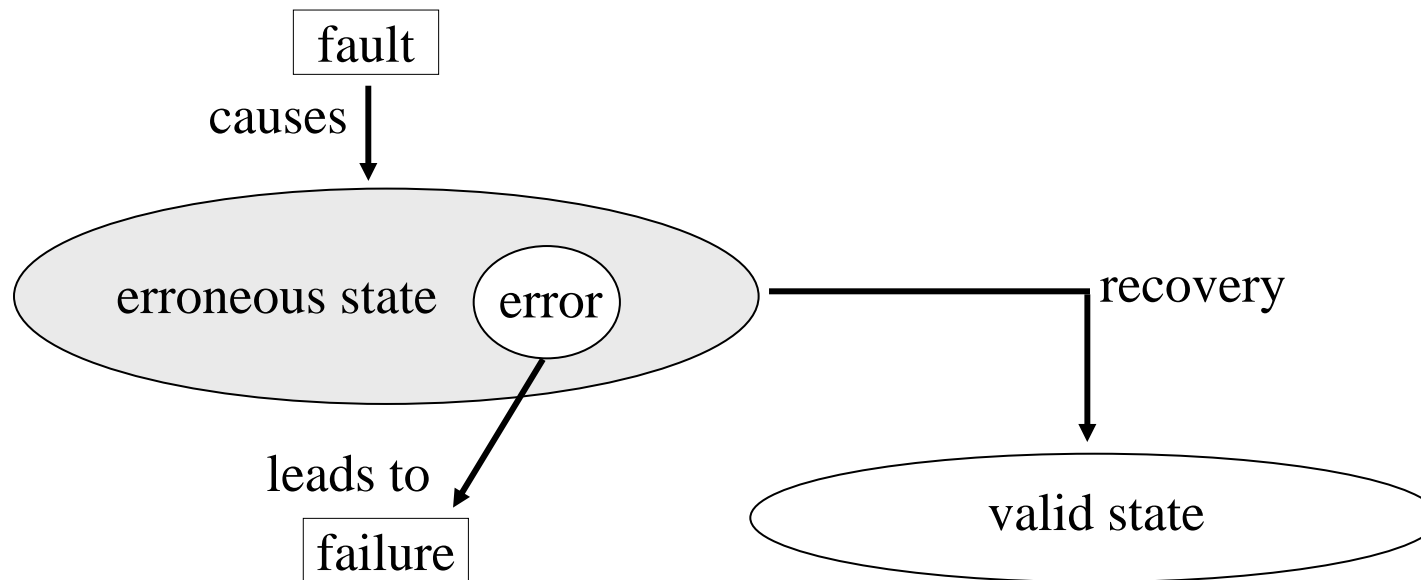


Checkpointing-Recovery

Fault Tolerance

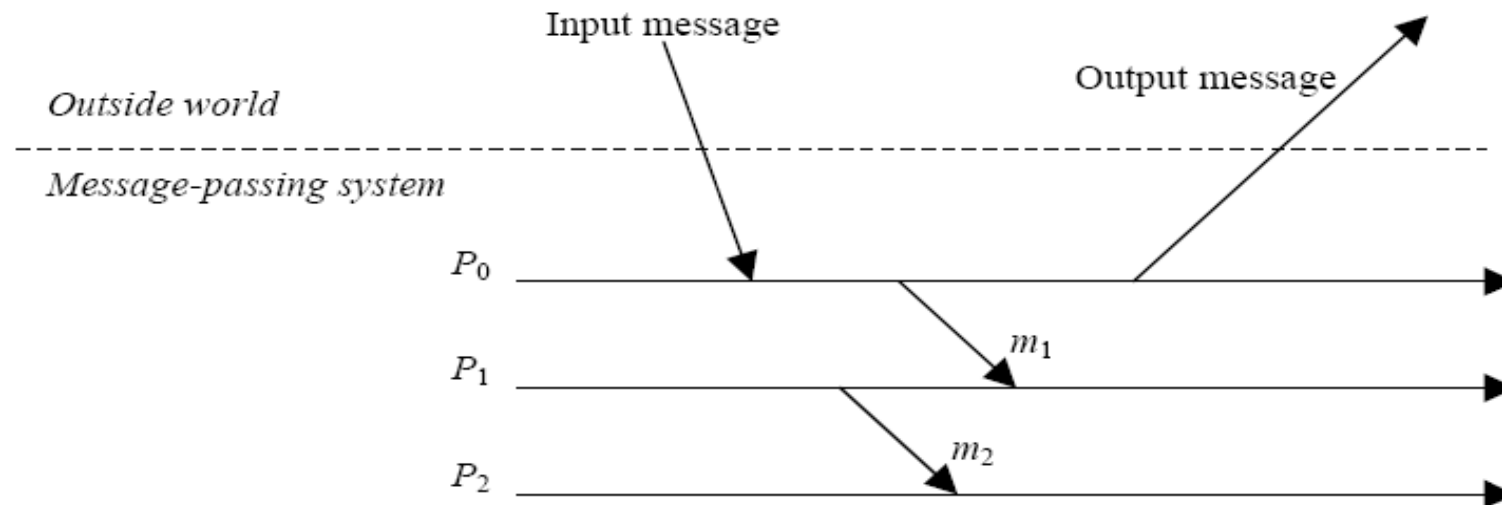


An error is a manifestation of a fault that can lead to a failure.

