Rollback Recovery Methods: a Quick Overview

CSEP590SG – University of Washington

Steve Gribble (gribble@cs.washington.edu)

[This material is taken from the paper "A Survey of Rollback-Recovery Protocols in Message-Passing Systems", by Elnozahy, Alvisi, Wang, and Johnson.]

Basic goal

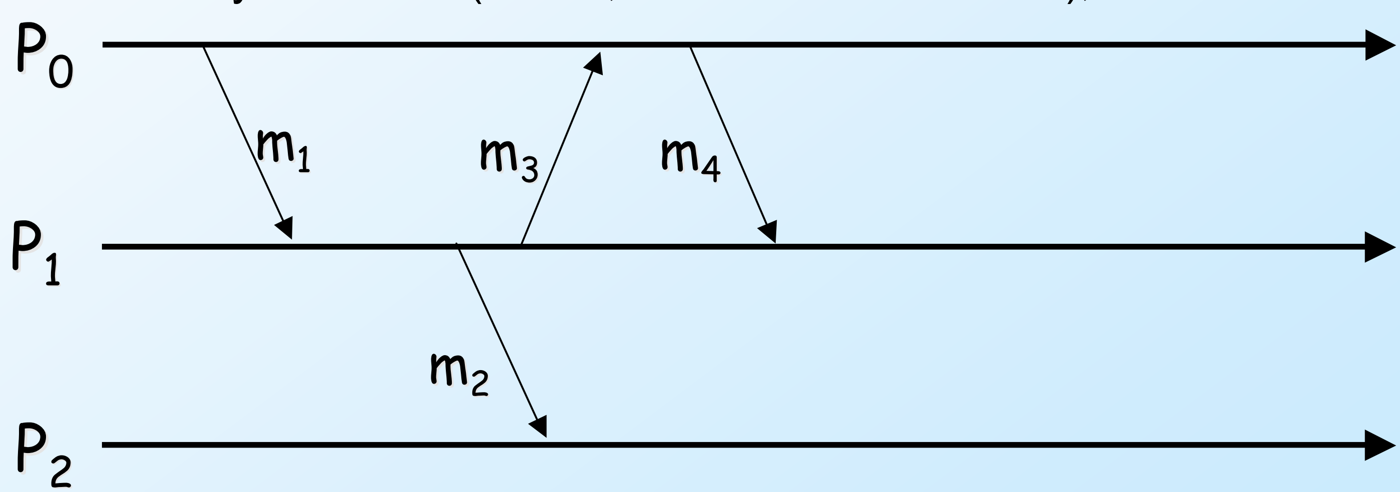
- **Fault tolerance of a long-running, distributed computation**
  - Ability to restart global computation to a "consistent" snapshot
  - Coordinate local process states and (causal) dependencies
- **Model: collection of processes, message-oriented computation**
  - Fail-stop: processes suddenly disappear when crash
    - No Byzantine failures (incorrect events are never generated)
- **Goal: recovery is transparent to both programmer and application**

Basic model

- **Finite number of processes in system**
  - Process “birth” is same as process doesn’t interact with other processes, outside world, until “birthday”
  - Process “death” must be that process doesn’t generate any events, or receive input from outside world after death
- **Communication network**
  - Message-oriented [don’t worry about bytestreams]
  - Arbitrary topology
  - Unreliable message delivery [lose, duplicate, reorder messages]
    - Some protocols assume reliable delivery, in which case system state includes channel state [why?]

