Rollback-Recovery Protocols in Message-Passing Systems

Hai Nguyen

Approaches to reliability

- Transactions: data-oriented applications

- Group communications: abstraction of ideal communication system

- Rollback recovery: long-running applications

"A system recovers correctly if its internal state is consistent with the observable behavior of the system before the failure"

Background and definitions

- System model

- Collection of application processes, communicating thru a network

- Processes have access to a *stable storage* device, where recovery info is periodically saved during failure-free execution

- Recovery info includes:

    - checkpoints

    - logs of interactions with I/O devices

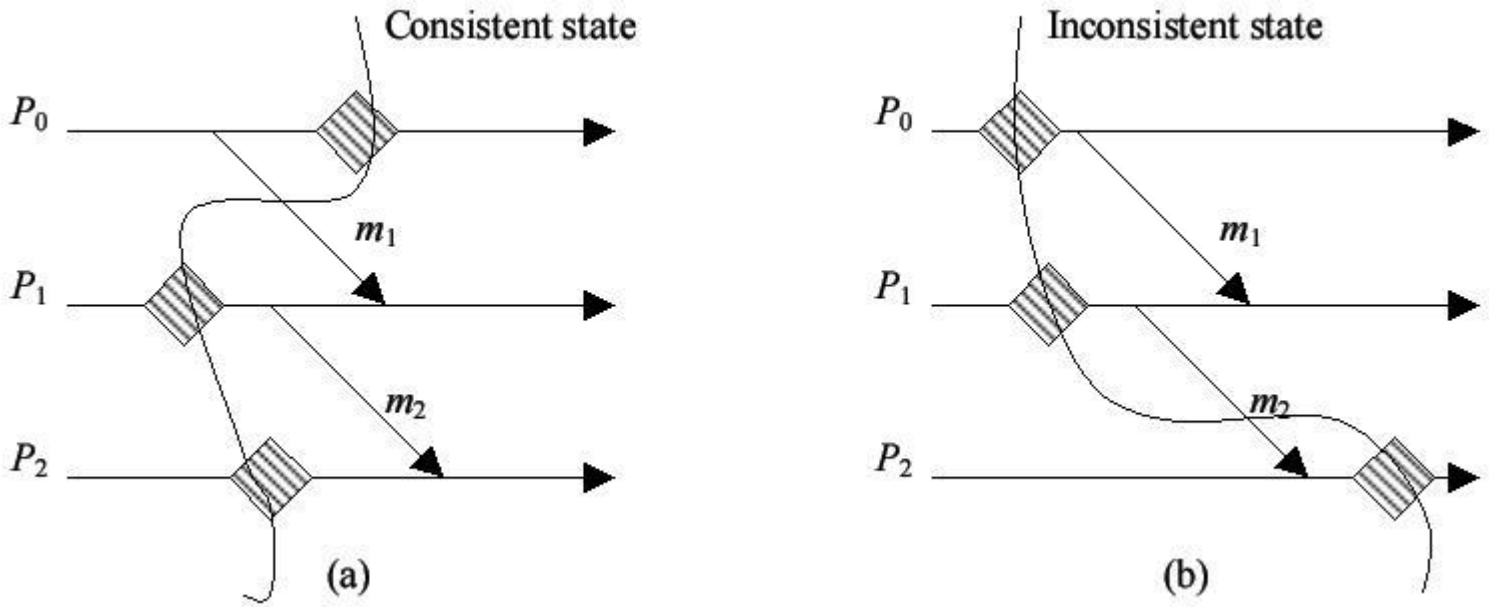
    - events that occur at each process

    - messages exchanged among processes

- Protocols may assume the communication subsystem is reliable & FIFO, or unreliable (lost, duplicate, reordered messages)

