CST 402 - DISTRIBUTED COMPUTING

Module – IV
Distributed shared memory and Failure Recovery

Module – IV
Lesson Plan

- **L1: Distributed shared memory – Abstraction and advantages**
- **L2: Shared memory mutual exclusion – Lamport's bakery algorithm**
- **L3: Checkpointing and rollback recovery – System model, consistent and inconsistent states**
- **L4: different types of messages**, Issues in failure recovery**.**
- **L5:** log based roll back recovery
- **L6:** log based roll back recovery

Distributed shared memory – Abstraction and advantages

- Distributed shared memory (DSM) is an abstraction provided to the programmer of a distributed system.
- It gives the impression of a single monolithic memory, as in traditional von Neumann architecture
- Programmers access the data across the network using only read and write primitives
- Programmers do not have to deal with send and receive communication primitives and the ensuing complexity of dealing explicitly with synchronization and consistency in the message passing model.
- A part of each computer's memory is earmarked for shared space, and the remainder is private memory

Distributed shared memory – Abstraction and advantages

- To provide programmers with the illusion of a single shared address space, a memory mapping management layer is required to manage the shared virtual memory space.

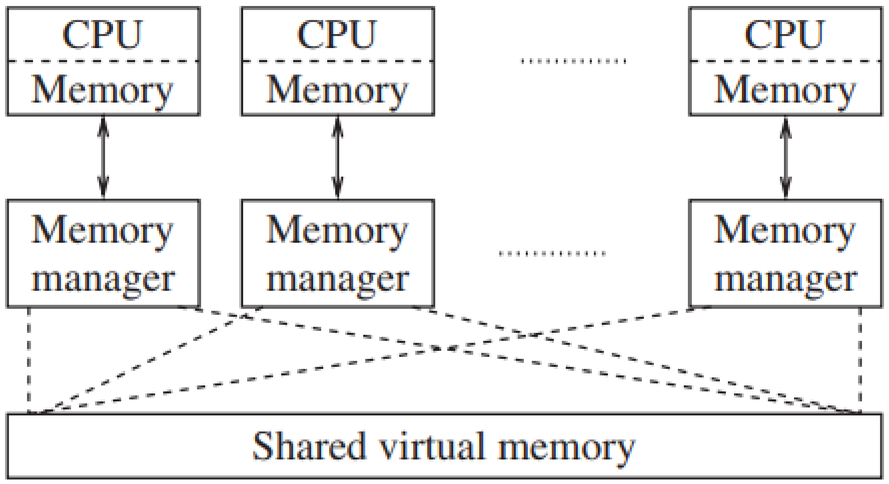


Figure 1: Abstract view of DSM

Distributed shared memory – Abstraction and advantages

DSM has the following advantages:

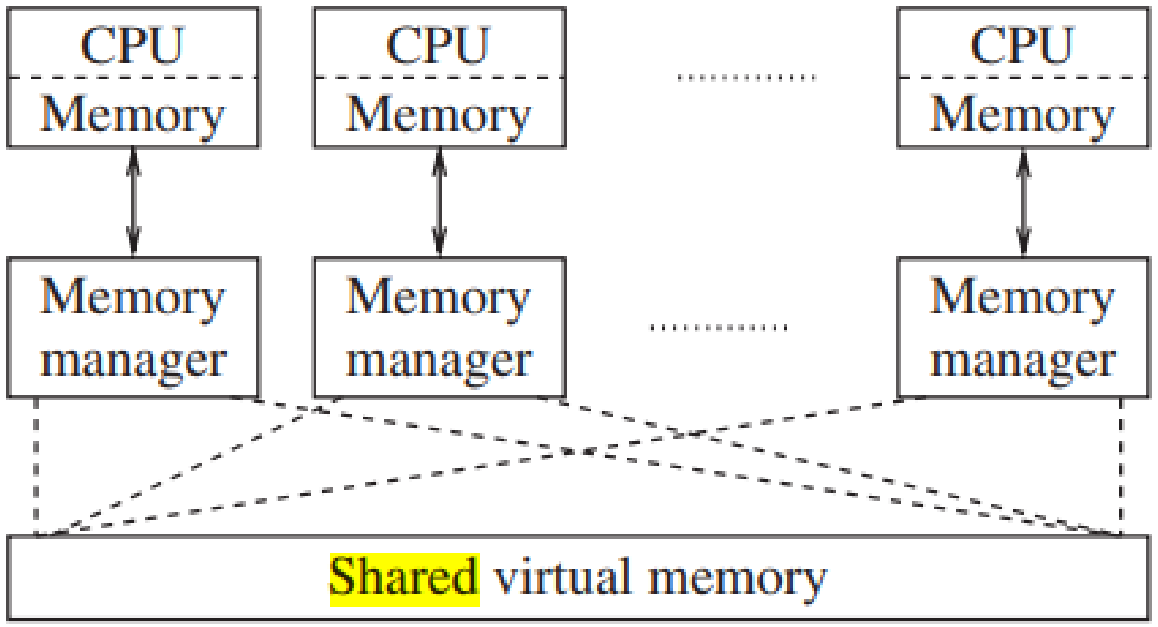
- 1. Communication across the network is achieved by the read/write abstraction that simplifies the task of programmers.
- 2. A single address space is provided, thereby providing the possibility of avoiding data movement across multiple address spaces, and simplifying passing-by-reference and passing complex data structures containing pointers.
- 3. If a block of data needs to be moved, the system can exploit locality of reference to reduce the communication overhead.
- 4. DSM is often cheaper than using dedicated multiprocessor systems, because it uses simpler software interfaces and off-the-shelf hardware.
- 5. There is no bottleneck presented by a single memory access bus

Distributed shared memory – Abstraction and advantages

- 6. DSM effectively provides a large (virtual) main memory.
- 7. DSM provides portability of programs written using DSM. This portability arises due to a common DSM programming interface, which is independent of the operating system and other low-level system characteristics.

Abstract view of DSM

- DSM is transparent to programmer



Distributed shared memory – Disadvantages

DSM has the following disadvantages:

1. Programmers are not shielded from having to know about various replica consistency models and from coding their distributed applications according to the semantics of these models.
2. As DSM is implemented under the covers using asynchronous message passing, the overheads incurred are at least as high as those of a message passing implementation. As such, DSM implementations cannot be more efficient than asynchronous message-passing implementations. The generality of the DSM software may make it less efficient.

DSM -disadvantages:

3. By yielding control to the DSM memory management layer, programmers lose the ability to use their own message-passing solutions for accessing shared objects. It is likely that the standard implementations of DSM have a higher overhead than a programmer-written implementation tailored for a specific application and system

Issues in designing a DSM system

The main issues in designing a DSM system are the following:

- ❑ Determining what semantics to allow for concurrent access to shared objects. The semantics needs to be clearly specified so that the programmer can code his program using an appropriate logic.
- ❑ Determining the best way to implement the semantics of concurrent access to shared data. One possibility is to use replication. Decide on degree of replication

➢ -partial replication at some sites, or full replication at all the sites.

➢ -decide on whether to use read-replication (replication for the read operations) or write-replication (replication for the write operations) or both.

Issues in designing a DSM system

- Selecting the locations for replication (if full replication is not used), to optimize efficiency from the system's viewpoint.

- Determining the location of remote data that the application needs to access, if full replication is not used.

- Reducing communication delays and the number of messages that are involved under the covers while implementing the semantics of concurrent access to shared data.

- four broad dimensions along which DSM systems can be classified and implemented:
  - Whether data is replicated or cached.
  - Whether remote access is by hardware or by software.
  - Whether the caching/replication is controlled by hardware or software.
  - Whether the DSM is controlled by the distributed memory manager

Table 12.1 Comparison of DSM systems (adapted from [29]).

