Distributed Algorithms

Faults and Recovery

Ludovic Henrio
CNRS – LIP (ens Lyon) – Cash team
ludovic.henrio@cnrs.fr

Sources:
- A survey of rollback-recovery protocols in message-passing systems (Elnozahy, Alvisi, Wang and Johnson)
- Distributed systems (Tanenbaum and Van Steen)

Outline

- Generalities: Faults, redundancy, stable storage
- Background + Recovery principles
- Rollback-recovery protocols
  - focus **Checkpointing protocols**
    Coordinated vs. uncoordinated
    Communication induced checkpointing
  - message logging
- Exercises

GENERALITIES ABOUT FAULTS AND RECOVERY

Failure Models

- Different types of failures.

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

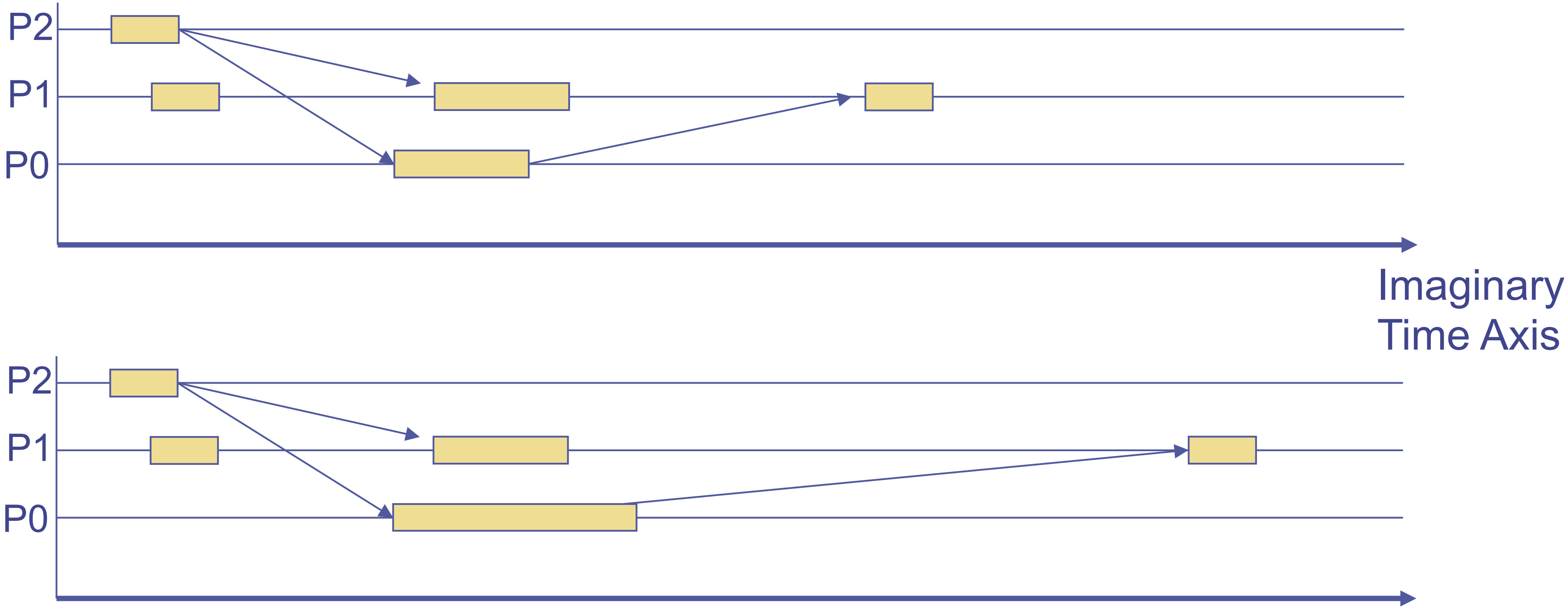
- A system is **k-fault tolerant** if it can survive faults in k components and still meet its specification

Stable storage – a prerequisite for recovery

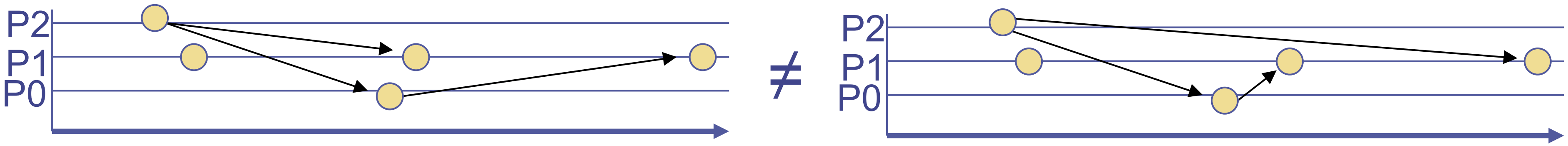
- In a system that tolerates only *a single failure*, stable storage may consist of **the volatile memory of another process**
- In a system that wishes to tolerate an arbitrary number of *transient failures*, *stable* storage may consist of **a local disk in each host.**
- In a system that tolerates *non-transient failures*, stable storage must consist of a **persistent medium outside the host on which a process is running.** A **replicated file system** is a possible implementation in such systems

BACKGROUND: MODELLING DISTRIBUTED EXECUTIONS

Execution representation: time diagram

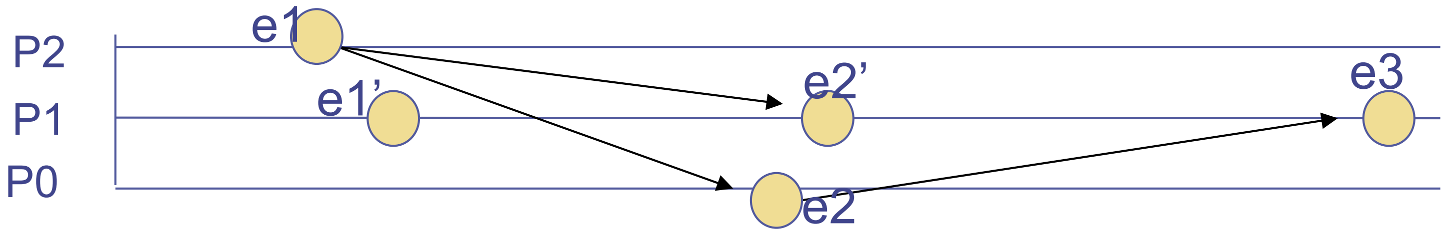


- these execution are identical -> event representation
- Only the order of message reception matters, whatever the transmission and execution duration



Happened-before relation: →

- When 2 events e1, e2,
  - Are local to a process Pi, e1 → e2
  - e1: message send on Pi, e2: corresponding message reception on Pj, e1 → e2
- Several events, e1, e2, e3 (transitivity)
  - If e1 → e2, and e2 → e3, then, e1 → e3
- Not all events are mandatorily related along →
  - Incomparable, independent, concurrent: ||
    - Non transitivity of ||
- Happened-before relation: also named Causality (partial order)



e1 →e2
e1 →e2'
e2 →e3
e1 →e3
e1' →e2'
e2' →e3
e1' →e3

e1 || e1'
e2 || e2'

e1' || e1
e2 || e1'
e2' || e2

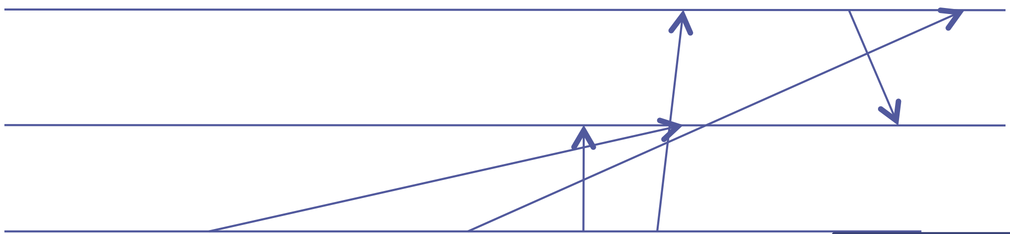
Happened Before [Lamport] = Asynchronous Communication

- asynchronous communications, any order is valid (provided messages are received after being sent)
- $(s,r) \in \Gamma$ **is** a communication
- $\prec_i$ local causality relation (total order on LOCAL events) ➔ sequentiality of local processes
- Global causality $\prec$,verifies at least:

