A Survey of Rollback-Recovery Protocols in Message-Passing Systems: Log-based rollback-recovery

Jonathan Chen

Motivations:

- Uncoordinated checkpoint based recovery is susceptible to a domino effect. Logs can be used to limit rollback.
- Checkpoints are coarse grained, and can still lose a substantial amount of work upon failure
- Logging increases the ability of a system to recover far beyond that of a purely checkpoint-based system.

Fundamentals of Log-based Recovery

- The piecewise deterministic assumption:
  - All nondeterministic events that a process executes can be identified.
    - Nondeterministic events vary based on implementation
      Examples of nondeterministic events:
      - receiving messages
      - receiving input from the external world
      - changing state w/in a process due to interrupt
  - Determinant - Information necessary to replay a nondeterministic event during recovery. Logged during failure-free operation
  - Other events can be uniquely determined by other factors in a process's execution.

- Checkpoints may reduce the amount of execution replayed. A failed process is rolled back to its last checkpoint and events are replayed in order.

Log-based Rollback-Recovery Protocols

- pessimistic logging
  - never creates orphan processes
  - higher failure-free performance costs

- optimistic logging
  - reduces failure-free performance
  - can create orphans, increasing recovery, garbage collecting and output commit

- causal logging
  - aims for low performance overhead and fast output commit
  - may require complex recover and garbage collection

Pessimistic Logging

- Assumes failures commonly occur at any nondeterministic event in a computation.

- The determinant of any non-deterministic event is prohibited from affecting computation unless logged to stable storage
  - synchronous logging - $\forall e : \neg Stable(e) \Rightarrow |Depend(e)| = 0$

Major Benefits of Pessimistic Logging

- The observable pre-failure state of each process is always recoverable
  - Processes can send messages to the outside world w/ no special protocol.
  - Recovery always starts from the most recent checkpoint.
  - Effects of failure are confined to the processes that fail.
  - Garbage collection is simple: reclaim everything before last checkpoint

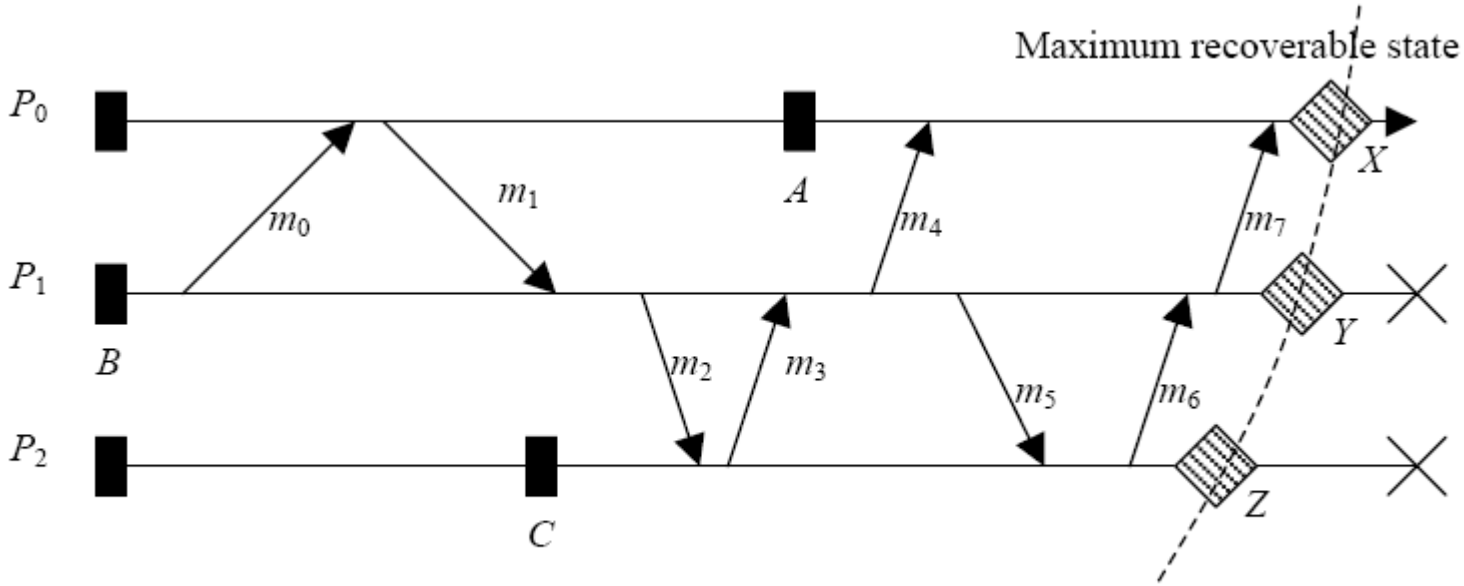

Figure 10. Pessimistic logging.