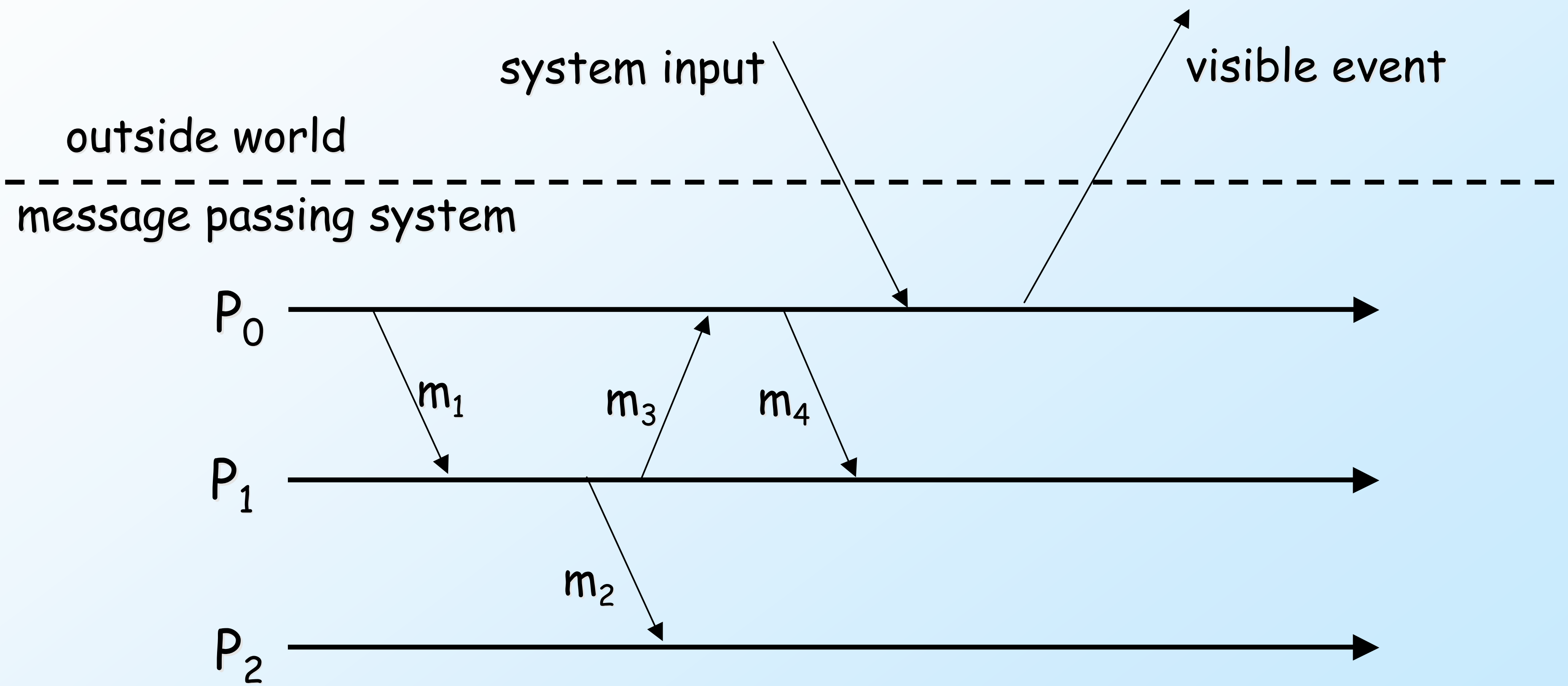
Picture of basic system

- **Process execution modeled as sequence of state intervals**
  - Deterministic computation started by a non-deterministic event
    - Non-determinism: in model, message reception
      - » what about message transmission?
    - In reality: also read physical clock, input from world, execute most system calls (failure, variable return values), …



