CST 402 – DISTRIBUTED COMPUTING

QUESTION – ANSWER BANK

Module – IV

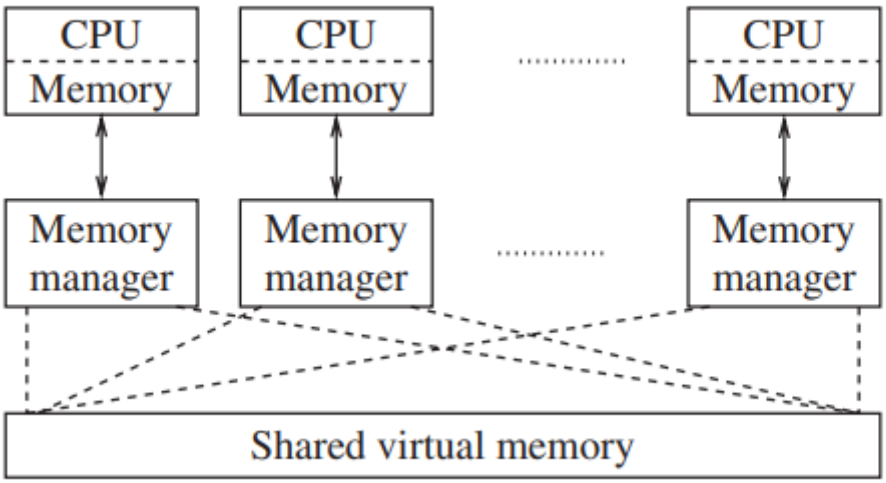
Distributed shared memory and Failure Recovery

PART A

Q1. State Distributed shared memory

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.
- 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.



Q2. State the Advantages of Distributed Shared memory.

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

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.

Q3. State the Disadvantages of Distributed Shared Memory.

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 messagepassing 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.

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

Q4. Explain the 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. One decision to be made is the degree of replication – partial replication at some sites, or full replication at all the sites. A further decision then is to decide on whether to use read-replication (replication for the read operations) or write-replication (replication for the write operations) or both.

• 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.

Q5. Describe Shared memory Mutual exclusion

Mutual exclusion is a concurrency control property which is introduced to prevent race conditions. It is the requirement that a process cannot enter its critical section while another concurrent process is currently present or executing in its critical section i.e only one process is allowed to execute the critical section at any given instance of time.

Mutual exclusion in single computer system Vs. distributed system:

In single computer system, memory and other resources are shared between different processes. The status of shared resources and the status of users is easily available in the shared memory so with the help of shared variable (For example: Semaphores) mutual exclusion problem can be easily solved.
In Distributed systems, we neither have shared memory nor a common physical clock and there for we can not solve mutual exclusion problem using shared variables. To eliminate the mutual exclusion problem in distributed system approach based on message passing is used. A site in distributed system do not have complete information of state of the system due to lack of shared memory and a common physical clock.

PART B

Q6. Explain Lamport's bakery algorithm

Lamport proposed the classical bakery algorithm for n-process mutual exclusion in shared memory systems
The algorithm is 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.
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.
In the entry section, a process chooses a timestamp for itself, and resets it to 0 in the exit section.
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.

(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*, $1 \leq i \leq n$.

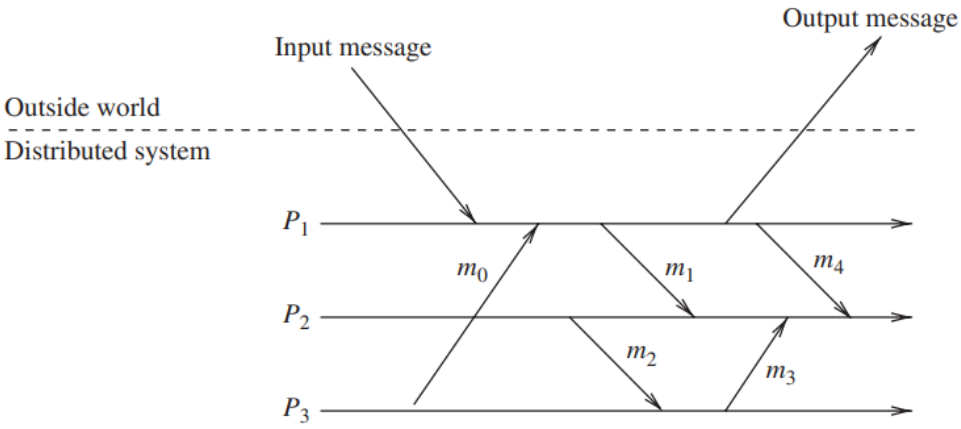
Q7 Explain Checkpointing and rollback recovery

- 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.
- In distributed systems, rollback recovery is complicated because 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, creating what is commonly called a rollback propagation
- 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.
- In a distributed system, if each participating process takes its checkpoints independently, then the system is susceptible to the domino effect.
- This approach is called independent or uncoordinated checkpointing.
- It is obviously desirable to avoid the domino effect and therefore several techniques have been developed to prevent it.
- One such technique is coordinated checkpointing where 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.
- Alternatively, communication-induced checkpointing forces each process to 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
- Logbased 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.