Type of DSM	Examples	Management	Caching	Remote access
Single-bus multiprocessor	Firefly, Sequent	by MMU	hardware control	by hardware
Switched multiprocessor	Alewife, Dash	by MMU	hardware control	by hardware
NUMA system	Butterfly, CM*	by OS	software control	by hardware
Page-based DSM	Ivy, Mirage	by OS	software control	by software
Shared variable DSM	Midway, Munin	by language runtime system	software control	by software
Shared object DSM	Linda, Orca	by language runtime system	software control	by software

Shared memory mutual exclusion

- Operating systems have traditionally dealt with multi-process synchronization using
  - algorithms based on first principles,
  - high-level constructs such as semaphores and monitors, and
  - special "atomically executed" instructions supported by special-purpose hardware (e.g., Test&Set, Swap, and Compare&Swap)

- These algorithms are applicable to all shared memory systems
- the bakery algorithm, which requires O (n) accesses in the entry section, irrespective of the level of contention
- fast mutual exclusion, which requires O (1) accesses in the entry section in the absence of contention.
- Bakery algorithm ->> illustrates technique in resolving concurrency

Lamport's bakery algorithm

- Lamport proposed the classical bakery algorithm for n-process mutual exclusion in shared memory systems
- So called because it mimics the actions that customers follow in a bakery store.
- A process wanting to enter the critical section picks a token number that is one greater than the elements in the array
- Processes enter the critical section in the increasing order of the token numbers.
- In case of **concurrent accesses** to *choosing* by multiple processes, the processes may have the same token number.
- In this case, a unique lexicographic order is defined on the tuple < token , pid>, and this dictates the order in which processes enter the critical section
- The algorithm can be shown to satisfy the three requirements of the critical section problem: (i) mutual exclusion, (ii) bounded waiting, and (iii) progress.

Lamport’s bakery algorithm

- In the entry section, a process chooses a timestamp for itself, and resets it to 0 in the exit section.

- In lines 1a–1c each process chooses a timestamp for itself, as the max of the latest timestamps of all processes, plus one

- These steps are non-atomic; thus multiple processes could be choosing timestamps in overlapping durations.

Lamport’s bakery algorithm

- When process i reaches line 1d, it has to check the status of each other process j, to deal with the effects of any race conditions in selecting timestamps
- In lines 1d–1f, process i serially checks the status of each other process j. If j is selecting a timestamp for itself, j’s selection may overlap with that of i
- Process-*i* needs to make sure process-*j* does not assign itself the same timestamp. Otherwise mutual exclusion could be violated as i would enter the CS, and subsequently, j, having a lower process identifier and hence a lexicographically lower timestamp, would also enter the CS. Hence, i waits for j’s timestamp to stabilize, i.e., choosing[j] to be set to false
- These steps are non-atomic; thus multiple processes could be choosing timestamps in overlapping durations.

Lamport’s bakery algorithm

- Once j’s timestamp is stabilized, i moves from line 1e to line 1f.
  - Either j is not requesting $\rightarrow$ j’s timestamp is 0
  - or j is requesting.
- Line 1f determines the relative priority between i and j.
  - The process with a lexicographically lower timestamp has higher priority and enters the CS; the other process has to wait (line 1g).
  - Hence, mutual exclusion is satisfied.

Lamport’s bakery algorithm

- Bounded waiting is satisfied because each other process j can “overtake” process i at most once after i has completed choosing its timestamp. The second time j chooses a timestamp, the value will necessarily be larger than i’s timestamp if i has not yet entered its CS.
- Progress is guaranteed because the lexicographic order is a total order and the process with the lowest timestamp at any time in the loop (lines 1d–1g) is guaranteed to enter the CS

Lamport’s bakery algorithm

```
(shared vars)
boolean: $choosing[1 \ldots n]$;
integer: $timestamp[1 \ldots n]$;
repeat
(1)   $P_i$ executes the following for the entry section:
(1a)  $choosing[i] \longleftarrow 1$;
(1b)  $timestamp[i] \longleftarrow \max_{k \in [1 \ldots n]}(timestamp[k]) + 1$;
(1c)  $choosing[i] \longleftarrow 0$;
(1d)  for $count = 1$ to $n$ do
(1e)        while $choosing[count]$ do no-op;
(1f)        while $timestamp[count] \neq 0$ and $(timestamp[count], count)$
               $< (timestamp[i], i)$ do
(1g)              no-op.
(2)   $P_i$ executes the critical section (CS) after the entry section
(3)   $P_i$ executes the following exit section after the CS:
(3a)  $timestamp[i] \longleftarrow 0$.
(4)   $P_i$ executes the remainder section after the exit section
until false;
```

Algorithm 12.5 Lamport's *n*-process bakery algorithm for shared memory mutual exclusion. Code shown is for process *Pi*, $1 \leq i \leq n$.

Checkpointing and rollback recovery

- Distributed systems - not fault-tolerant and the vast computing potential is often hampered by their susceptibility to failures
- Techniques to add reliability and high availability to distributed systems include
  - transactions, group communication, and rollback recovery
- rollback recovery protocols, which restore the system back to a consistent state after a failure

Checkpointing and rollback recovery

- Rollback recovery treats a distributed system application as a collection of processes that communicate over a network.
- It achieves fault tolerance by periodically saving the state of a process during the failure-free execution, enabling it to restart from a saved state upon a failure to reduce the amount of lost work.
- The saved state is called a checkpoint, and the procedure of restarting from a previously checkpointed state is called rollback recovery.
- A checkpoint -> saved on →the stable storage or the volatile storage

Checkpointing and rollback recovery

- Rollback recovery - complicated → messages induce inter-process dependencies during failure-free operation
- Upon a failure of one or more processes in a system, these dependencies may force some of the processes that did not fail to roll back→rollback propagation

Checkpointing and rollback recovery

- To see why rollback propagation occurs, consider the situation where the sender of a message m rolls back to a state that precedes the sending of m.
- The receiver of m must also roll back to a state that precedes m’s receipt;
- otherwise, the states of the two processes would be inconsistent because they would show that message m was received without being sent, which is impossible in any correct failure-free execution.
- This phenomenon of cascaded rollback is called the **domino effect**.

Checkpointing and rollback recovery

- Independent or uncoordinated checkpointing
  - if each participating process takes its checkpoints independently,
  - the system is susceptible to the domino effect
- Need To avoid Domino effect
- coordinated checkpointing → processes coordinate their checkpoints to form a system-wide consistent state.
  - In case of a process failure, the system state can be restored to such a consistent set of checkpoints, preventing the rollback propagation.

- Communication-induced checkpointing
  - each process take checkpoints based on information piggybacked on the application messages it receives from other processes
  - Checkpoints are taken such that a system-wide consistent state always exists on stable storage, thereby avoiding the domino effect.