Performance Impact of Pessimistic Logging

- Synchronous logging can result in high performance overhead. Some special hardware exists to reduce the overhead
  - fast, non-volatile semiconductor memory as stable storage
  - special bus guaranteeing atomic logging of messages w/in a system

Sender-Based Message Logging (SBML) scheme:

- logged in sender rather than receiver
- 2-step logging of determinants:
  - content of message *m* logged upon sending
  - sender obtains ordering information in confirmation of receipt.
- limited to a single failure
- cannot handle internal nondeterministic events

Relaxed Logging Atomicity

- Determinants aren't logged until the receiving process communicates with another process.
- $\forall e : \neg Stable(e) \Rightarrow |Depend(e)| \leq 1$
- Multiple events can be logged together, decreasing the frequency of accesses to stable storage.
- Interprocess communication and output commit often require logging
- Assuming reliable channels may cause difficulty in dealing w/ separately logging and sending messages.

Optimistic Logging

- Optimistically assumes that logging will complete before a failure occurs.
  - keeps a volatile log of determinants which is periodically flushed to stable storage
  - does not force the application to block while writing determinants.
  - little failure-free overhead.

- Determinants in volatile log are lost upon failure.
  - Processes may become orphans if the failed process sent a message during an unrecoverable state interval.

Optimistic Logging (cont'd)

- Orphaned processes must roll back to remove effects of receiving an orphaned message
- Causal dependencies must be tracked during failure-free execution.
- Multiple checkpoints are maintained, complicating garbage collection.
- Output commit becomes complicated since specific determinants must be logged so output cannot be revoked as a result of a rollback.

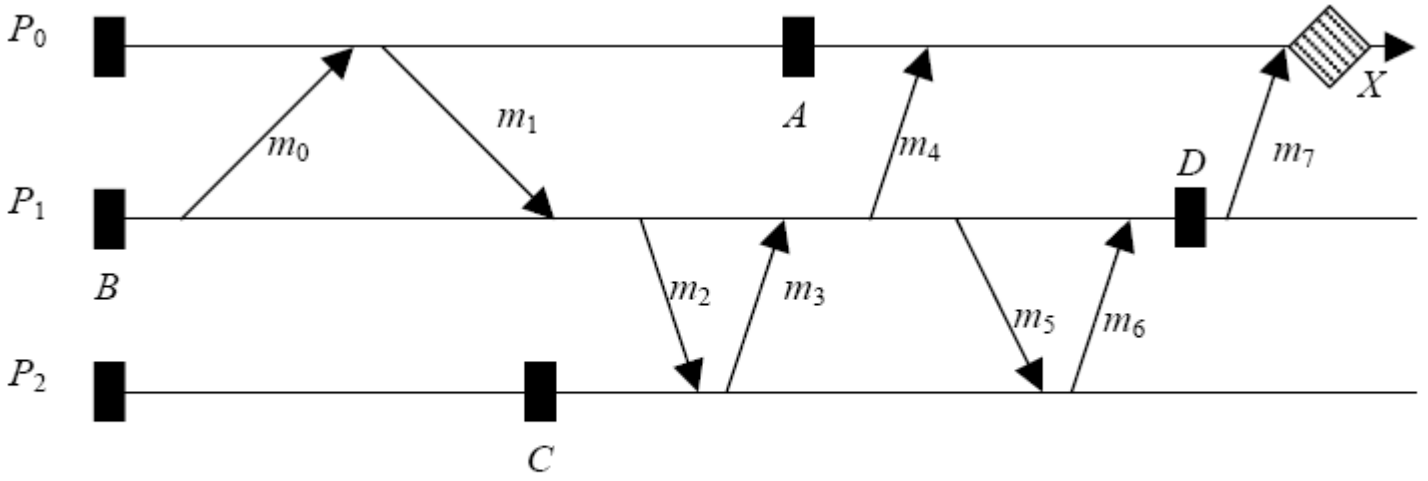

Figure 11. Optimistic logging.