Q8. Explain 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



System model

- Rollback-recovery protocols generally make assumptions about the reliability of the inter-process communication.
- Some protocols assume that the communication subsystem delivers messages reliably, in first-in-first-out (FIFO) order, while other protocols assume that the communication subsystem 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
- System model
- Rollback-recovery protocols therefore must maintain information about the internal interactions among processes and also the external interactions with the outside world.

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.

System model

- 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
- A local checkpoint is shown in the process-line by the symbol " | ".
- 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.
- Intuitively, 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.

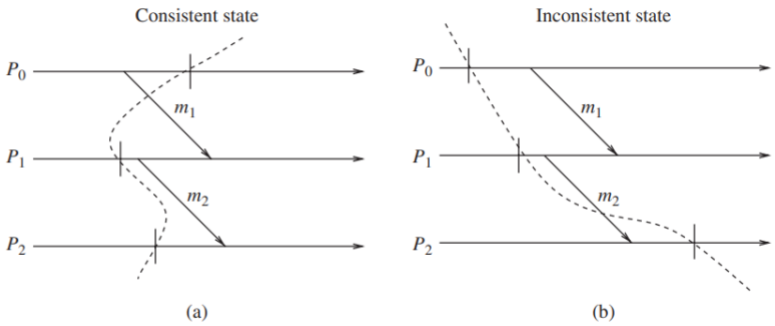


Figure 13.2 Examples of consistent and inconsistent states [13].

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

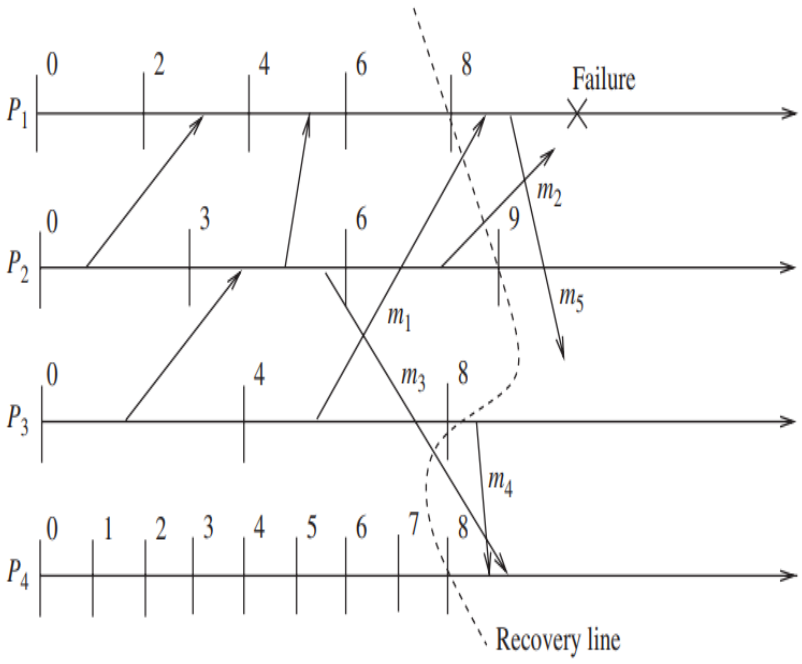
- Note that the consistent state in Figure 13.2(a) shows message m1 to have been sent but not yet received, but that is alright.
- The state in Figure 13.2(a) is consistent because it represents a situation in which every message that has been received, there is a corresponding message send event
- The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

Q9. 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.

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.

Figure 13.3 Different types of messages [25].



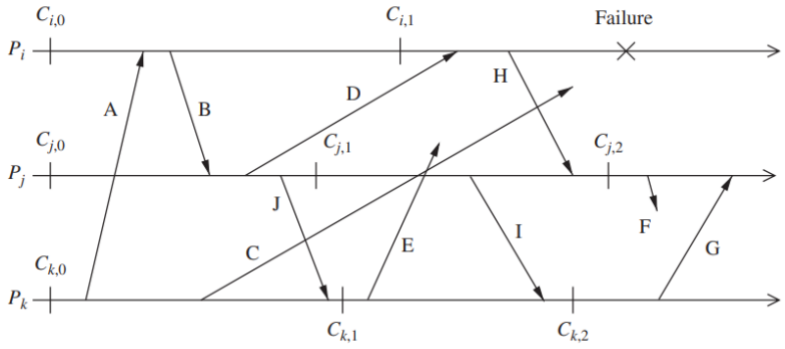
On the other hand, if a system model assumes lossy communication channels, then in-transit messages can be omitted from system state.