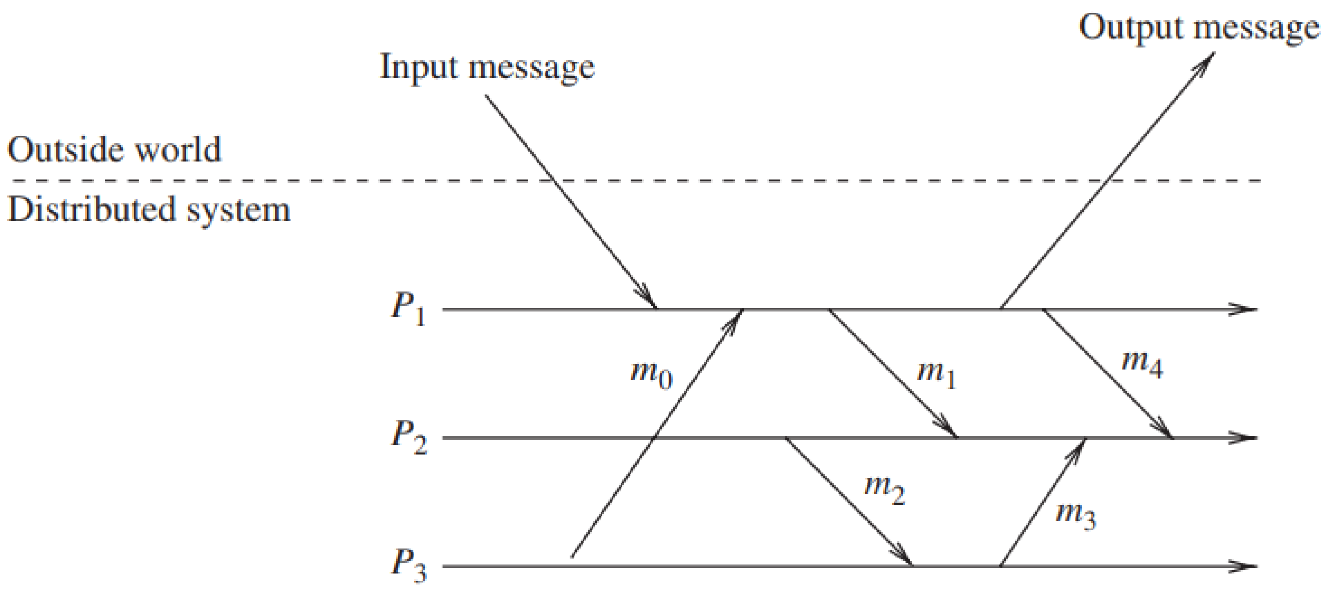
Checkpointing and rollback recovery

- checkpoint-based rollback recovery → relies only on checkpoints to achieve fault-tolerance
- Log-based rollback recovery combines checkpointing with logging of nondeterministic events.
  - Log-based rollback recovery relies on the piecewise deterministic (PWD) assumption, which postulates that all non-deterministic events that a process executes can be identified and that the information necessary to replay each event during recovery can be logged in the event’s determinant.
  - By logging and replaying the non-deterministic events in their exact original order, a process can deterministically recreate its pre-failure state even if this state has not been checkpointed.
  - Log-based rollback recovery in general enables a system to recover beyond the most recent set of consistent checkpoints.

Checkpointing and rollback recovery: System model

System model

- A distributed system consists of a fixed number of processes, P1, P2… PN , which communicate only through messages.
- Processes cooperate to execute a distributed application and interact with the outside world by receiving and sending input and output messages, respectively



Checkpointing and rollback recovery: System model

- Rollback-recovery protocols generally make assumptions about the reliability of the inter-process communication. i.e. The communication subsystem

- ➢ delivers messages reliably, in first-in-first-out (FIFO) order,
- ➢ can lose, duplicate, or reorder messages.
- The choice between these two assumptions usually affects the complexity of checkpointing and failure recovery.

- A system recovers correctly if its internal state is consistent with the observable behavior of the system before the failure
- Rollback-recovery protocols therefore must maintain information about the internal interactions among processes and also the external interactions with the outside world.

Rollback recovery : System model

A local checkpoint

- In distributed systems, all processes save their local states at certain instants of time.

- This saved state is known as a local checkpoint.

✓ A local checkpoint is a snapshot of the state of the process at a given instance and the event of recording the state of a process is called local checkpointing.

- The contents of a checkpoint depend upon the application context and the checkpointing method being used.

Rollback recovery : System model

Local checkpointing

- Depending upon the checkpointing method used, a process may keep several local checkpoints or just a single checkpoint at any time.

- We assume that a process stores all local checkpoints on the stable storage so that they are available even if the process crashes.

- We also assume that a process is able to roll back to any of its existing local checkpoints and thus restore to and restart from the corresponding state

- Let $C_{i,k}$ denote the kth local checkpoint at process P_i. Generally, it is assumed that a process P_i takes a checkpoint $C_{i,0}$ before it starts execution.

- A local checkpoint is shown in the process-line by the symbol “ | ”.

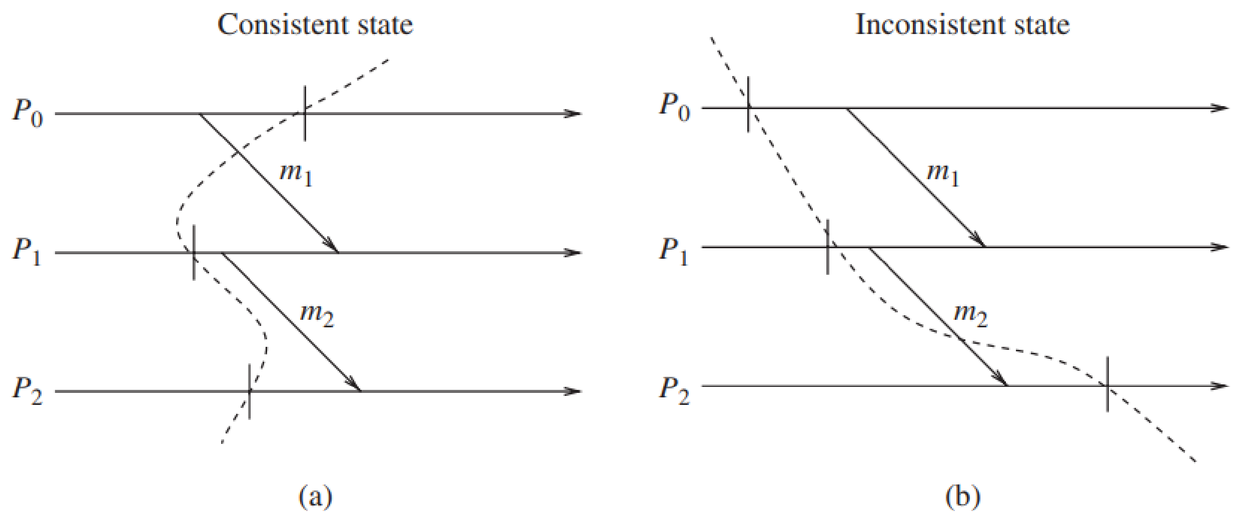
Checkpointing and rollback recovery

consistent and inconsistent states

- A global state of a distributed system is a collection of the individual states of all participating processes and the states of the communication channels.
- A consistent global state is one that may occur during a failure-free execution of a distributed computation.
- More precisely, **a consistent system state** is one in which a process's state reflects a message receipt, then the state of the corresponding sender must reflect the sending of that message

Checkpointing and rollback recovery

Figure 13.2 Examples of consistent and inconsistent states [13].



The state in Figure 13.2(a) is consistent and the state in Figure 13.2(b) is inconsistent.

- Note that the consistent state in Figure 13.2(a) shows message m1 to have been sent but not yet received, but that is alright.
- The state in Figure 13.2(a) is consistent because it represents a situation in which every message that has been received, there is a corresponding message send event

Checkpointing and rollback recovery

- The state in Figure 13.2(b) is inconsistent because process P2 is shown to have received m2 but the state of process P1 does not reflect having sent it. Such a state is impossible in any failure-free, correct computation.
- Inconsistent states occur because of failures. For instance, the situation shown in Figure 13.2(b) may occur if process P1 fails after sending message m2 to process P2 and then restarts at the state shown in Figure 13.2(b).
- Thus, a local checkpoint is a snapshot of a local state of a process and a global checkpoint is a set of local checkpoints, one from each process.
- A consistent global checkpoint is a global checkpoint such that no message is sent by a process after taking its local checkpoint that is received by another process before taking its local checkpoint

Checkpointing and rollback recovery

- The consistency of global checkpoints strongly depends on the flow of messages exchanged by processes and an arbitrary set of local checkpoints at processes may not form a consistent global checkpoint
- The fundamental goal of any rollback-recovery protocol is to bring the system to a consistent state after a failure.
- The reconstructed consistent state is not necessarily one that occurred before the failure.
- It is sufficient that the reconstructed state be one that could have occurred before the failure in a failure-free execution, provided that it is consistent with the interactions that the system had with the outside world.

