

## DISTRIBUTED COMPUTING

### MODULE 4

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, as they would in a uniprocessor system. 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.

#### # DISTRIBUTED SHARED MEMORY ABSTRACTION

The DSM abstraction is illustrated in Figure 1. 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.

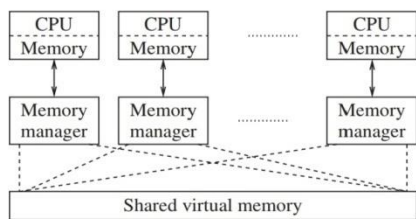


Fig 1- Abstract view of DSM

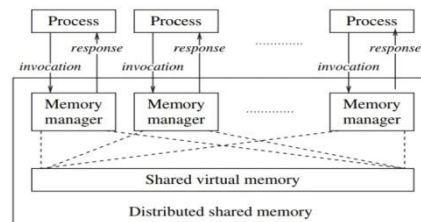


Fig 2- Detailed abstraction of DSM and interaction with application processes

#### DSM has the following advantages:

- Communication across the network is achieved by the read/write abstraction that simplifies the task of programmers.
- 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.
- If a block of data needs to be moved, the system can exploit locality of reference to reduce the communication overhead.
- DSM is often cheaper than using dedicated multiprocessor systems, because it uses simpler software interfaces and off-the-shelf hardware.
- There is no bottleneck presented by a single memory access bus.
- DSM effectively provides a large (virtual) main memory.
- 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.

Although a familiar (i.e., read/write) interface is provided to the programmer (see Figure 2) there is a catch to it. The data needs to be shared in some fashion. When multiple processors wish to access the same data object, a decision about how to handle concurrent accesses needs to be made. The main point of allowing concurrent access (by different processors) to the same data object is to increase throughput. But in concurrent access, the semantics of what value a read operation returns to the program needs to be specified. Programmers ultimately need to understand this semantics, which may differ from the Von Neumann semantics, because the program logic depends greatly on this semantics. This compromises the assumption that the DSM is transparent to the programmer.

#### DSM systems, we look at its disadvantages:

- 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.
- 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.

- 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 vanilla implementations of DSM have a higher overhead than a programmer-written implementation tailored for a specific application and 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.

#### 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 managers, by the operating system, or by the language runtime system.

#### # Shared Memory Mutual Exclusion: Lamport's Bakery Algorithm

---

```
(shared vars)
boolean: choosing[1..n];
integer: timestamp[1..n];
repeat
(1)  $P_i$  executes the following for the entry section:
(1a)  $choosing[i] \leftarrow 1$ ;
(1b)  $timestamp[i] \leftarrow \max_{k \in [1..n]}(timestamp[k]) + 1$ ;
(1c)  $choosing[i] \leftarrow 0$ ;
(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)  $timestamp[i] \leftarrow 0$ .
(4)  $P_i$  executes the remainder section after the exit section
until false;
```

---

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

---

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  $choosing[1..n]$ . 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 for process *i* is given above. 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. 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 interval may have overlapped with that of *i*, leading to an unknown order of timestamp values. Process *i* needs to make sure that any other process *j* (*j* < *i*) that had begun to execute line 1b concurrently with itself and may still be executing line 1b 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*. Once *j*'s timestamp is stabilized, *i* moves from line 1e to line 1f. Either *j* is not requesting (in which case *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.

*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.

The bakery algorithm have lead to several important results:

*Space complexity*: A lower bound of *n* registers, specifically, the timestamp array, has been shown for the shared memory critical section problem. Thus, one cannot hope to have a more space-efficient algorithm for distributed shared memory mutual exclusion.

*Time complexity*: (*n*) time for Bakery algorithm. Although this algorithm guarantees mutual exclusion and progress, unfortunately, this fast algorithm has a price – in the worst case, it does not guarantee bounded delay.

## # 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 can be saved on either the stable storage or the volatile storage depending on the failure scenarios to be tolerated.

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**. If each process takes its checkpoints independently, then the system cannot avoid the domino effect – this scheme is called independent or uncoordinated checkpointing.

### Techniques that avoid domino effect

1. **Coordinated checkpointing rollback recovery** - Processes coordinate their checkpoints to form a system-wide consistent state

2. **Communication-induced check pointing rollback recovery** - Forces each process to take checkpoints based on information piggybacked on the application.
3. **Log-based rollback recovery** - Combines check pointing with logging of nondeterministic events • relies on piecewise deterministic (PWD) assumption.

### System model

- A distributed system consists of a fixed number of processes,  $P_1, P_2, \dots, P_N$ , 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.
- Rollback-recovery protocols generally make assumptions about the reliability of the inter-process communication.
- Some protocols assume that the communication uses first-in-first-out (FIFO) order, while other protocols assume that the communication subsystem can lose, duplicate, or reorder messages.
- A generic correctness condition for rollback-recovery can be defined as follows: "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.