Bigger picture

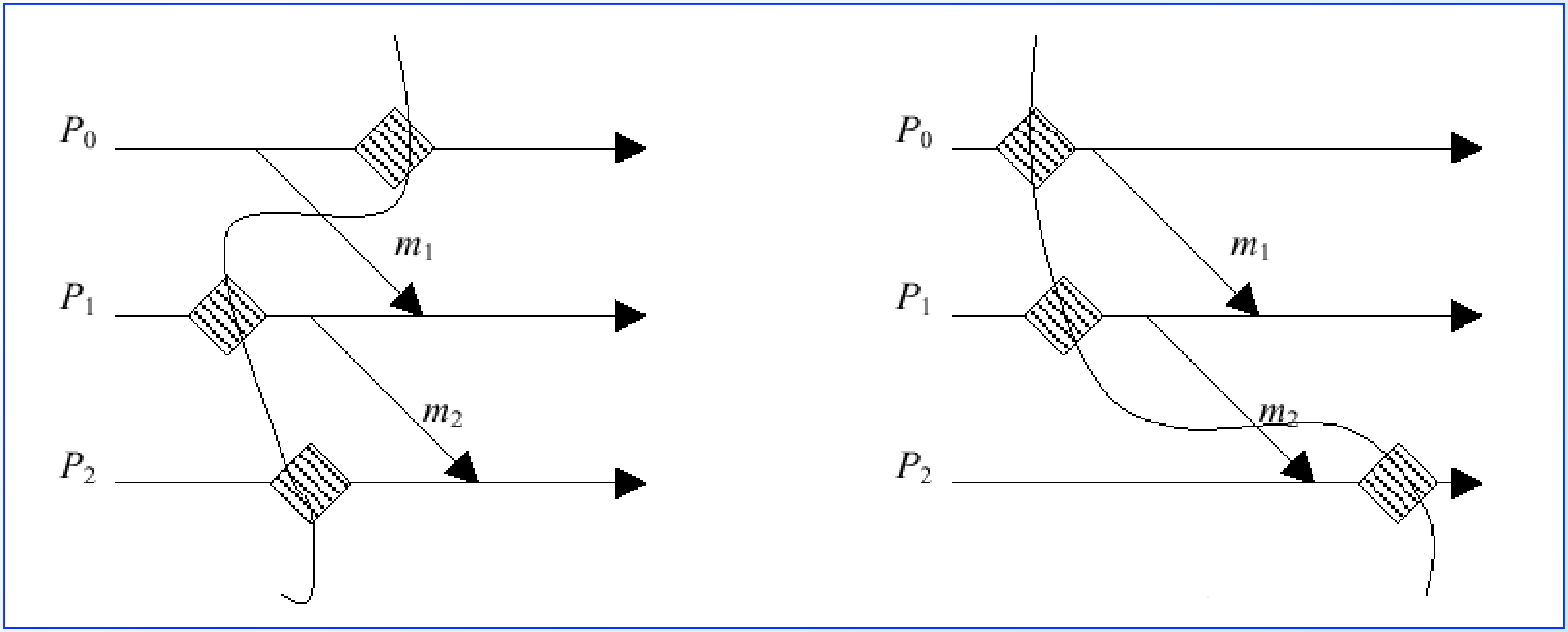
- **The "outside world" matters too**



A computation

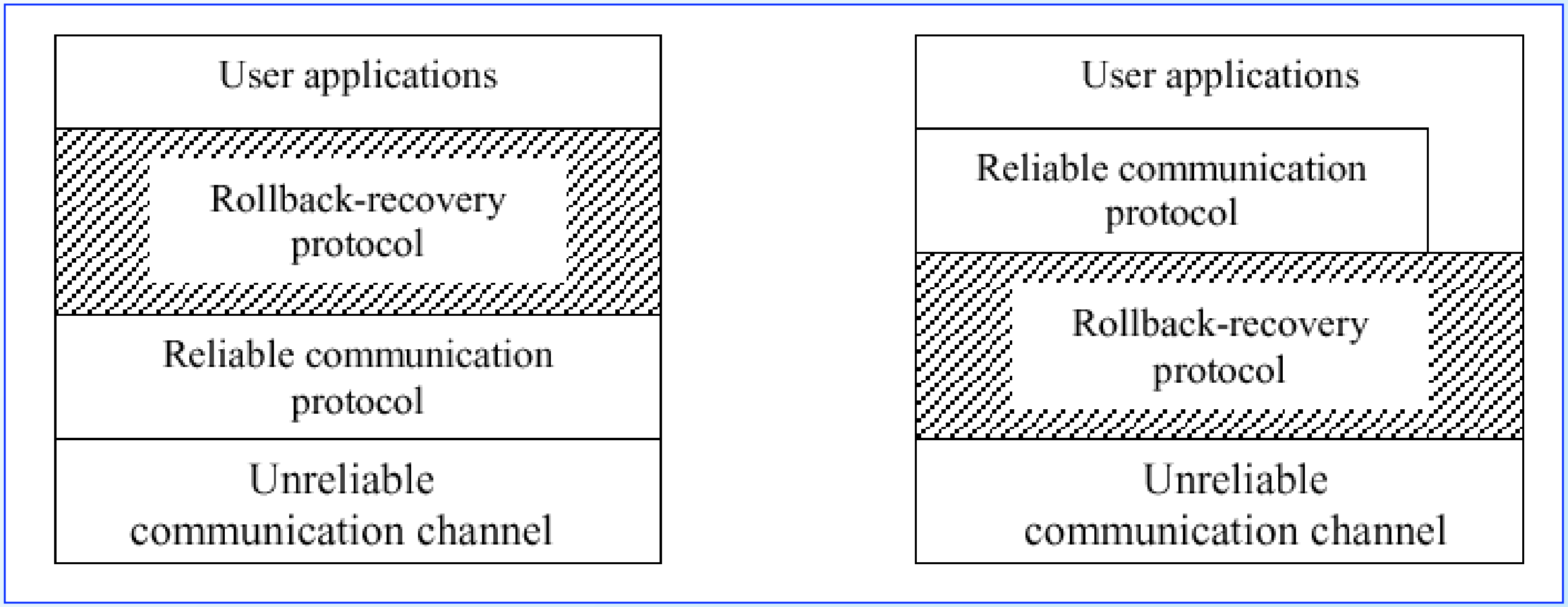
- **A "computation" represents the evolution of the system state over time**
  - System state means {process state}, possibly state of channels
  - "Consistent system state": may occur in failure-free, correct execution
    - Iff. If a process's state reflects a message receipt, then state of corresponding sender reflects sending that message
      - Is this the same as Lamport's causal ordering?
- **Goal of rollback recovery protocol:**
  - Bring system back into consistent state when inconsistencies occur because of a failure.
    - Reconstructed state may not be one that occurred before the failure. It is sufficient that it "could" have occurred.

Consistent vs. Inconsistent State