Interactions with the outside world

- A distributed application often interacts with the outside world to receive input data or deliver the outcome of a computation.
- If a failure occurs, the outside world cannot be expected to roll back. (Printer,ATM)
- Outside world→special process the “outside world process” (OWP).
- outside world should see a consistent behavior of the system despite failures

Interactions with the outside world

- output commit problem
- before sending output to the OWP, the system must ensure that the state from which the output is sent will be recovered despite any future failure.
- recovery protocols must arrange to save these input messages so that they can be retrieved when needed for execution replay after a failure.
- →on the stable storage before allowing the application program to process it.
- An interaction with the outside world to deliver the outcome of a computation is shown on the process-line by the symbol "||"

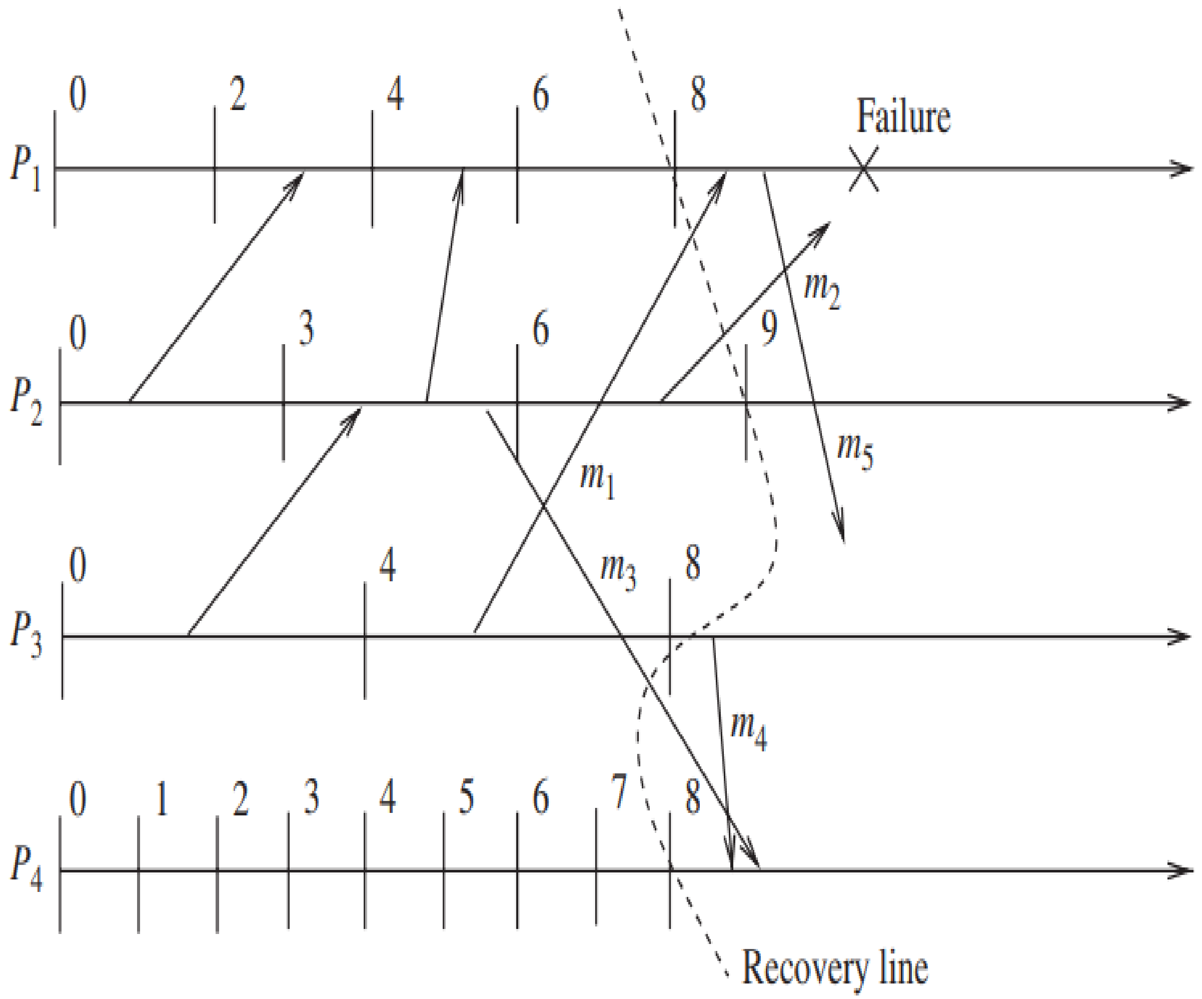
Different types of messages

- A process failure and subsequent recovery may leave messages that were perfectly received (and processed) before the failure in abnormal states.

- This is because a rollback of processes for recovery may have to rollback the send and receive operations of several messages
- Four processes.
- Process P1 fails at the point indicated and the whole system
- recovers to the state indicated by the recovery line; that is, to global state

  $\{C_{1,8}, C_{2,9}, C_{3,8}, C_{4,8}\}.$

Figure 13.3 Different types of messages [25].



$\{C_{1,8}, C_{2,9}, C_{3,8}, C_{4,8}\}$.

Different types of messages

1. **In-transit messages**

- In Figure the global state shows that message m1 has been sent but not yet received. We call such a message an in-transit message
- When in-transit messages are part of a global system state, these messages do not cause any inconsistency.
- However, depending on whether the system model assumes reliable communication channels, rollback-recovery protocols may have to guarantee the delivery of in-transit messages when failures occur.
- For reliable communication channels, a consistent state must include in-transit messages because they will always be delivered to their destinations in any legal execution of the system.
- if a system model assumes lossy communication channels, then in-transit messages can be omitted from system state.

Different types of messages

2. Lost messages

- Messages whose send is not undone but receive is undone due to rollback are called lost messages.

- This type of messages occurs when the process rolls back to a checkpoint prior to reception of the message while the sender does not rollback beyond the send operation of the message.

- In Figure 13.3, message m1 is a lost message.

Different types of messages

3. Delayed messages

- Messages whose receive is not recorded because
- ➢ the receiving process was either down or
- ➢ the message arrived after the rollback of the receiving process, are called delayed messages.
- For example, messages m2 and m5 in Figure 13.3 are delayed messages.

