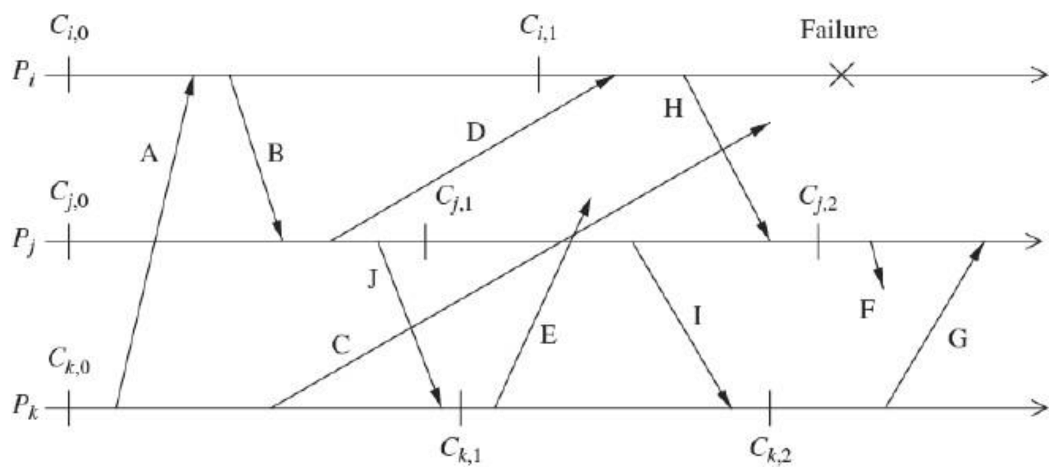
4.3 Issues in failure recovery

In a failure recovery, we must not only restore the system to a consistent state, but also appropriately handle messages that are left in an abnormal state due to the failure and recovery



The computation comprises of three processes Pi, Pj , and Pk, connected through a communication network. The processes communicate solely by exchanging messages over fault- free, FIFO communication channels.

Processes Pi, Pj , and Pk have taken checkpoints

- 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 : $\{C_{i,1}, C_{j,1}, C_{k,1}\}$

- The rollback of process Pi to checkpoint Ci,1 created an orphan message H
- Orphan message I is created due to the roll back of process Pj to checkpoint Cj,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 Pj, but the receive event has been undone at process Pi.
  - 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

4.4 Checkpoint-based recovery

Checkpoint-based rollback-recovery techniques can be classified into three categories:

1. U*ncoordinated checkpointing*
2. *Coordinated checkpointing*
3. *Communication-induced checkpointing*

1. Uncoordinated Checkpointing

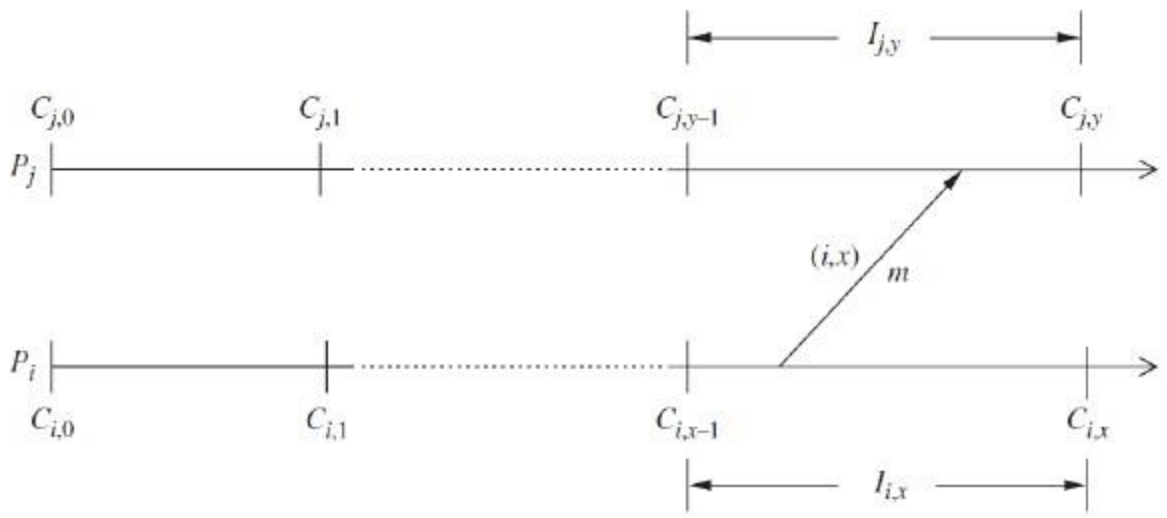
- Each process has autonomy in deciding when to take checkpoints
- Advantages