Optimistic Logging Recovery

- Synchronous recovery:
  - All processes run a recovery protocol to determine the maximum recoverable system state, then performs the rollbacks.
  - Relies on dependency tracking and logged information.

  - Dependency tracking:
    - a state interval index is maintained to distinguish the state intervals of a process

    - direct – state interval index is sent along w/ outgoing messages

    - transitive – each process maintains a vector of the highest state intervals of other processes' which the current process depends on
      - faster output commit and recover at the cost of higher failure-free overhead

Optimistic Logging Recovery (cont'd)

- Asynchronous recovery:
  - A failed process restarts and broadcasts a rollback announcement to begin an incarnation.
  - When a process receives the rollback announcement it determines whether or not it is now an orphan.
    - If yes, roll back and broadcast own rollback announcement.
    - Processes may have multiple incarnations. Dependencies must be tracked for every incarnation of each process.
      - When a process P_i delivers a message to process P_j, P_j sends a rollback announcement to P_i to make sure P_i does not depend on invalid states of P_j's previous incarnations.

Exponential rollbacks

- a single failure can cause a process to roll back an exponential number of times.

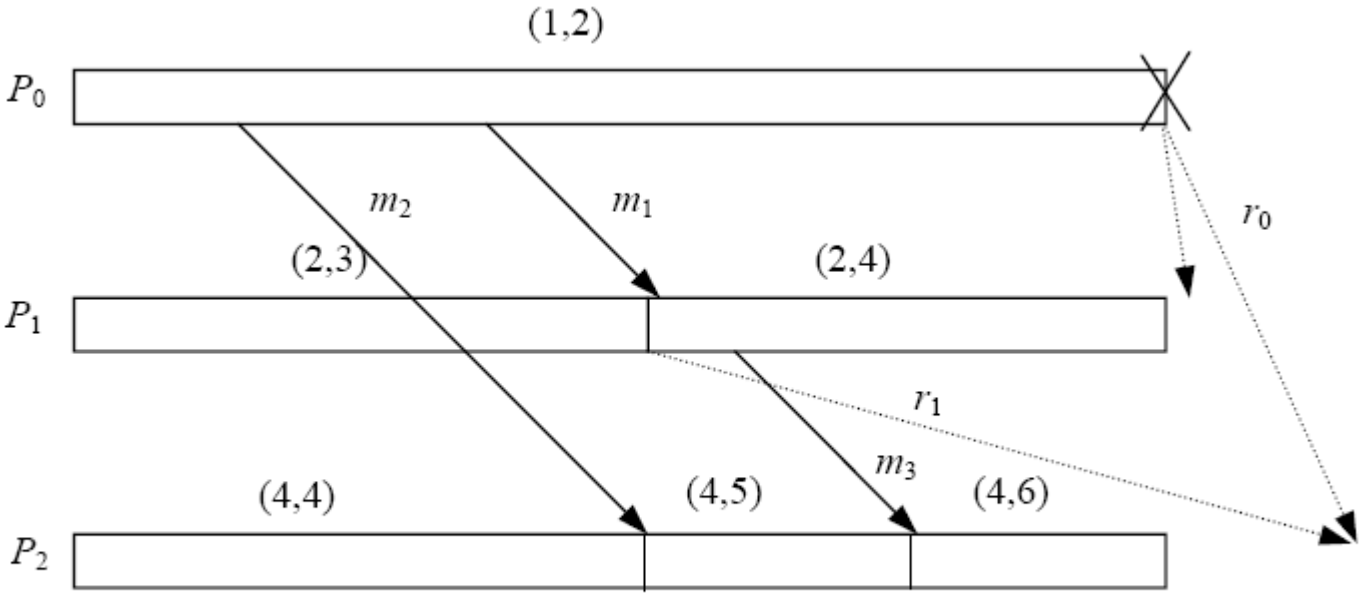

Figure 12. Exponential rollbacks.

- P_i rolls back 2^{i-1} times in response to P_0's failure.

Exponential rollback (cont'd)

- In order to eliminate exponential rollback, several approaches can be taken to make sure a process is rolled back at most one time.
  - distinguish between failure and rollback announcements, requiring processes receive the failure notification before rolling back.
  - piggyback original rollback announcement w/ subsequent rollback announcements.

Causal Logging

- Avoids synchronous access to stable storage except during output commit
- Allows processes to commit output independently and never creates orphans.
- Limits rollback of a failed process to the most recent checkpoint on stable storage.

