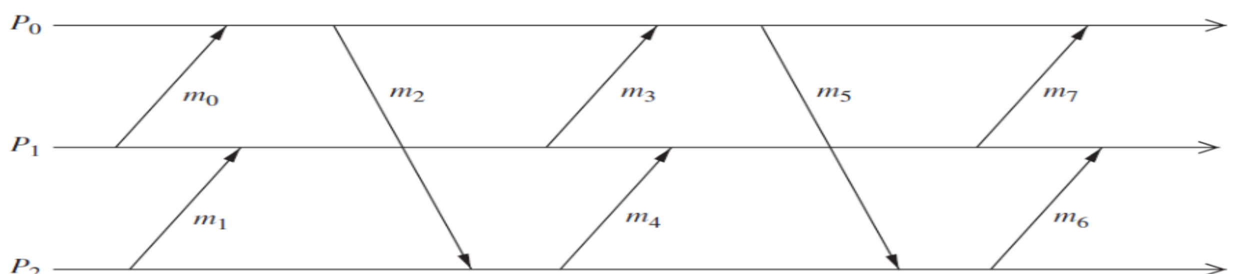
LOG-BASED ROLLBACK RECOVERY

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*.

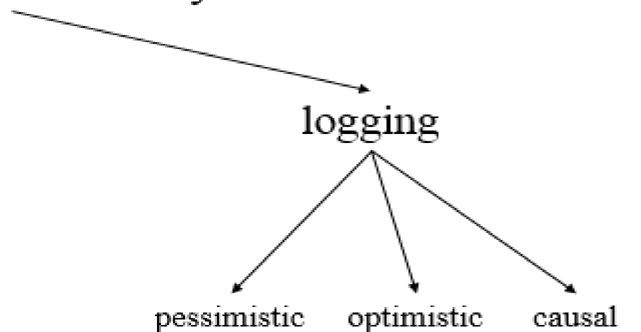
• ***Log*(*e*):** the set of processes that have logged a copy of *e*'s determinant in their volatile memory.

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

always-no-orphans condition

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

<u>Types</u>



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*

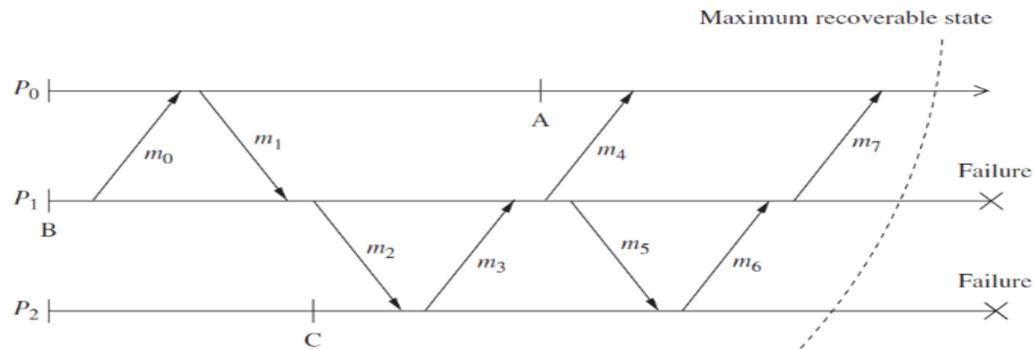
  $- \forall e: \neg Stable(e) \Rightarrow |Depend(e)| = 0$

- Thai is, if an event has not been logged on the stable storage, then no process can depend on it.