Drilling down on network channel state

- **Two models:**
  - reliable communications substrate is underneath recovery
  - or, reliability is is implemented above recovery mechanisms



More on channels

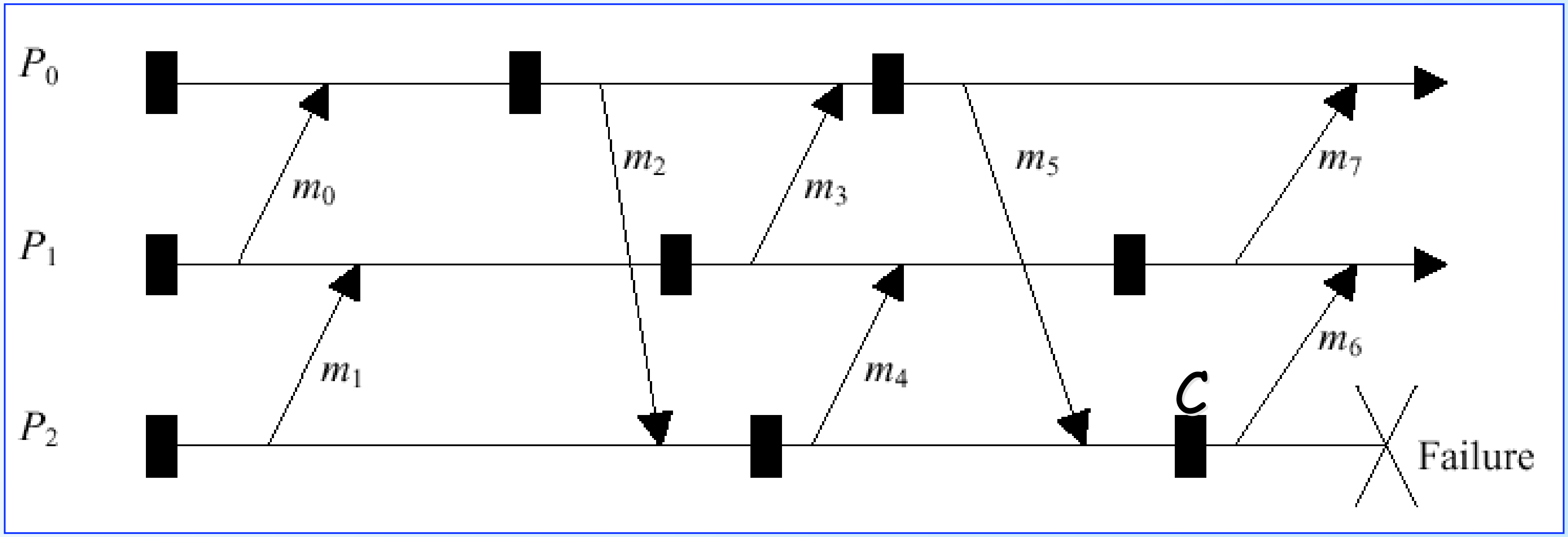
- **Counterintuitive:**
  - If reliability is implemented above the recovery protocol, then the recovery protocol can simply ignore all channel state
    - Assuming a reliability protocol complicates recovery!!
- **To wit: if reliability is below**
  - TCP-like protocols ensure message delivery during failure-free execution, but cannot promise delivery if either endpoint fails
    - Delivery state is shared across endpoints
  - So, if failure occurs on receiver:
    - Recovery protocol must ensure sender's TCP does not time out, as receiver will eventually recover
    - (TCP timeout changes computation of sender application)