- Ensures stable or locally available determinants to any process whose state it causally precedes.
  - A process which depends on a failed process can guide the recovery of the failed process.
  - With multiple failed processes, determinants can still be lost, but only if no other processes has a dependency.

Tracking Causality:

- Non-stable determinants are piggybacked on sent messages.
- Manetho system – causal information maintained in antecedence graph.
  - Every process has a complete history of events effecting its state.
  - Local antecedence graph is piggybacked on message send.
    - In practice, only updates need to be sent.
- Limiting number of tolerated failures reduces overhead from stable storage access.

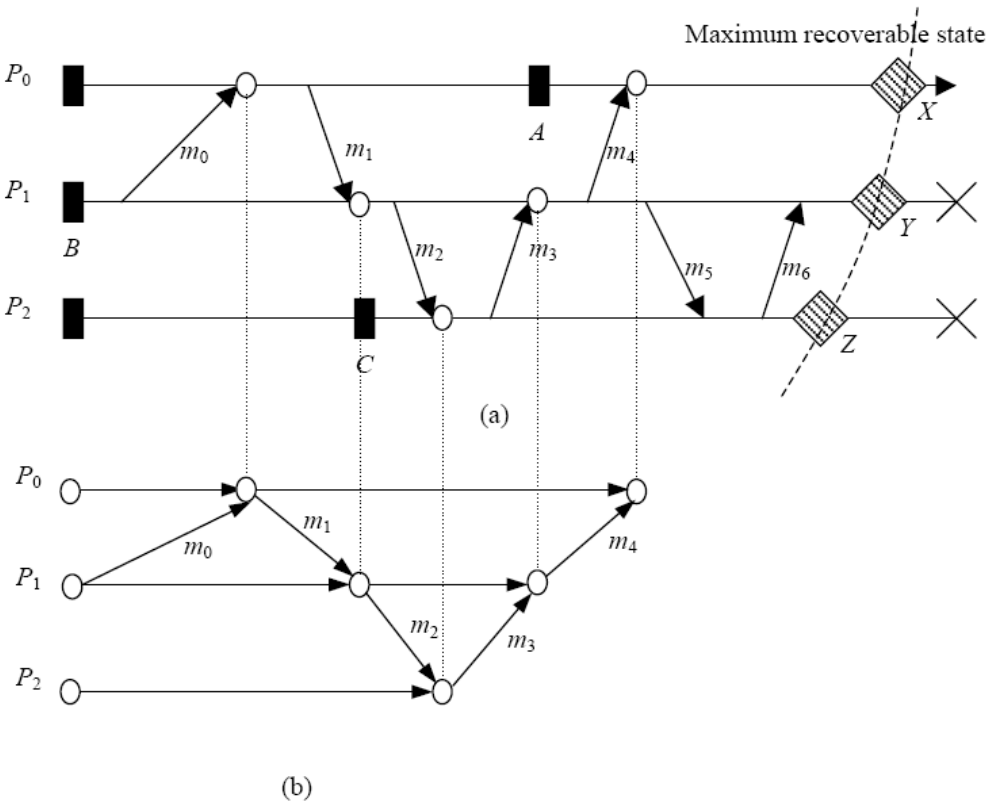

Figure 13. Causal logging. (a) Maximum recoverable states, and (b) antecedence graph of P_0 at state X.

Log-based Recovery: Implementation Issues

- Failure-free overhead
  - messages held in local memory of each process
    - reduced communication throughput and latency
  - flushing the volatile log to stable storage
  - additional bandwidth used to communicate to stable storage if this is accessed through a network.
- Garbage Collection
- Recovery complexity

Implementation Issues (cont'd)

- Support of nondeterministic events
  - System calls:
    - return deterministic values
    - logged but not re-executed – return different value on replay
    - logged and re-executed – modifications to environment.
  - Asynchronous signals
    - tracked via instruction counters – number of instructions between asynchronous interrupts

Coordinated vs. Uncoordinated Checkpointing in Log-based Recovery

- Combining log-based recovery w/ coordinated checkpointing
  - simplified garbage collection
  - fast output commit
  - no need to maintain large logs on stable storage in sender-based logging implementations
    - reduced overhead
    - message logging w/ coordinated checkpointing shown to outperform message logging w/ uncoordinated checkpointing.

Comparison of rollback-recovery protocols

	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

Table 1 A comparison between various flavors of rollback-recovery protocols.