Background and definitions

- Consistent system states



Classification

- Checkpoint-based

  relies only on checkpoints – the states of the participating processes

- Log-based

  combines checkpointing with logging of nondeterministic events

Checkpoint-based rollback recovery

1. Uncoordinated checkpointing
2. Coordinated checkpointing
3. Communication-induced checkpointing

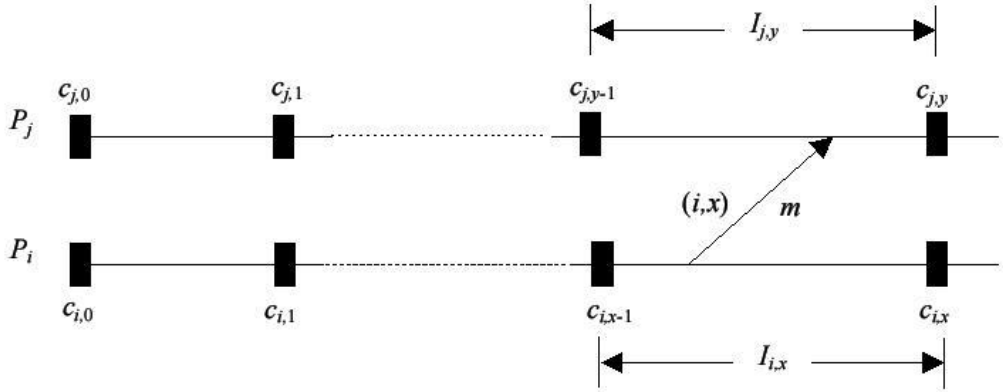
1. Uncoordinated checkpointing

-Each process independently decides when to take checkpoints

- $c_{i,x}$: x[th] checkpoint of process P_i

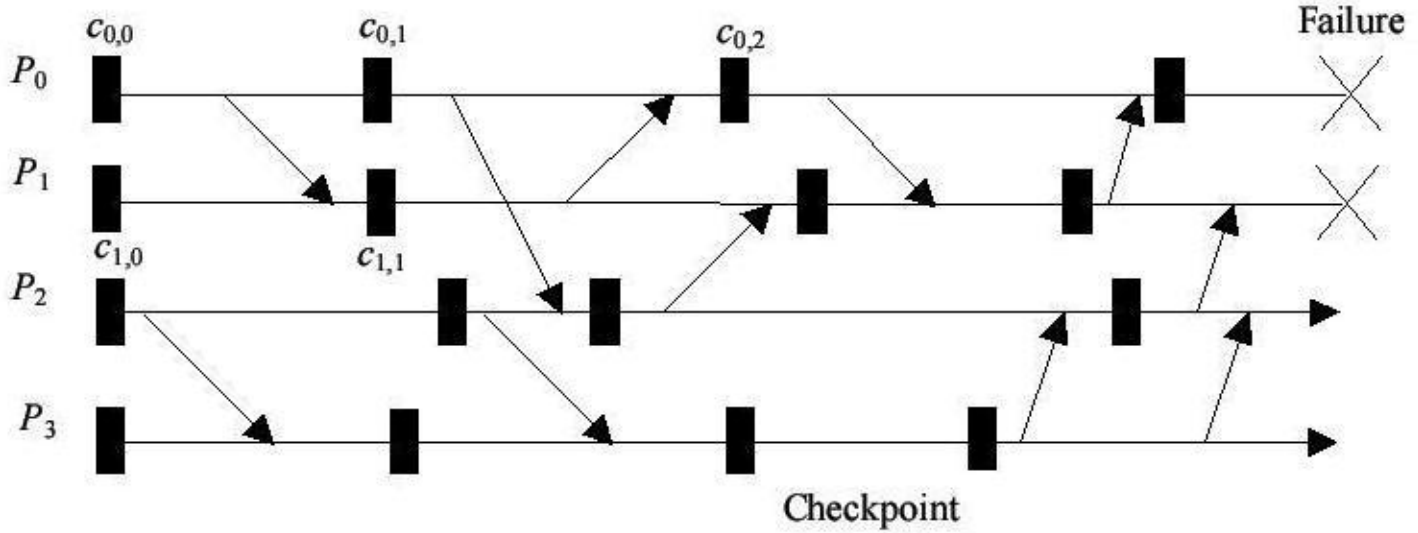
- $I_{j,y}$: checkpoint interval

Maximum number of useful checkpoints that must be kept on stable storage cannot exceed N(N+1) / 2



Recovering process		Other processes
Broadcasts *dependency request*	→	Stop execution
		↓
		Reply with dependency info (saved and current)
Calculates *recovery line*	←	
↓		
Broadcasts *rollback request* containing recovery line	→	If state belongs to recovery line, resume execution. Else, rollback to an earlier checkpoint