Failure Recovery:

- backward recovery
 - operation-based (do-undo-redo logs)
 - state-based (checkpointing/logging)
- forward recovery

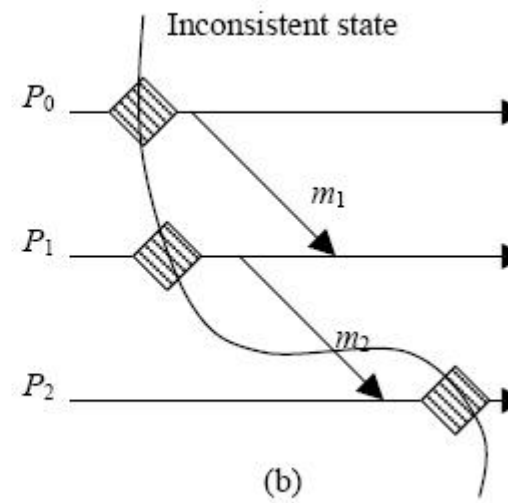
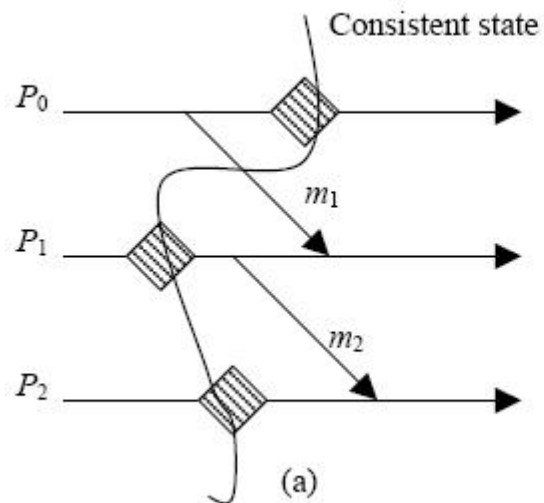
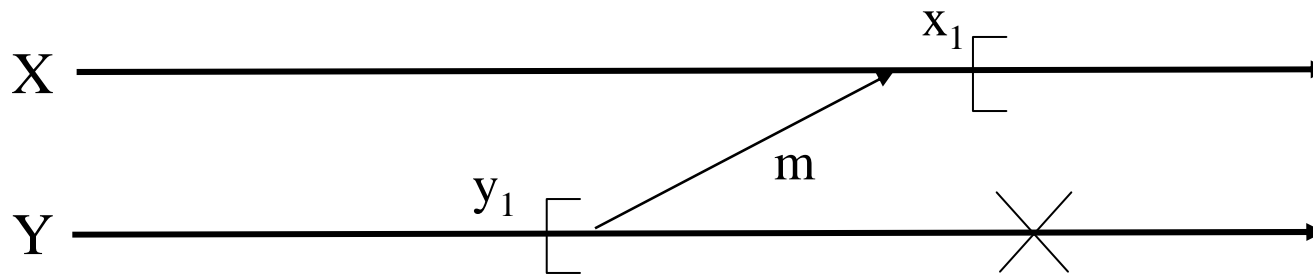
System Model



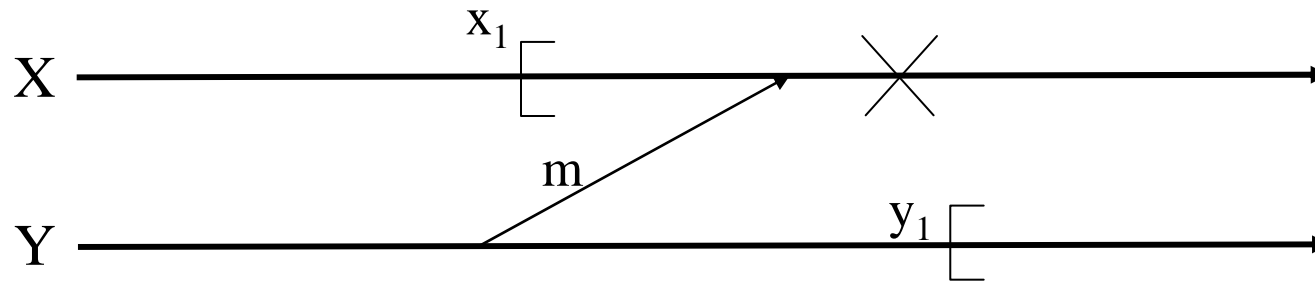
Basic approaches

- checkpointing : copying/restoring the state of a process
- logging : recording/replaying messages