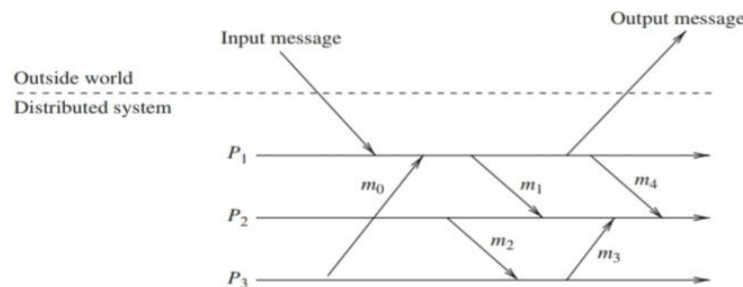


FIG: An example of a distributed system with three processes

### 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.

Assumptions that we make:

- A process stores all local checkpoints on the stable storage so that they are available even if the process crashes.
- 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  $k$ th 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 " $|$ ".

### Consistent system 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

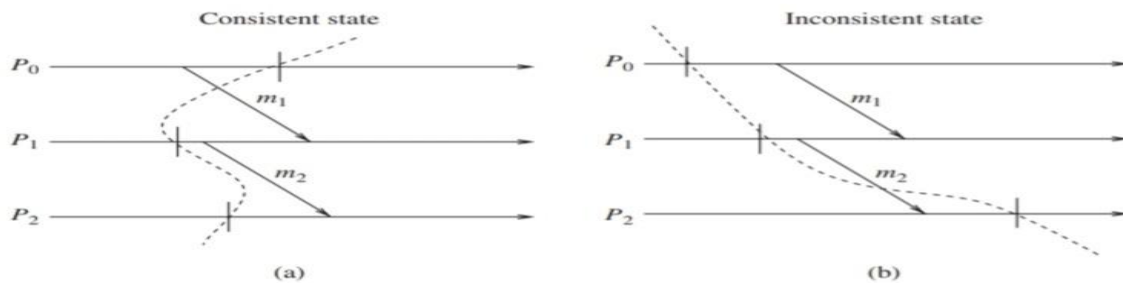
Consistent global state

- 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 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 point that is received by another process before taking its 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.



For instance, Figure shows two examples of global states.

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

Note that the consistent state in Figure (a) shows message  $m_1$  to have been sent but not yet received, but that is alright.

The state in Figure (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 (b) is inconsistent because process  $P_2$  is shown to have received  $m_2$  but the state of process  $P_1$  does not reflect having sent it.

Such a state is impossible in any failure-free, correct computation. Inconsistent states occur because of failures.

### Interactions with outside world

A distributed system 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. For example, a printer cannot roll back the effects of printing a character.

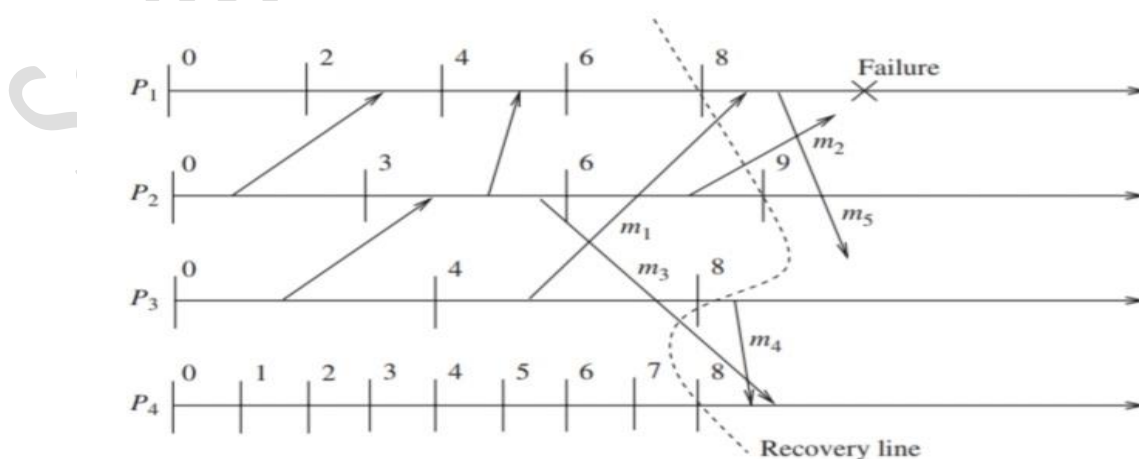
#### Outside World Process (OWP)

- It is a special process that interacts with the rest of the system through message passing.
- It is therefore necessary that the outside world see a consistent behavior of the system despite failures.
- Thus, 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.