$a \prec_i b \Rightarrow a \prec b$

$s \prec r \quad \text{if } (s,r) \in \Gamma$

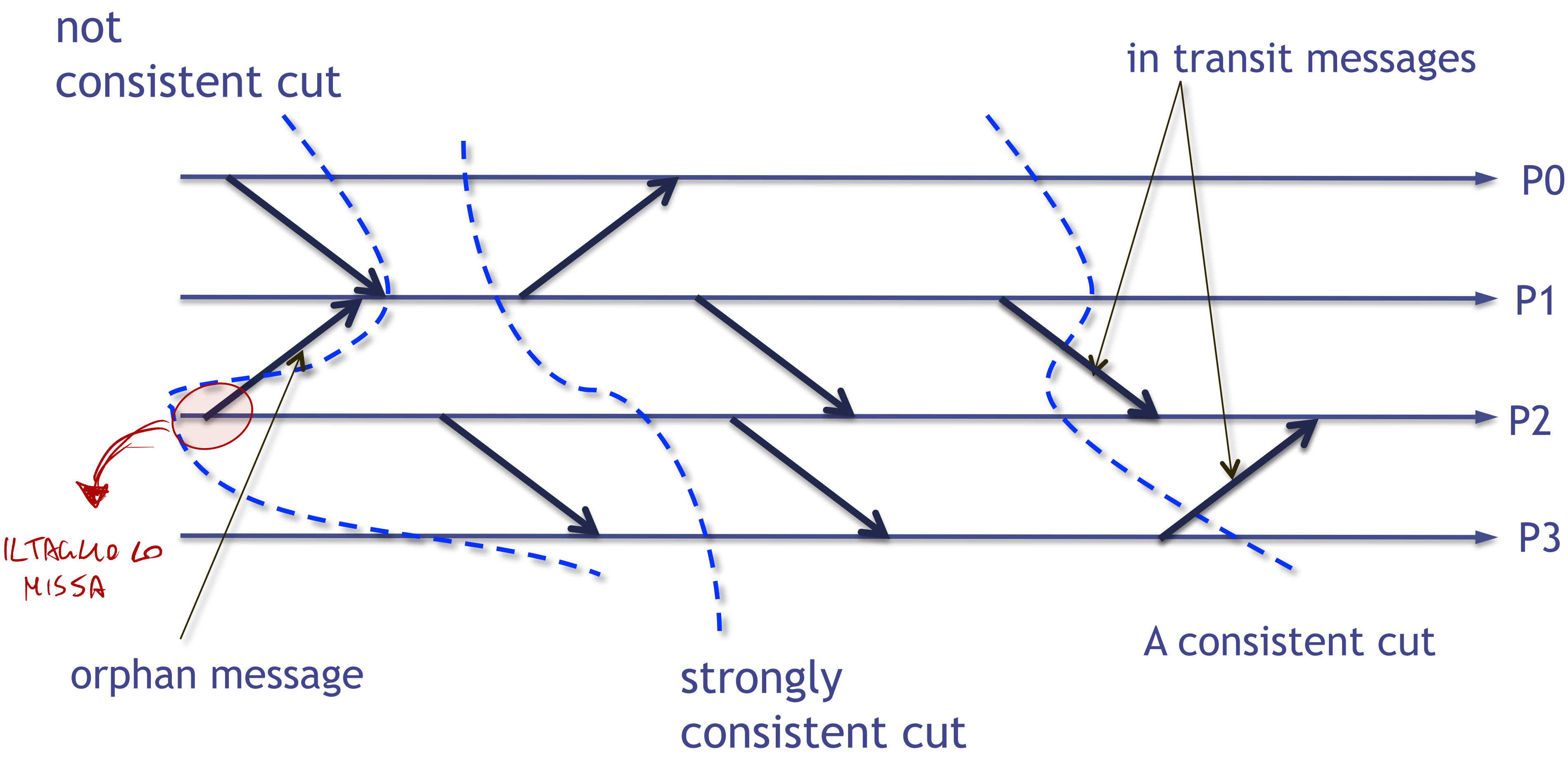
+ transitivity: If $e1 \prec e2$, and $e2 \prec e3$, then, $e1 \prec e3$



If $\prec$ is a partial order (antisymetric) then it represents a valid asynchronous communication i.e. there must be no cycle of different events

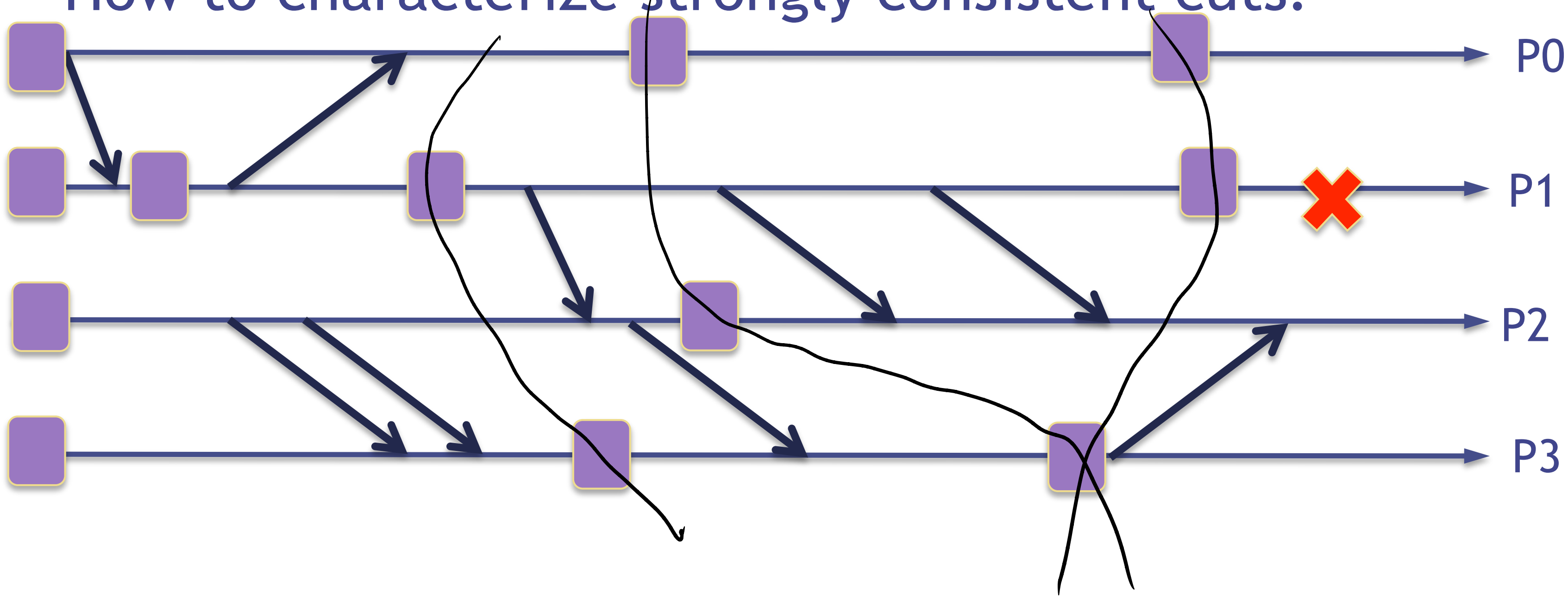
Question: Do you know what FIFO message ordering is? Causal ordering? How to characterise it?

Cuts / consistent cuts



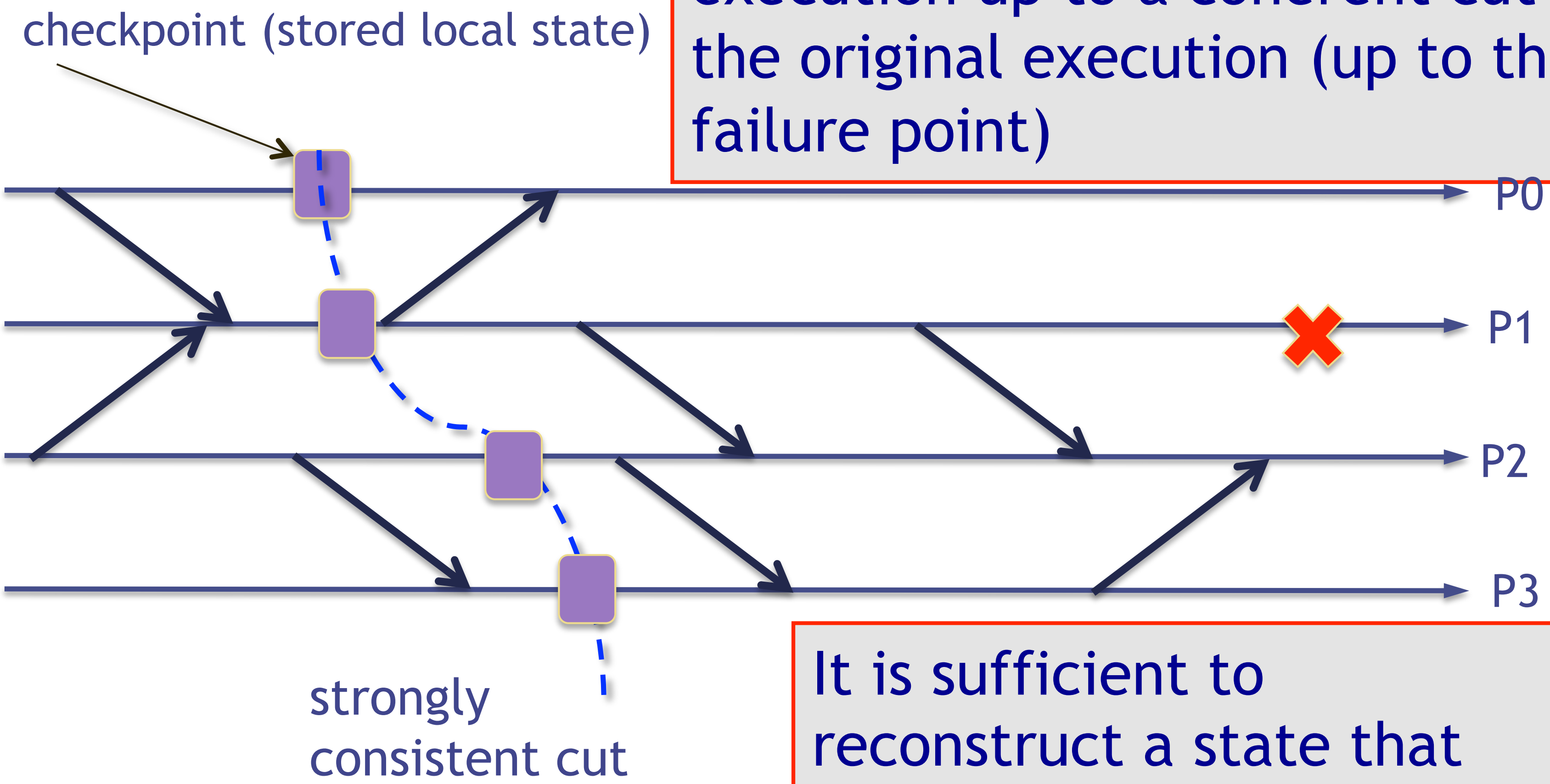
exercise

- Find a few consistent cuts in the figure below (passing by)
- Order the according to happened before
- Characterise a consistent cut based on the happened before relation
- How to characterize strongly consistent cuts?



