that a checkpoint is useless if and only if it is part of a Z cycle
- ◆ If we avoid Z-cycles \rightarrow we avoid useless checkpoints

Communication-induced checkpointing

- ◆ **Model-based checkpointing**

- ◆ A model is set up to detect the possibility that Z-paths occurs
- ◆ Each process can locally decide whether a forced checkpoint is required

- ◆ **Problems**

- ◆ Forced checkpoints are un-coordinated
- ◆ Several additional checkpoints can be forced by distinct processes

Communication-induced checkpointing

◆ Index-based protocols

- ◆ If there are two checkpoints $C_{i,x}$ and $C_{j,y}$ such that $C_{i,x} \rightarrow C_{j,y}$ then $ts(C_{i,x}) \geq ts(C_{j,y})$ (timestamps)
- ◆ consecutive local checkpoints have increasing checkpoints
- ◆ timestamps are piggybacked on messages

◆ Briatico et al.

- ◆ When a process receives a timestamp greater than the local one, a forced checkpoint is taken (with same timestamp)
- ◆ Guarantees that checkpoints having the same index at different processes form a consistent state

Communication-induced checkpointing

◆ Index-based protocols

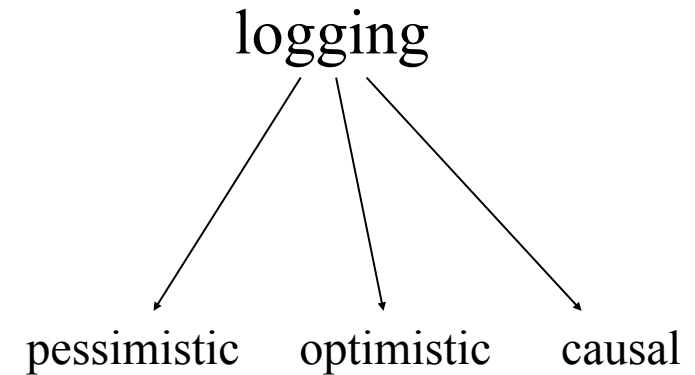
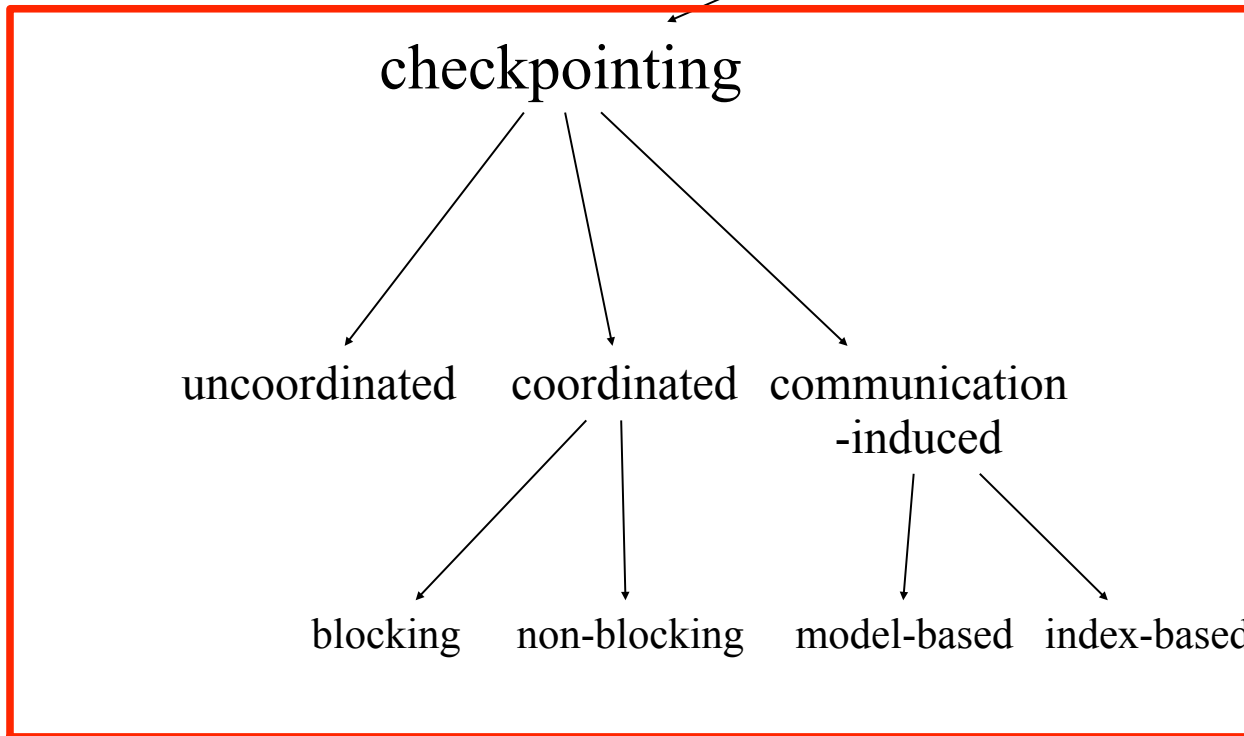
- ◆ If there are two checkpoints $C_{i,x}$ and $C_{j,y}$ such that $C_{i,x} \rightarrow C_{j,y}$ then $ts(C_{i,x}) \geq ts(C_{j,y})$ (timestamps)
- ◆ consecutive local checkpoints have increasing checkpoints
- ◆ timestamps are piggybacked on messages