1. Uncoordinated checkpointing



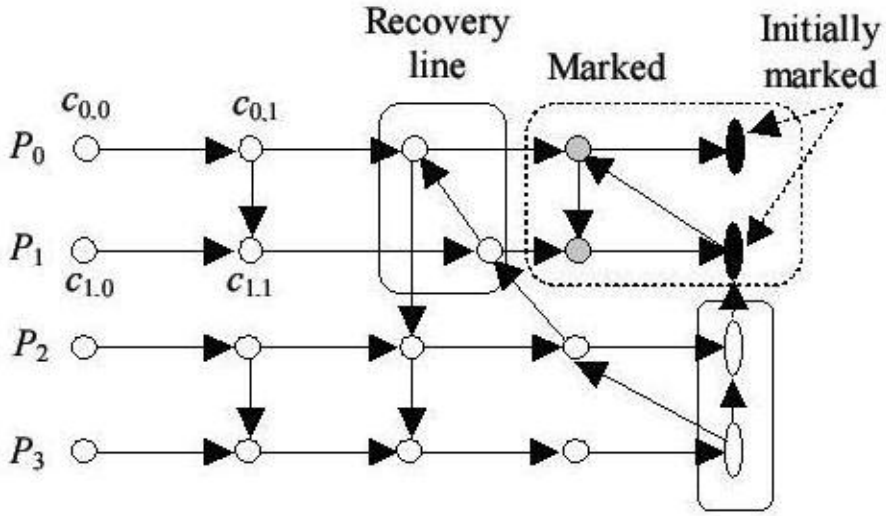
(a) Example execution

Rollback dependency graph

Directed edge drawn from $c_{i,x}$ to $c_{j,y}$ if either

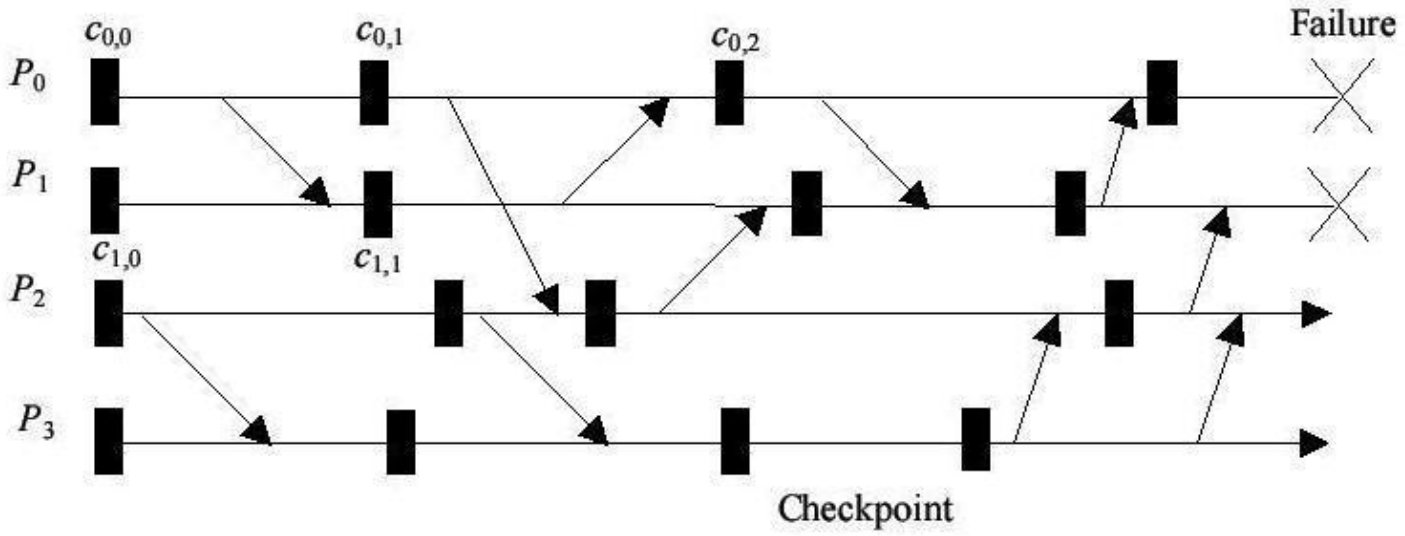
(1) $i \neq j$, and a message m is sent from $I_{i,x}$ and received in $I_{j,y}$, or