Different types of messages

4. Orphan messages

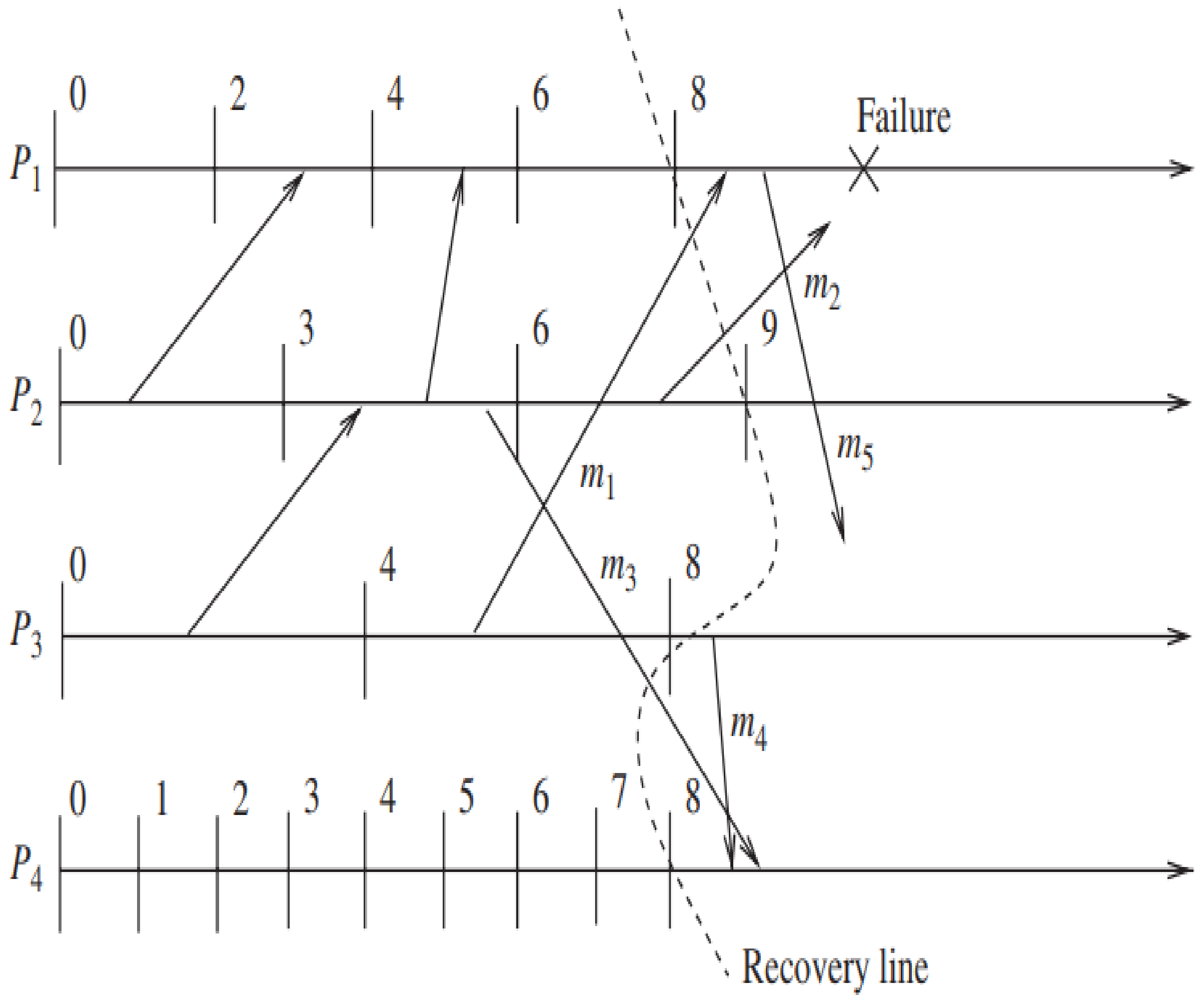
- Messages with receive recorded but message send not recorded are called orphan messages.
- For example, a rollback might have undone the send of such messages, leaving the receive event intact at the receiving process.
- Orphan messages do not arise if processes roll back to a consistent global state.

Different types of messages

5. Duplicate messages

- Duplicate messages arise due to message logging and replaying during process recovery
- m4,m5
- message m4 was sent and received before the rollback. However, due to the rollback of process P4 to C48 and process P3 to C38, both send and receipt of message m4 are undone. When process P3 restarts from C38, it will resend message m4. If P4 replays message m4, then message m4 is called a duplicate message.

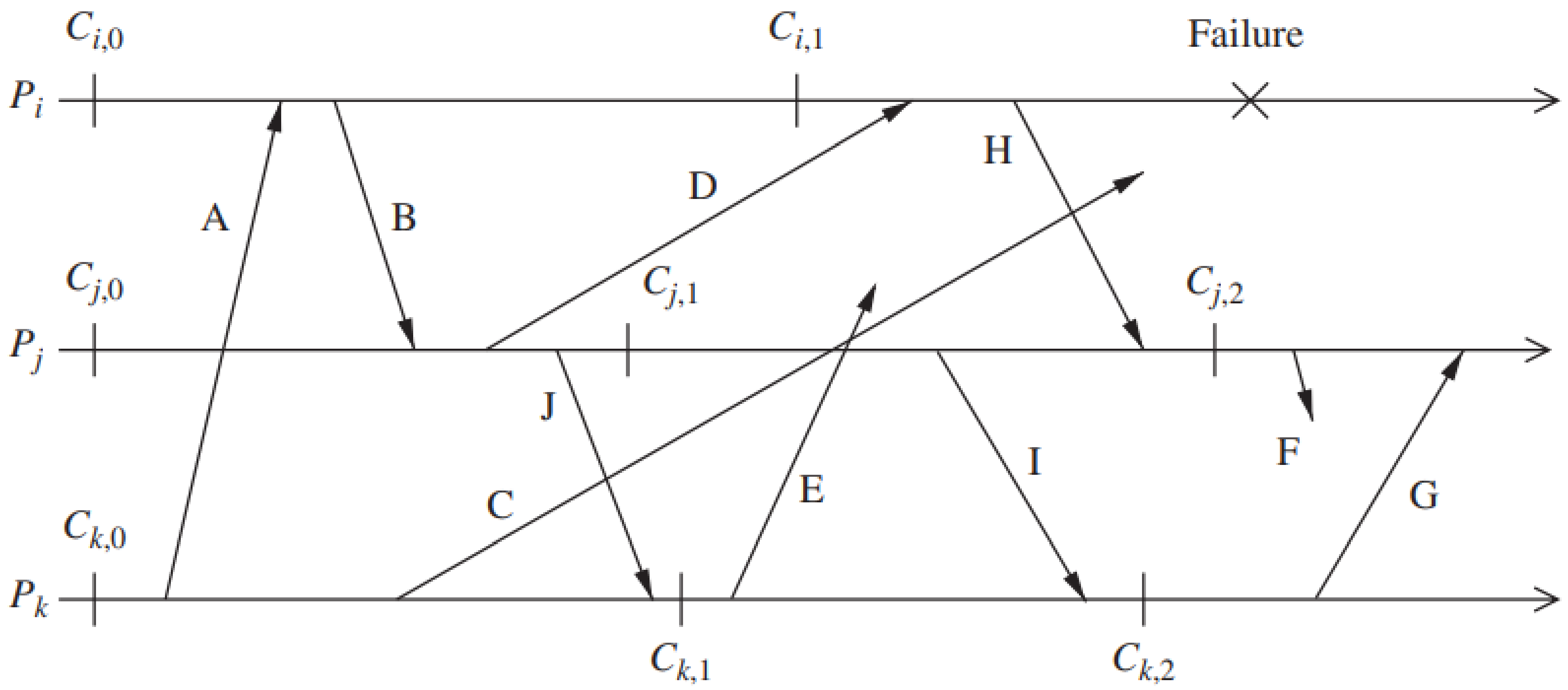
Figure 13.3 Different types of messages [25].