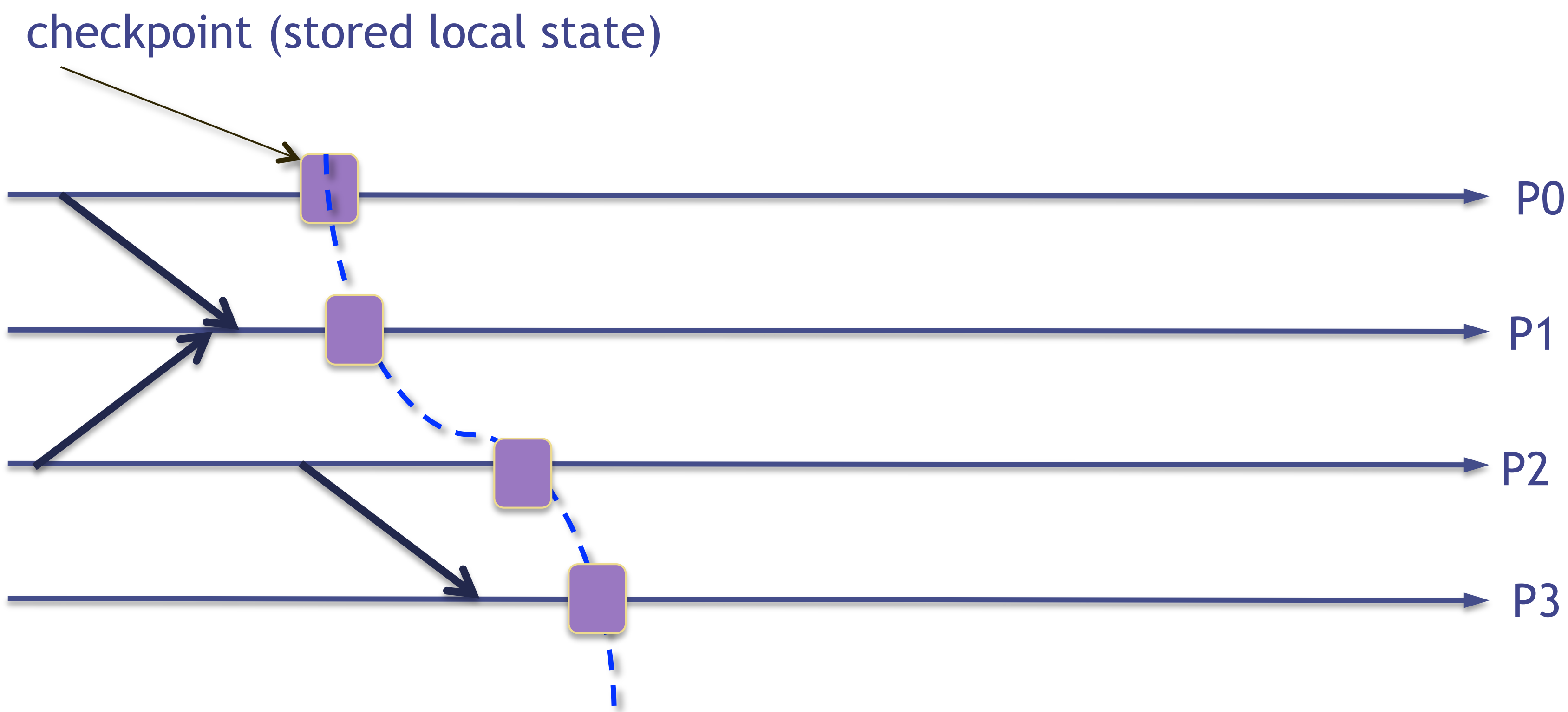
Recovery: Principles:

A recoverable state contains enough information to replay an execution up to a coherent cut of the original execution (up to the failure point)



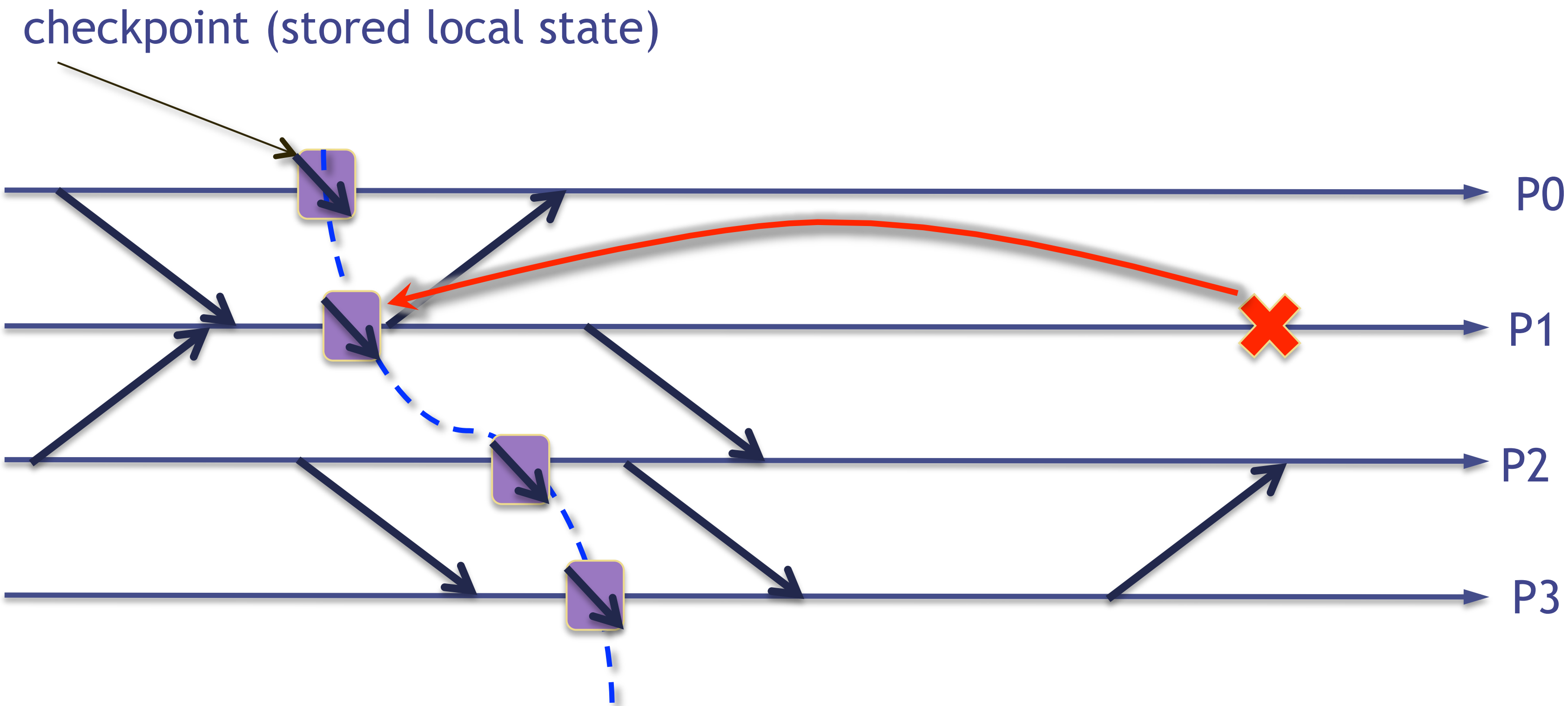
It is sufficient to reconstruct a state that *could* have occurred in a failure-free execution

Recovery: Principles:
1 - Checkpointing



Restart all, or almost all, processes from a consistent cut and let a new execution run

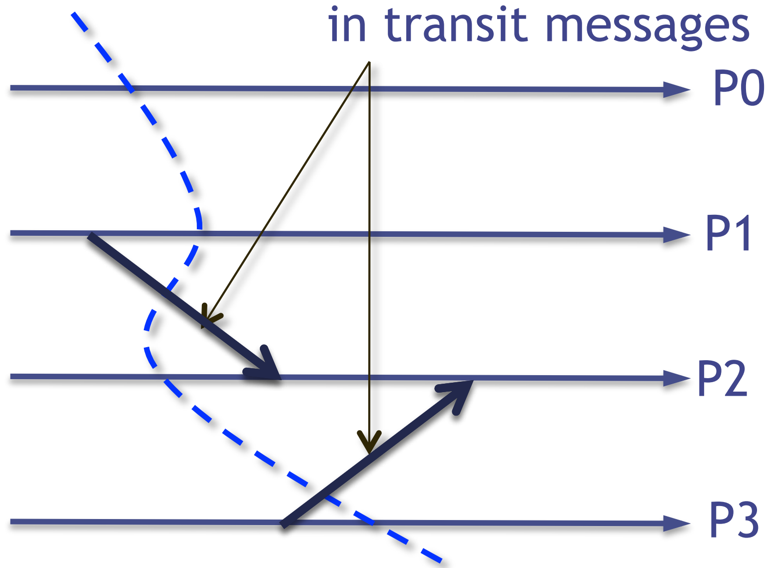
Recovery: Principles:
2 – Message Logging



Only one (or a few) process recover and use message information to replay the previous execution until reaching the failure point