    The lower runtime overhead during normal execution

- Disadvantages
    1. Domino effect during a recovery
    2. Recovery from a failure is slow because processes need to iterate to find a consistent set of checkpoints
    3. Each process maintains multiple checkpoints and periodically invoke a garbage collection algorithm
    4. Not suitable for application with frequent output commits
- The processes record the dependencies among their checkpoints caused by message exchange during failure-free operation
- The following direct dependency tracking technique is commonly used in uncoordinated checkpointing.

Direct dependency tracking technique

- Assume each process Pi starts its execution with an initial checkpoint Ci,0
- Ii,x : checkpoint interval, interval between Ci,x−1 and Ci,x
- When Pj receives a message m during Ij,y , it records the dependency from Ii,x to Ij,y, which is later saved onto stable storage when Pj takes Cj,y



- When a failure occurs, the recovering process initiates rollback by broadcasting a *dependency request* message to collect all the dependency information maintained by each process.
- When a process receives this message, it stops its execution and replies with the dependency information saved on the stable storage as well as with the dependency information, if any, which is associated with its current state.
- The initiator then calculates the recovery line based on the global dependency information and broadcasts a *rollback request* message containing the recovery line.
- Upon receiving this message, a process whose current state belongs to the recovery line simply resumes execution; otherwise, it rolls back to an earlier checkpoint as indicated by the recovery line.

2. Coordinated Checkpointing

In coordinated checkpointing, processes orchestrate their checkpointing activities so that all local checkpoints form a consistent global state

Types

1. 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
2. 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.

