Checkpointing & Rollback Recovery

Chapter 13

Anh Huy Bui
Jason Wiggs
Hyun Seok Roh

Introduction

- Rollback recovery protocols
  - restore the system back to a consistent state after a failure
  - achieve fault tolerance by periodically saving the state of a process during the failure-free execution
  - treats a distributed system application as a collection of processes that communicate over a network
- Checkpoints
  - the saved states of a process
- Why is rollback recovery of distributed systems complicated?
  - messages induce inter-process dependencies during failure-free operation
- Rollback propagation
  - the dependencies may force some of the processes that did not fail to roll back
  - This phenomenon is called "*domino effect*"

Introduction

- If each process takes its checkpoints independently, then the system can not avoid the domino effect
  - this scheme is called independent or uncoordinated checkpointing
- Techniques that avoid domino effect
  - Coordinated checkpointing rollback recovery
    - processes coordinate their checkpoints to form a system-wide consistent state
  - Communication-induced checkpointing rollback recovery
    - forces each process to take checkpoints based on information piggybacked on the application
  - Log-based rollback recovery
    - combines checkpointing with logging of non-deterministic events
    - relies on piecewise deterministic (PWD) assumption

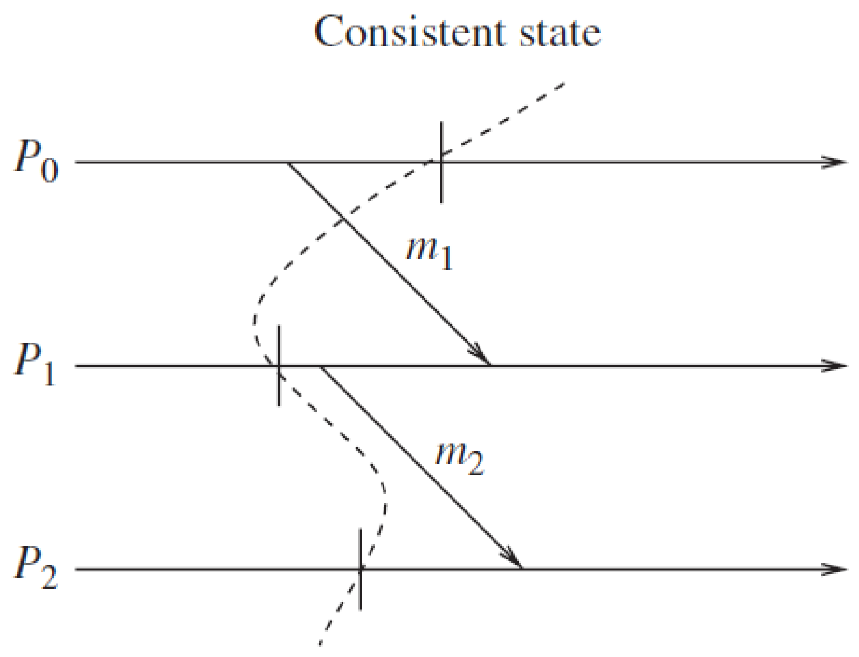
A local checkpoint

- All processes save their local states at certain instants of time
- A local check point is a snapshot of the state of the process at a given instance
- Assumption
  - A process stores all local checkpoints on the stable storage
  - A process is able to roll back to any of its existing local checkpoints
- $C_{i,k}$
  - The *k*th local checkpoint at process P_i
- $C_{i,0}$
  - A process P_i takes a checkpoint $C_{i,0}$ before it starts execution

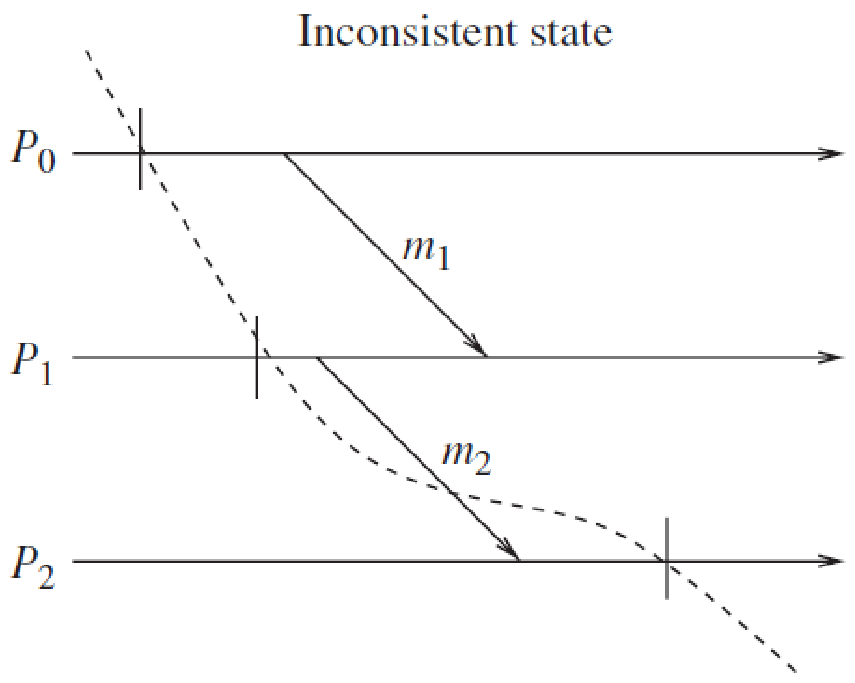
Consistent states

- A global state of a distributed system
  - a collection of the individual states of all participating processes and the states of the communication channels
- Consistent global state
  - a global state that may occur during a failure-free execution of distribution of distributed computation
  - if a process's state reflects a message receipt, then the state of the corresponding sender must reflect the sending of the message
- A global checkpoint
  - a set of local checkpoints, one from each process
- A consistent global checkpoint
  - a global checkpoint such that no message is sent by a process after taking its local point that is received by another process before taking its checkpoint

Consistent states - examples



(a)



(b)

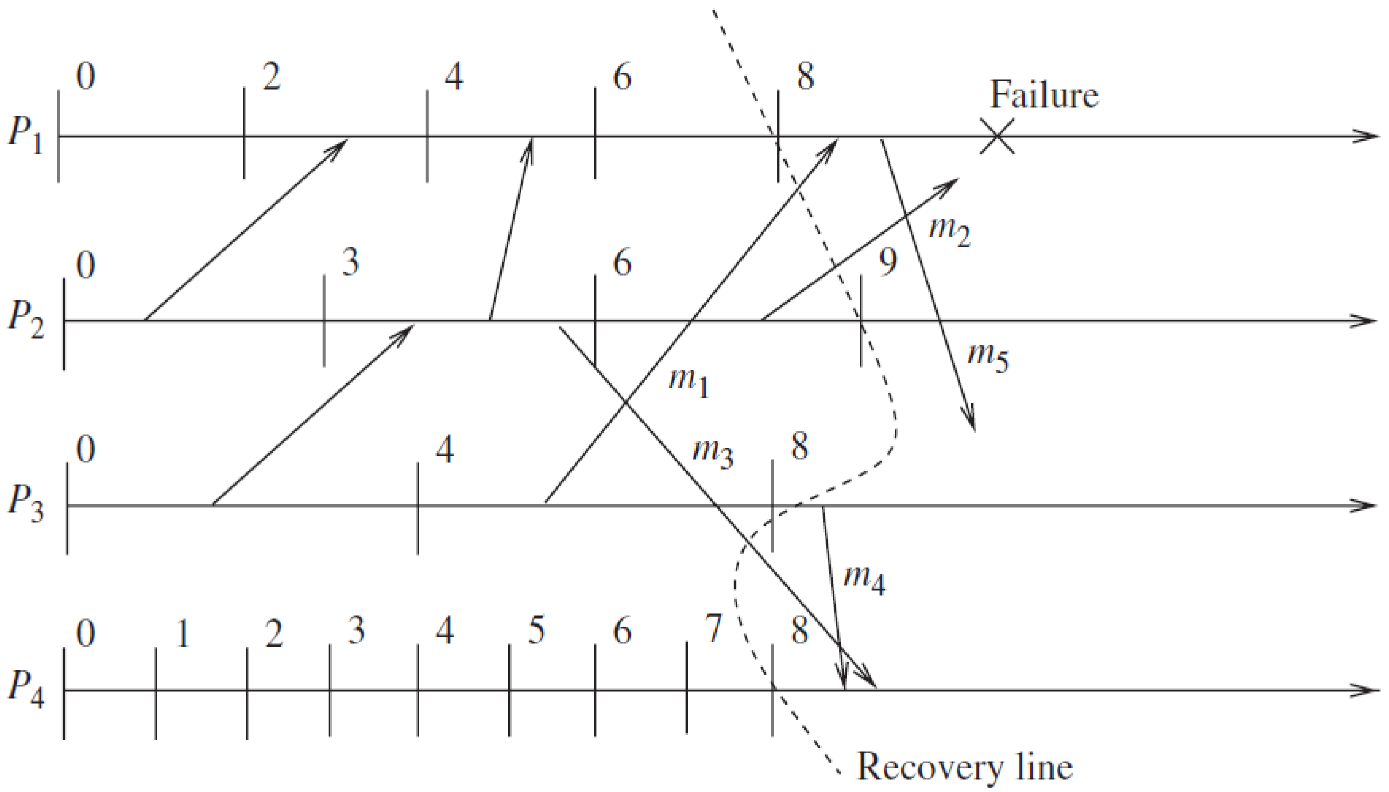
Interactions with outside world

- A distributed system often interacts with the outside world to receive input data or deliver the outcome of a computation
- Outside World Process (OWP)
  - a special process that interacts with the rest of the system through message passing
- A common approach
  - save each input message on the stable storage before allowing the application program to process it
- Symbol "||"
  - An interaction with the outside world to deliver the outcome of a computation

