A Survey of Rollback-Recovery Protocols in Message-Passing Systems

E.N. Elnozahy, L. Alivisi, Y.-M. Wang, and D.B. Johnson

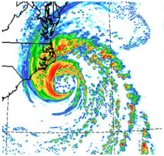
Presented by Denis Conan

June 2024

Outline

1 Context: Fault-tolerance of long-running applications

- Long-running HPC[1] applications for: environment, climate, new energies, health, biology, transport, geophysics, astrophysics, plasma physics, laser physics, human and social sciences



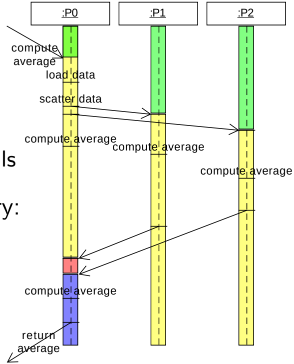

- Too many processes to replicate (active replication), then fault-tolerance through passive replication
  - Save recovery information periodically during failure-free execution
    - Processes' states into what are called *checkpoints*
    - Messages of the interactions into logs
  - In case of a crash, recover from an intermediate state by rollbacking

1. High Performance Computing (See Module CSC5001)

2 Background and Problem Definition

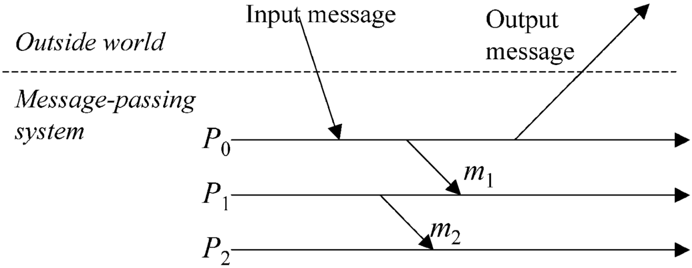
2.1 System Model and Correctness Condition

- Processes communicate by exchanging messages
- No partitioning, reliable network, but all processes may fail simultaneously
- Fail-stop model: Failures are correctly detected
- The piece-wise deterministic assumption:
  - Process execution = sequence of state intervals, each started by a nondeterministic event
  - Process execution = deterministic in state intervals
- Generic correctness condition for rollback-recovery:
  - A system recovers correctly if its internal state is consistent with the observable behaviour of the system before the failure



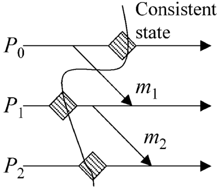
2.2 Communication with the Outside World

- Nondeterministic events (e.g. timeouts) can be modelled as input messages
  - Remember that the aim is to be able to replay these events
- The OW process cannot maintain state, and cannot rollback

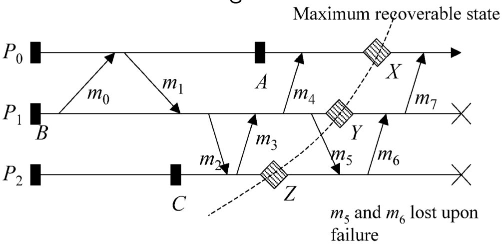


2.3 In-transit messages, determinant, logging, orphan message

- In-transit message: Since we assume reliable co
  a message seen as sent, but not yet received,
  must be stored with receiver's checkpoint
  so that the receiver "replays" the receipt
  when rollbacking



- Determinant of a message = all information necessary to replay the event
- Logging = saving message determinants in stable storage
- Orphan message =
  a message (which determinant)
  was not logged
  before the failures
  and that cannot be recovered

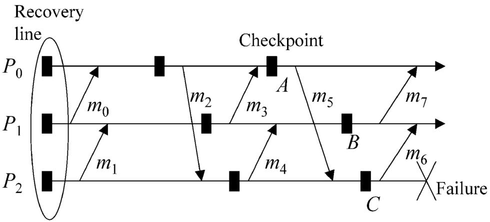


3 Approach 1: Checkpoint-based rollback-recovery

- More in the article on Communication-Induced Checkpointing

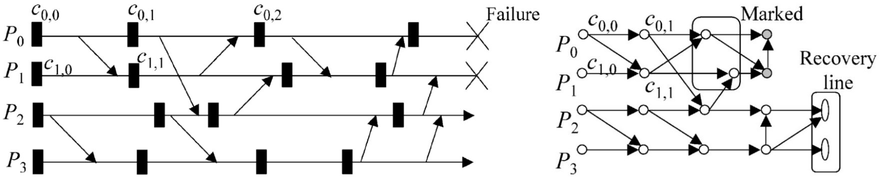
3.1 Uncoordinated Checkpointing I

- Each process takes a checkpoint whenever it wants, without coordination
- The pros:
  - Choose the right time to decrease the size of the checkpoint (proc. memory)
- The cons:
  - A checkpoint may be useless (will never be part of a global consistent state)
  - Maintain lot of checkpoints before garbage collection
  - Be subject to the domino effect



3.1 Uncoordinated Checkpointing II

- Determine the maximum recovery line, i.e. max. consistent global checkpoint
  - Track checkpoint interval, piggyback it in messages, and record checkpoint dependencies
  - In case of failure, compute the recovery line in the dependency graph by starting with the checkpoints of the faulty processes