Orphan Message



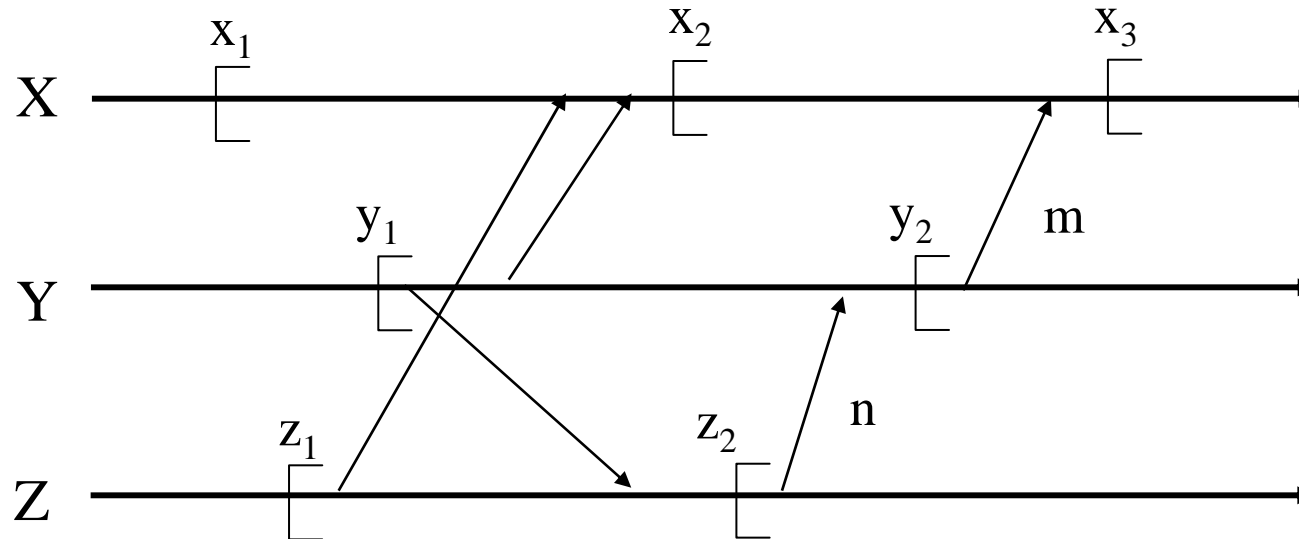
Lost Messages



Regenerating lost messages on recovery:

- if implemented on unreliable communication channels, the application is responsible
- if implemented on reliable communication channels, the recovery algorithm is responsible

Domino Effect



Cases:

- X fails after x_3
- Y fails after sending message m
- Z fails after sending message n

Other Issues

■ Output commit

- the state from which messages are sent to the “outside world” can be recovered
- affects latency of message delivery to “outside world” and overhead of checkpoint/logging