Checkpointing protocols

- **Basic hammer: each process periodically saves its state on stable storage**
  - State contains enough information to restart process execution
- **Goal is to construct a "*consistent global checkpoint*"**
  - Set of local checkpoints, one from each process, forming consistent system state.
  - Can restart system from any consistent global checkpoint after failure
    - generally want to use the most recent consistent global checkpoint [called *recovery line*]

What makes this hard: Domino Effect

- **Suppose P2 fails, and rolls back to checkpoint C**
  - Where is the recovery line?



Answer:

- **Rollback “invalidates” sending of message m6, so P1 must roll back to B to invalidate the receipt of message**
  - Otherwise P1 becomes an “orphan process”
- **But, rollback of P1 invalidates sending of m7, so P0 must roll back to A.**
- **Etc., until you get all the way back to the beginning.**

Getting around the Domino effect

- **Must be careful about *coordinating* checkpoints**
  - Simplest way: execute some sort of consensus process to synchronously begin checkpoint at all processes
    - E.g., 2-phase commit
    - Very expensive!
- **Another way: log events to supplement checkpoints**
  - Log non-deterministic events after checkpoint
  - Checkpoint + log guarantees that a process computation proceeds identically to prefailure computation
    - Identical until first non-logged, non-deterministic event after the last checkpoint

What about outside events?