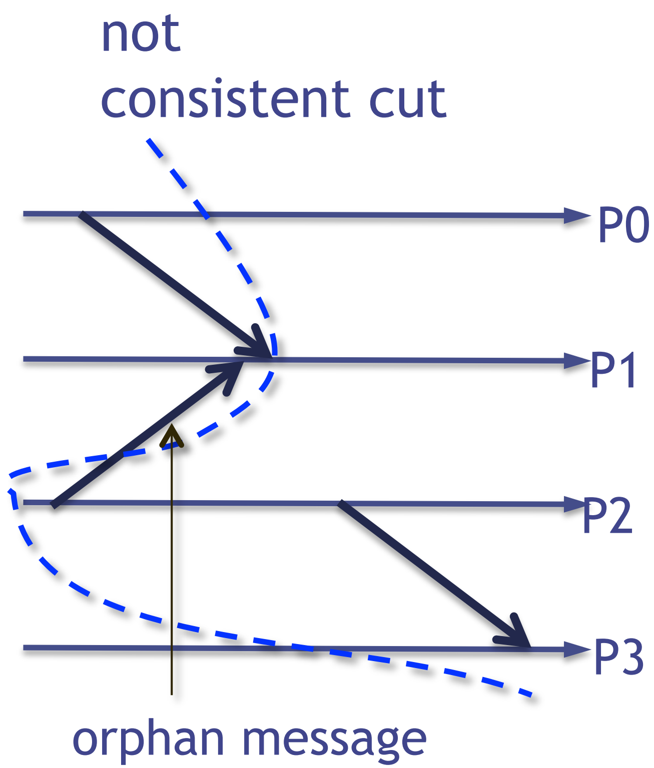
In transit messages

- If message delivery is not guaranteed, they are not a problem!
- But if the communication protocol is reliable, they must be taken into account
  ➡ We have to store them (they are part of the recoverable state)



Orphan messages

- If P2 fails and restarts from the cut, the message will be re-emitted and received twice by P1
  - Either avoid using inconsistent cuts (in general for checkpointing)
  - Or avoid re-emitting the message (in general for message logging) **and replay the same execution**

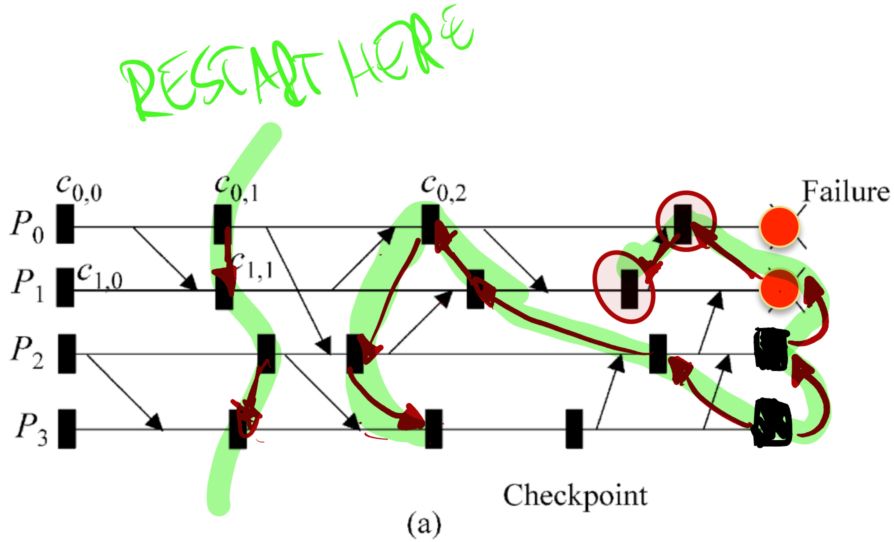


RECOVERY: CHECKPOINTING MECHANISMS

Checkpoint-based rollback recovery – Uncoordinated checkpointing

- **Hypothesis: Fail stop**
- Each process takes checkpoints from time to time
- Upon failure we first restart enough machines
- Then we compute the recovery line
  - A process (eg the failed one) initiates the process
  - Collects dependencies information from all the processes
  - Computes the recovery line and triggers recovery

Example: exercise 1

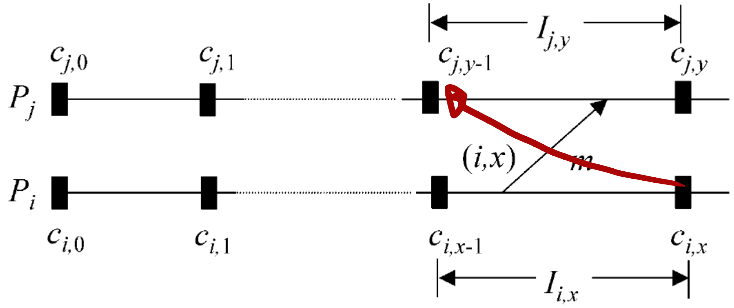


(a)

rollback-dependency graph [Bhargava and Lian 1988] in which each node represents a checkpoint and a directed edge is drawn from $c_{i,x}$ to $c_{j,y}$ if either:

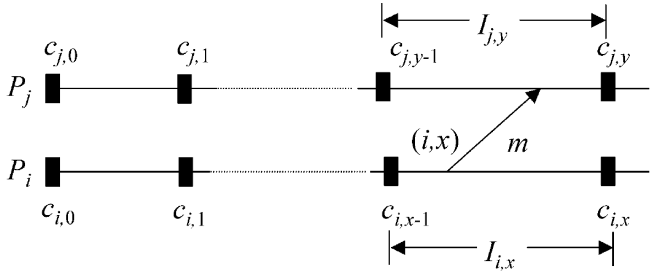
(1) $i \neq j$, and a message m is sent from $I_{i,x}$ and received in $I_{j,y}$, or

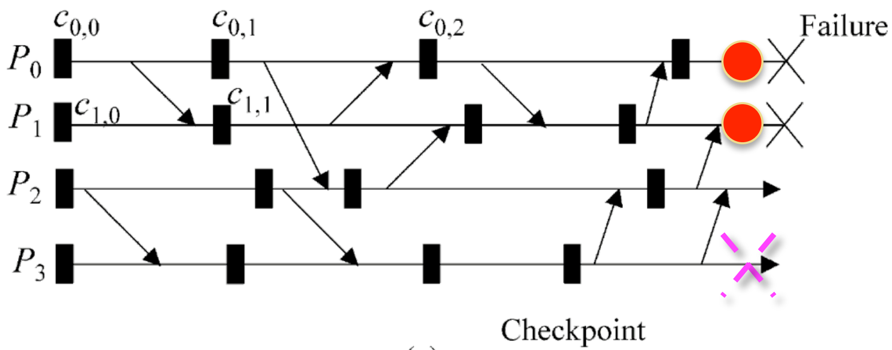
(2) $i = j$ and $y = x + 1$.



The algorithm used to compute the recovery line first marks the graph ~~nodes~~ corresponding to the states of processes *P0 and P1 at the failure point* (red ellipses). It then uses reachability analysis to mark all reachable nodes from any of the initially marked nodes. The union of the *last unmarked nodes* over the entire system forms the recovery line,

Example: exercise 1



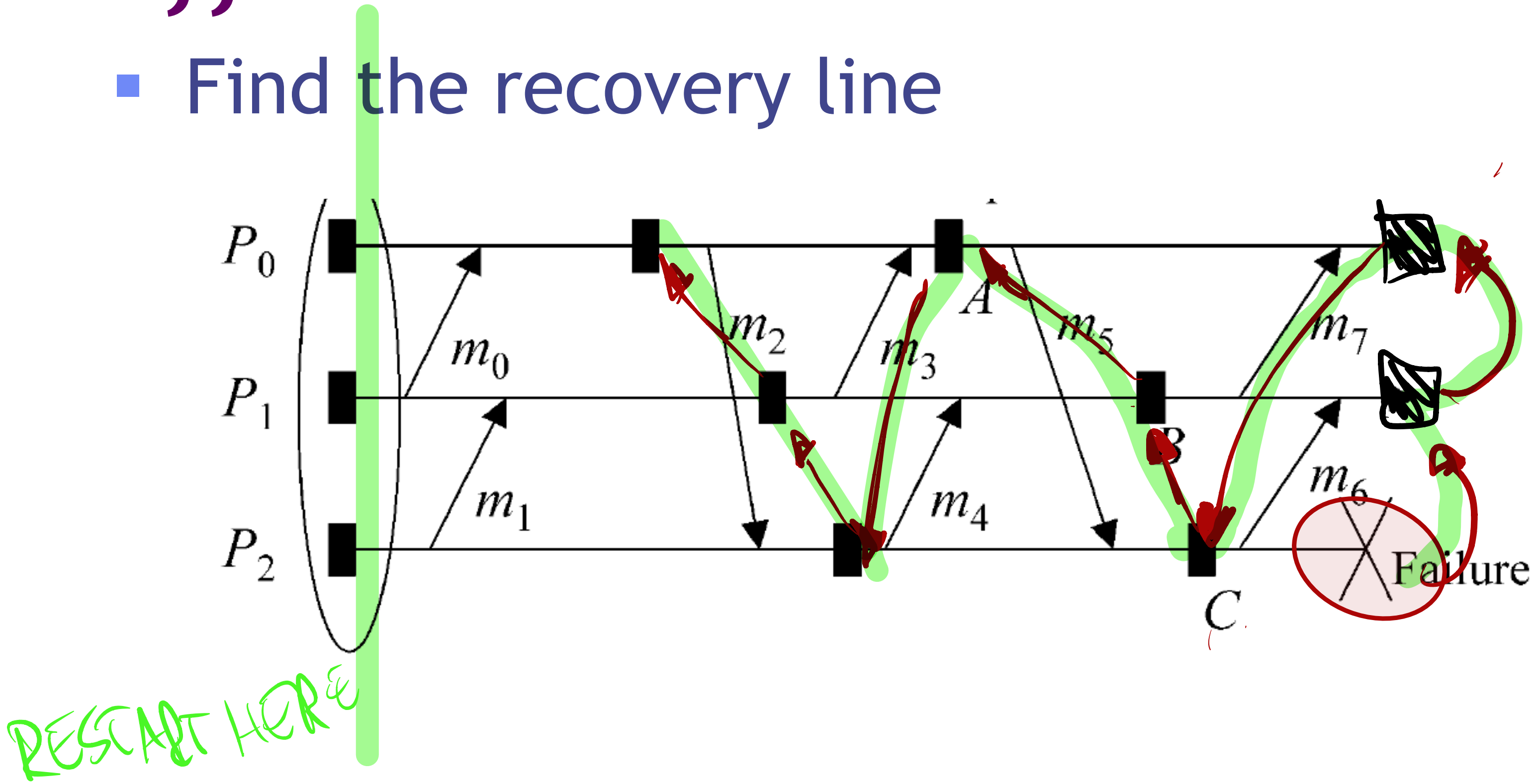


(a)