■ Stable storage

- survives process failures
- contains checkpoint/logging information

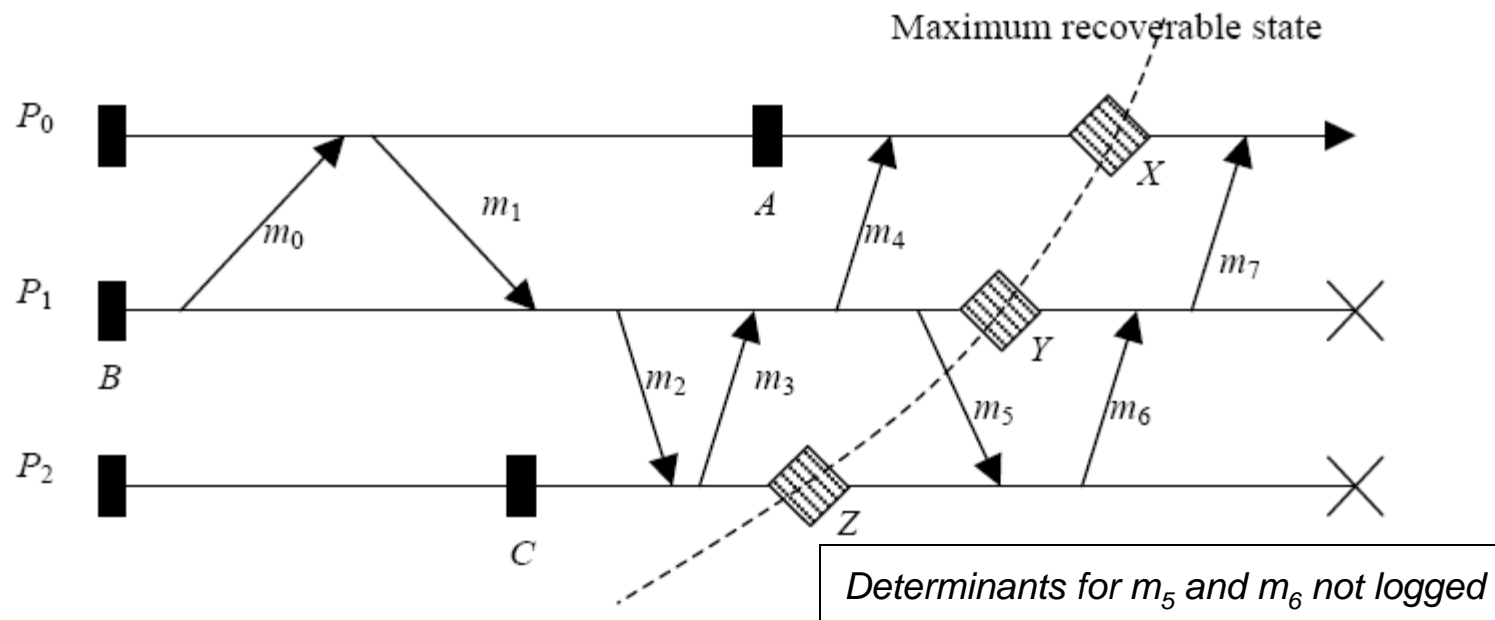
■ Garbage collection

- removal of checkpoints/logs no longer needed

Logging Protocols

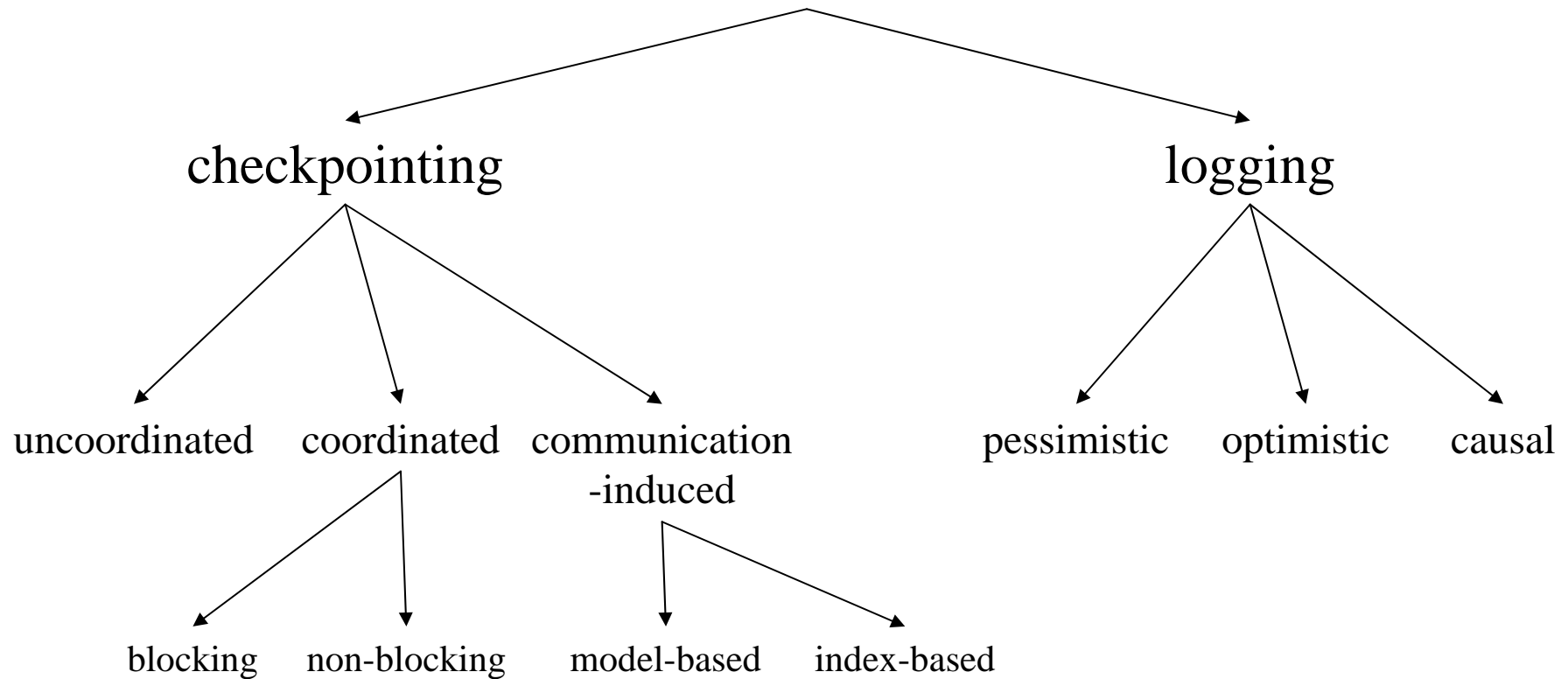
Elements

- Piecewise deterministic (PWD) assumption – the system state can be recovered by replaying message receptions
- Determinant – record of information needed to recover receipt of message