◆ Helary et al.

- ◆ If checkpoints timestamps always increase along a Z-path, then no Z-cycles can form
- ◆ (note: always increase different from “not decrease”)

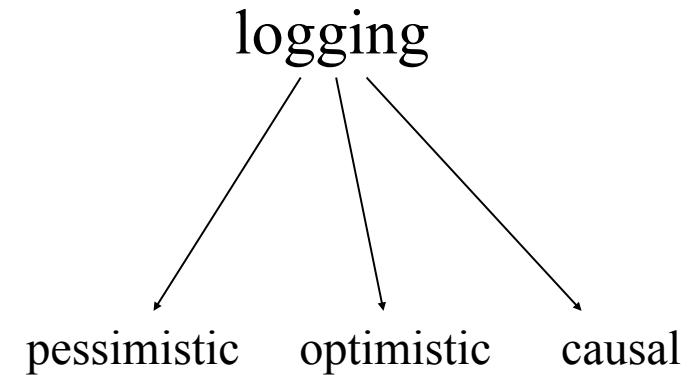
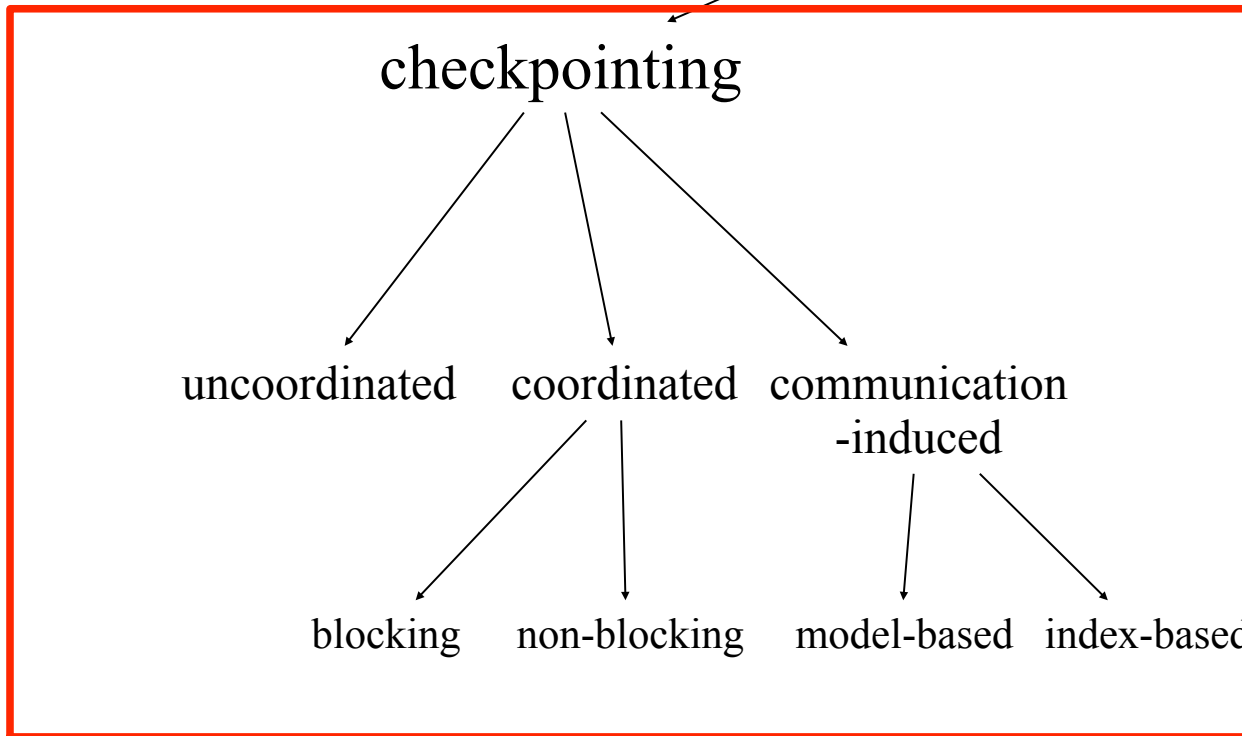
Rollback-recovery - Taxonomy