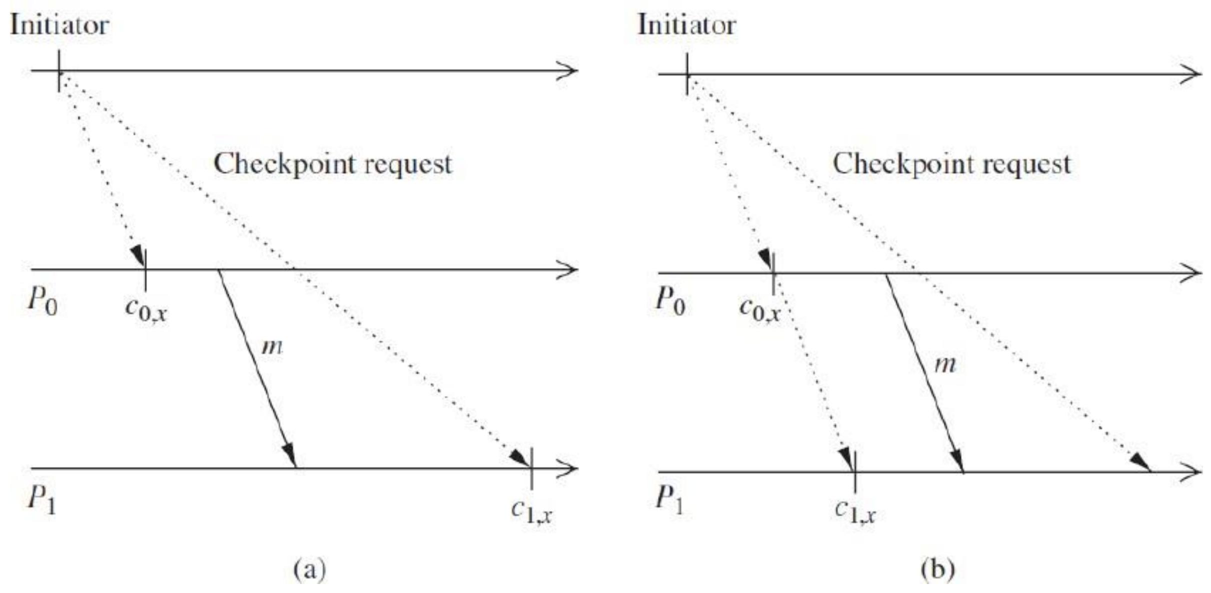
Example (a) : Checkpoint inconsistency

- Message m is sent by $P0$ after receiving a checkpoint request from the checkpoint coordinator
- Assume m reaches $P1$ before the checkpoint request
- This situation results in an inconsistent checkpoint since checkpoint $C1,x$ shows the receipt of message m from $P0$, while checkpoint $C0,x$ does not show m being sent from $P0$

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

Coordinated Checkpointing



Impossibility of min-process non-blocking checkpointing

- A min-process, non-blocking checkpointing algorithm is one that forces only a minimum number of processes to take a new checkpoint, and at the same time it does not force any process to suspend its computation.

Algorithm

- The algorithm consists of two phases. During the first phase, the checkpoint initiator identifies all processes with which it has communicated since the last checkpoint and sends them a request.
- Upon receiving the request, each process in turn identifies all processes it has communicated with since the last checkpoint and sends them a request, and so on, until no more processes can be identified.
- During the second phase, all processes identified in the first phase take a checkpoint. The result is a consistent checkpoint that involves only the participating processes.
- In this protocol, after a process takes a checkpoint, it cannot send any message until the second phase terminates successfully, although receiving a message after the checkpoint has been taken is allowable.

3. Communication-induced Checkpointing

Communication-induced checkpointing is another way to avoid the domino effect, while allowing processes to take some of their checkpoints independently. Processes may be forced to take additional checkpoints

Two types of checkpoints

1. Autonomous checkpoints
2. Forced checkpoints

The checkpoints that a process takes independently are called *local* checkpoints, while those that a process is forced to take are called *forced* checkpoints.

- Communication-induced check pointing 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 check pointing, no special coordination messages are exchanged

Two types of communication-induced checkpointing

1. Model-based checkpointing
2. Index-based checkpointing.

Model-based checkpointing

- Model-based checkpointing prevents patterns of communications and checkpoints that could result in inconsistent states among the existing checkpoints.
- No control messages are exchanged among the processes during normal operation. All information necessary to execute the protocol is piggybacked on application messages
- There are several domino-effect-free checkpoint and communication model.
- The MRS (mark, send, and receive) model of Russell avoids the domino effect by ensuring that within every checkpoint interval all message receiving events precede all message-sending events.

Index-based checkpointing.

- Index-based communication-induced checkpointing assigns monotonically increasing indexes to checkpoints, such that the checkpoints having the same index at different processes form a consistent state.

4.5 Log-based rollback recovery

A log-based rollback recovery makes use of deterministic and nondeterministic events in a computation.