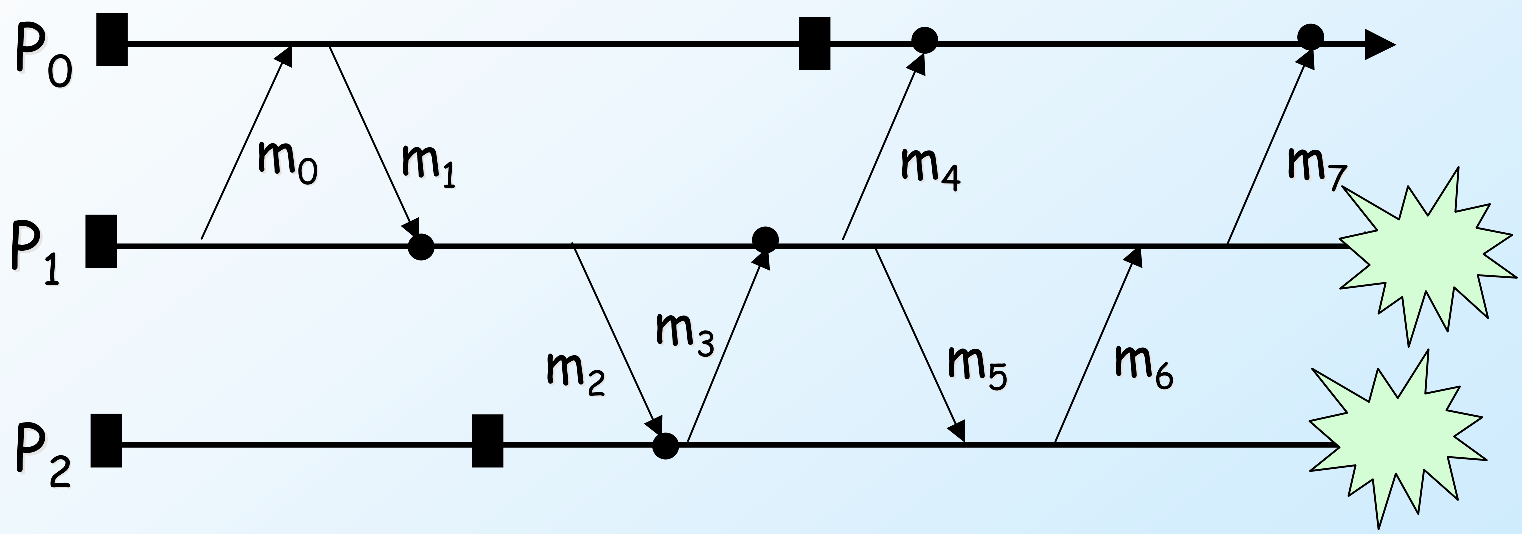
- **Input events:**
  - must log them, since not guaranteed that outside world is recoverable
- **Output events:**
  - this is the Lowell paper
    - locally, must log before generating output event
    - globally, must ensure consistent checkpoint before generating output event
  - expensive to handle, but necessary
    - alternative is “compensation events”

Logging Protocols

- **Non-deterministic events (incl. input) must be logged**
  - Alternative: checkpoints must be taken before process induces a side-effect after non-deterministic event
  - Logs depend on piecewise deterministic (PWD) assumption
    - Ability for application to log a "determinant" of non-deterministic events
    - Determinant contains all info necessary to replay event after failure
- **Process state interval is *recoverable* if:**
  - enough information in checkpoints/logs to replay execution up to that state interval, despite any future failures in system
- **State interval is *stable* if:**
  - Determinant of non-deterministic event that started it is in the log

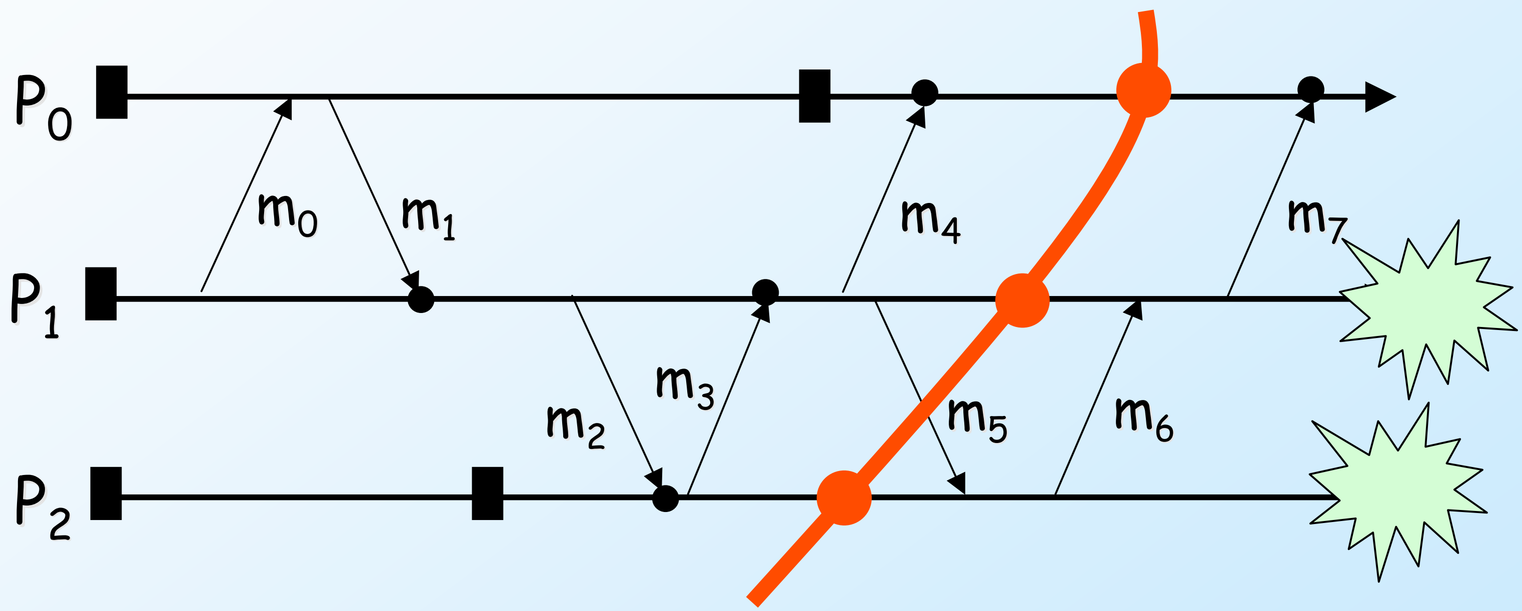
- **Q: does recoverable interval → stable interval?**
- **Q: does stable interval → recoverable interval?**