(2) $i = j$ and $y = x + 1$



(b) rollback-dependency graph

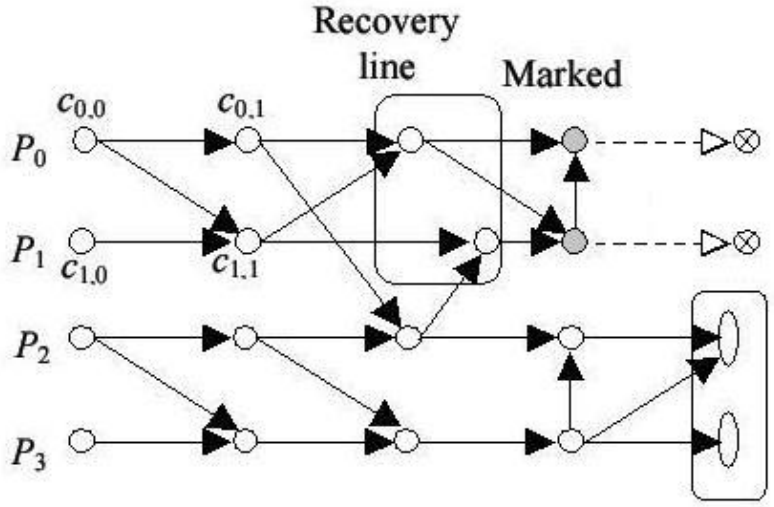
1. Uncoordinated checkpointing



(a) Example execution

Checkpoint graph

- Similar to dependency graph except directed edge drawn from $c_{i,x-1}$ to $c_{j,y}$
- Produces same recovery line



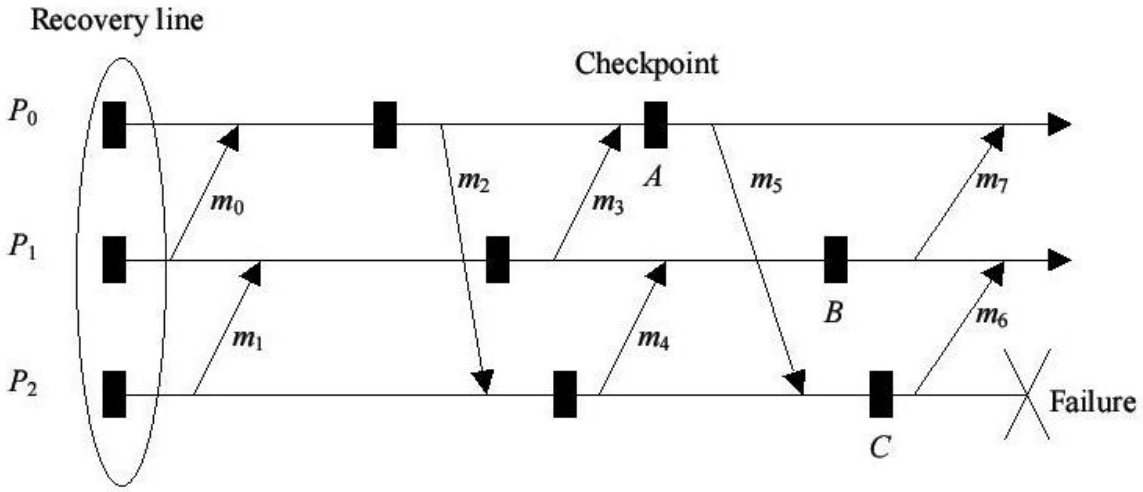
(c) checkpoint graph.