1 - build the rollback dependency graph
2 – What is the recovery line?
3 – What if P3 fails instead? ( )

Exercise 1 contd: the *domino effect*

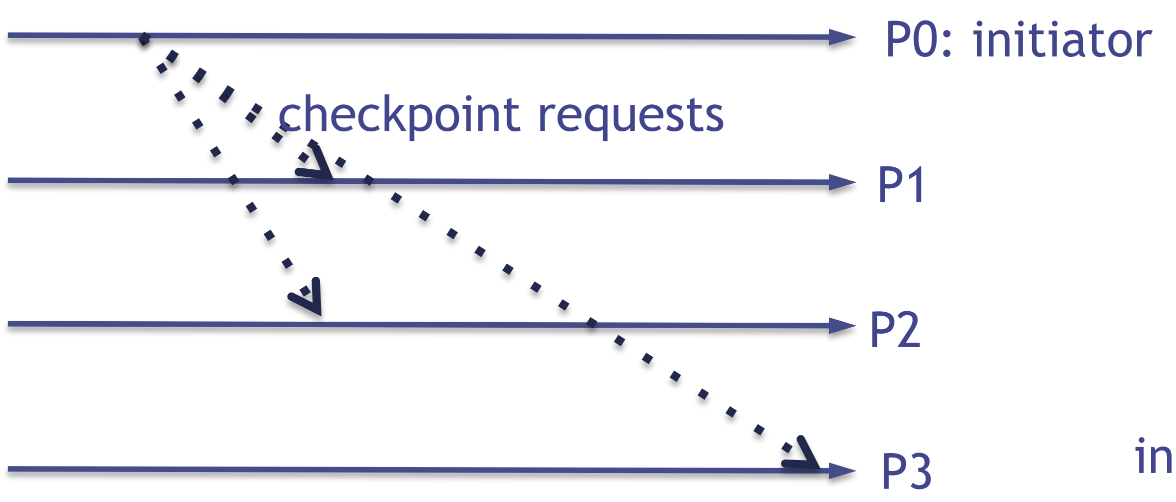
- Find the recovery line

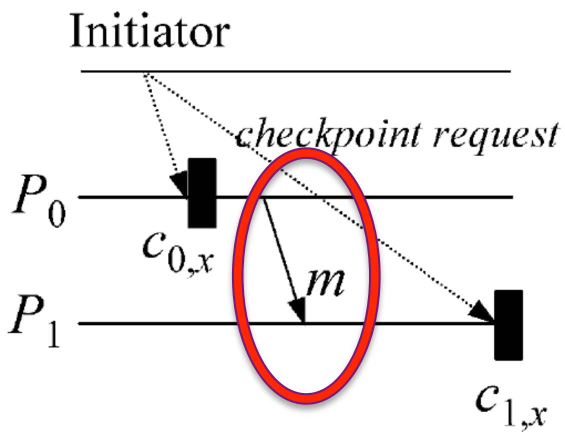


Conclusion: let us synchronize checkpoints !!!

Coordinated checkpointing

- There is an initiator process for the checkpointing
  - Only one (or 2) checkpoint per process (always consistent)
  - large latency: processed blocked until checkpoint is finished



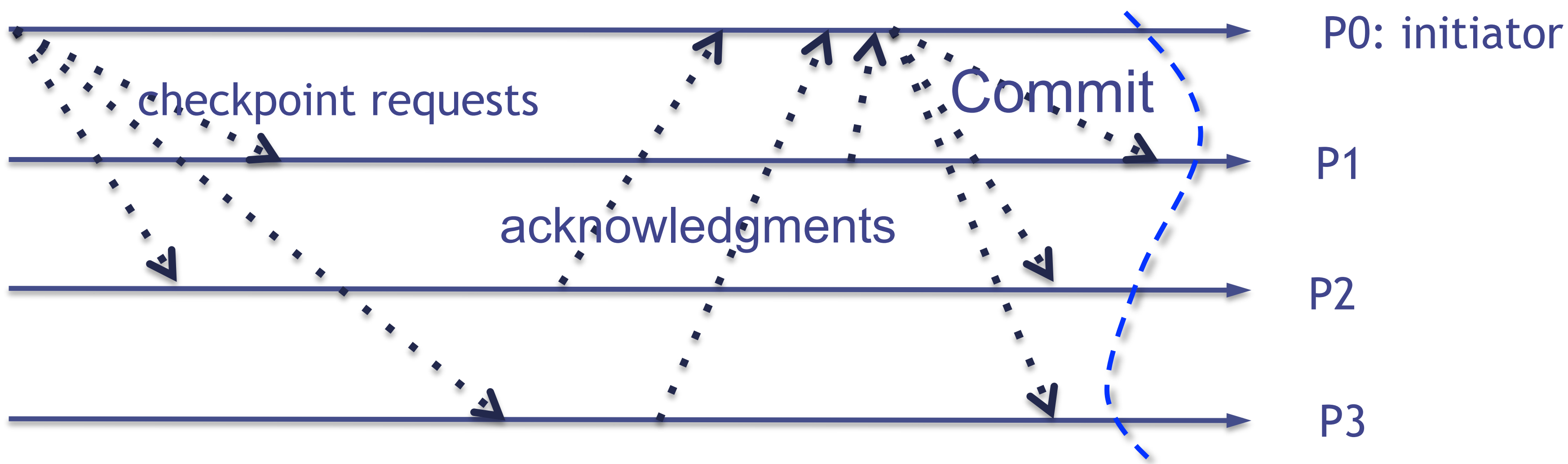


inconsistency if communications are not blocked until the end of the checkpointing phase

Coordinated checkpointing (2)

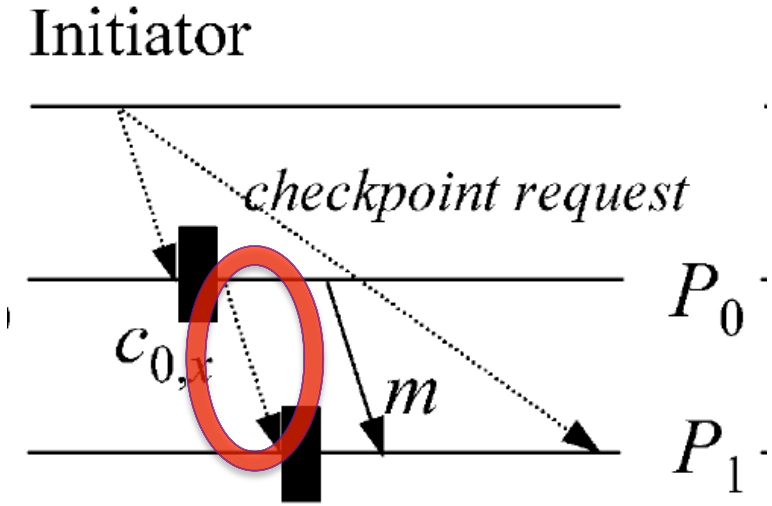
- Algorithm:
  - block communications while the protocol executes
  - An initiator takes a checkpoint and broadcasts a request message to all processes
  - When a process receives this message, it
    - stops its execution,
    - flushes all the communication channels,
    - takes a *tentative checkpoint, and*
    - sends an acknowledgment message back
  - the coordinator receives acknowledgments from all processes, and broadcasts a commit message
  - After receiving the commit each process removes the old checkpoint, the new one becomes permanent

Coordintated Checkpointing (3)
Overall execution graph

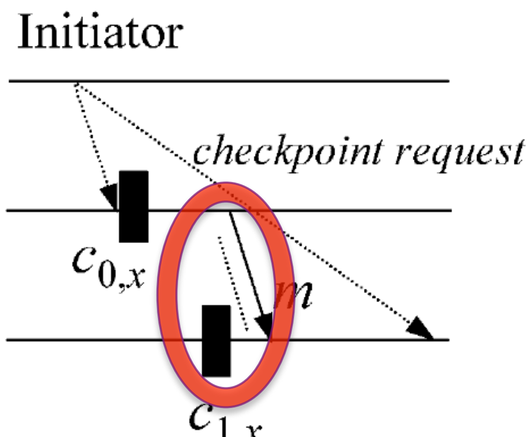


Solutions to avoid blocked states

- if communication channels are FIFO: propagate the checkpoint request before sending any other message



- Or piggyback checkpoint request on first message => take the checkpoint before taking the message into account