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.

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.

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.
- 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



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.

Q10. Explain the 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. 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
- 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.
- 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
- 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.
- 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

Q11. Log-based rollback recovery

- A log-based rollback recovery makes use of deterministic and nondeterministic events in a computation

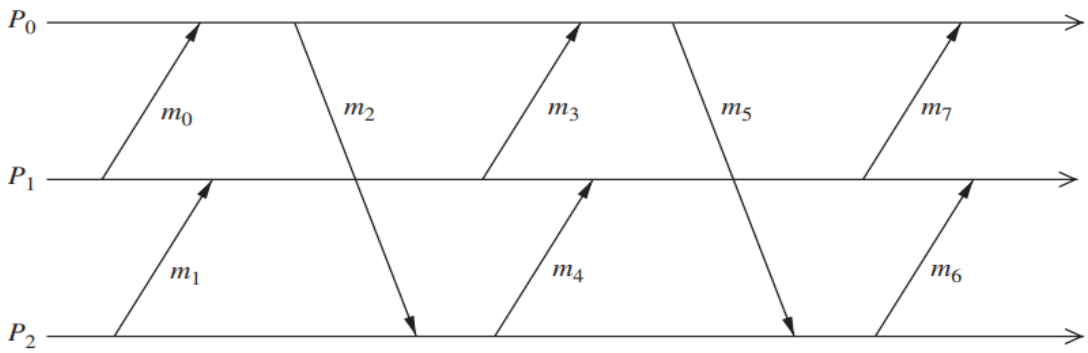
  Deterministic and non-deterministic events

  Log-based rollback recovery exploits the fact that a process execution can be modeled as a sequence of deterministic state intervals, each starting with the execution of a non-deterministic event.

  A non-deterministic event can be the receipt of a message from another process or an event internal to the process.

  For example, in Figure the execution of process P0 is a sequence of four deterministic intervals.

  The first one starts with the creation of the process, while the remaining three start with the receipt of messages m0, m3, and m7, respectively.



- Send event of message m2 is uniquely determined by the initial state of P0 and by the receipt of message m0, and is therefore not a non-deterministic event.
- 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.
- Additionally, each process also takes checkpoints to reduce the extent of rollback during recovery.
- After a failure occurs, the failed processes recover by using the checkpoints and logged determinants to replay the corresponding non-deterministic events precisely as they occurred during the pre-failure execution.
- Because execution within each deterministic interval depends only on the sequence of non-deterministic events that preceded the interval's beginning,
- The no-orphans consistency condition
- Let e be a non-deterministic event that occurs at process p. We define the following
- 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.

$$\forall(e) : \neg Stable(e) \Longrightarrow Depend(e) \subseteq Log(e).$$

This property is called the always-no-orphans condition

- Log-based rollback-recovery protocols guarantee that upon recovery of all failed processes, the system does not contain any orphan process,
  Log-based rollback-recovery protocols are of three types:
  1. pessimistic logging,
  2. optimistic logging,
  3. causal logging

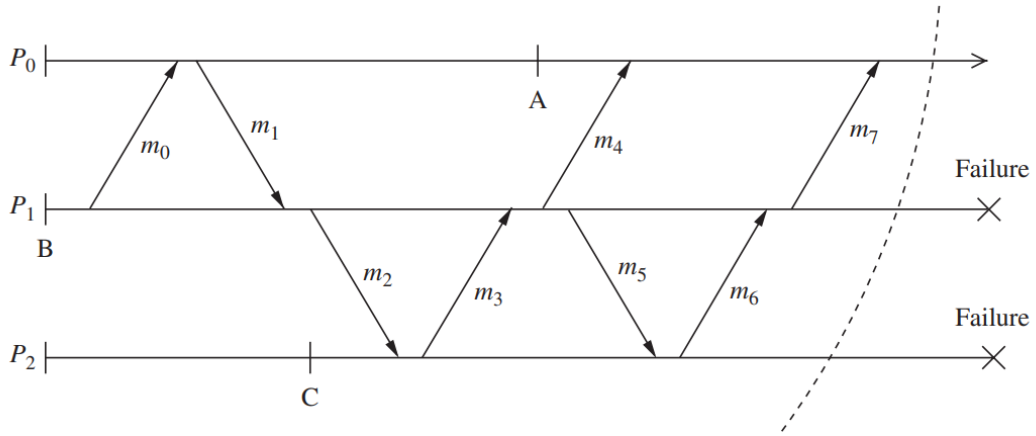
1. Pessimistic logging

- Pessimistic logging protocols assume that a failure can occur after any non-deterministic event in the computation.
- This assumption is "pessimistic" since in reality failures are rare
- In their most straightforward form, pessimistic protocols log to the stable storage the determinant of each non-deterministic event before the event affects the computation.
- Pessimistic protocols implement the following property, often referred to as synchronous logging, which is a stronger than the always-no-orphans condition