Rollback-Recovery



Rollback-recovery - Taxonomy

Rollback-Recovery



Log-based rollback recovery

- ◆ **Message Logging**
 - ◆ Can avoid domino effect
 - ◆ Works with coordinated checkpoint
 - ◆ Works with uncoordinated checkpoint
 - ◆ Can reduce cost of output commit
 - ◆ **More difficult to implement**
- ◆ **Not limited to “messages” though; the name comes from historical reasons**

Log-based rollback recovery

- ◆ **Deterministic state intervals**
 - ◆ Started by a nondeterministic event
 - ◆ Message receipt
 - ◆ Internal events
 - ◆ Note: message sending is not non-deterministic
- ◆ **Assumption**
 - ◆ All non-deterministic events can be identified and their corresponding determinant can be logged to stable storage

Log-based rollback-recovery

◆ How it works

- ◆ Each process logs on stable storage the determinants of all non-deterministic events
- ◆ Additionally, checkpoints are taken to reduce the extent of rollback during recovery

◆ In case of recovery

- ◆ Checkpoints are used to restore state
- ◆ Nondeterministic events are replayed precisely as they occurred
- ◆ Recovery works up to the first non-deterministic event whose determinant is not logged

◆ What are determinants, exactly?

Orphan process

◆ Definition

- ◆ A process p becomes an orphan when p does not fail and p 's state depends on the execution of a nondeterministic event e whose determinant cannot be recovered
 - ◆ from stable storage
 - ◆ from the volatile memory of a surviving process

◆ The management of the orphan process problem may have effects on:

- ◆ failure-free performance overhead
- ◆ latency of output commit
- ◆ garbage collection
- ◆ recovery mechanism

Logging “flavors”

◆ Pessimistic

- ◆ Orphans never created
- ◆ Simple recovery, garbage collection and output commit
- ◆ High failure-free overhead

◆ Optimistic

- ◆ Allow orphans to be created
- ◆ Reduce failure overhead
- ◆ Orphans complicate recovery, garbage collection, output commit

◆ Causal

- ◆ Tries to combine the best of pessimistic, optimistic
 - ◆ Low performance overhead
 - ◆ Fast output commit
- ◆ But still complicate recovery, garbage collection