3.2 Coordinated Checkpointing

- Orchestrate checkpointing actions in order to form consistent global states
  - Distributed snapshots algorithm "à la" Chandy&Lamport'1985
  - Storage of in-transit messages in order to be able to replay them
- Cons
  - Synchronisation of all processes $\Longrightarrow$ not possible to choose timing individually
- Principle of Communication-Induced Checkpointing (more in the article)
  - Piggyback checkpointing information into application messages
    so that a process knows
    whether taking an uncoordinated-checkpoint "now" may be useless
    or not taking a checkpoint "now" will render other checkpoints useless

4 Approach 2: Log-based rollback-recovery

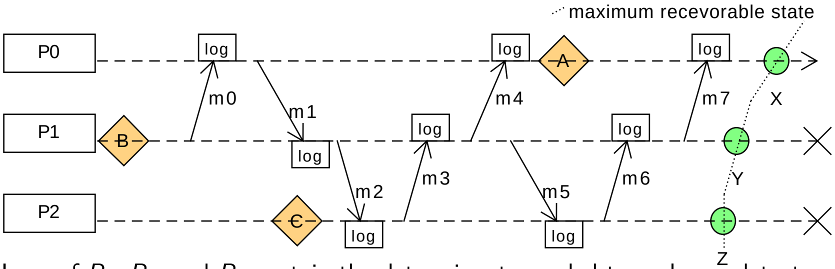
- More in the article on Causal Logging

4.1 Always-no-orphans condition

- Reminder: Piece-wise deterministic assumption
- Log-based rollback-recovery enables rollback-recovery beyond the most recent set of consistent checkpoints
  - Reminder: Determinant of event e = all information necessary to replay e
  - Execution can be reconstructed up to the first nondeterministic event whose determinant is not logged
- Corollary: checkpointing is not necessary before sending to the outside world
- Preliminary definitions about logging
  - $Depend(e)$: set of processes that are affected by a nondeterministic event e
  - $Log(e)$: set of processes that have logged a copy of e's determinant (in their volatile memory)
  - $Stable(e)$: true if e's determinant is logged on stable storage
- Always-no-orphans condition $\equiv \forall e : \neg Stable(e) \implies Depend(e) \subseteq Log(e)$

4.2 Pessimistic Logging I

- "Pessimictic" = a failure can occur after any nondeterministic event
- Synchronous logging: Log the determinant to stable storage before delivery
  - $\forall e : \neg Stable(e) \implies |Depend(e)| = 0$



Logs of P_0, P_1, and P_2 contain the determinants needed to replay or detect the replay of messages $[m_0, m_4, m_7]$, $[m_1, m_3, m_6]$, and $[m_2, m_5]$, respectively

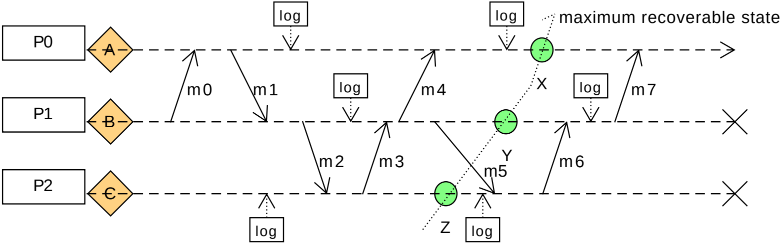
4.2 Pessimistic Logging II

- The pros:
  - Processes can send messages to the outside world without running a special protocol
  - Processes restart from their most recent checkpoint
  - Recovery and garbage collection are simplified
- The cons:
  - Performance penalty, and, in reality, failures are rare
    - Countermeasures:
      - If only one failure, sender-based message logging: keep the determinant in the volatile memory of its sender
        $\Longrightarrow$ Avoid the overhead of accessing stable storage
      - Defer logging until the receiver sends another message
        $\Longrightarrow$ Relax logging atomicity (i.e., some form of asynchrony)

4.3 Optimistic Logging

- Log determinants asynchronously to stable storage
  - In volatile memory and periodically flushed to stable storage
- If a process fails, the determinants in its volatile memory will be lost

$\implies$ This solution does not implement the always-no-orphans condition



$\neg Stable(m_5) \wedge Depend(m_5) = \{P_2, P_1, P_0\} \not\subseteq Log(m_5) \implies m_5,\ P_0..P_2$ orphan

- What if using causal logging? ... P_0 knows determinants of m_5, m_6, m_7 ... see in the article

5 Conclusion

- Concepts
  - Checkpoint, piece-wise determinism, rollback-recovery, outside world
  - In-transit message, determinant, logging, orphan message
  - Checkpoint-based rollback-recovery, un/coordinated Checkpointing,
    - Without logging, message to the outside world $\Longrightarrow$ checkpointing
  - Log-based rollback-recovery, no-orphans condition, pessimistic/optimistic logging
- More in the article
  - Communication-Induced Checkpointing (zigzag path/cycle)
  - Causal logging
  - Implementation issues