Taxonomy

Rollback-Recovery



Uncoordinated Checkpointing

Rollback-Recovery

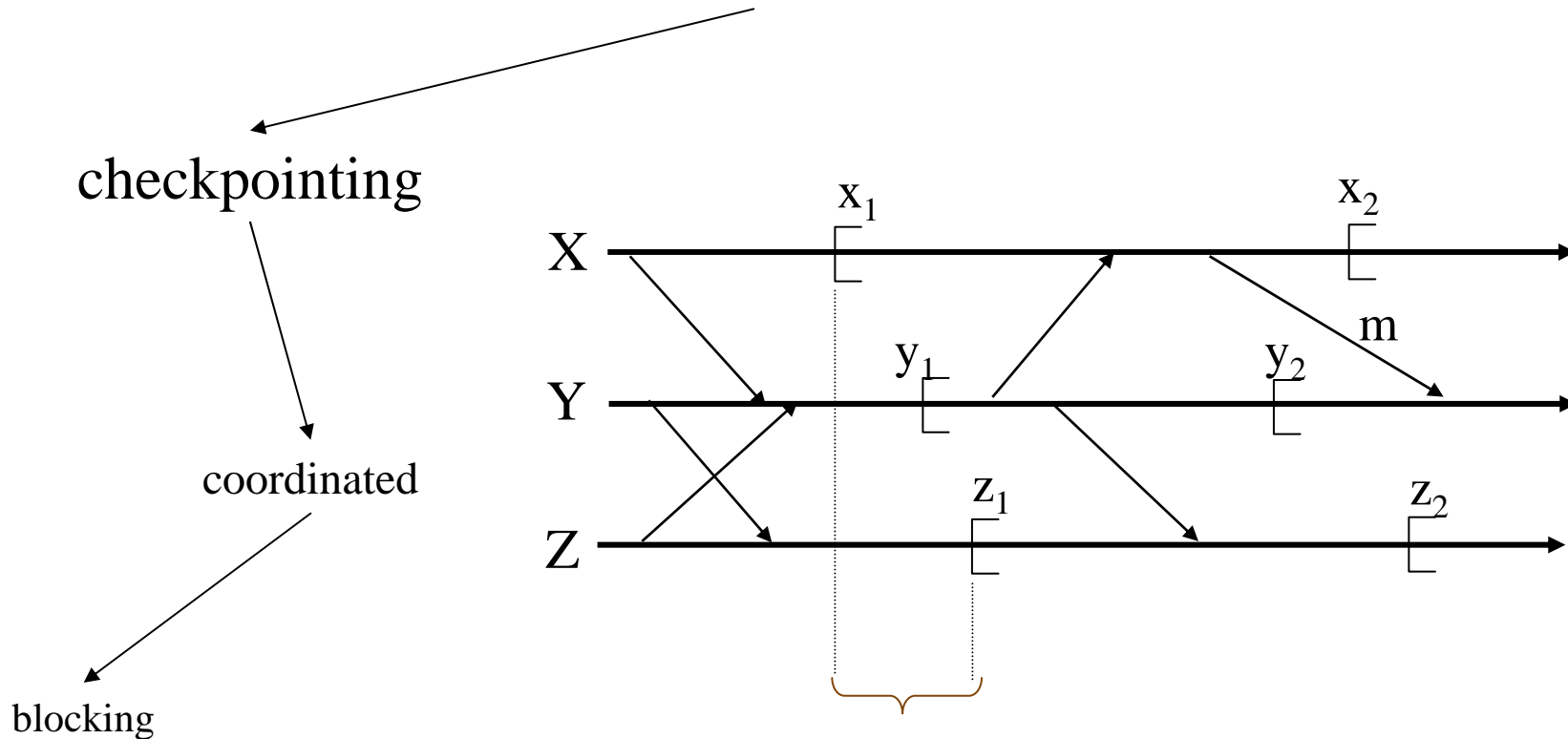
checkpointing

uncoordinated

- susceptible to domino effect
- can generate useless checkpoints
- complicates storage/GC
- not suitable for frequent output commits

Cordinated/Blocking Protocols

Rollback-Recovery

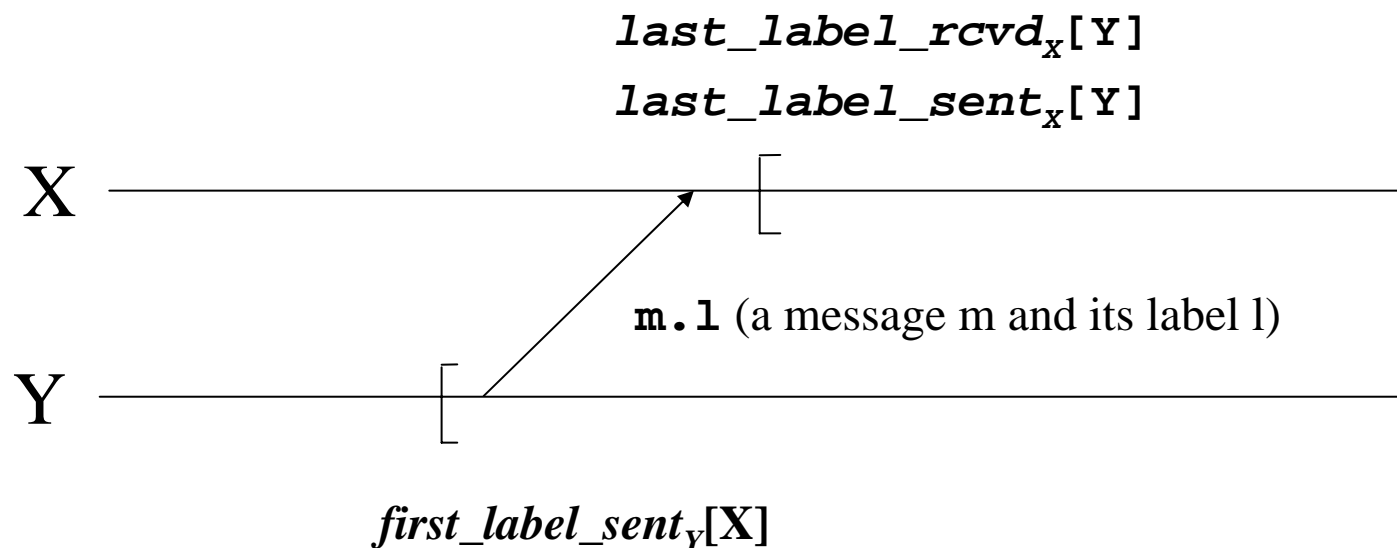


- no messages can be in transit during checkpointing
- $\{x_1, y_1, z_1\}$ forms "recovery line"

Coordinated/Blocking Notation

Each node maintains:

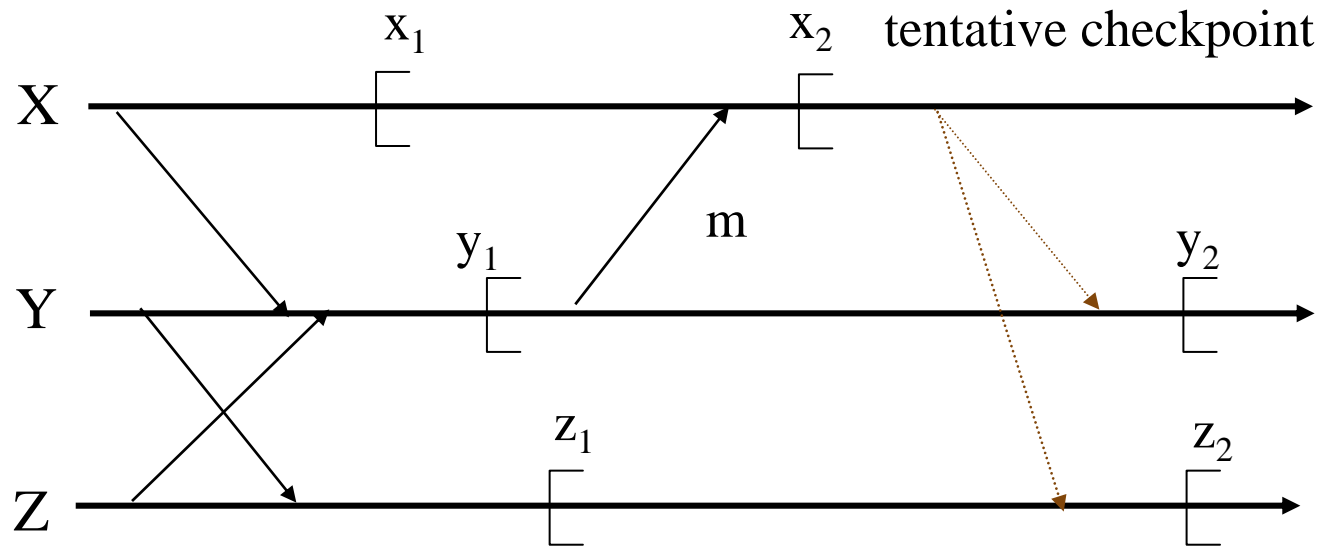
- a monotonically increasing counter with which each message from that node is labeled.
- records of the last message from/to and the first message to all other nodes.



Note: “sl” denotes a “smallest label” that is $<$ any other label and
 “ll” denotes a “largest label” that is $>$ any other label

Coordinated/Blocking Algorithm

- (1) When must I take a checkpoint?
- (2) Who else has to take a checkpoint when I do?



- (1) When I (Y) have sent a message to the checkpointing process, X, since my last checkpoint:

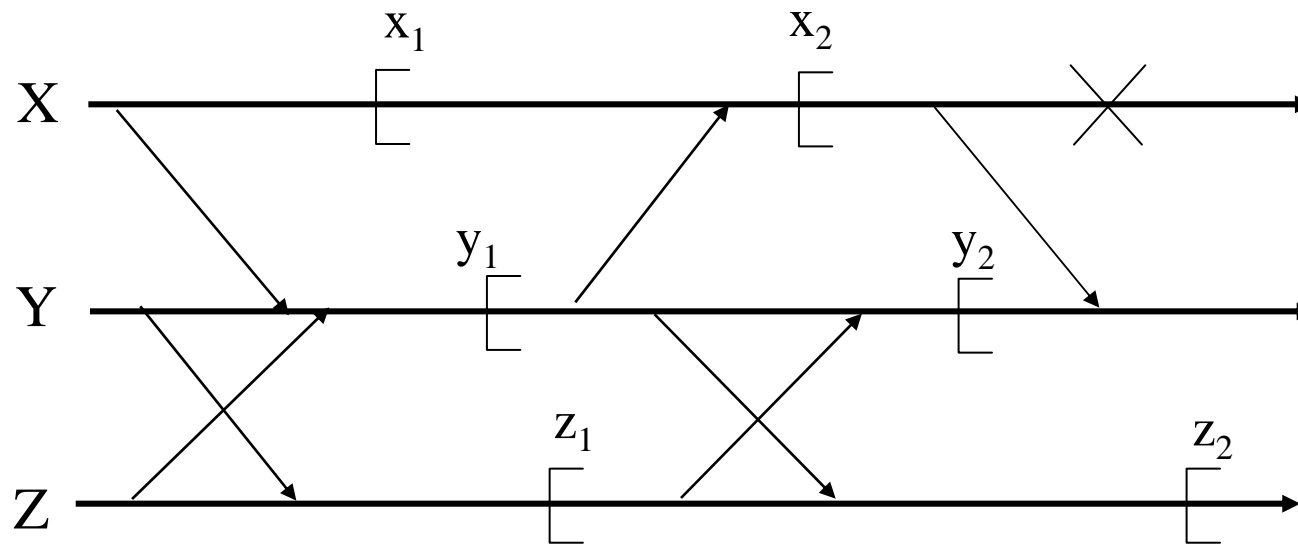
$$\mathit{last_label_rcvd}_x[Y] \geq \mathit{first_label_sent}_y[X] > s1$$

- (2) Any other process from whom I have received messages since my last checkpoint.

$$\mathit{ckpt_cohort}_x = \{Y \mid \mathit{last_label_rcvd}_x[Y] > s1\}$$

Coordinated/Blocking Algorithm

- (1) When must I rollback?
- (2) Who else might have to rollback when I do?