Specification

- ◆ **Let e be a nondeterministic event executed by p**
- ◆ **$depend(e)$**
 - ◆ the set of processes that are affected by e plus any process whose state depends on e according to Lamport's happened before
- ◆ **$log(e)$**
 - ◆ the set of processes that have logged a copy of e in their *volatile* memory
- ◆ **$stable(e)$**
 - ◆ a predicate that is true if e 's determinant is logged on stable storage

Specification

- ◆ **Always-no-orphan**
 - ◆ $not\ stable(e) \Rightarrow depend(e) \subseteq log(e)$
 - ◆ if a process depends on a event e , either e is on stable storage, or the process has a copy of the determinant of event e
 - ◆ otherwise, it is orphan
- ◆ **Compare it with “eventually-no-orphan”**
 - ◆ processes may become orphan but eventually they recover to a non-orphan state

Pessimistic logging

◆ Pessimistic

- ◆ Failures can happen – at any time!
- ◆ (In reality, failures are rare)

◆ How it works

- ◆ Each nondeterministic event is logged on stable storage before it can affect the computation
- ◆ Synchronous logging: $not\ stable(e) \Rightarrow depend(e) = \emptyset$
- ◆ If an event has not been logged to stable storage, no process can depend on it

Pessimistic logging

◆ Advantages

- ◆ Interaction with OWP without running a special protocol
- ◆ Process restart from their most recent checkpoint upon a failure
 - ◆ limits the amount of computation that must be replayed
 - ◆ frequency of checkpoints → trade-off overhead/protection
- ◆ Recovery is simple
 - ◆ The effects of a failure are confined to the processes that fail
 - ◆ Processes never become orphans
- ◆ Garbage collection is simple
 - ◆ Collect everything before last checkpoint