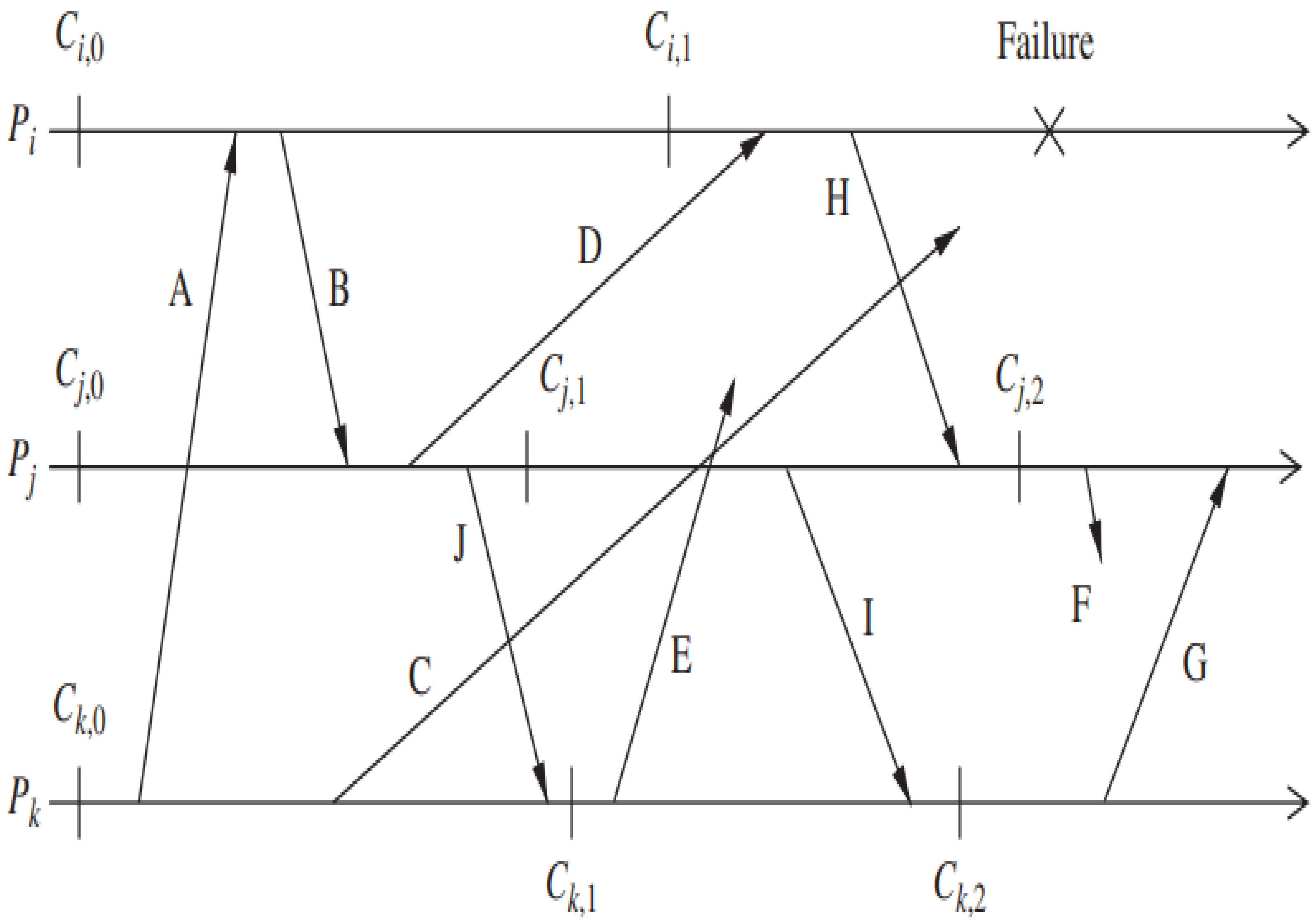


$\{C_{1,8}, C_{2,9}, C_{3,8}, C_{4,8}\}$.

Issues in failure recovery

In a failure recovery, we must not only restore the system to a consistent state, but also appropriately handle messages that are left in an abnormal state due to the failure and recovery




$C_{i,0}$
$C_{i,1}$
Failure
P_i
A
B
D
H
$C_{j,0}$
$C_{j,1}$
$C_{j,2}$
P_j
J
F
I
E
G
C
$C_{k,0}$
P_k
$C_{k,1}$
$C_{k,2}$

Issues in failure recovery

- The computation comprises of three processes Pi, Pj, and Pk, connected through a communication network.
- The processes communicate solely by exchanging messages over fault-free, FIFO communication channels.
- Processes Pi, Pj, and Pk have taken checkpoints {Ci0, Ci1}, {Cj0, Cj1, Cj2}, and {Ck0, Ck1}, respectively, and these processes have exchanged messages A to J
- Suppose process Pi fails at the instance indicated in the figure.
- All the contents of the volatile memory of Pi are lost and, after Pi has recovered from the failure, the system needs to be restored to a consistent global state from where the processes can resume their execution.
- Process Pi's state is restored to a valid state by rolling it back to its most recent checkpoint Ci1.

Issues in failure recovery

- To restore the system to a consistent state, the process Pj rolls back to checkpoint Cj1 because the rollback of process Pi to checkpoint Ci1 created an orphan message H
- Note that process Pj does not roll back to checkpoint Cj2 but to checkpoint Cj1, because rolling back to checkpoint Cj2 does not eliminate the orphan message H.
- Even this resulting state is not a consistent global state, as an orphan message I is created due to the roll back of process Pj to checkpoint Cj1.
- To eliminate this orphan message, process Pk rolls back to checkpoint Ck1.
- The restored global state {Ci1, Cj1, Ck1} is a consistent state as it is free from orphan message
- Although the system state has been restored to a consistent state, several messages are left in an erroneous state which must be handled correctly.

Issues in failure recovery

- Messages A, B, D, G, H, I, and J had been received at the points indicated in the figure and messages C, E, and F were in transit when the failure occurred.

- Restoration of system state to checkpoints {Ci1, Cj1,Ck1} automatically handles messages A, B, and J because the send and receive events of messages A, B, and J have been recorded, and both the events for G, H, and I have been completely undone.

- These messages cause no problem and we call messages A, B, and J normal messages and messages G, H, and I vanished messages

Issues in failure recovery

- Messages C, D, E, and F are potentially problematic.
- Message C is in transit during the failure and it is a delayed message.
- The delayed message C has several possibilities:
  - C might arrive at process Pi before it recovers,
  - it might arrive while Pi is recovering, or
  - it might arrive after Pi has completed recovery.
- Each of these cases must be dealt with correctly.

Issues in failure recovery

- Message D is a lost message since the send event for D is recorded in the restored state for process Pj, but the receive event has been undone at process Pi.
- Process Pj will not resend D without an additional mechanism, since the send D at Pj occurred before the checkpoint and the communication system successfully delivered D

Issues in failure recovery

- Messages E and F are delayed orphan messages and pose perhaps the most serious problem of all the messages.
- When messages E and F arrive at their respective destinations, they must be discarded since their send events have been undone.
- Processes, after resuming execution from their checkpoints, will generate both of these messages, and recovery techniques must be able to distinguish between messages like C and those like E and F.

Issues in failure recovery

- Lost messages like D can be handled by having processes keep a message log of all the sent messages.

- So when a process restores to a checkpoint, it replays the messages from its log to handle the lost message problem.

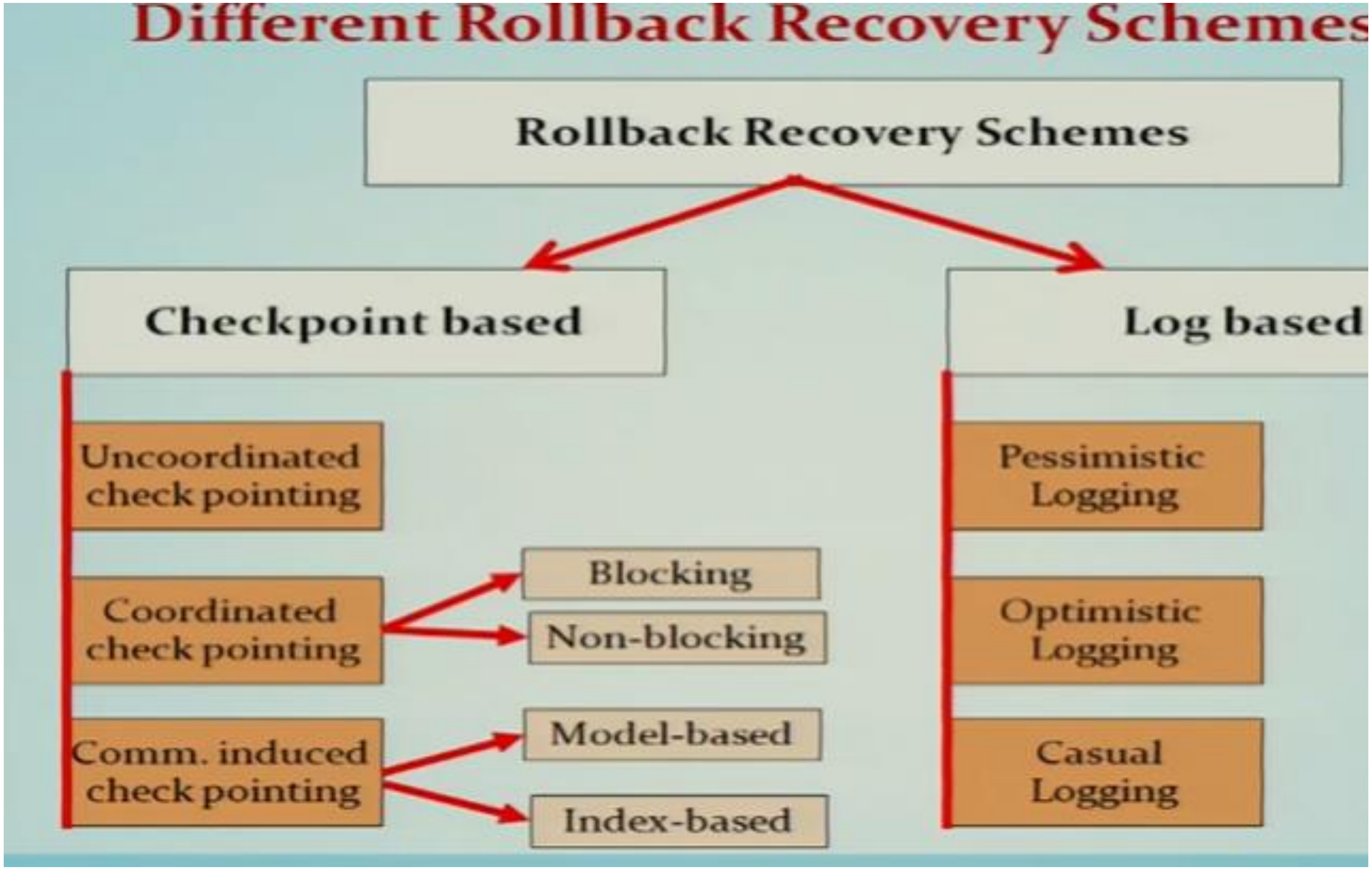
- However, message logging and message replaying during recovery can result in duplicate messages