- (1) When I, Y, have received a message from the restarting process, X, since X's last checkpoint.

$$last_label_rcvd_y(X) > last_label_sent_x(Y)$$

- (2) Any other process to whom I can send messages.

$$roll_cohort_y = \{z \mid Y \text{ can send message to } z\}$$

Taxonomy

Rollback-Recovery

```
graph TD; A[Rollback-Recovery] --> B[checkpointing]; B --> C[coordinated]; C --> D[non-blocking]
```

checkpointing

coordinated

non-blocking

Approach:

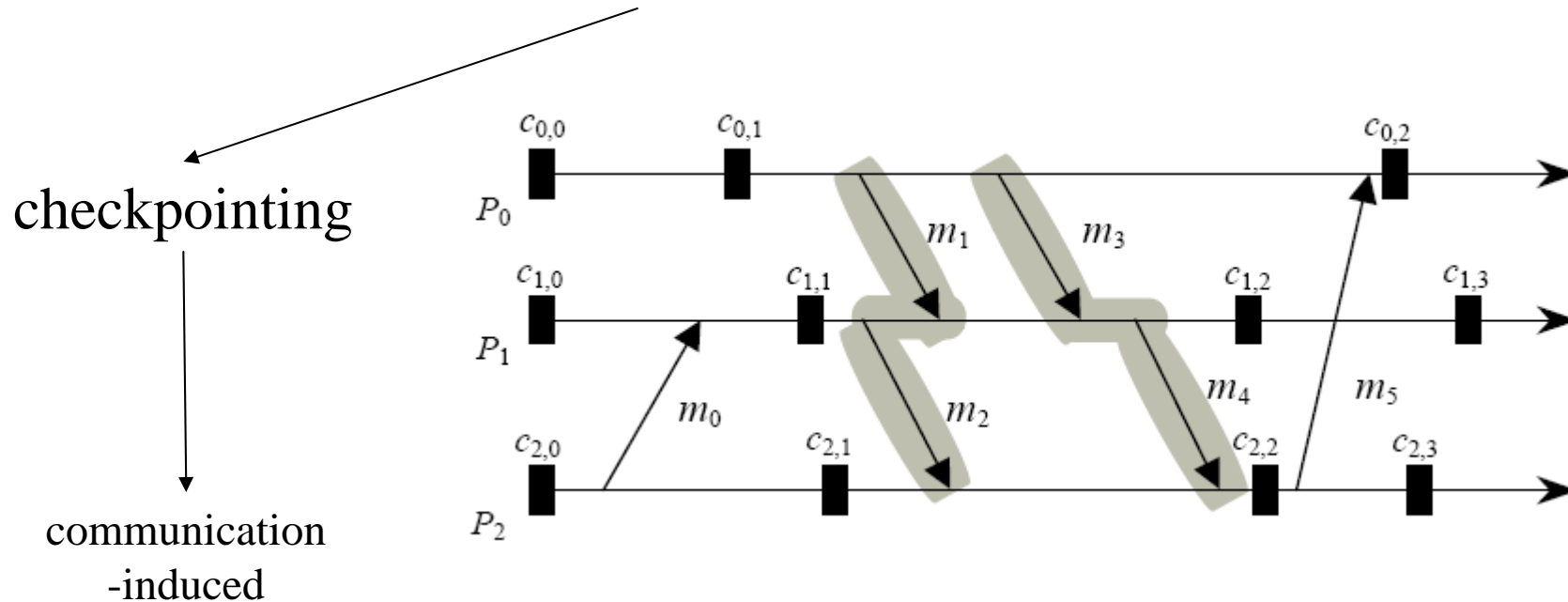
“tag” message to trigger checkpointing

Example:

global-state recording algorithm

Communication-Induced Checkpointing

Rollback-Recovery



Z-path: $[m_1, m_2]$ and $[m_3, m_4]$

Z-cycle: $[m_3, m_4, m_5]$

Checkpoints (like $c_{2,2}$) in a z-cycle are useless

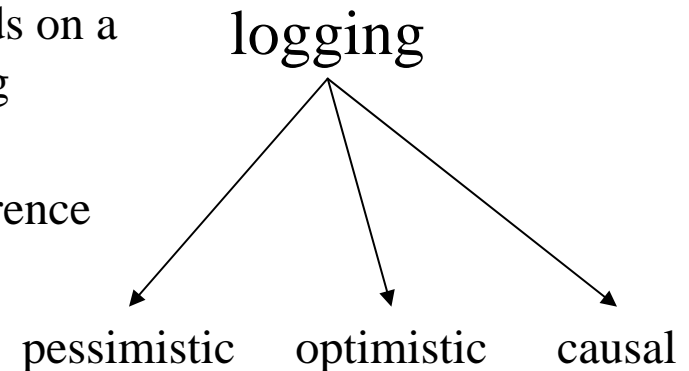
Cause checkpoints to be taken to avoid z-cycles

Logging

Rollback-Recovery

Orphan process: a non-failed process whose state depends on a non-deterministic event that cannot be reproduced during recovery.

Determinant: the information need to “replay” the occurrence of a non-deterministic event (e.g., message reception).