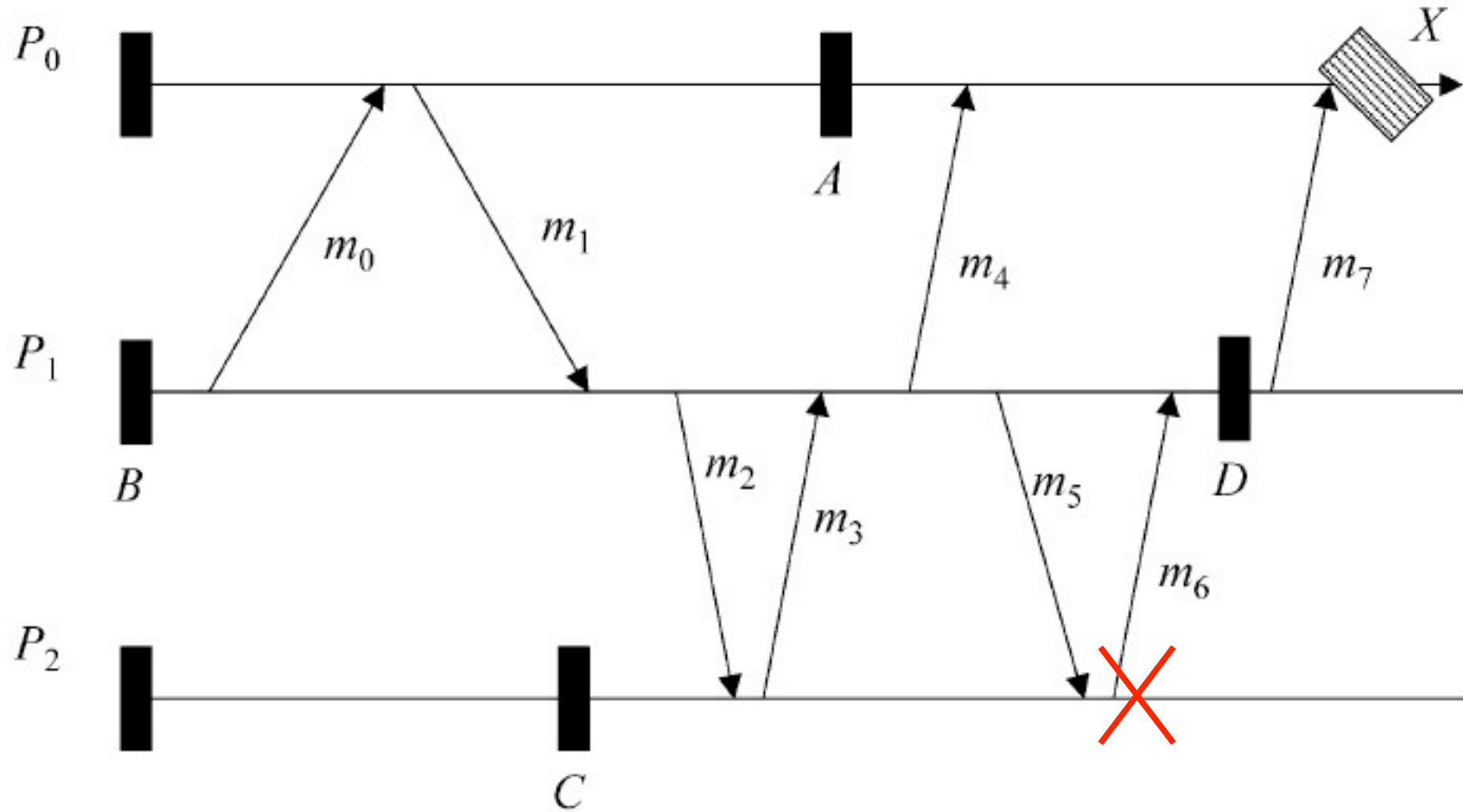
◆ Disadvantages

- ◆ Performance penalty for synchronous logging

Optimistic logging

- ◆ **Optimistic**
 - ◆ Failures are rare
 - ◆ They do not occur before determinant is logged to stable storage
- ◆ **How it works**
 - ◆ Determinants are logged to volatile memory
 - ◆ Periodically flushed to stable storage
 - ◆ No blocking is necessary
- ◆ **Problems: if a process fails before determinant is written to stable storage**
 - ◆ determinants will be lost
 - ◆ other processes may become orphans
 - ◆ always-no-orphans is not guaranteed

Optimistic logging



Optimistic logging

- ◆ **Problems**
 - ◆ Garbage collection is complex
 - ◆ Recovery is complex
 - ◆ Output commit requires global coordination

Causal logging

- ◆ **Advantages of optimistic logging**

- ◆ Logging is performed asynchronously
- ◆ (Except for output commit)

- ◆ **Advantages of pessimistic logging**

- ◆ Each process can independently perform output commit
- ◆ Always-no-orphans is satisfied