1. Uncoordinated checkpointing

Advantage: each process may take checkpoint when most convenient

Disadvantages

- Domino effect

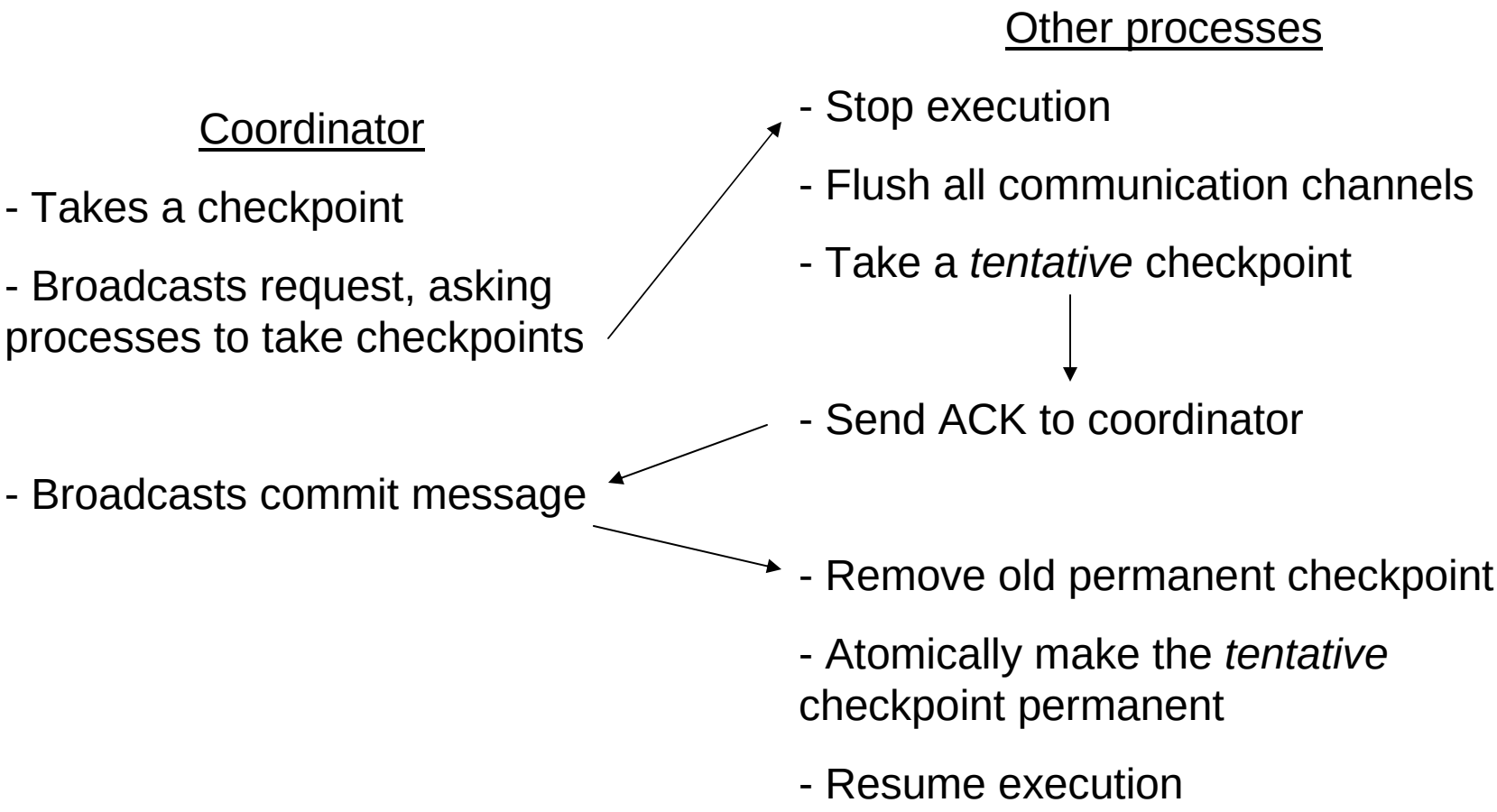


- Useless checkpoints
- Each process must maintain multiple checkpoints
- Garbage collector must be invoked periodically
- Not suitable for apps with frequent output commits

2. Coordinated checkpointing

Straightforward approach:

Block communications while checkpointing executes



Problem: messages that could make a checkpoint inconsistent are also blocked

2. Coordinated checkpointing

Non-blocking Checkpoint Coordination

- FIFO channels:
  - Precede 1st post-checkpoint message on each channel by a checkpoint request
  - Force each process to take a checkpoint upon receiving 1st checkpoint request

Non-FIFO channels:

- Checkpoint request piggybacked on every post-checkpoint message

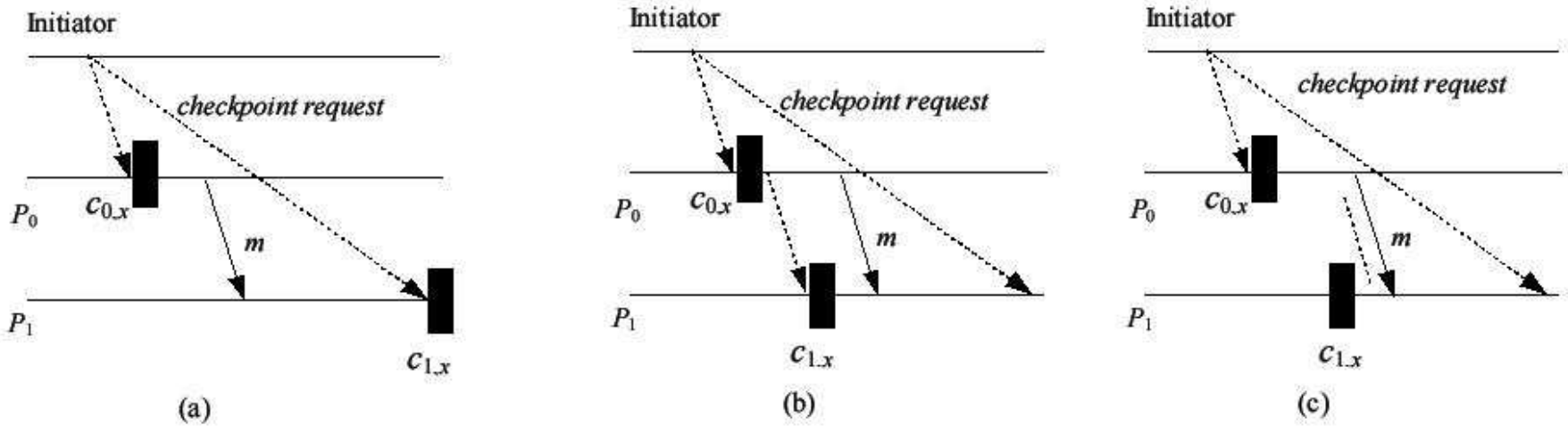


Figure 8. Non-blocking coordinated checkpointing: (a) checkpoint inconsistency; (b) with FIFO channels; (c) non-FIFO channels (short dashed line represents piggybacked *checkpoint request*).

2. Coordinated checkpointing

Advantages

- Simplified recovery

- Avoids domino effect

- Less stable storage requirement (1 checkpoint / process)

- No need for garbage collection

Disadvantages

- Large latency in committing output (may improve by minimal checkpoint coordination)

3. Communication-induced checkpointing

- Hybrid approach

- Processes take 2 kinds of checkpoints
  - *Local* : taken independently
  - *Forced* : must be taken to guarantee progress of recovery line, and prevent useless checkpoints

- No special coordination messages for forced checkpoints

- Protocol-specific info piggybacked on application message, for receiver to determine if it should take a forced checkpoint

3. Communication-induced checkpointing

How to detect possibly useless check points?

Z-path and Z-cycles

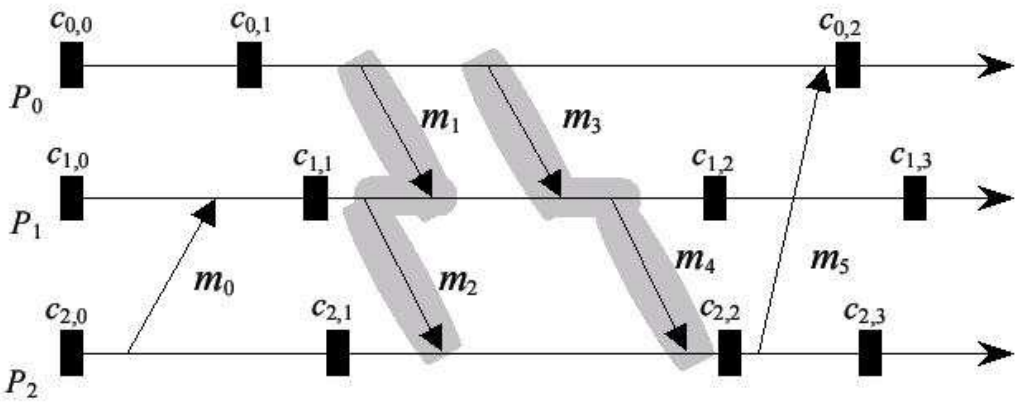


Figure 9. Z-paths and Z cycles.