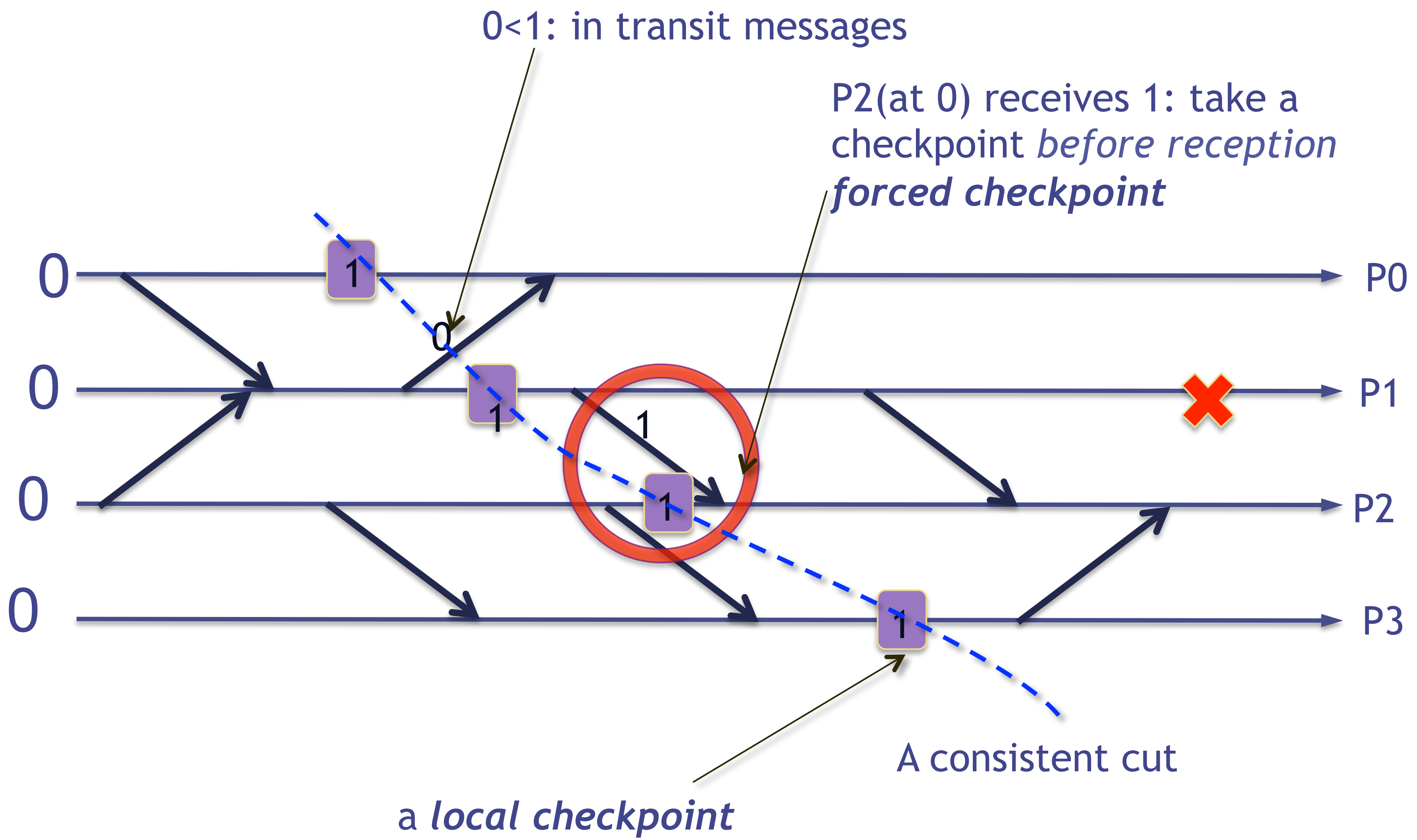


Question: is FIFO necessary when piggybacking?

Communication Induced Checkpointing

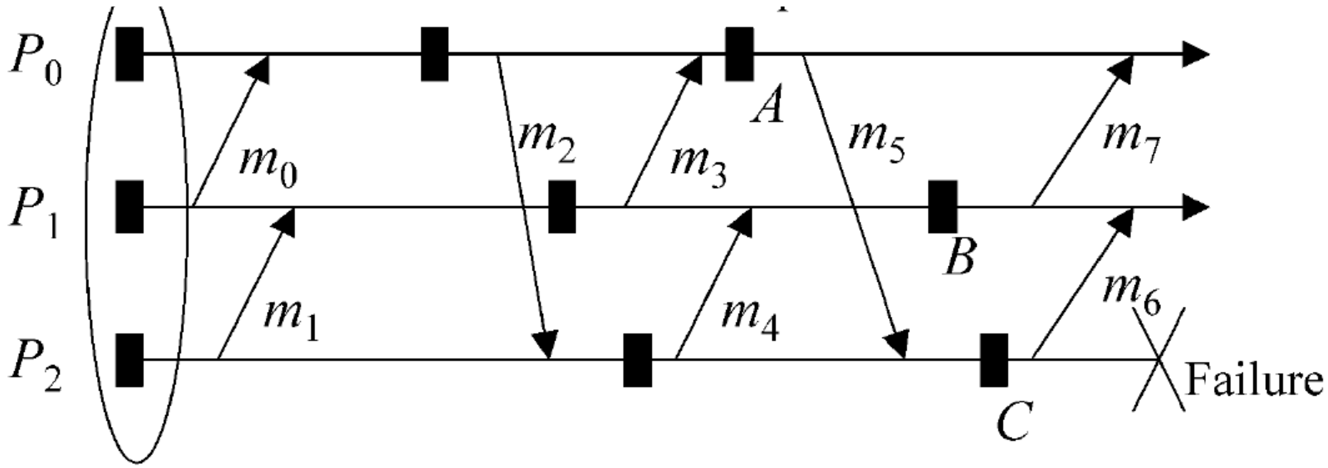
- 2 kinds of checkpoints: *local* and *forced*
- prevent the creation of useless checkpoints
- no coordination message: only piggybacks information
- Simplest = index-based:
  - processes piggyback timestamps (increasing timestamps for a given process)
  - For example [Briatico et al.] forces a checkpoint upon receiving a message with a greater index than the local index
  - A recovery line consists of checkpoints with the same index

Communication Induced Checkpointing (2)



Exercise

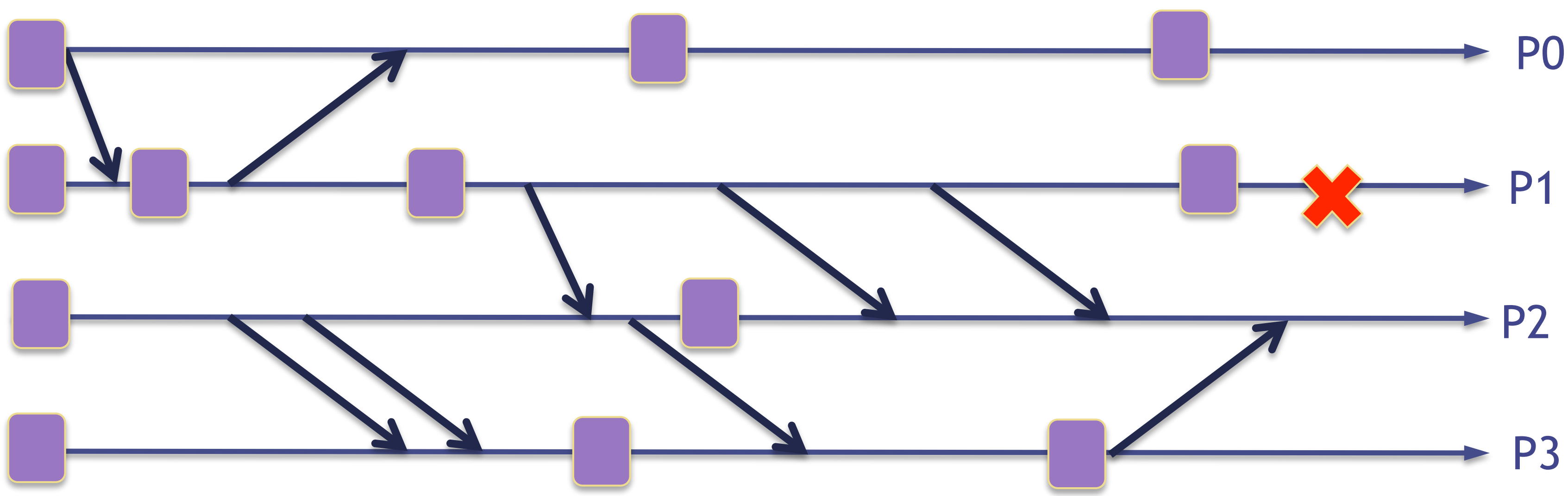
- show that the domino effect of exercise 1 is not possible anymore: assign index to checkpoints, add forced checkpoints and give piggybacked indexes on messages (black boxes are the local checkpoints)



- check with different failure points

exercise contd.

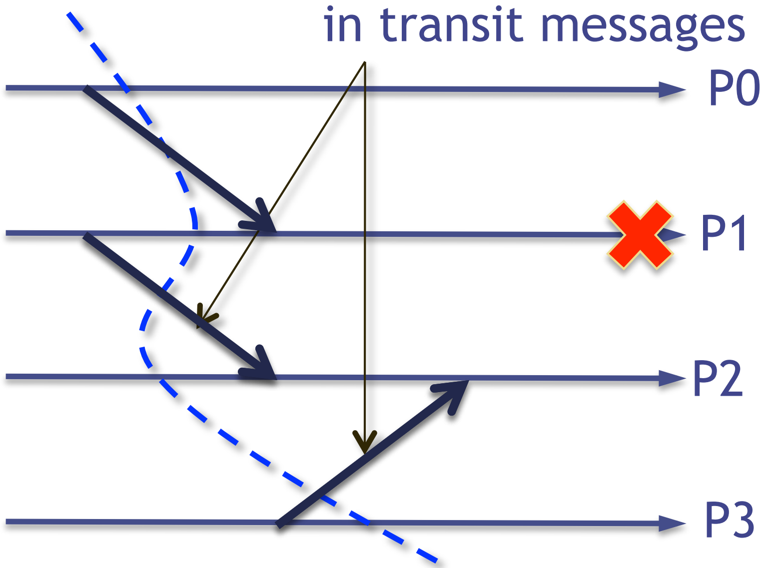
- what to do if more than 1 number of difference between indices?
- What does it mean when the piggybacked index is smaller than the current checkpoint?
  What can be done / can we use this information?