Messages

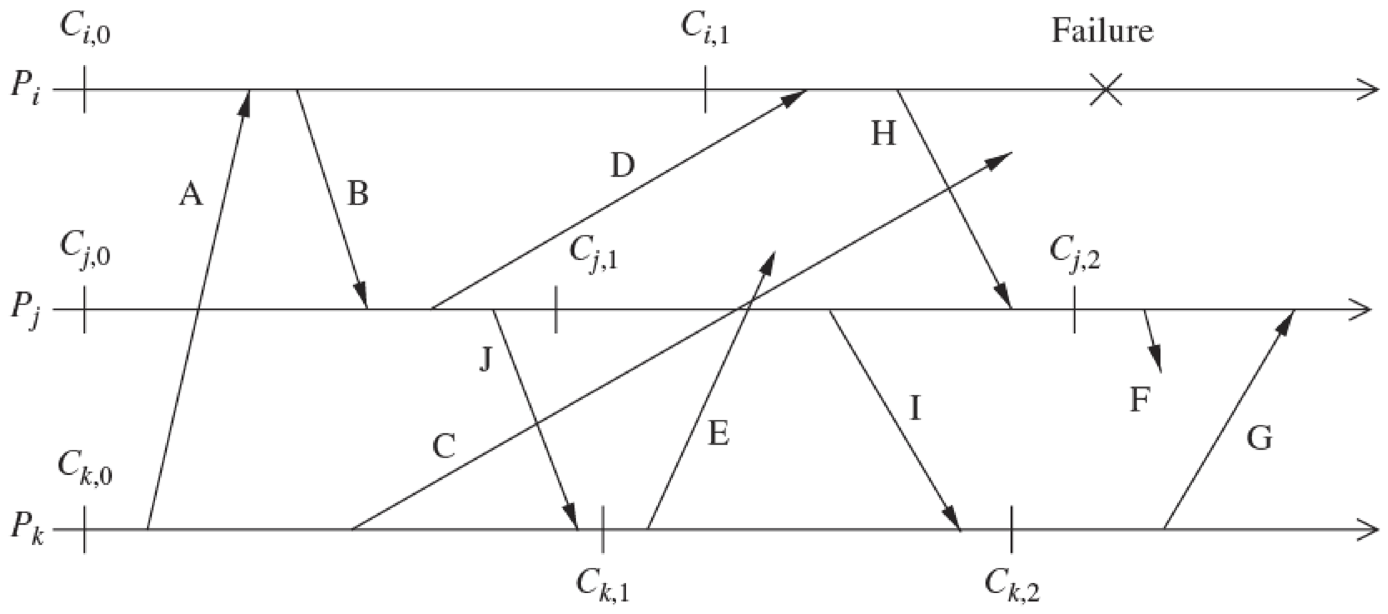
- In-transit message
  - messages that have been sent but not yet received
- Lost messages
  - messages whose ‘send’ is done but ‘receive’ is undone due to rollback
- Delayed messages
  - messages whose ‘receive’ is not recorded because the receiving process was either down or the message arrived after rollback
- Orphan messages
  - messages with ‘receive’ recorded but message ‘send’ not recorded
  - do not arise if processes roll back to a consistent global state
- Duplicate messages
  - arise due to message logging and replaying during process recovery

Messages – example



- In-transit
  - m_1, m_2
- Lost
  - m_1
- Delayed
  - m_1, m_5
- Orphan
  - none
- Duplicated
  - m_4, m_5

Issues in failure recovery



- Checkpoints : $\{C_{i,0}, C_{i,1}\}$, $\{C_{j,0}, C_{j,1}, C_{j,2}\}$, and $\{C_{k,0}, C_{k,1}, C_{k,2}\}$
- Messages : A - J
- The restored global consistent state : $\{C_{i,1}, C_{j,1}, C_{k,1}\}$

Issues in failure recovery

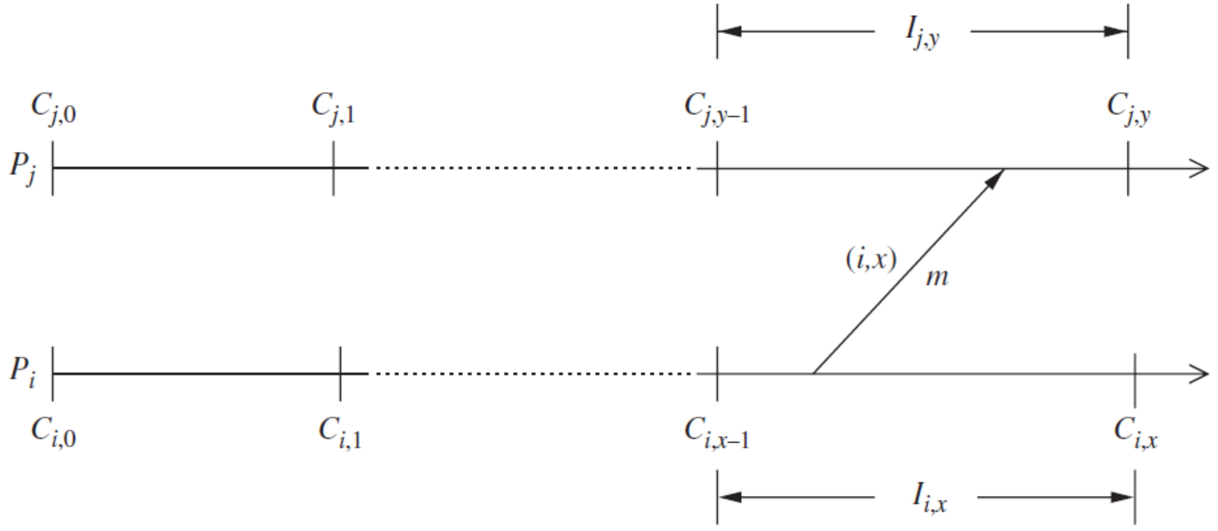
- The rollback of process P_i to checkpoint $C_{i,1}$ created an orphan message H
- Orphan message I is created due to the roll back of process P_j to checkpoint $C_{j,1}$
- Messages C, D, E, and F are potentially problematic
  - Message C: a delayed message
  - Message D: a lost message since the send event for D is recorded in the restored state for P_j, but the receive event has been undone at process P_i.
  - Lost messages can be handled by having processes keep a message log of all the sent messages
  - Messages E, F: delayed orphan messages. After resuming execution from their checkpoints, processes will generate both of these messages

Uncoordinated Checkpointing

- Each process has autonomy in deciding when to take checkpoints
- Advantages
  - The lower runtime overhead during normal execution
- Disadvantages
  - Domino effect during a recovery
  - Recovery from a failure is slow because processes need to iterate to find a consistent set of checkpoints
  - Each process maintains multiple checkpoints and periodically invoke a garbage collection algorithm
  - Not suitable for application with frequent output commits
- The processes record the dependencies among their checkpoints caused by message exchange during failure-free operation

Direct dependency tracking technique

- Assume each process P_i starts its execution with an initial checkpoint $C_{i,0}$
- $I_{i,x}$: checkpoint interval, interval between $C_{i,x-1}$ and $C_{i,x}$
- When P_j receives a message m during $I_{j,y}$, it records the dependency from $I_{i,x}$ to $I_{j,y}$, which is later saved onto stable storage when P_j takes $C_{j,y}$



Coordinated Checkpointing

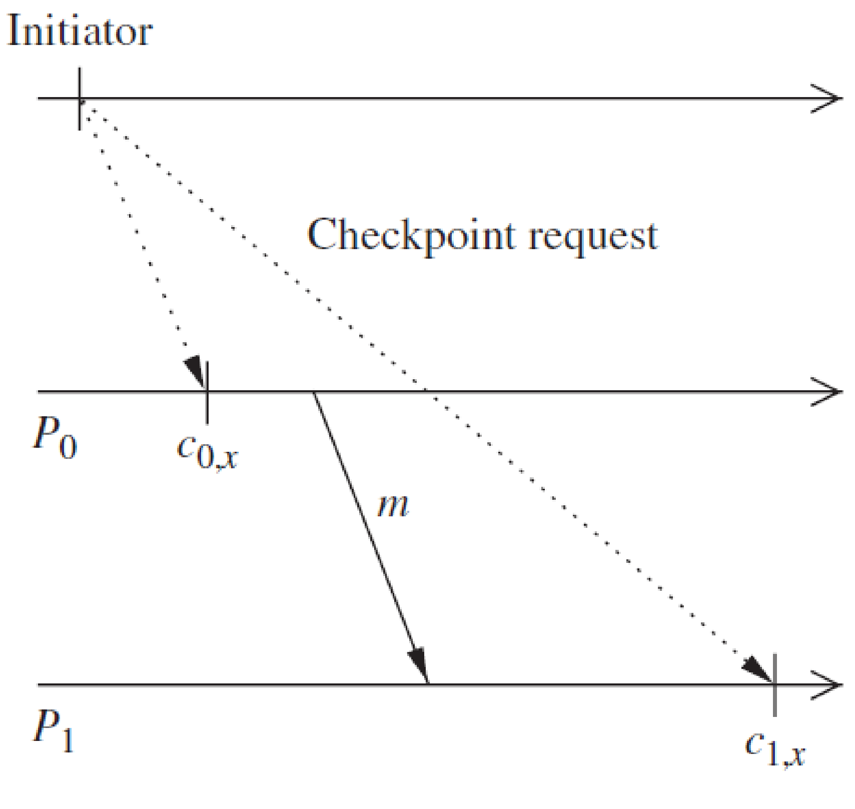
- Blocking Checkpointing
  - After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpointing activity is complete
  - Disadvantages
    - the computation is blocked during the checkpointing

- Non-blocking Checkpointing
  - The processes need not stop their execution while taking checkpoints
  - A fundamental problem in coordinated checkpointing is to prevent a process from receiving application messages that could make the checkpoint inconsistent.