- E.g. when process Pj replays messages from its log, it will regenerate message J. Process Pk, which has already received message J, will receive it again, thereby causing inconsistency in the system state.

Issues in failure recovery

- Overlapping failures further complicate the recovery process.
- A process Pj that begins rollback/recovery in response to the failure of a process Pi can itself fail and develop amnesia with respect process Pi’s failure
- a mechanism must be introduced to deal with amnesia and the resulting inconsistencies

Different Rollback Recovery Schemes



Checkpoint-based recovery

- In the checkpoint-based recovery approach, the state of each process and the communication channel is checkpointed frequently so that, upon a failure, the system can be restored to a globally consistent set of checkpoints.
- It does not rely on the PWD assumption, and so does not need to detect, log, or replay non-deterministic events.
- less restrictive and simpler to implement than log-based rollback recovery
- does not guarantee that prefailure execution can be deterministically regenerated after a rollback
- may not be suitable for applications that require frequent interactions with the outside world

Checkpoint-based recovery

- three categories:
  - uncoordinated checkpointing,
  - coordinated checkpointing, and
  - communication-induced checkpointing

Checkpoint-based recovery

- **1. Uncoordinated checkpointing**
- each process has autonomy in deciding when to take checkpoints.
- eliminates the synchronization overhead as there is no need for coordination between processes
- it allows processes to take checkpoints when it is most convenient or efficient.
- The main advantage is the lower runtime overhead during normal execution, because no coordination among processes is necessary.
- allows each process to select appropriate checkpoints positions

Uncoordinated checkpointing
Disadvantages

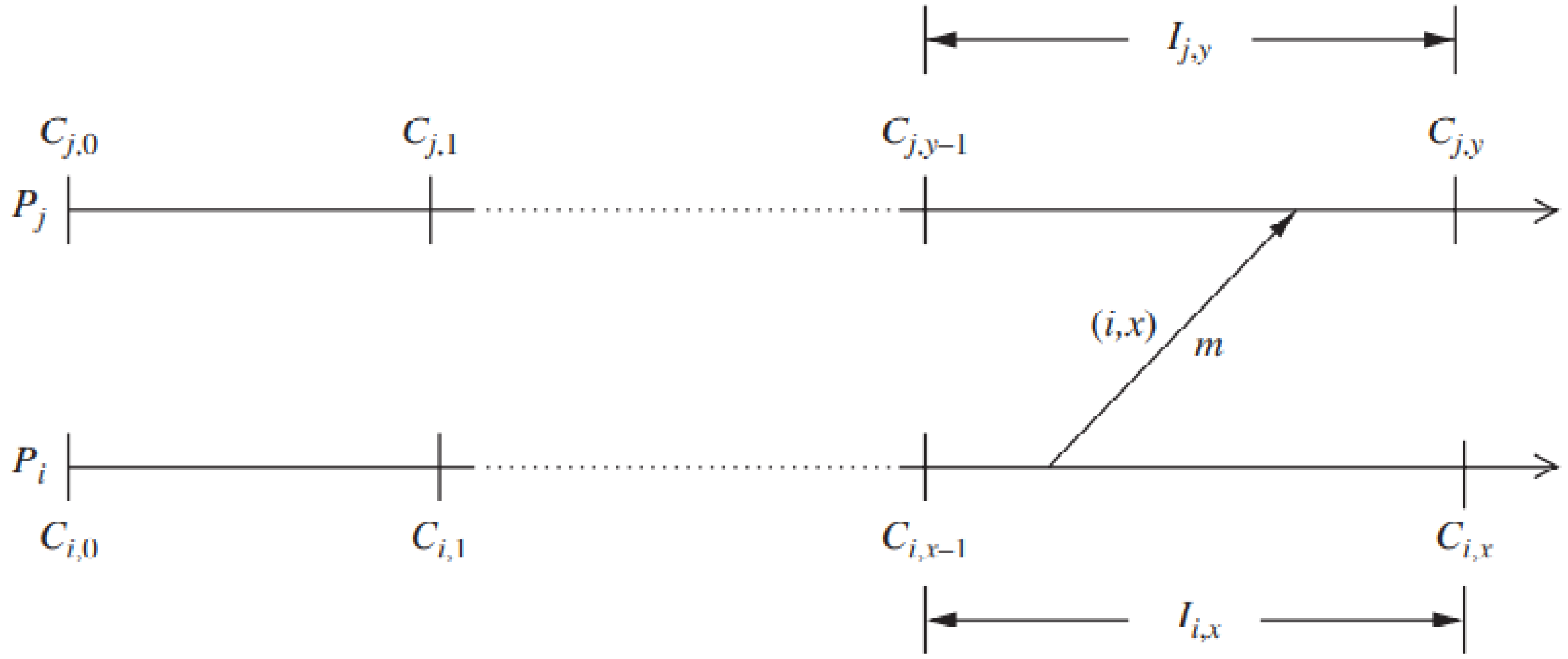
1) possibility of the domino effect during a recovery→loss of a large amount of useful work.
2) Second, recovery from a failure is slow because processes need to iterate to find a consistent set of checkpoints. (useless checkpoints)
3) Third, uncoordinated checkpointing forces each process to maintain multiple checkpoints, and to periodically invoke a garbage collection algorithm to reclaim the checkpoints that are no longer required.
4) Fourth, it is not suitable for applications with frequent output commits because these require global coordination to compute the recovery line, negating much of the advantage of autonomy.

Uncoordinated checkpointing

- we need to determine a consistent global checkpoint to rollback to, when a failure occurs
- the processes record the dependencies among their checkpoints caused by message exchange during failure-free operation
- direct dependency tracking technique used

Uncoordinated checkpointing

Figure 13.5 Checkpoint index and checkpoint interval [13].