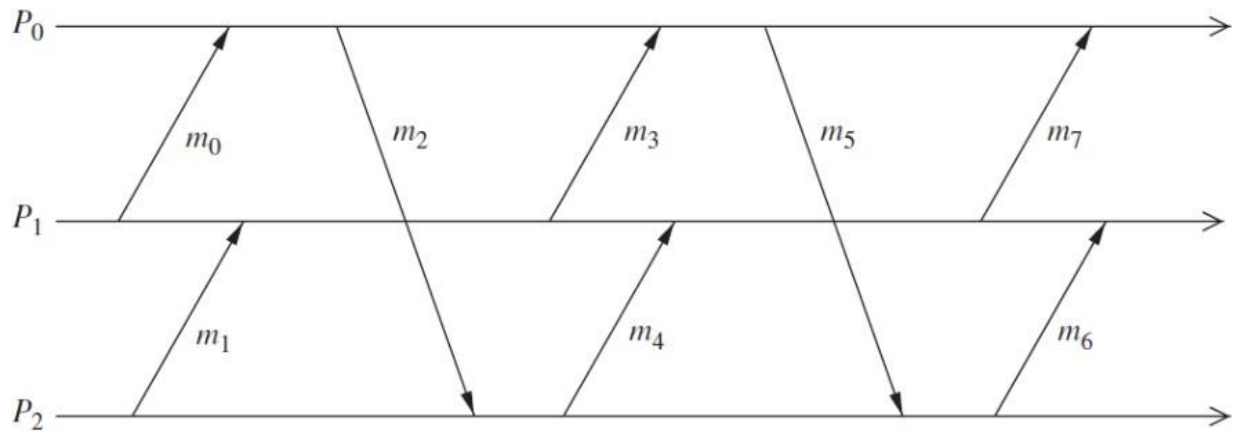
Deterministic and non-deterministic events

- Log-based rollback recovery exploits the fact that a process execution can be modeled as a sequence of deterministic state intervals, each starting with the execution of a non-deterministic event.
- A non-deterministic event can be the receipt of a message from another process or an event internal to the process.
- Note that a message send event is *not* a non-deterministic event.
- For example, in Figure, 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.
- Log-based rollback recovery assumes that all non-deterministic events can be identified and their corresponding determinants can be logged into the stable storage.
- <u>Determinant</u>: the information need to “replay” the occurrence of a non-deterministic

event (e.g., message reception).

- During failure-free operation, each process logs the determinants of all non-deterministic events that it observes onto the stable storage. Additionally, each process also takes checkpoints to reduce the extent of rollback during recovery.

Log-based Rollback Recovery



The no-orphans consistency condition

Let *e* be a non-deterministic event that occurs at process *p*. We define the following:

• *Depend*(*e*): the set of processes that are affected by a non-deterministic event *e.*

• *Log*(*e*): the set of processes that have logged a copy of *e*'s determinant in their volatile memory.

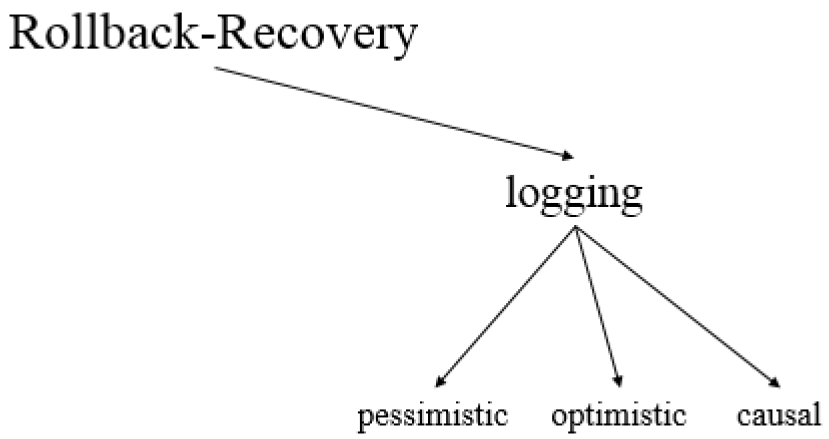
• *Stable*(*e*): a predicate that is true if *e*'s determinant is logged on the stable storage.