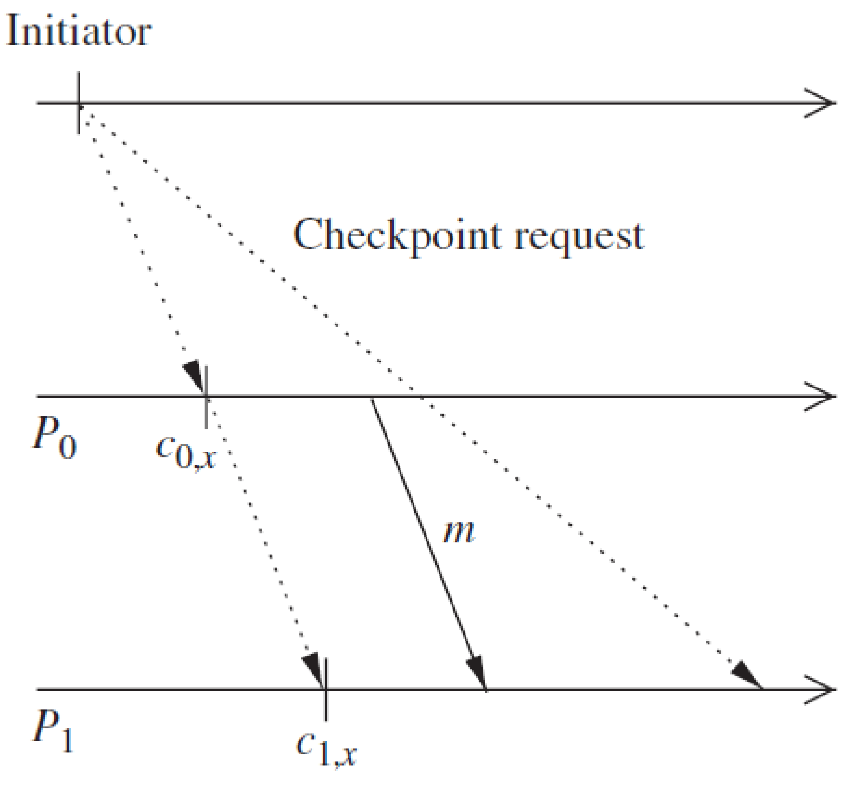
Coordinated Checkpointing

- Example (a) : checkpoint inconsistency
  - message m is sent by P_0 after receiving a checkpoint request from the checkpoint coordinator
  - Assume m reaches P_1 before the checkpoint request
  - This situation results in an inconsistent checkpoint since checkpoint $C_{1,x}$ shows the receipt of message m from P_0, while checkpoint $C_{0,x}$ does not show m being sent from P_0
- Example (b) : a solution with FIFO channels
  - If channels are FIFO, this problem can be avoided by preceding the first post-checkpoint message on each channel by a checkpoint request, forcing each process to take a checkpoint before receiving the first post-checkpoint message

Coordinated Checkpointing



(a)



(b)

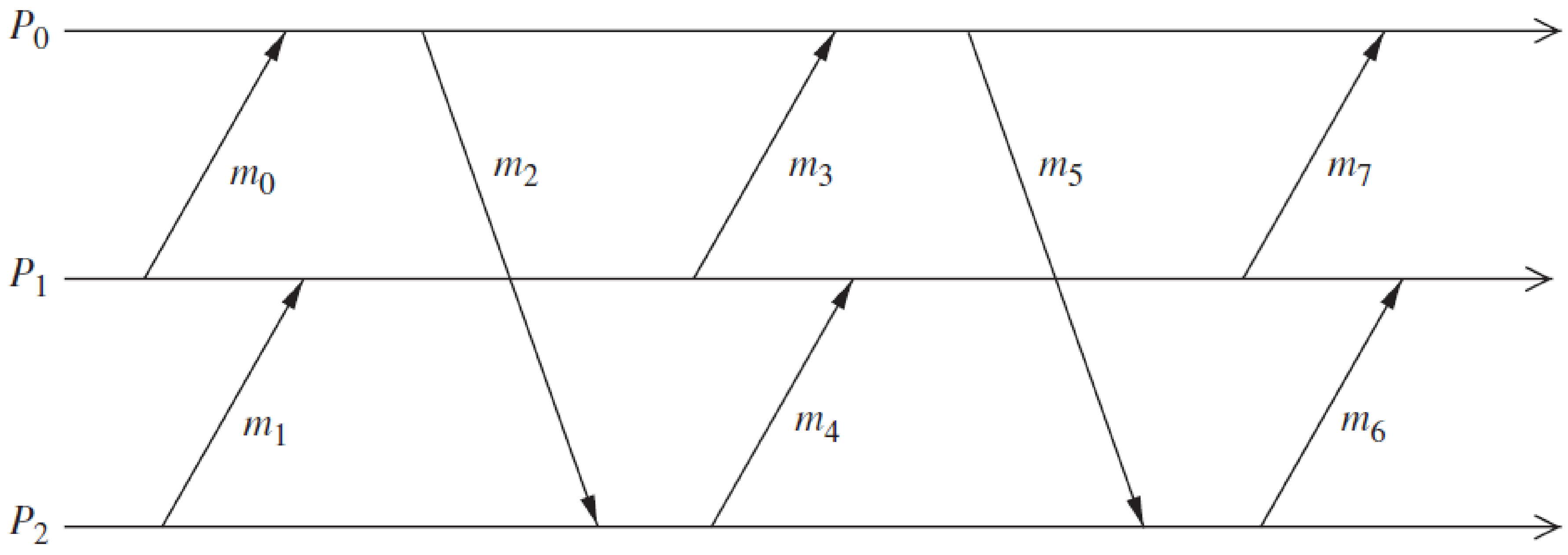
Communication-induced Checkpointing

- Two types of checkpoints
  - autonomous and forced checkpoints
- Communication-induced checkpointing piggybacks protocol-related information on each application message
- The receiver of each application message uses the piggybacked information to determine if it has to take a forced checkpoint to advance the global recovery line
- The forced checkpoint must be taken before the application may process the contents of the message
- In contrast with coordinated checkpointing, no special coordination messages are exchanged
- Two types of communication-induced checkpointing
  - model-based checkpointing and index-based checkpointing.

Log-based Rollback Recovery

- A log-based rollback recovery makes use of deterministic and nondeterministic events in a computation.
- Deterministic and Non-deterministic events
  - Non-deterministic events can be the receipt of a message from another process or an event internal to the process
  - a message send event is *not* a non-deterministic event.
  - the execution of process P_0 is a sequence of four deterministic intervals
  - Log-based rollback recovery assumes that all non-deterministic events can be identified and their corresponding determinants can be logged into the stable storage
  - During failure-free operation, each process logs the determinants of all non-deterministic events that it observes onto the stable storage

Log-based Rollback Recovery



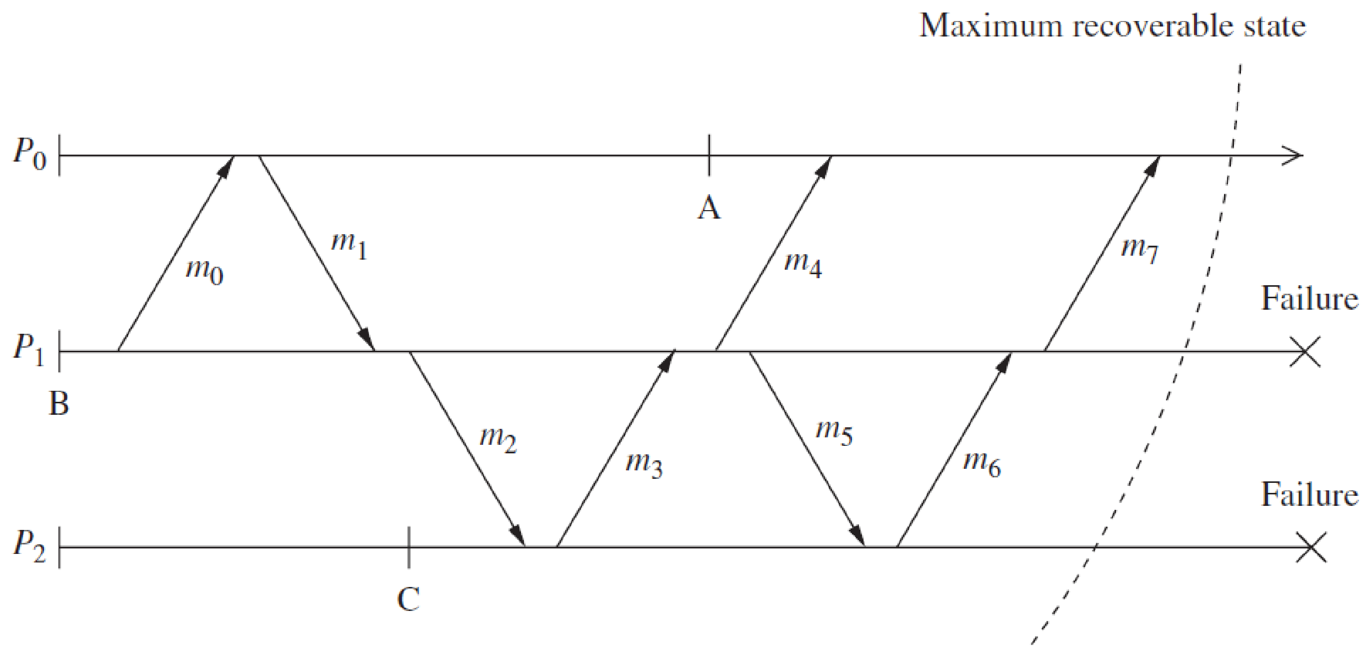
No-orphans consistency condition

- Let *e* be a non-deterministic event that occurs at process *p*
- *Depend*(*e*)
  - the set of processes that are affected by a non-deterministic event *e*. This set consists of *p*, and any process whose state depends on the event *e* according to Lamport's *happened before* relation
- *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
  - $\forall(e) : \neg \mathit{Stable}(e) \Rightarrow \mathit{Depend}(e) \subseteq \mathit{Log}(e)$

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
- *synchronous logging*
  - $\forall e: \neg \text{Stable}(e) \Rightarrow |\text{Depend}(e)| = 0$
  - if an event has not been logged on the stable storage, then no process ca n depend on it.
  - stronger than the always-no-orphans condition