Example:

Suppose processes *P*1 and *P*2 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**

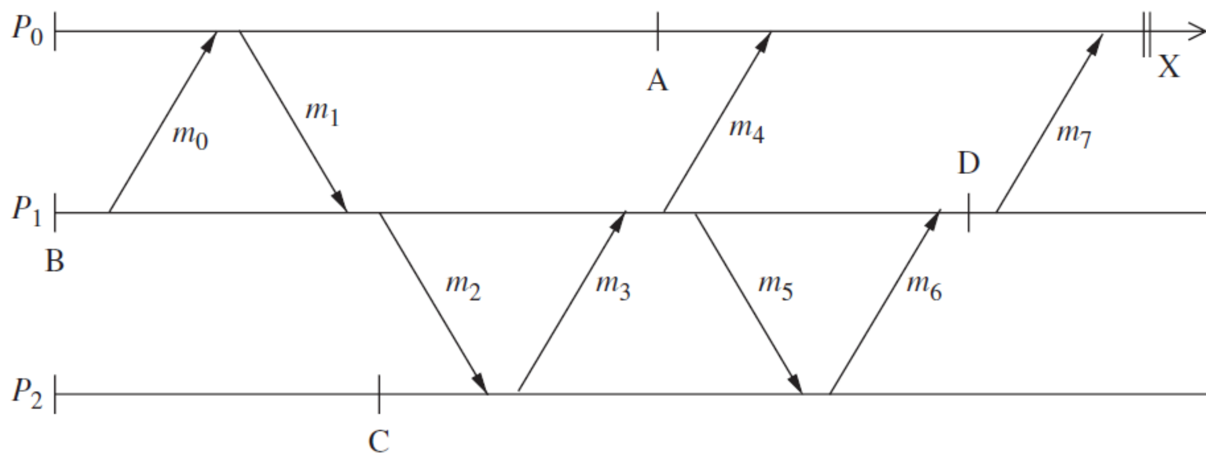
  Two steps.

  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

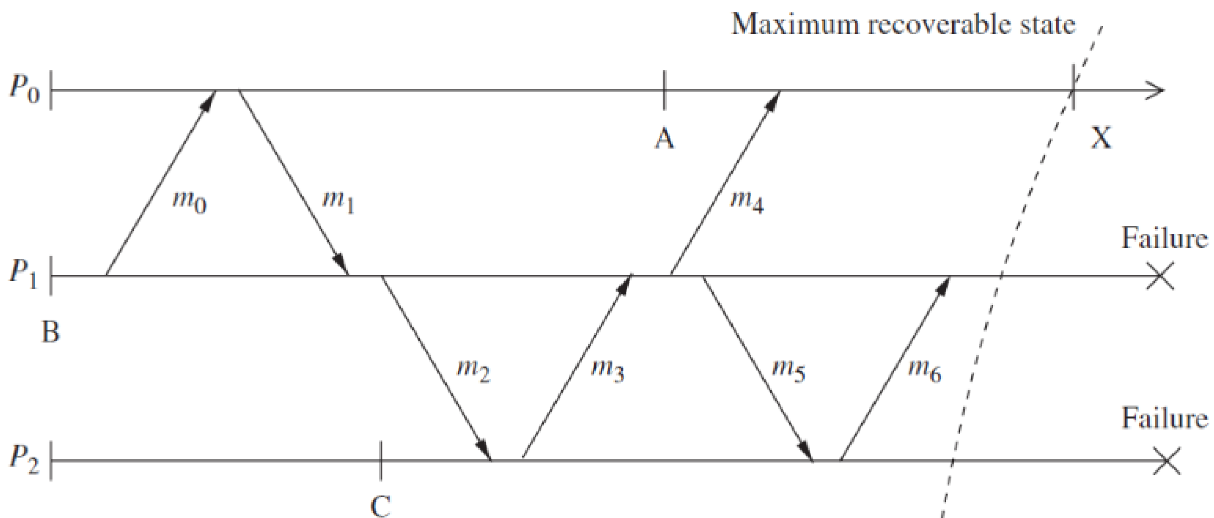


- Consider the example shown in Figure. Suppose process *P*2 fails before the determinant for *m*5 is logged to the stable storage. Process *P*1 then becomes an orphan process and must roll back to undo the effects of receiving the orphan message *m*6. The rollback of *P*1 further forces *P*0 to roll 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, $P0$ has the order in which $P2$should replay message $m2$ to be consistent with both $P0$ and $P1$.
- The content of these messages is obtained from the sender log of $P0$ or regenerated deterministically during the recovery of $P1$ and $P2$.
- Note that information about messages $m5$ and $m6$ 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.