In transit messages

- Remember that if the communication protocol is reliable, they must be stored
  - ➡ It is easy to store them with the next checkpoint of the message sender (sender-based) or reciever.
  - ➡ Receiver-based: checkpoint already stored
  - ➡ Sender-based: messages are sent again upon recovery

Question:
Can we optimize the recovery process and avoid re-sending in-transit messages to processes that have not failed? How?

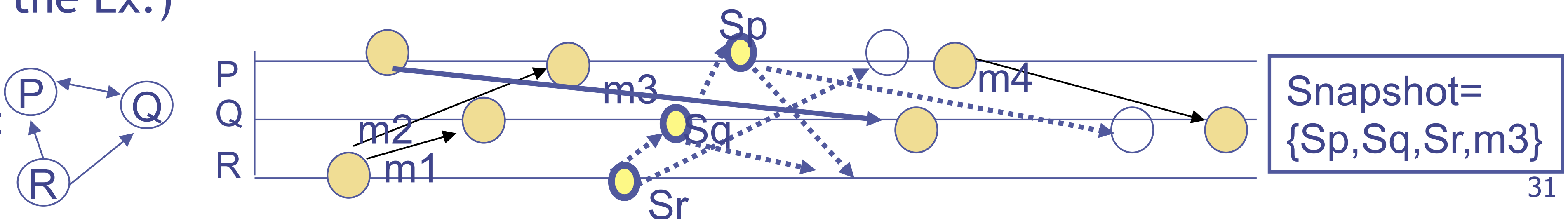


Exercise: Another protocol -- Distributed Snapshot algorithm for FIFO channels [Chandy-Lamport]

- Channels are FIFO. Messages are not lost.
- Snapshot algo. executes concurrently with the application
- Special "control" message
  - When receiving it for the 1st time through a channel:
    - Pi records its state, and channel state = empty
    - Pi forwards control message to all its outgoing neighbors
  - Messages received through the other incoming channels after a 1st received "control" msg are logged
  - When not the 1st time:

    Pi adds to its state all logged msgs that came from this channel so far
- Any process may initiate the algo. at any time (triggers one control msg for itself), concurrent executions of the protocol must be distinguishable
- Terminated: all Pi received control msg from all incoming channels
- Logged msgs on P->Q, logged by Q="msgs sent by P to Q while P and Q already logged their state, and Q waited the control msg from P" (m3 in the Ex.)

Ex.:



Questions

- Why is FIFO necessary for Chandy-Lamport algorithm?

How are orphan messages avoided?

- What about in transit messages: how are they managed with Chandy Lamport algorithm?
- Two processes P and Q are connected in a ring, they constantly rotate a message m (but might perform some local compuation before re-sending the msg). At any time, there is only one copy of m in the system. Each process's state consists of the number of times it has received m, P sends first. At a certain point, P has the message and its state is 101. Immediately after sending m, P initiates the snapshot algorithm. Explain the operations of the algorithm in this case and give the possible global state(s) reported by it.

RECOVERY: MESSAGE LOGGING MECHANISMS

Message Logging

- Hypothesis: ***piecewise determinism*** = all non-deterministic events can be identified and their determinants can be stored on stable storage.
- An execution is a sequence of deterministic events (replayed) and non-deterministic events (logged and simulated from log)
- determinants of non-deterministic events are stored during failure-free execution
- + checkpoints to avoid recovering from the start
- Additional hypothesis: It is possible to prevent a message from being sent or received

Message Logging

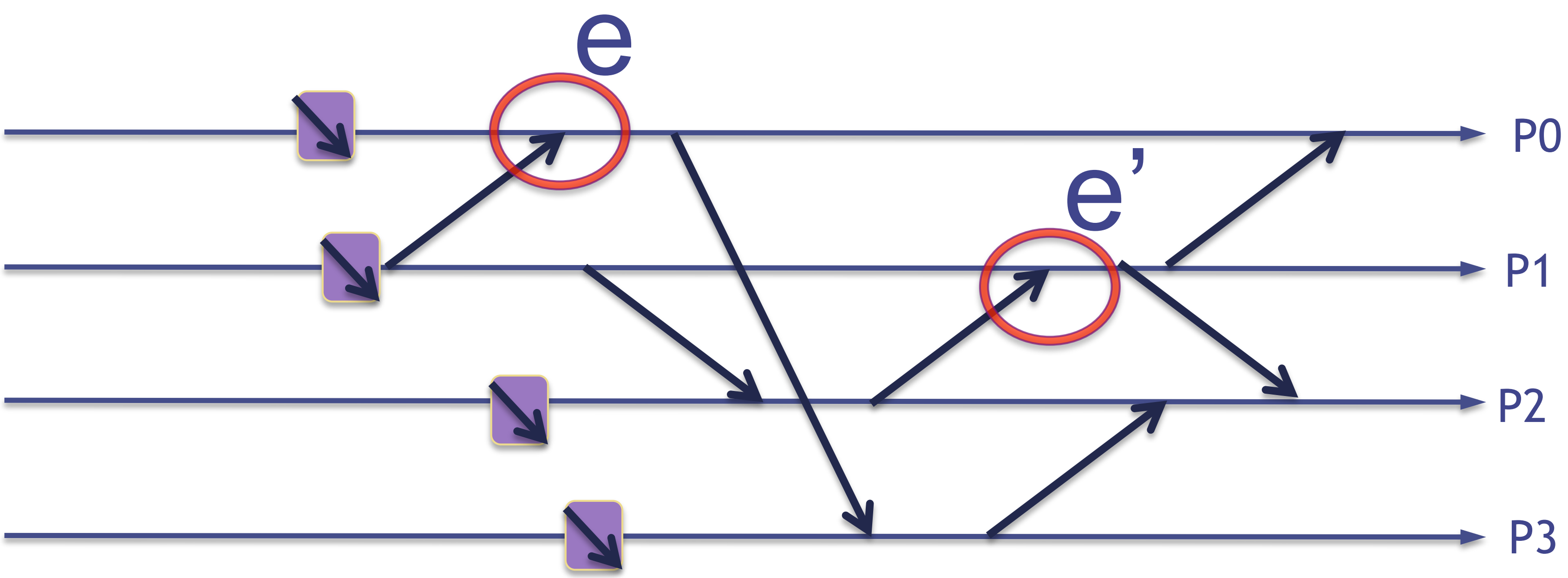
- A ***process is orphan*** if it depends on the execution of a non-logged non-deterministic event
- Always no ***orphan process***
  - Log(e) = set of processes locally storing the event e
  - Stable(e) if e's determinant is logged on stable storage
  - Depend(e) processes affected by a non-deterministic event e

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

else the process is said orphan

Tiny exercise

- Question: what is depend(e) in the example below?
- What about depend(e')

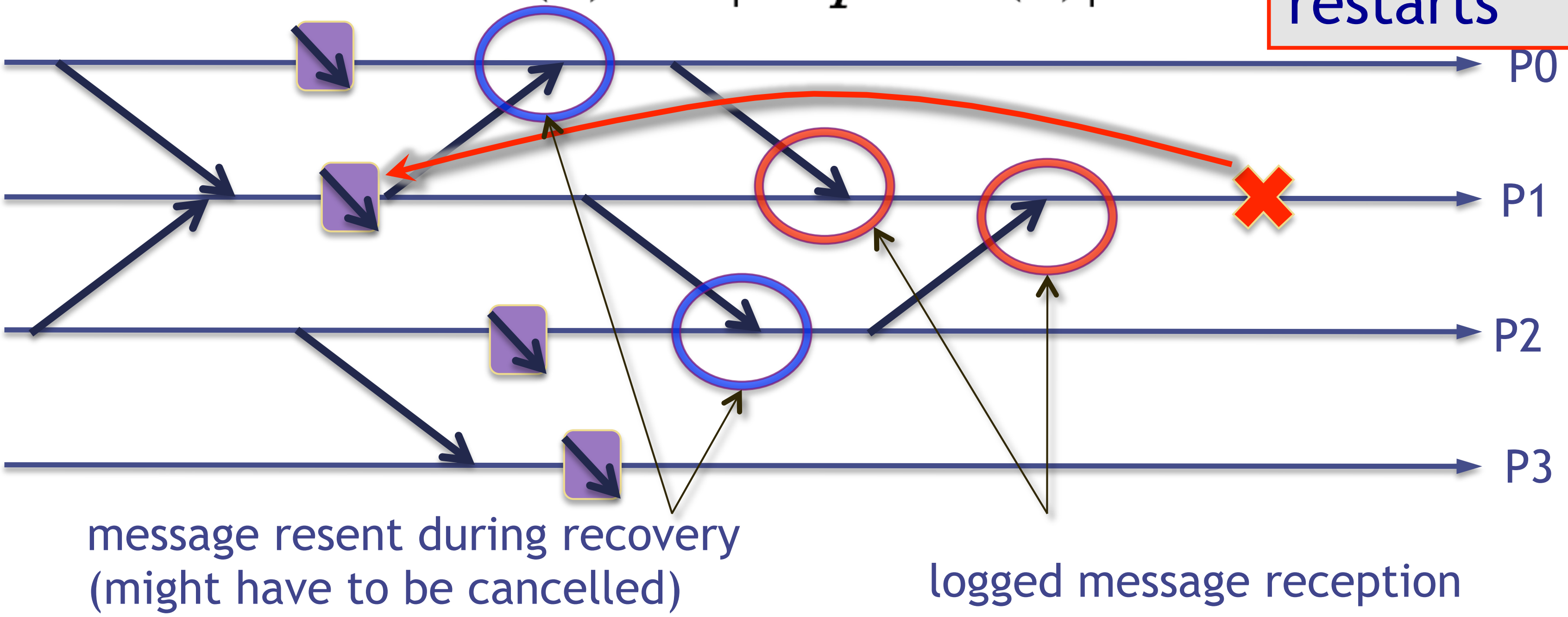


Pessimistic message logging

- orphan processes are never created but requires a lot of synchronizations with the stable storage
- Logs the determinant of ND events before executing them