Pessimistic Logging

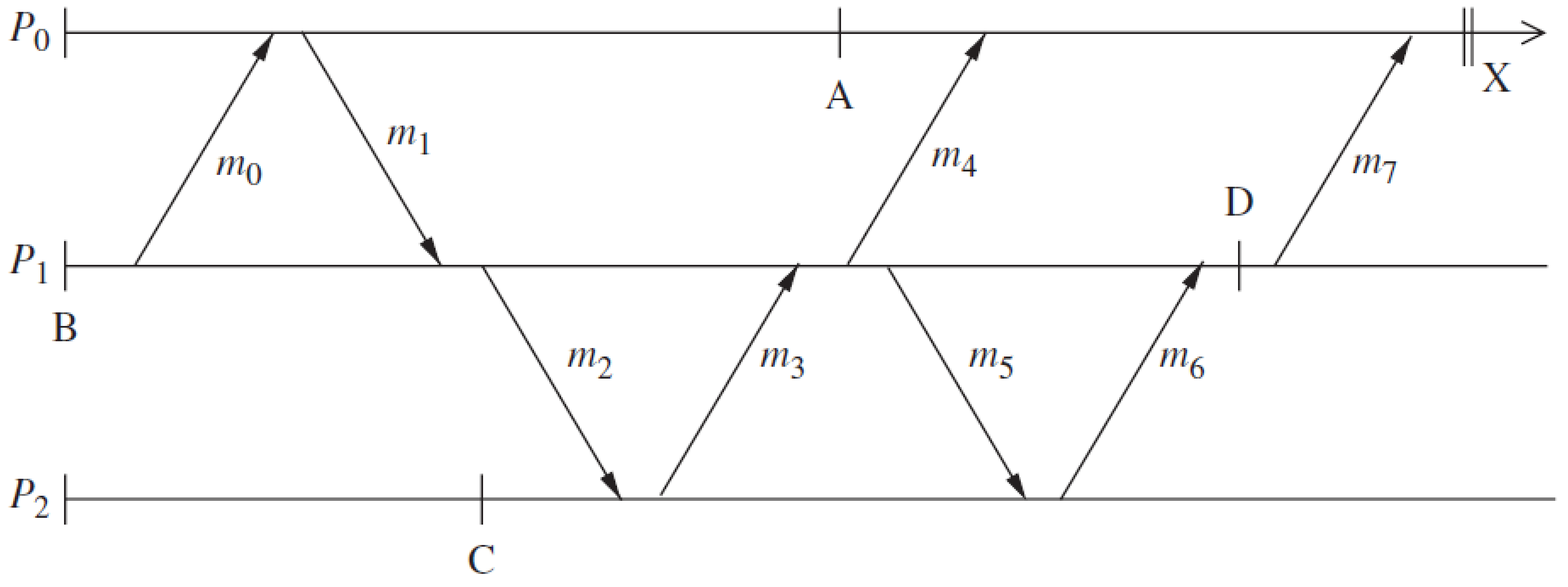


- Suppose processes P_1 and P_2 fail as shown, restart from checkpoints B and C, and roll forward using their determinant logs to deliver again the same sequence of messages as in the pre-failure execution
- Once the recovery is complete, both processes will be consistent with the state of P_0 that includes the receipt of message m_7 from P_1

Optimistic Logging

- Processes log determinants asynchronously to the stable storage
- Optimistically assume that logging will be complete before a fail ure 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 collect ion 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

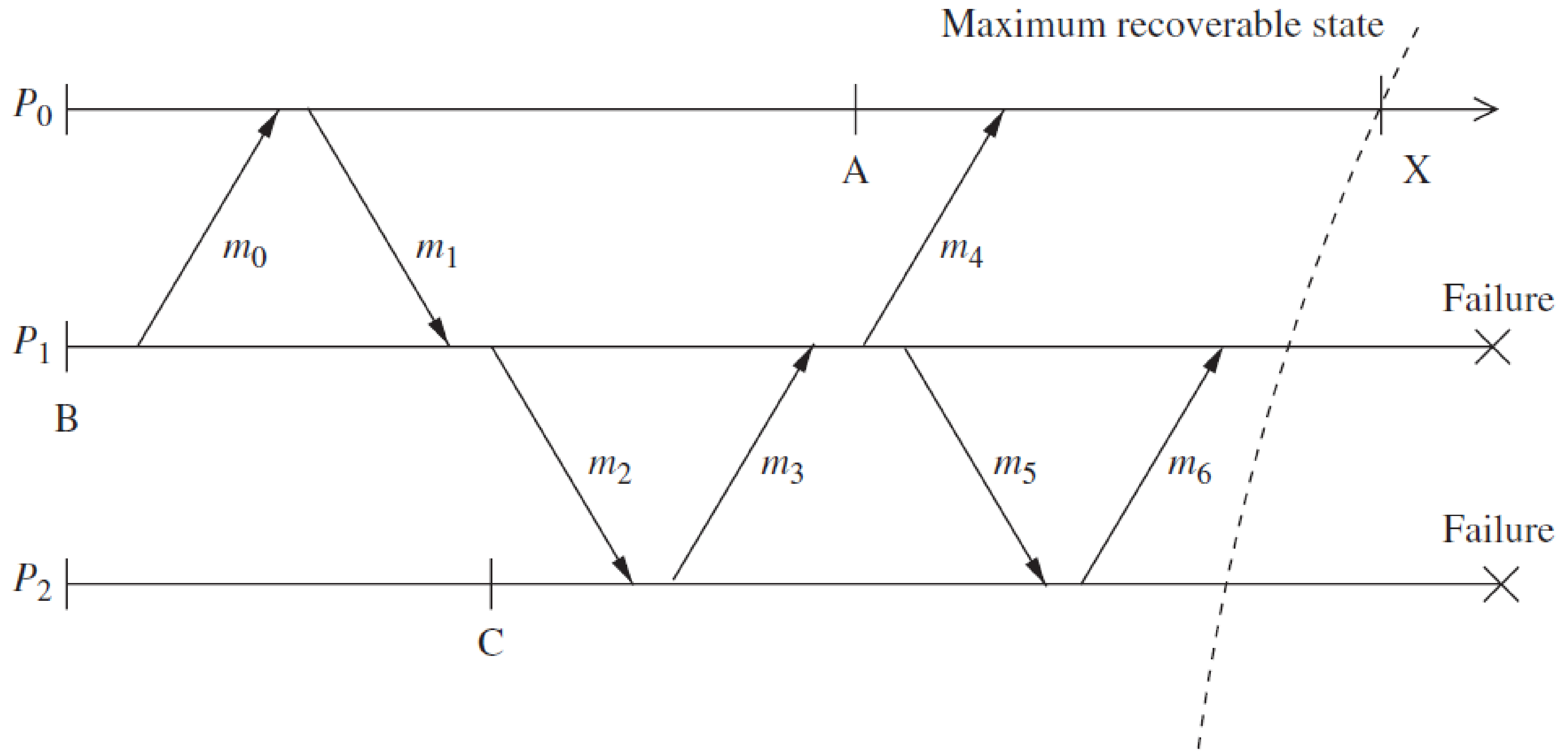
Optimistic Logging



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

Causal Logging



Koo-Toueg coordinated checkpointing algorithm

- A coordinated checkpointing and recovery technique that takes a consistent set of checkpointing and avoids domino effect and livelock problems during the recovery

- Includes 2 parts: the checkpointing algorithm and the recovery algorithm

Koo-Toueg coordinated checkpointing algorithm(cont.)

- Checkpointing algorithm
  - Assumptions: FIFO channel, end-to-end protocols, communication failures do not partition the network, single process initiation, no process fails during the execution of the algorithm
  - Two kinds of checkpoints: permanent and tentative
    - Permanent checkpoint: local checkpoint, part of a consistent global checkpoint
    - Tentative checkpoint: temporary checkpoint, become permanent checkpoint when the algorithm terminates successfully

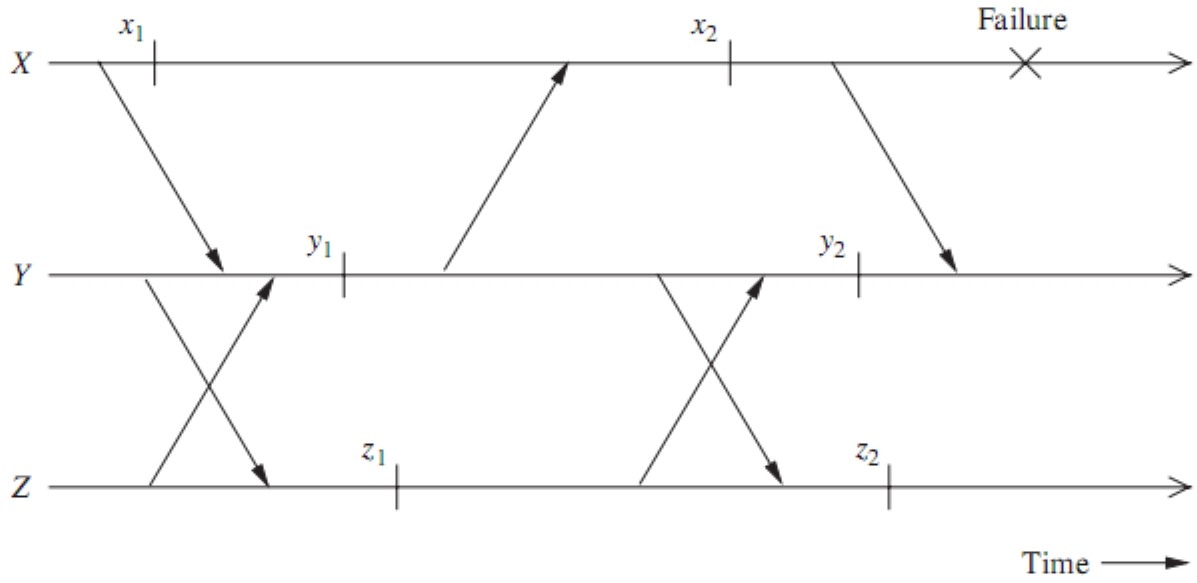
Koo-Toueg coordinated checkpointing algorithm(cont.)

- Checkpointing algorithm
  - 2 phases
    - The initiating process takes a tentative checkpoint and requests all other processes to take tentative checkpoints. Every process can not send messages after taking tentative checkpoint. All processes will finally have the single same decision: do or discard
    - All processes will receive the final decision from initiating process and act accordingly
  - Correctness: for 2 reasons
    - Either all or none of the processes take permanent checkpoint
    - No process sends message after taking permanent checkpoint
  - Optimization: maybe not all of the processes need to take checkpoints (if not change since the last checkpoint)

Koo-Toueg coordinated checkpointing algorithm(cont.)

- The rollback recovery algorithm
  - Restore the system state to a consistent state after a failure with assumptions: single initiator, checkpoint and rollback recovery algorithms are not invoked concurrently
  - 2 phases
    - The initiating process send a message to all other processes and ask for the preferences – restarting to the previous checkpoints. All need to agree about either do or not.
    - The initiating process send the final decision to all processes, all the processes act accordingly after receiving the final decision.

Koo-Toueg coordinated checkpointing algorithm(cont.)



- Correctness: resume from a consistent state
- Optimization: may not to recover all, since some of the processes did not change anything

Juang-Venkatesan algorithm for asynchronous checkpointing and recovery

- Assumptions: communication channels are reliable, delivery messages in FIFO order, infinite buffers, message transmission delay is arbitrary but finite
- Underlying computation/application is event-driven: process P is at state s, receives message m, processes the message, moves to state s' and send messages out. So the triplet (*s, m, msgs_sent*) represents the state of P
- Two type of log storage are maintained:
  - Volatile log: short time to access but lost if processor crash. Move to stable log periodically.
  - Stable log: longer time to access but remained if crashed

Juang-Venkatesan algorithm for asynchronous checkpointing and recovery(cont.)