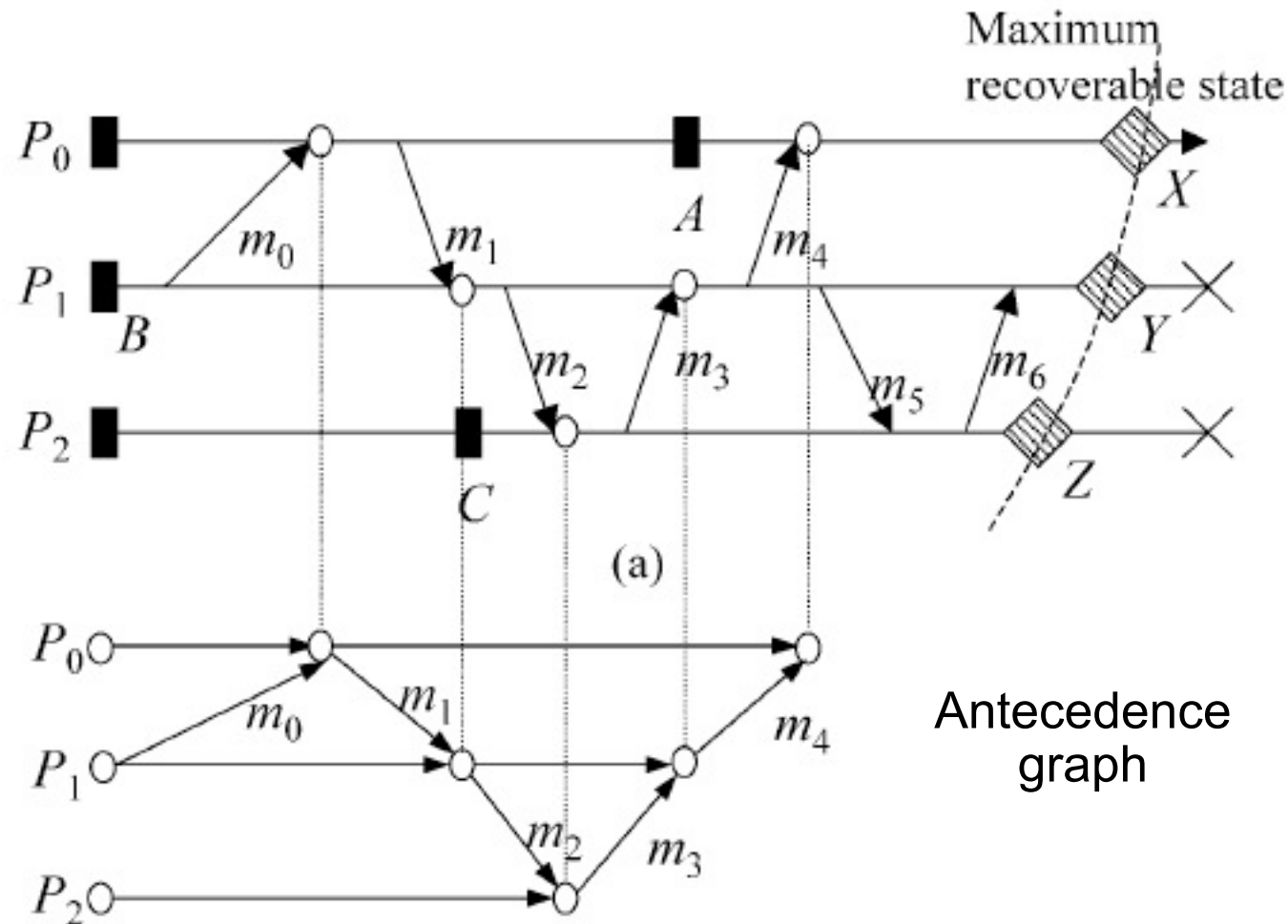
- ◆ **How it works**

- ◆ The determinant of each event that causally precedes the state of a process is either stable or available locally to that process
- ◆ Determinants of causally-preceding events are piggy-backed on messages

Causal logging

What happens

- ◆ m_5 m_6 are lost
- ◆ m_0 m_1 m_2 m_3 m_4 determinants are logged on P_0
- ◆ Determinants contain order of receipt
- ◆ Message sender logs the content
- ◆ P_0 will be able to “guide” the other processes to replay messages in the right order
- ◆ Content is recovered is either recovered from the sender log of P_0 or deterministically-regenerated by replaying events



Causal logging - implementations

- ◆ **Manetho (Elnozahy 1993)**

- ◆ Several optimizations

- ◆ If p sends several messages to q , it can piggyback only the difference between the various antecedence graphs

- ◆ If also q sends messages to p , the information piggybacked by q to p needs not to be resent to q

- ◆ Other optimizations are possible – depends on message semantics

- ◆ Very low overhead after all

Causal logging - implementations

- ◆ **Family-Based Logging (Alvisi 1996)**
 - ◆ Configurable tolerance – up to f failures
 - ◆ Log determinants in at least $f+1$ other processes
 - ◆ Unlike Manetho, piggybacking can stop after $f+1$ other processes have logged an event
 - ◆ For $f < N$, stable storage is accessed only for checkpointing
 - ◆ Applications pay only the overhead that corresponds to the number of failures they are willing to tolerate
 - ◆ Manetho $\rightarrow f = N$

Table I. A Comparison Between Various Flavors of Rollback-Recovery Protocols

	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