Suppose a set of processes Ψ crashes. A process *p* in Ψ becomes an orphan when *p* itself does not fail and *p*'s state depends on the execution of a nondeterministic event *e* whose determinant cannot be recovered from the stable storage or from the volatile memory of a surviving process. storage or from the volatile memory of a surviving process. Formally, it can be stated as follows

always-no-orphans condition

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

Types



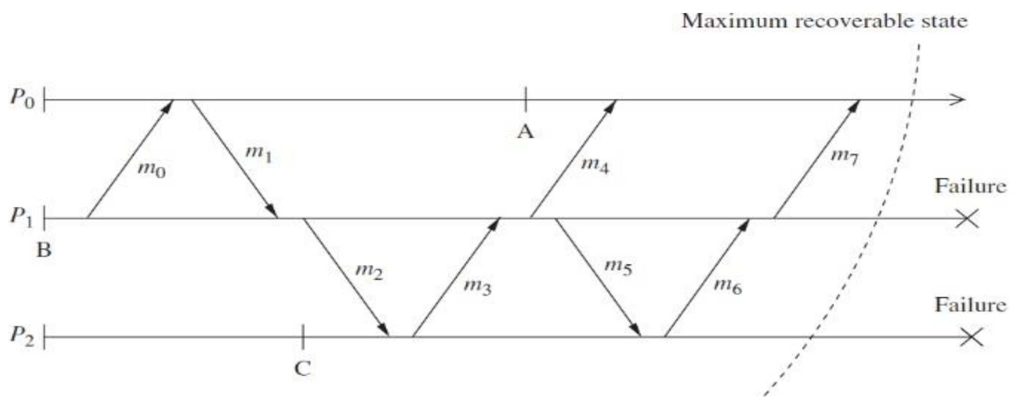
1. Pessimistic Logging

- Pessimistic logging protocols assume that a failure can occur after any non-deterministic event in the computation. However, in reality failures are rare
- Pessimistic protocols implement the following property, often referred to as *synchronous logging*, which is a stronger than the always-no-orphans condition
- *Synchronous logging*

  – ∀e: ¬Stable(e) ⇒ |Depend(e)| = 0
- Thai is,if an event has not been logged on the stable storage, then no process can depend on it.

Example:

- Suppose processes $P1$ and $P2$ 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 $P0$ that includes the receipt of message $m7$ from $P1$



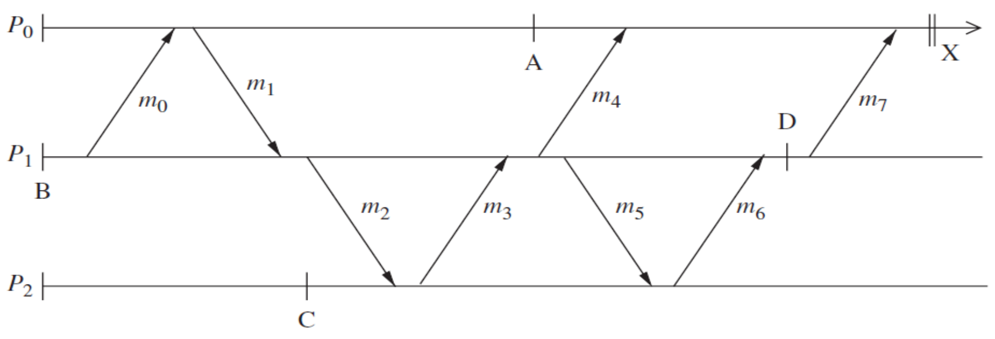
- Disadvantage: performance penalty for synchronous logging
- Advantages:
    - immediate output commit
    - restart from most recent checkpoint
    - recovery limited to failed process(es)
    - simple garbage collection
- Some pessimistic logging systems reduce the overhead of synchronous logging without relying on hardware. For example, the *sender-based message logging* (SBML) protocol keeps the determinants corresponding to the delivery of each message *m* in the volatile memory of its sender.
- The ***sender-based message logging* (SBML) protocol**

  Two steps.

  1. First, before sending m, the sender logs its content in volatile memory.
  2. Then, when the receiver of m responds with an acknowledgment that includes the order in which the message was delivered, the sender adds to the determinant the ordering information.