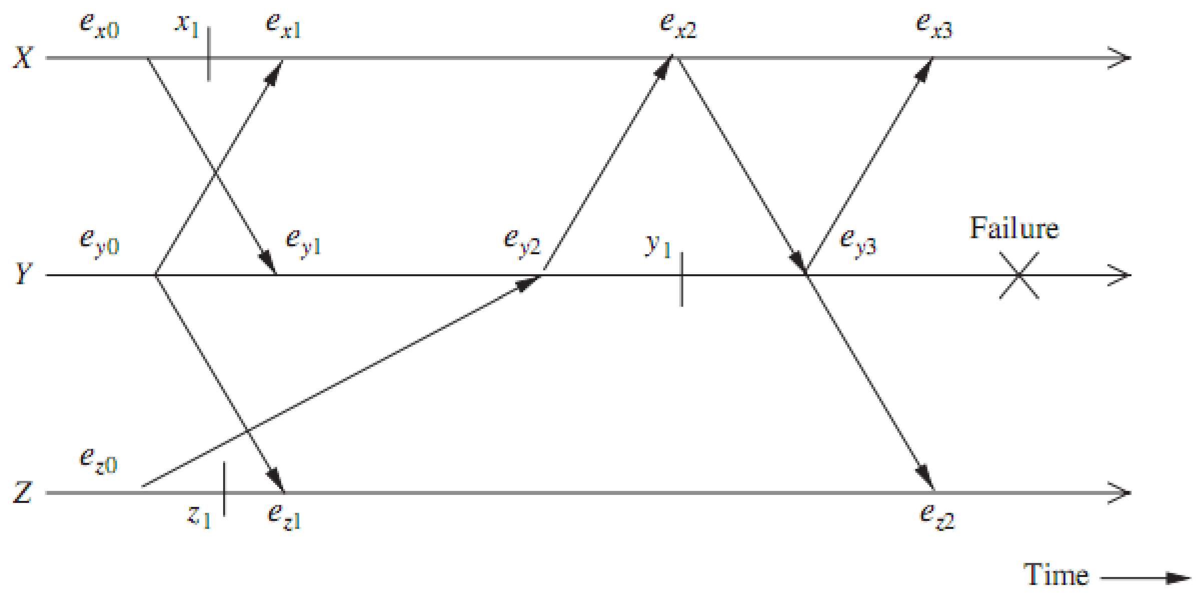
- Asynchronous checkpointing:
  - After executing an event, the triplet is recorded without any synchronization with other processes.
  - Local checkpoint consist of set of records, first are stored in volatile log, then moved to stable log.
- Recovery algorithm
  - Notations:
    - $RCVD_{i \leftarrow j}(CkPt_i)$: number of messages received by p_i from p_j, from the beginning of computation to checkpoint $CkPt_i$
    - $SENT_{i \rightarrow j}(CkPt_i)$: number of messages sent by p_i to p_j, from the beginning of computation to checkpoint $CkPt_i$
  - Idea:
    - From the set of checkpoints, find a set of consistent checkpoints
    - Doing that based on the number of messages sent and received

Juang-Venkatesan algorithm for asynchronous checkpointing and recovery(cont.)

```
**Procedure RollBack_Recovery**: processor $p_i$ executes the following:
  STEP (a)
  if processor $p_i$ is recovering after a failure then
    $CkPt_i$ := latest event logged in the stable storage
  else
    $CkPt_i$ := latest event that took place in $p_i$ {The latest event at $p_i$ can be
    either in stable or in volatile storage.}
  end if
  STEP (b)
  for $k = 1$ to $N$ {$N$ is the number of processors in the system} do
    for each neighboring processor $p_j$ do
      compute $SENT_{i \to j}(CkPt_i)$
      send a $ROLLBACK(i, SENT_{i \to j}(CkPt_i))$ message to $p_j$
    end for
    for every $ROLLBACK(j, c)$ message received from a neighbor $j$ do
      if $RCVD_{i \leftarrow j}(CkPt_i) > c$ {Implies the presence of orphan messages}
        then
        find the latest event $e$ such that $RCVD_{i \leftarrow j}(e) = c$ {Such an event $e$
        may be in the volatile storage or stable storage.}
        $CkPt_i := e$
      end if
    end for
  end for {for $k$}
```

Juang-Venkatesan algorithm for asynchronous checkpointing and recovery(cont.)

- Example

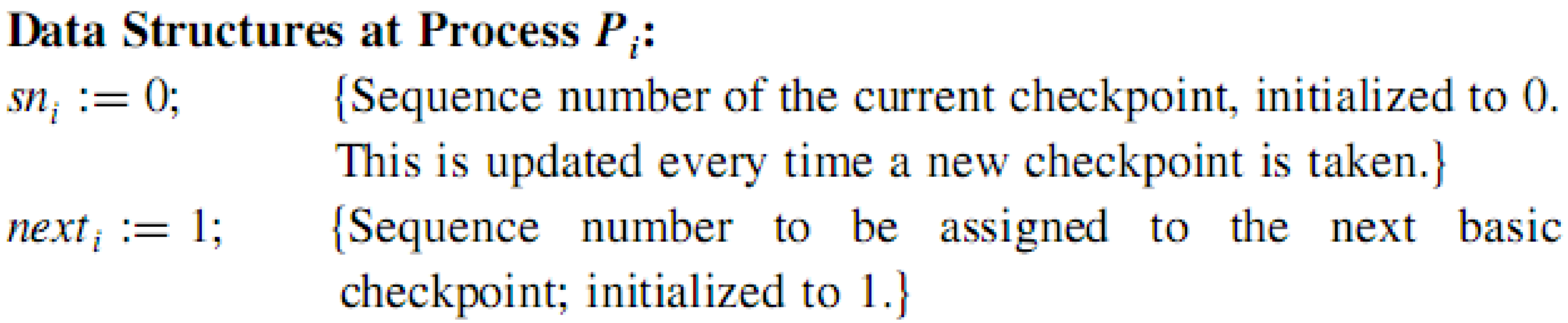


Manivannan-Singhal algorithm

- Observation: there are some checkpoints useless (i.e. never included in any consistent global checkpoint), even none of them are useful
- Combine the coordinated and uncoordinated checkpointing approaches
  - Take checkpoint asynchronously
  - Use communication-induced checkpointing to eliminates the useless checkpoint
  - Every checkpoint lies on a consistent checkpoint, determine the recovery line is easy and fast
- Idea
  - Each checkpoint of a process has a unique sequence number – local number, increased periodically
  - When a process send out a message, its sequence number is piggybacked
  - When a process received a message, if the received sequence number > its sequence number, it is forced to take checkpoint, and any basic checkpointing with smaller sequence number is skipped

Manivannan-Singhal – Checkpointing Alg. (1)

- Checkpointing algorithm
  - Checkpoints satisfy the following interesting properties
    - $C_{i,m}$ of P_i is concurrent with $C_{*,m}$ of all other processes
    - Checkpoints $C_{*,m}$ of all processes form a consistent global checkpoint
    - Checkpoint $C_{i,m}$ of P_i is concurrent with earliest checkpoint $C_{j,n}$ with $m \leq n$



Data Structures at Process P_i:

```
sn_i := 0;       {Sequence number of the current checkpoint, initialized to 0.
                  This is updated every time a new checkpoint is taken.}
next_i := 1;     {Sequence number to be assigned to the next basic
                  checkpoint; initialized to 1.}
```

Manivannan-Singhal – Checkpointing Alg. (2)

```
When it is time for process $P_i$ to increment $next_i$:
$next_i := next_i + 1$;        {$next_i$ is incremented at periodic time intervals of $X$
                                time units}

When process $P_i$ sends a message $M$:
$M.sn := sn_i$;        {sequence number of the current checkpoint is appended
                        to $M$}
send ($M$);
```

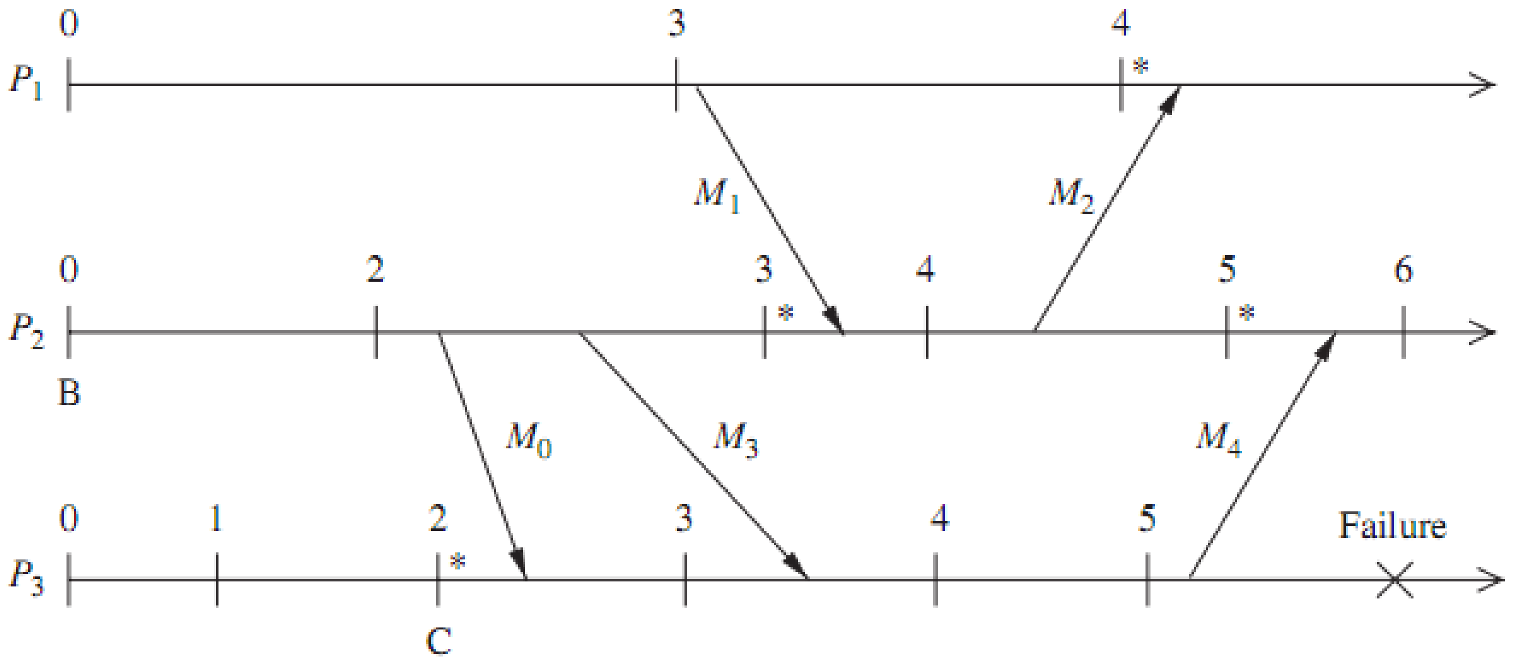
For a forced checkpoint

```
Process $P_j$ receives a message from process $P_i$:
if $sn_j < M.sn$ then             {if sequence number of the current checkpoint
    Take checkpoint $C$;           is less than checkpoint number received in the
    $C.sn := M.sn$;                  message, then take a new checkpoint before
    $sn_j := M.sn$;                              processing the message}
Process the message.
```