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

Only 1 process restarts



Pessimistic message logging (2)

- only the failed processes recovers
- simple
- restart from last checkpoint, recovery simple and very fast
- garbage collection simple
- Easier to take into account outside world
- performance penalty due to synchronous logging
- NB: if message delivery is not guaranteed then logging does not have to be synchronous, it is only necessary to log a reception before sending the next message

Optimistic message logging (principles)

- Determinant kept locally, and sometimes stored on global storage
- Track causal dependencies between messages
- synchronous recovery: compute the maximum recoverable state
- Asynchronous: trigger recovery of causally related processes during the recovery process

  >> Risk of exponential rollbacks

Summary

- In fault tolerance strong (interesting) results require strong assumptions, or a lot of redundancy and inefficiency
- Fortunately in practice most system are reliable enough
- What was not presented:
  - safe communications
  - details of optimistic message logging
  - causal logging
  - complex protocols in general
  - redundancy and basic coherence, safety algorithm (course placed on a higher protocol level)

To conclude: 2 summarising tables next slides

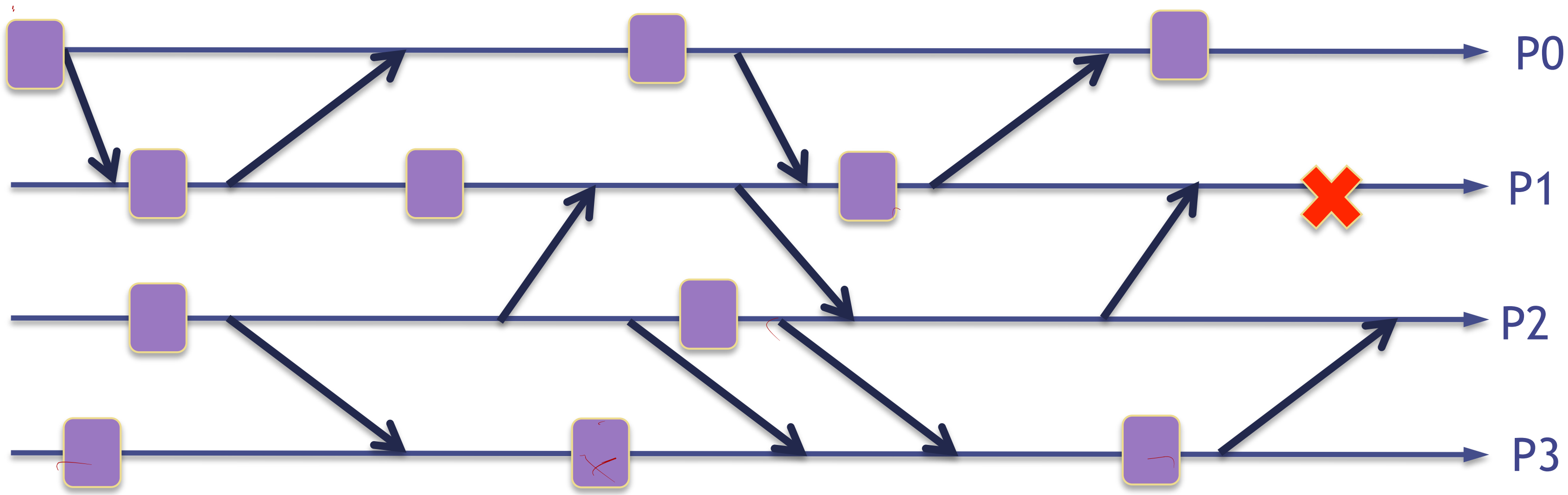
	Uncoordinated Checkpointing	Coordinated Checkpointing	Comm. Induced Checkpointing	Pessimistic Logging	Optimistic Logging	Causal Logging
PWD assumed?	No	No	No	Yes	Yes	Yes
Checkpoint/ process	Several	1	Several	1	Several	1
Domino effect	Possible	No	No	No	No	No
Orphan processes	Possible	No	Possible	No	Possible	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

Advantages and drawbacks of ML/CP (simplified!)

	Target system	Overhead
Checkpointing	small and medium size	Rather low
Message logging	large scale	Medium or high

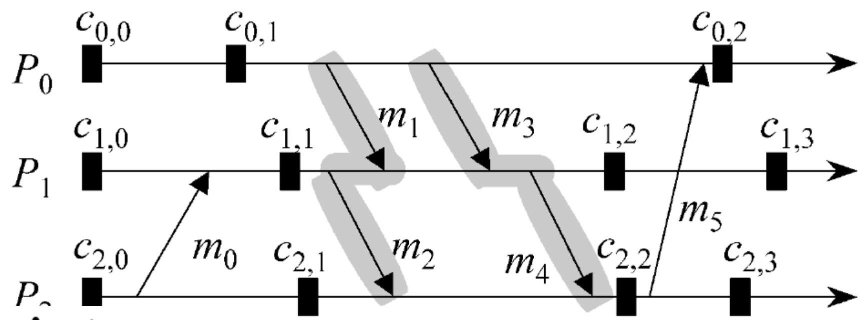
Homework

- Build the rollback dependency graph for the execution below
- What is the recovery line?
- How would you extend the rules of the rollback dependency graph to also avoid in-transit message?
- What is the new recovery line on the execution below?
- Considering that the purple squares are the forced checkpoints, run a CIC protocol on the execution below. What is a valid recovery line in this case (CIC)?



EXERCISES

Exercise: Z-paths



Given two checkpoints $c_{i,x}$ and $c_{j,y}$, a Z-path exists between $c_{i,x}$ and $c_{j,y}$ if and only if one of the following two conditions holds:

1. $x < y$ and $i = j$; or
2. There exists a sequence of messages $[m_0, m_1, \ldots, m_n]$, $n \mu 0$, such that:
   - $c_{i,x} \mapsto send_i(m_0)$;
   - $\forall l < n$, either $deliver_k(m_l)$ and $send_k(m_{l+1})$ are in the same checkpoint interval, or $deliver_k(m_l) \mapsto send_k(m_{l+1})$; and
   - $deliver_j(m_n) \mapsto c_{j,y}$

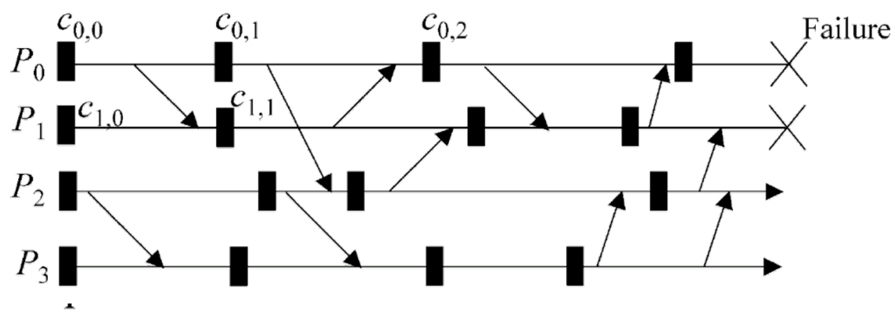
where $send_i$ and $deliver_i$ are communication events executed by process P_i. In

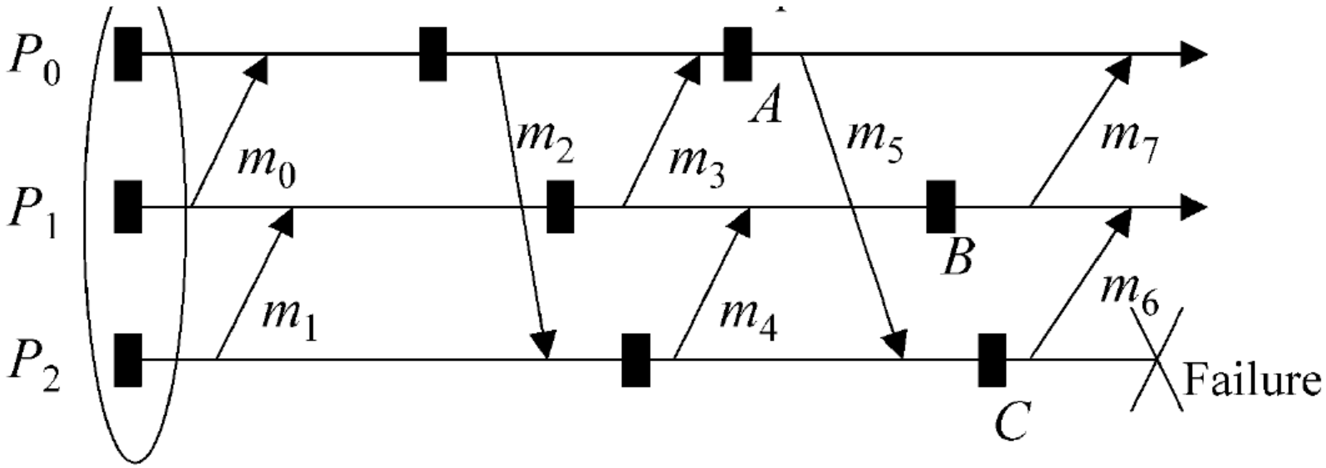
A Z-cycle is a Z-path that begins
and ends with the same checkpoint. e.g. m5 m3 m4

Exercise: link between Z-paths and checkpoint dependencies

1 – draw the rollback dependency graph for the execution of the previous slide

2 – find some Z-path and all Z-cycles in the executions mentioned in the course:





Equivalence?

- Zpaths have been used to prove correctness of some CIC protocols, because a checkpoint in a Z-cycle is not useful.
- Exercise: *On the preceding examples, show that the checkpoints in Z-cycles would not be used upon recovery according to the checkpoint dependency graph*
- Of course, this is not a proof of equivalence. Can you give a hint why and checkpoint in a cycle would not be used in a checkpoint