$$\forall e : \neg Stable(e) \Longrightarrow |Depend(e)| = 0.$$

- That is, if an event has not been logged on the stable storage, then no process can depend on it.
- In addition to logging determinants, processes also take periodic checkpoints to minimize the amount of work that has to be repeated during recovery.
- When a process fails, the process is restarted from the most recent checkpoint and the logged determinants are used to recreate the prefailure execution.
- in Figure During failure-free operation the logs of processes P0, P1, and P2 contain the determinants needed to replay messages m0, m4, m7, m1, m3, m6, and m2, m5, respectively.
- Suppose processes P1 and P2 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.

- This guarantees that P1 and P2 will repeat exactly their pre-failure execution and re-send the same messages.
- Hence, once the recovery is complete, both processes will be consistent with the state of P0 that includes the receipt of message m7 from P1.
- In a pessimistic logging system, the observable state of each process is always recoverable.



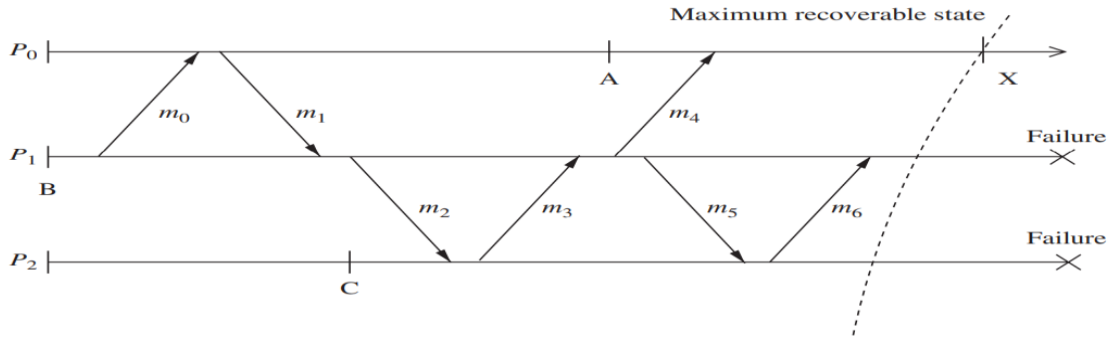
- fast non-volatile semiconductor memory can be used to implement the stable storage.
- Another approach is to limit the number of failures that can be tolerated.
- The overhead of pessimistic logging is reduced by delivering a message or executing an event and deferring its logging until the process communicates with another process or with the outside world.
- Some pessimistic logging systems reduce the overhead of synchronous logging without relying on hardware.
- For example, the sender-based message logging (SBML) protocol keeps the determinants corresponding to the delivery of each message m in the volatile memory of its sender.

2. Optimistic logging

- In optimistic logging protocols, processes log determinants asynchronously to the stable storage .
- These protocols optimistically assume that logging will be complete before a failure occurs.
- Determinants are kept in a volatile log, and are periodically flushed to the stable storage.
- Optimistic logging protocols do not implement the always-no-orphans condition.
- The protocols allow the temporary creation of orphan processes which are eventually eliminated.
- To perform rollbacks correctly, optimistic logging protocols track causal dependencies during failure free execution.
- Upon a failure, the dependency information is used to calculate and recover the latest global state of the pre-failure execution in which no process is in an orphan.

3. Causal 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.
- Moreover, causal logging limits the rollback of any failed process to the most recent checkpoint on the stable storage, thus minimizing the storage overhead and the amount of lost work.
- Causal logging protocols make sure that the always-no-orphans property holds by ensuring that the determinant of each non-deterministic event that causally precedes the state of a process is either stable or it is available locally to that process.



- Process P0 at state X will have logged the determinants of the nondeterministic events that causally precede its state according to Lamport's happened-before relation.
- These events consist of the delivery of messages m0, m1, m2, m3, and m4.
- The determinant of each of these non-deterministic events is either logged on the stable storage or is available in the volatile log of process P0.
- The determinant of each of these events contains the order in which its original receiver delivered the corresponding message.
- The message sender, as in sender-based message logging, logs the message content. Thus, process P0 will be able to "guide" the recovery of P1 and P2