For a basic checkpoint

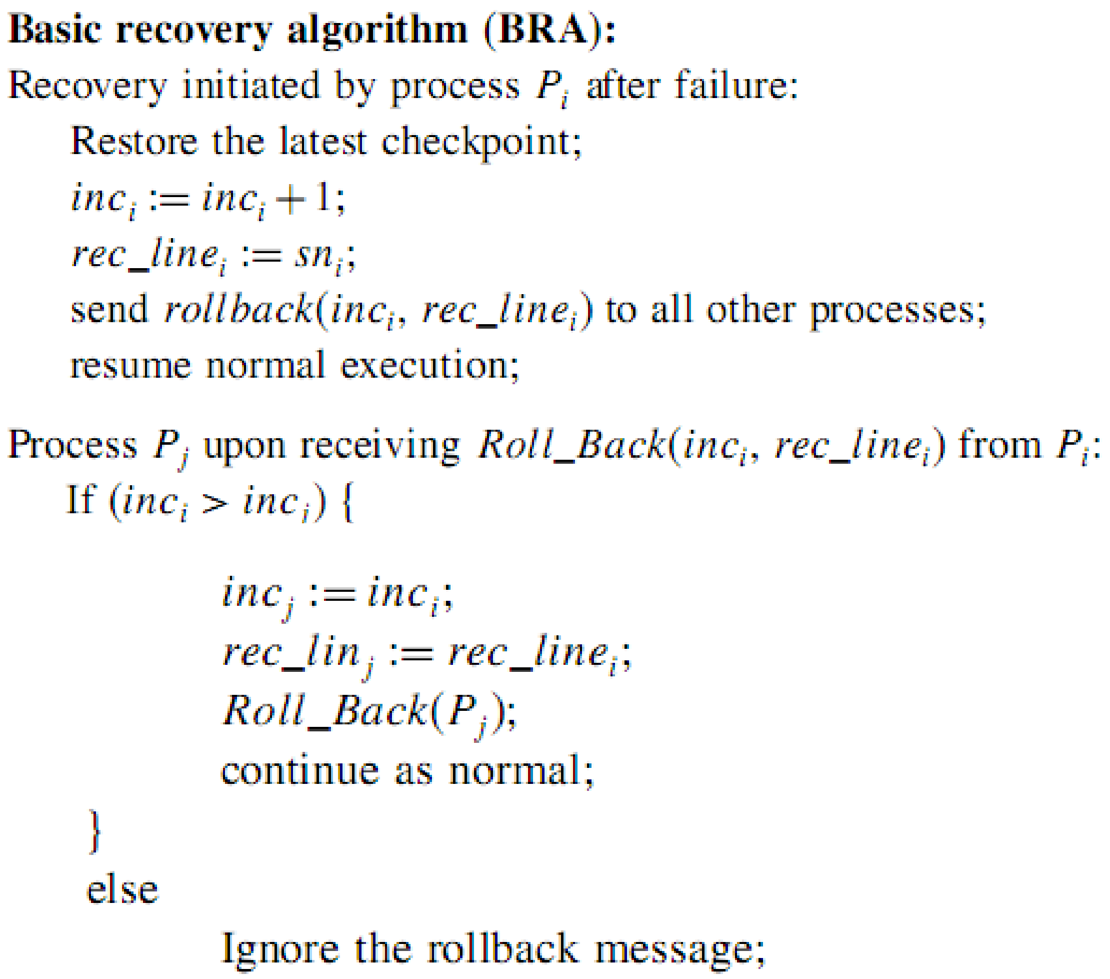
```
When it is time for process $P_i$ to take a basic checkpoint:
if $next_i > sn_i$ then           {skips taking a basic checkpoint if $next_i \leq sn_i$
    Take checkpoint $C$;            (i.e., if it already took a *forced* checkpoint
    $sn_i := next_i$;                   with sequence number $\geq next_i$)}
    $C.sn := sn_i$;
```

Manivannan-Singhal – Checkpointing Ex



- M_1 forces P_2 to take a forced checkpoint with sequence number 3 before processing M_1 because M_1.sn> sn2

Manivannan-Singhal – Recovery Alg. (1)



Basic recovery algorithm (BRA):

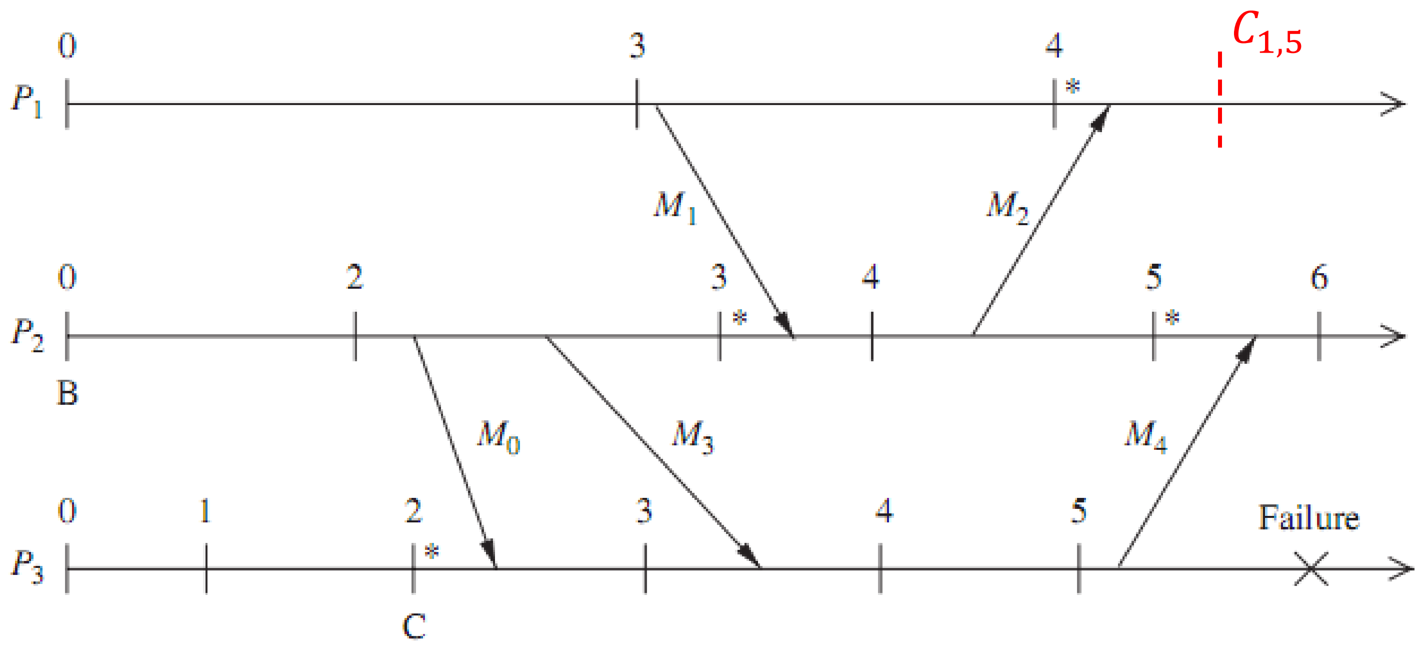
```
Recovery initiated by process $P_i$ after failure:
    Restore the latest checkpoint;
    $inc_i := inc_i + 1$;
    $rec\_line_i := sn_i$;
    send $rollback(inc_i, rec\_line_i)$ to all other processes;
    resume normal execution;

Process $P_j$ upon receiving $Roll\_Back(inc_i, rec\_line_i)$ from $P_i$:
    If $(inc_i > inc_j)$ {
            $inc_j := inc_i$;
            $rec\_lin_j := rec\_line_i$;
            $Roll\_Back(P_j)$;
            continue as normal;
     }
     else
            Ignore the rollback message;
```

Manivannan-Singhal – Recovery Alg. (2)

```
Procedure $Roll\_Back(P_j)$:
    If $(rec\_line_j > sn_j)$ {
        Take checkpoint $C$;
        $C.sn := rec\_line_j$;
        $sn_j := C.sn$;
    }
    else
    {
        Find the earliest checkpoint $C$ with $C.sn \geq rec\_line_j$;
        $sn_j := C.sn$;
        Restore checkpoint $C$;
        Delete all checkpoints after $C$;
    }
```