Pop quiz



- **What is the "maximum recoverable state"?**
  - (most recent recoverable consistent system state)

maximum recoverable state



Recap: 2 main strategies for recovery

- **Checkpoint-based rollback recovery**
  - Depend only on sequence of checkpoints to recover system
    - No logging of events
  - Challenge: overcoming domino effect to find “recovery line”
- **Log-based rollback recovery**
  - In addition to checkpoints, log non-deterministic events
    - Essentially adds to checkpoint by logging non-deterministic decisions since last checkpoint
  - Challenge: overcoming cost of (synchronously) logging events

Uncoordinated Checkpointing

- **Checkpoint-based recovery, but uncoordinated: maximum autonomy across processes**
  - Purely local policy dicates when to record a checkpoint
  - Requires "dependency graphs" to calculate recovery line
    - Dependency information piggybacked on messages
- **Problems:**
  - domino effect
  - "useless" checkpoints that will never be part of a recovery line
  - need for global "garbage collection" to reclaim no-longer-necessary checkpoints

Coordinated checkpoint recovery

- **Recovery line is constructed by cooperation**
  - Synchronous (blocking) checkpoints: two-phase commit, computation ceases during checkpoint
  - Asynchronous (nonblocking) checkpoints: Lamport's snapshot
    - Eliminate FIFO by piggybacking marker on **all** post-checkpoint messages
      - marker gets through on first message that gets through
  - Synchronized physical clocks: at time T, each process takes checkpoint, and then "freezes" to account for skew
    - Freeze time = max clock error + max failure detection time
    - Abort if detect failure
  - Communication-induced checkpoints: hybrid approach (Lowell)
    - Autonomous *local checkpoints*, but occasional *forced checkpoints*
      - e.g., when receive message

Logging protocols

- **Protocols phrased in terms of consistency conditions**
  - *No-orphans:* the set of processes that depend on a non-deterministic event is a subset of those that have logged it
- **Various flavors:**
  - Pessimistic: synchronously log all non-deterministic events
    - Observable state of each process can always be recovered
      - processes can output to world without a special protocol!
      - processes can always restart from most recent checkpoint!
      - process failure never affects other processes!
    - Can relax this slightly by only logging an event when the process is about to affect another process (e.g., output to world, or send message to process)

Log-based recovery cont.

- **More flavors:**
  - Optimistic: log non-deterministic events asynchronously
    - "hope" that entry makes it to disk before failure
      - those that don't are lost on failure
      - need to compute recovery line
    - Recovery can be synchronous or asynchronous
    - Orphans are possible, need to roll them back
  - Causal: piggyback causal dependency on messages
    - Non-deterministic event is either stable on log, or its determinant is piggybacked on all messages sent from that process
      - and transitively through "happens-before" relationship
    - Non-failed process can "guide" recovery of others