Uncoordinated checkpointing

- Let $C_{i,x}$ be the xth checkpoint of process P_i, where i is the process i.d. and x is the checkpoint index
- Let $I_{i,x}$ denote the *checkpoint interval* or simply *interval* between checkpoints $C_{i,x-1}$ and $C_{i,x}$.
- When process P_i at interval $I_{i,x}$ sends a message m to P_j, it piggybacks the pair (i, x) on m. When P_j receives m during interval $I_{j,y}$, it records the dependency from $I_{i,x}$ to $I_{j,y}$, which is later saved onto stable storage when P_j takes checkpoint $C_{j,y}$.

Direct dependency tracking technique in **uncoordinated checkpointing**

- When a failure occurs, the recovering process initiates rollback by broadcasting a *dependency request* message to collect all the dependency information maintained by each process.

- When a process receives this message, it stops its execution and replies with the dependency information saved on the stable storage as well as with the dependency information, if any, which is associated with its current state.

- The initiator then calculates the recovery line based on the global dependency information and broadcasts a *rollback request* message containing the recovery line.

- Upon receiving this message, a process whose current state belongs to the recovery line simply resumes execution; otherwise, it rolls back to an earlier checkpoint as indicated by the recovery line

2. Coordinated checkpointing

- In coordinated checkpointing, processes orchestrate their checkpointing activities so that all local checkpoints form a consistent global state .
- Coordinated checkpointing simplifies recovery and is not susceptible to the domino effect, since every process always restarts from its most recent checkpoint.
- Requires each process to maintain only one checkpoint on the stable storage, reducing the storage overhead and eliminating the need for garbage collection.

Coordinated checkpointing :disadvantage

- disadvantage →large latency is involved in committing output, as a global checkpoint is needed before a message is sent to the OWP
- Also, delays and overhead are involved everytime a new global checkpoint is taken

Coordinated checkpointing: approaches

- Since perfectly synchronized clocks are not available, the following approaches are used to guarantee checkpoint consistency:
- ❑ either the sending of messages is blocked for the duration of the protocol, or
- ❑ checkpoint indices are piggybacked to avoid blocking

Blocking Coordinated checkpointing

- to block communications while the checkpointing protocol executes.
- After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpointing activity is complete.

- The coordinator takes a checkpoint and broadcasts a *request message* to all processes, asking them to take a checkpoint.
- When a process receives this message, it stops its execution, flushes all the communication channels, takes a tentative checkpoint, and sends an acknowledgment message back to the coordinator.

Blocking Coordinated checkpointing contd…

- After the coordinator receives acknowledgments from all processes, it broadcasts a commit message that completes the two-phase checkpointing protocol.

- After receiving the commit message, a process removes the old permanent checkpoint and atomically makes the tentative checkpoint permanent and then resumes its execution and exchange of messages with other processes.

- A problem with this approach is that the computation is blocked during the checkpointing

Non-blocking checkpoint Coordination

- In this approach 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 requires all processes to participate in every checkpoint.

Non-blocking checkpoint Coordination contd…

Consider the example in Figure 13.6(a) : 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.

- 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,

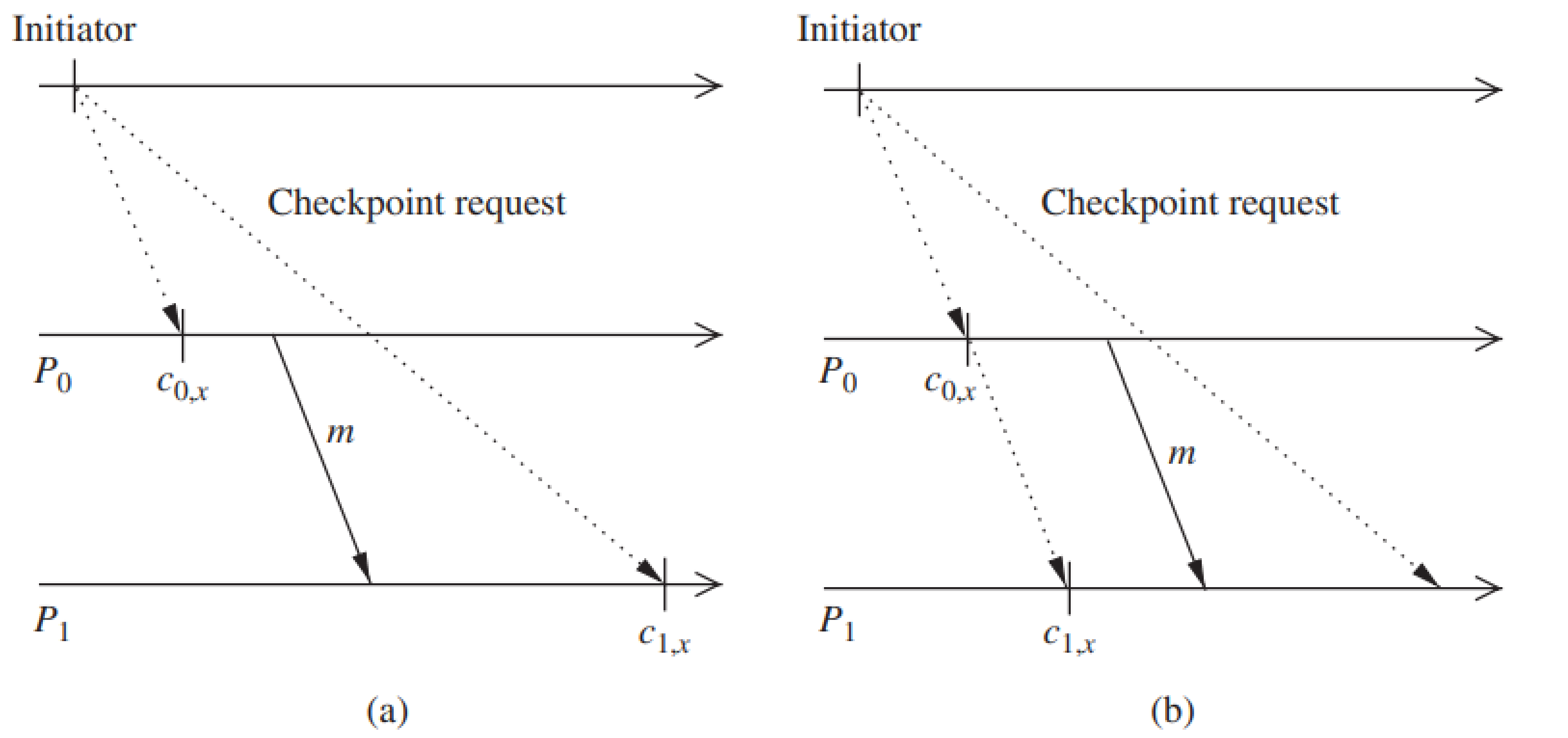


Figure 13.6 Non-blocking coordinated checkpointing: (a) checkpoint inconsistency; (b) a solution with FIFO channels [13].

3. Communication-induced checkpointing

- Communication-induced checkpointing is another way to avoid the domino effect, while allowing processes to take some of their checkpoints independently. Processes may be forced to take additional checkpoints.
- Communication-induced checkpointing reduces or completely eliminates the useless checkpoints.

communication-induced checkpointing contd…

- In communication-induced checkpointing, processes take two types of checkpoints, namely, autonomous and forced checkpoints.
- The checkpoints that a process takes independently are called local checkpoints, while those that a process is forced to take are called forced checkpoints.
- Communication-induced checkpointing piggybacks protocol-related information on each application message.

communication-induced checkpointing contd…

- 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, possibly incurring some latency and overhead.
- It is desirable in these systems to minimize the number of forced checkpoints.
- In contrast with coordinated checkpointing, no special coordination messages are exchanged

communication-induced checkpointing contd…

- There are two types of communication-induced checkpointing :
  - modelbased checkpointing and
  - index-based checkpointing.

- In model-based checkpointing, the system maintains checkpoints and communication structures that prevent the domino effect or achieve some even stronger properties.
- In index-based checkpointing, the system uses an indexing scheme for the local and forced checkpoints, such that the checkpoints of the same index at all processes form a consistent state