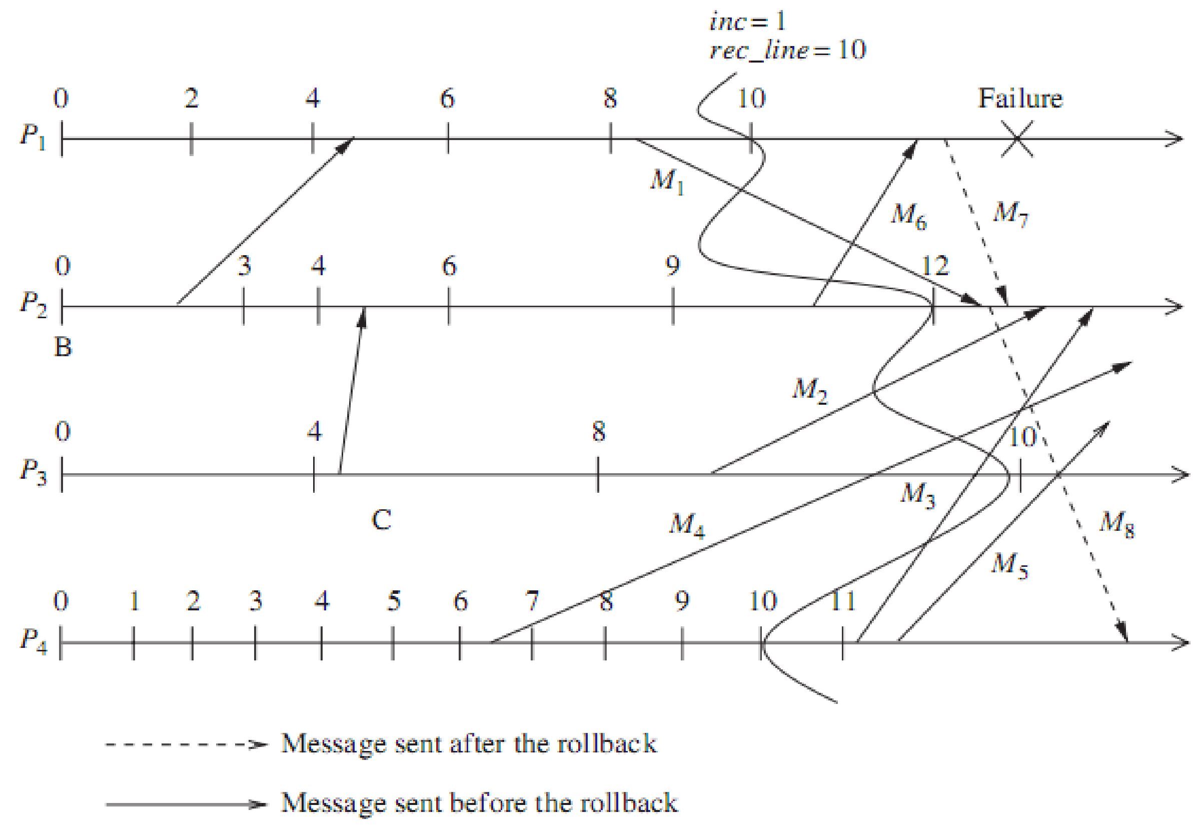
Manivannan-Singhal – Recovery Ex



- When P_3 recovers,
  - broadcast rollback(inc3, rec_lin3) where inc3 = 1 and rec_line3 = 5
  - P_2 rollback to $C_{2,5}$
  - P_1 does not have a checkpoint with sequence number ≥ 5. So it takes a local check point and assign 5 as its sequence number

Manivannan-Singhal quasi-synchronous checkpointing algorithm(cont.)

- Comprehensive handling messages during recovery
  - Handling the replay of messages
  - Handle of received messages



inc = 1
rec_line = 10
Failure
P1
P2
P3
P4
B
C
M1
M2
M3
M4
M5
M6
M7
M8
Message sent after the rollback
Message sent before the rollback

Peterson-Kearns algorithm – Definition (1)

- Based on optimistic recovery protocol
- Rollback based on vector time
- Ring configuration : each processor knows its successor on the ring
- Each process P_i has a vector clock $V_i[j]$, $0 \leq j \leq$ N-1
- $V_i(e_i)$: the clock value of an event e_i which occurred at P_i
- $V_i(p_i)$: the current vector clock time of P_i and e_i denotes the most recent event in P_i, thus $V_i(p_i) = V_i(e_i)$
- e_j^i : i th event on P_j
- s : A send event of the underlying computation
- $\sigma(s)$: The process where send event s occurs
- ρ(s) : The process where the receive event matched with send event s occurs
- f_j^i : The i th failure on P_j

Peterson-Kearns algorithm – Definition (2)

- ck_j^i : The *i* th state checkpoint on P_j. The check point resides on the stable stoarge
- rs_j^i : The i th restart event on P_j
- rb_j^i : The i th rollback event on P_j
- *LastEvent* (f_j^i) = e' iff $e' \mapsto rs_j^i$
- $C_{i,k}(m)$: The arrival of the final polling wave message for rollback from failure f_i^m at process P_k
- $w_{i,k}(m)$: The response to this final polling wave by P_k. If no response is required, $w_{i,k}(m)$ = $C_{i,k}(m)$
- The final polling wave for recovery from failure f_i^m:

$$PW_i(m) = \bigcup_{k=0}^{N-1} w_{i,k}(m) \ \ \bigcup_{k=0}^{N-1} C_{i,k}(m)$$

- *tk(i, m).ts* : the token with failure f_i^m and restart event rs_i^m
- *tk(i, m).inc* : incarnation number of in the token

Peterson-Kearns Alg. – Informal Description (1)

- Step 1
  - When a process P_i restarts after failure, it retrieves its latest checkpoint, including its vector time value, and roll back to it
- Step 2
  - The message log is replayed
- Step 3
  - The recovering process executes a restart event rs_i^m to begin the global rollback protocol
  - creates a token message containing the vector timestamp of the restart event
  - sends the token to its successor process

Peterson-Kearns Alg. – Informal Description (2)

- Step 4
  - The token is circulated through all the processes on the ring (propagation rule : from P_i to $P_{(i+1) \bmod N}$)
  - When the token arrives at process P_j, the timestamp in the token is used to determine whether P_j must roll back

    If tk(i, m).ts < $V_j(p_j)$,

    then P_j must roll back to an earlier state because an orphan event has occurred at P_j

    Otherwise, the state of P_j is not changed
- Step 5
  - When the token returns to the originating process, the roll back recovery is complete

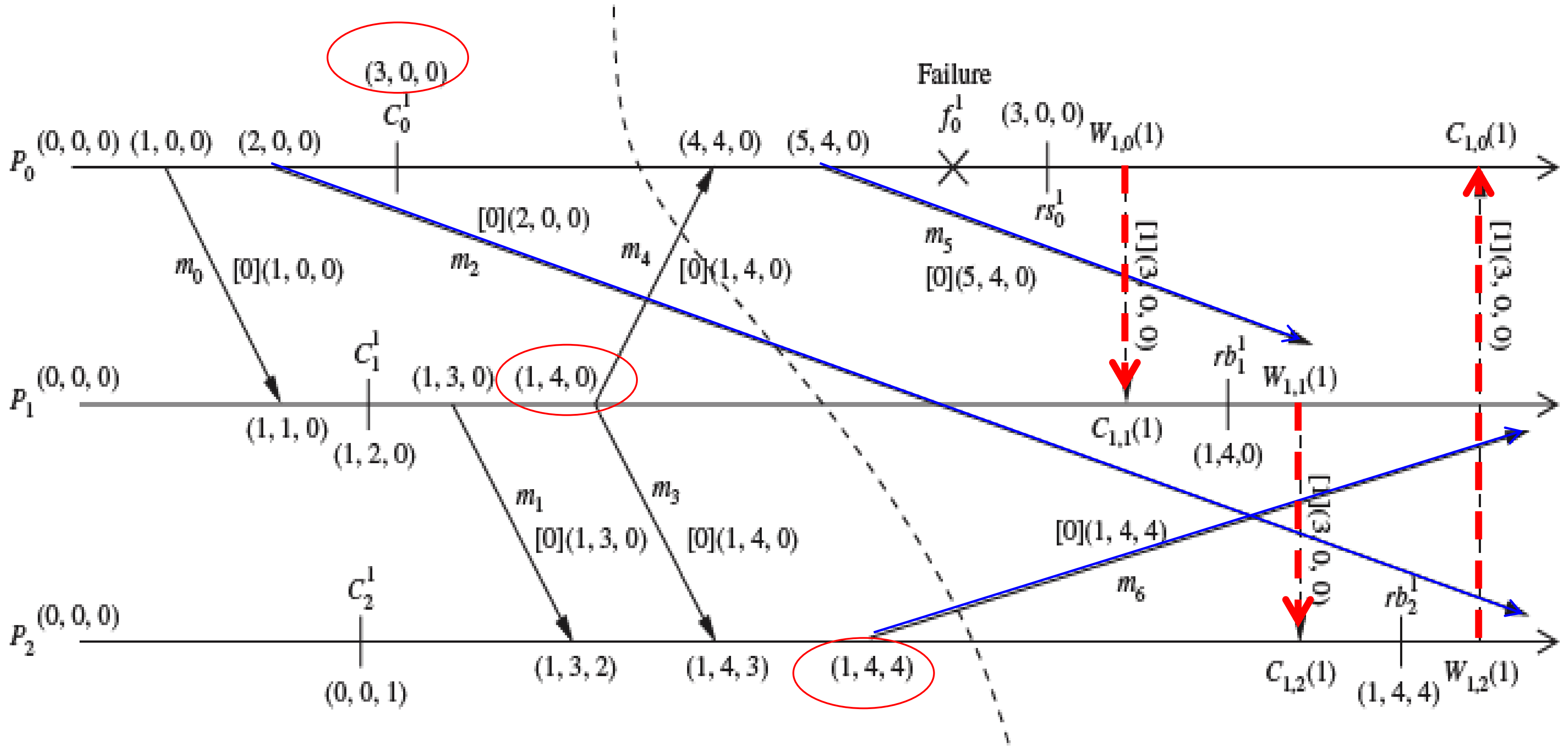
Peterson-Kearns Alg. – Formal Description (1)

- described as set of six rules, CRB1 to CRB6
- CRB1
  - A formerly failed process creates and propagates a token, event $w_{i,i}(m)$, only after restoring the state from the latest checkpoint and executing the message log from the stable storage
- CRB2
  - The restart event increments the incarnation number at the recovering process, and the token carries the vector timestamp of the restart event and the newly incremented incarnation number
- CRB3
  - A non-failed process will propagate the token only after it has rolled back

Peterson-Kearns Alg. – Formal Description (2)

- CRB4
  - A non-failed process will propagate the token only after it has incremented its incarnation number and has stored the vector timestamp of the token and the incarnation number of the token in its *OrVect* set
- CRB5
  - When the process that failed, recovered, and initiated the token, receives its token back, the rollback is complete
- CRB6
  - Messages that were in transit and which were orphaned by the failure and subsequent restart and recovery must be discarded

Peterson-Kearns - example



Helary-Mostefaoui-Netzer-Raynal protocol (1)