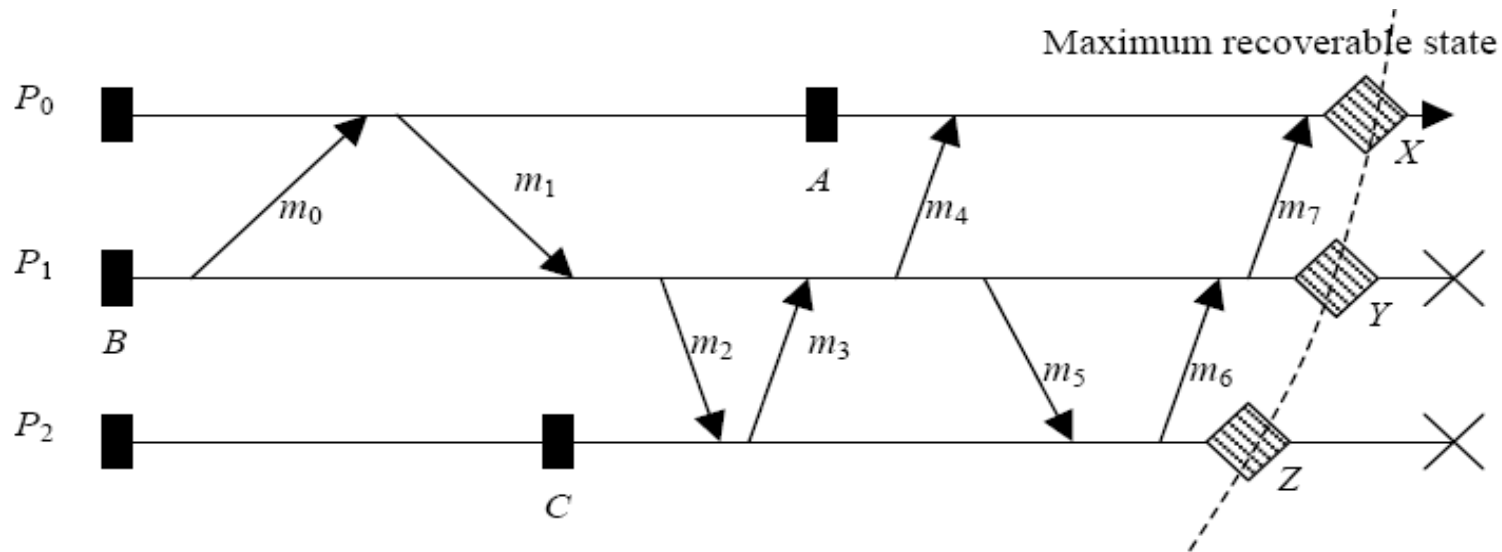


Avoid orphan processes by guaranteeing:

$$\text{For all } e : \text{not } Stable(e) \Rightarrow Depend(e) < Log(e)$$

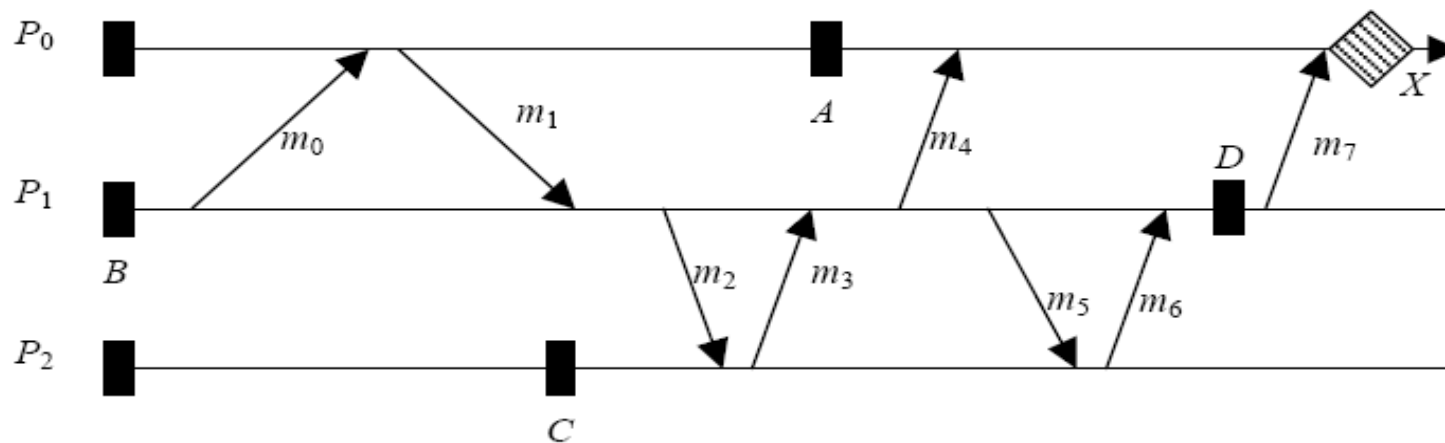
where: $Depend(e)$ – set of processes affected by event e
 $Log(e)$ – set of processes with e logged on volatile memory
 $Stable(e)$ – set of processes with e logged on stable storage

Pessimistic Logging



- Determinant is logged to stable storage before message is delivered
- Disadvantage: performance penalty for synchronous logging
- Advantages:
 - immediate output commit
 - restart from most recent checkpoint
 - recovery limited to failed process(es)
 - simple garbage collection

Optimistic Logging



- determinants are logged asynchronously to stable storage
- consider: P_2 fails before m_5 is logged
- advantage: better performance in failure-free execution
- disadvantages:
 - coordination required on output commit
 - more complex garbage collection

Causal logging

- combines advantages of optimistic and pessimistic logging
- based on the set of events that causally precede the state of a process
- guarantees determinants of all causally preceding events are logged to stable storage or are available locally at non-failed process
- non-failed process “guides” recovery of failed processes
- piggybacks on each message information about causally preceding messages
- reduce cost of piggybacked information by send only difference between current information and information on last message