A Z-path exists between $c_{i,x}$ and $c_{j,y}$ iff either:

1. $x < y$ and $i = j$; or
2. There exists a sequence of messages $[m_0, m_1, \ldots, m_n]$, $n \geq 0$, such that
   - $c_{i,x} \mapsto send_i(m_0)$;
   - $\forall\ l < n$, either $deliver_k(m_l)$ and $send_k(m_{l+1})$ are in the same checkpoint interval, or $deliver_k(m_l) \mapsto send_k(m_{l+1})$; and
   - $deliver_j(m_n) \mapsto c_{j,y}$

3. Communication-induced checkpointing

Type 1: *Model-based* Protocols

- Prevent patterns of communications and checkpoints that could result in Z-cycles

- Use heuristics to detect potential forming of patterns

- Insert forced checkpoints to prevent unfavorable patterns

- May independently force checkpoints → conservative, no coordination

- E.g. take checkpoint before every *receive* that is not separated from its previous *send* by a checkpoint; or take checkpoint before every *send*

3. Communication-induced checkpointing

Type 2: *Index-based* Protocols

- Guarantee: (1) if $c_{i,m} \mapsto c_{j,n}$, then $ts(c_{j,n}) \geq ts(c_{i,m})$
  (2) consecutive local checkpoints of a process have increasing timestamps
- Checkpoints with same index at different process form consistent state
- Equivalent to *model-based*; may have fewer forced checkpoints
- E.g. take checkpoint upon receiving a message with index greater than local index
- E.g. if checkpoints' timestamps always increase along a Z-path, no Z-cycle can form

3. Communication-induced checkpointing

Potential advantages:

- Avoids domino effect

- Processes can take local checkpoints when convenient

- Scalability: processes don’t have to participate in a globally coordinated checkpoint

Checkpointing protocols in practice

- Initially, communication overhead was greater than stable storage access overhead → uncoordinated > coordinated

- Modern system: overhead of coordinating checkpoints negligible compared to that of saving states → overhead of coordinated ≈ uncoordinated

- Problems with uncoordinated checkpointing:
    - finding recovery line is complex
    - domino effect
    - storage overhead of saving multiple checkpoints / process
    - garbage collection overhead

- CIC: study shows poor scalability
    - Forced checkpoints at random → hard to predict amount of stable storage → affects local checkpoints placing policy
    - Autonomy of local checkpoints may not hold (too many forced checkpoints)

Implementation issues

- Checkpointing
- Communication
- Stable storage

Implementation issues

Checkpointing implementation

- Suspending is bad

- Concurrent: use memory protection hardware to continue execution while checkpoint is saved; if page fault → copy to buffer

- Incremental: avoids rewriting portions of process states that don't change
    - using dirty bits, parity, direct comparison, signature

Implementation issues

Communication protocols

Mask processes' or remote ports' identity and location

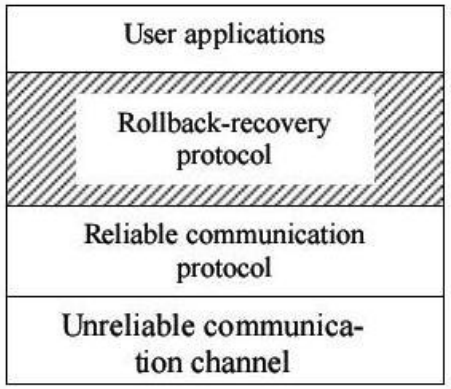
Reliable channel:

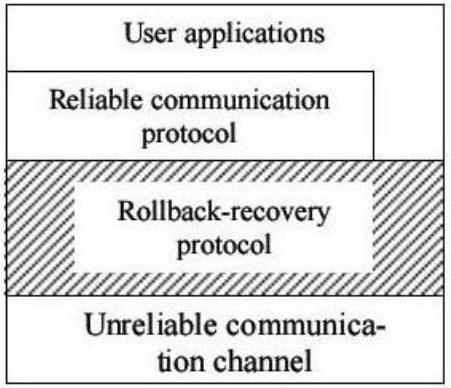
- Mask the timeout from the application at sender, make in-transit messages available to receiver after recovers

- Reestablish connection, clean up old connection, identify (+suppress) retransmitted messages

- Recover "lost" messages: save on sender

Unreliable channel: in-transit messages lost due to process failures appear same as lost due to communication failures





Implementation issues

Stable storage

Outside world process: cannot fail + recover

- before sending; after receiving

Only an abstraction, implementations differ in what failures can tolerate

- Single failure: volatile memory of another process
- Transient failures: local disk in each host
- Non-transient failures: persistent medium outside host

Stable storage implementation should access disk directly, not on top of FS (due to FS & network implementations)

Garbage collection

- Useless info can be dropped along the way (before *recovery line*)
- Consider: how much to keep, how often to call GC