- Communication-induced checking protocol
- Some coordination is required in taking local checkpoints
- Achieve the coordination by piggybacking control information on application messages
- Basic checkpoints
  - Processes take local checkpoints independently
- Forced checkpoints
  - The protocol directs processes to take additional local checkpoints
  - A process takes a forces checkpoint when it receives a message and its predicate becomes true
- No local checkpoint is useless
- Takes as few forced checkpoints as possible

Helary-Mostefaoui-Netzer-Raynal protocol (2)

- Based on **Z-path and Z-cycle theory**
  - A useless checkpoint exactly corresponds to the existence of a Z-cycle in the distributed computation
  - The protocol prevents Z-Cycles

- A **Z-path** exists from local check point **A** to local checkpoint **B** iff (i) **A** precedes **B** in the same process, or (ii) a sequence of message [m_1, m_2, . . . , m_q] (q $\geq$ 1) exists such that
  - (1) **A** precedes **send($\mathbf{m_1}$)** in the same process, and
  - (2) for each $\mathbf{m_i}$, i < q**, delivery($\mathbf{m_i}$)** is in the same or earlier interval as **send($\mathbf{m_{i+1}}$)**, and
  - (3) **delivery($\mathbf{m_q}$)** precedes **B** in the same process

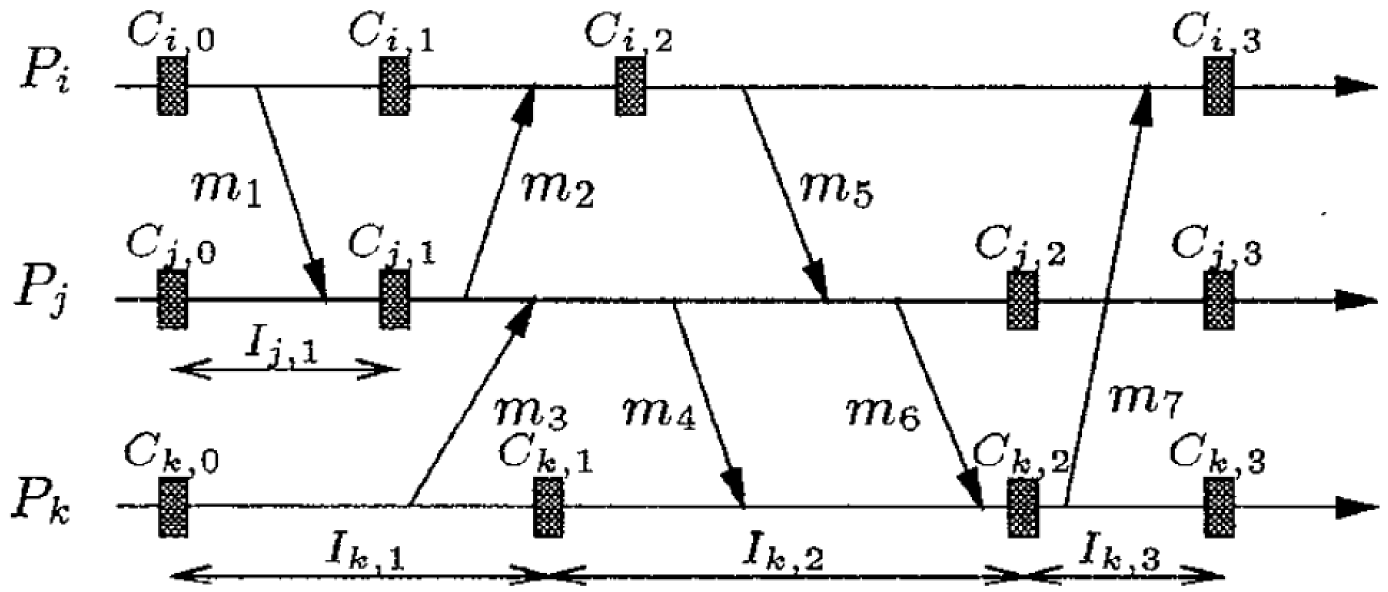
Helary-Mostefaoui-Netzer-Raynal protocol (3)

- A Z-path from a local checkpoint $C_{i,x}$ to the same local checkpoint $C_{i,x}$ is called a **Z-cycle** (i.e., it involves the local checkpoint $C_{i,x}$)

- In a Z-path $[m_1, m_2, \dots, m_q]$, two consecutive messages m_α and $m_{\alpha+1}$ form a **Z-pattern** iff send($m_{\alpha+1}$) → delivery(m_α)

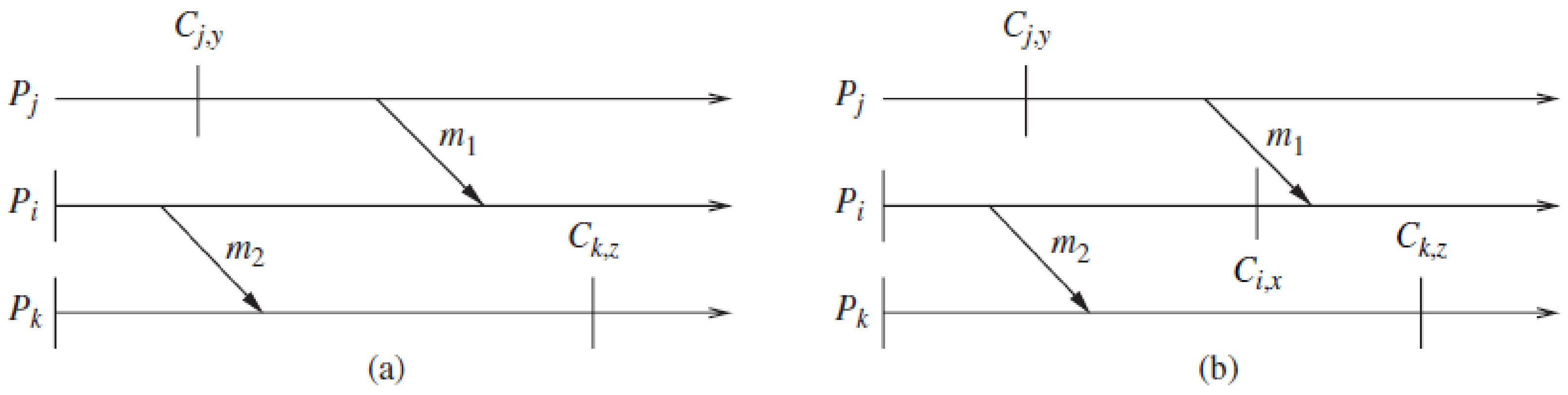
- Theorem : For any pair of checkpoints $C_{j,y}$ and $C_{k,z}$, such that there is a Z-path from $C_{j,y}$ to $C_{k,z}$, $C_{j,y}.t < C_{k,z}.t$ implies that there is no Z-cycle

H-M-N-R protocol – Z-path & Z-cycle ex.



- $[m_3, m_2]$ is a Z-path from $C_{k,0}$ to $C_{i,2}$
- $[m_5, m_4]$ and $[m_5, m_6]$ are two Z-paths from $C_{i,2}$ to $C_{k,2}$
- $[m_3, m_2]$ and $[m_5, m_4]$ are two Z-patterns
- The Z-path $[m_7, m_5, m_6]$ is a Z-cycle that involves the local checkpoint $C_{k,2}$

H-M-N-R protocol – forced checkpoints ex.



- (a) $\boldsymbol{m_1.t \le m_2.t}$: $C_{j,y}.t < m_1.t < m_2.t < C_{k,z}.t$. Hence, the Z-pattern $[m_1, m_2]$ is consistent with the assumption of the above theorem
- (b) $\boldsymbol{m_1.t > m_2.t}$: A safe strategy to prevent Z-cycle formation is to direct P_i to take a forced checkpoint $C_{i,x}$ before delivering m_1. This “breaks” $[m_1, m_2]$, so it is no longer a Z-pattern
- How to implement “taking a forced checkpoint”?
  - P_i takes a forced checkpoint if C is true, where C $\equiv$ ($\exists$ k: $sent\_to_i[k] \wedge m_1.t > min\_to_i[k]$)

Helary-Mostefaoui-Netzer-Raynal protocol – Alg.

```
Procedure take-checkpoint:
        $\forall k$ do $sent\_to_i[k]$ := false end do;
        $\forall k$ do $min\_to_i[k] := +\infty$ end do;
        $\forall k \neq i$ do $taken_i[k]$ := true end do;
        $clock_i[i] := clock_i[i] + 1$;
        Save the current local state with a copy of $clock_i[i]$;
        /* let $C_{i,x}$ denote this checkpoint. We have $C_{i,x}.t = clock_i[i]$ */
        $ckpt_i[i] := ckpt_i[i] + 1$;
(S0) initialization:
        $\forall k$ do $clock_i[k] := 0$; $ckpt_i[k] := 0$ end do;
        $taken_i[i] := false$;
        $take\_checkpoint$;
(S1) When $P_i$ sends a message to $P_k$:
        if $\neg$ $sent\_to_i[k]$ then $sent\_to_i[k] := true$; $min\_to_i[k] := clock_i[i]$
           end if;
        Send ($m$, $clock_i$, $ckpt_i$, $taken_i$) to $P_k$;
```

```
(S2) When $P_i$ receives ($m$, $clock_i$ $i$, $ckpt_i$, $taken_i$) from $P_j$:
    /* $m.clock[j]$ is the Lamport's timestamp of $m$ (i.e., $m.t$) */
    if $(\exists k : sent\_to_i[k] \wedge (m.clock[j] > min\_to_i[k]) \wedge$
       $((m.clock[j] > max(clock_i[k], m.clock[k])) \vee$
    $(m.ckpt[i] = ckpt_i[i] \wedge m.taken[i])))$
            then $take\_checkpoint$ /*forced checkpoint */
    end if;
    $clock_i[i] := max(clock_i[i], m.clock[j])$; /* update of the
       scalar clock $lc_i \equiv clock_i[i]$ */
    $\forall k \neq i$ do
          $clock_i[k] := max(clock_i[k], m.clock[k])$;
          case
                $m.ckpt[k] < ckpt_i[k] \rightarrow$ skip
                $m.ckpt[k] > ckpt_i[k] \rightarrow ckpt_i[k] := m.ckpt[k]$;
                  $taken_i[k] := m.taken[k]$
                $m.ckpt[k] < ckpt_i[k] \rightarrow taken_i[k] := taken_i[k] \vee$
                  $m.taken[k]$
            end case
         end do
    deliver ($m$);
```

The End

- Question portion