2. Optimistic Logging

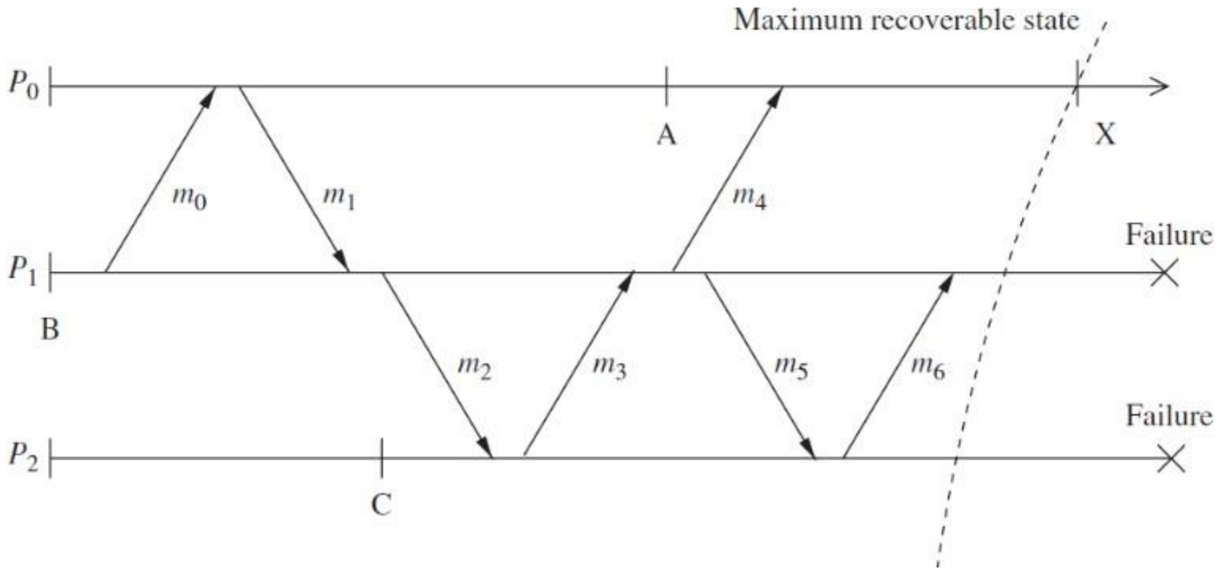
- Processes log determinants asynchronously to the stable storage
- Optimistically assume that logging will be complete before a failure occurs
- Do not implement the *always-no-orphans* condition
- To perform rollbacks correctly, optimistic logging protocols track causal dependencies during failure free execution
- Optimistic logging protocols require a non-trivial garbage collection 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



- 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.
- **Advantage: better performance in failure-free execution**
- **Disadvantages:**
    - **coordination required on output commit**
    - **more complex garbage collection**
- Since determinants are logged asynchronously, output commit in optimistic logging protocols requires a guarantee that no failure scenario can revoke the output. For example, if process *P*0 needs to commit output at state X, it must log messages *m*4 and*m*7 to the stable storage and ask *P*2 to log *m*2 and *m*5. In this case, if any process fails, the computation can be reconstructed up to state X.

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



- Consider the example in Figure Messages *m*5 and *m*6 are likely to be lost on the failures of P1 and P2 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.
- The determinant of each of these events contains the order in which its original receiver delivered the corresponding message.
- The message sender, as in sender-based message logging, logs the message content. Thus, process *P*0 will be able to "guide" the recovery of *P*1 and *P*2 since it knows the order in which *P*1 should replay messages *m*1 and *m*3 to reach the state from which *P*1 sent message *m*4.
- Similarly, *P*0 has the order in which *P*2 should replay message *m*2 to be consistent with both *P*0 and *P*1.
- The content of these messages is obtained from the sender log of *P*0 or regenerated deterministically during the recovery of *P*1 and *P*2.
- Note that information about messages *m*5 and *m*6 is lost due to failures. These messages may be resent after recovery possibly in a different order.
- However, since they did not causally affect the surviving process or the outside world, the

resulting state is consistent.

- Each process maintains information about all the events that have causally affected its state.