A common approach is to save each input message 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. Process  $P_1$  fails at the point indicated and the whole system recovers to the state indicated by the recovery line; that is, to global state  $\{C_{1,8}, C_{2,9}, C_{3,8}, C_{4,8}\}$ .



### @ In-transit messages

#### Messages that have been sent but not yet received

In Figure 13.3, the global state  $\{C_{1,8}, C_{2,9}, C_{3,8}, C_{4,8}\}$  shows that message  $m_1$  has been sent but not yet received. We call such a message an *in-transit* message. Message  $m_2$  is also an in-transit message. When in-transit messages are part of a global system state, these messages do not cause any inconsistency. 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.

### @ Lost messages

#### Messages whose "send" is done but "receive" is undone due to rollback.

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, message  $m_1$  is a lost message.

### @ 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  $m_2$  and  $m_5$  in Figure are delayed messages.

### @ 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.

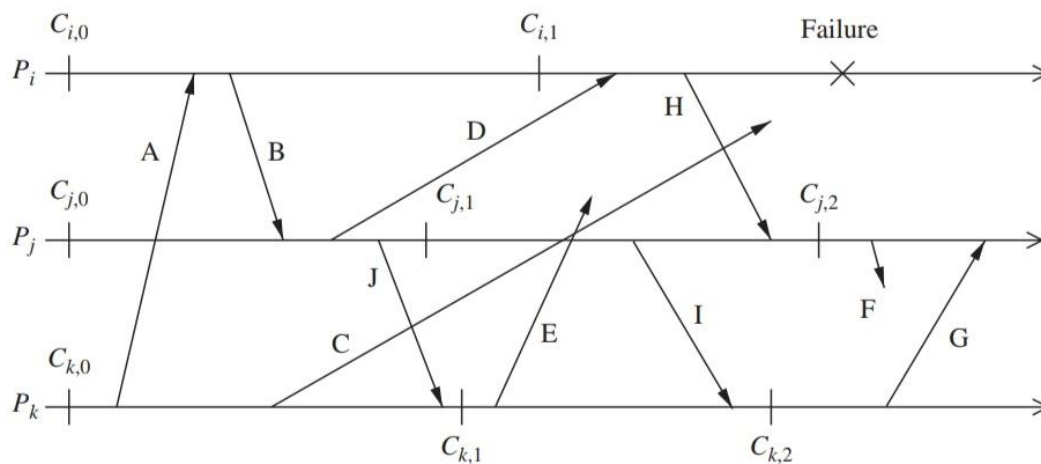
### @ Duplicate messages

Duplicate messages arise due to message logging and replaying during process recovery. For example, in Figure, message  $m_4$  was sent and received before the rollback. However, due to the rollback of process  $P_4$  to  $C_{4,8}$  and process  $P_3$  to  $C_{3,8}$ , both send and receipt of message  $m_4$  are undone. When process  $P_3$  restarts from  $C_{3,8}$ , it will resend message  $m_4$ . Therefore,  $P_4$  should not replay message  $m_4$  from its log. If  $P_4$  replays message  $m_4$ , then message  $m_4$  is called a *duplicate* message.

Message  $m_5$  is an excellent example of a duplicate message. No matter what, the receiver of  $m_5$  will receive a duplicate  $m_5$  message.

### 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.



- The computation comprises of three processes  $P_i$ ,  $P_j$ , and  $P_k$ , connected through a communication network. The processes communicate solely by exchanging messages over fault free, FIFO communication channels.
- Processes  $P_i$ ,  $P_j$ , and  $P_k$ , have taken checkpoints  $\{C_{i,0}, C_{i,1}\}$ ,  $\{C_{j,0}, C_{j,1}, C_{j,2}\}$ , and  $\{C_{k,0}, C_{k,1}\}$ , respectively, and these processes have exchanged messages A to J



- Suppose process  $P_i$  fails at the instance indicated in the figure. All the contents of the volatile memory of  $P_i$  are lost and, after  $P_i$  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  $P_i$ 's state is restored to a valid state by rolling it back to its most recent checkpoint  $C_{i,1}$ . To restore the system to a consistent state, the process  $P_j$  rolls back to checkpoint  $C_{j,1}$  because the rollback of process  $P_i$  to checkpoint  $C_{i,1}$  created an orphan message  $H$  (the receive event of  $H$  is recorded at process  $P_j$  while the send event of  $H$  has been undone at process  $P_i$ ).
- $P_j$  does not roll back to checkpoint  $C_{j,2}$  but to checkpoint  $C_{j,1}$ . An orphan message  $I$  is created due to the roll back of process  $P_j$  to checkpoint  $C_{j,1}$ . To eliminate this orphan message, process  $P_k$  rolls back to checkpoint  $C_{k,1}$ .
- **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  $P_i$  before it recovers, it might arrive while  $P_i$  is recovering, or it might arrive after  $P_i$  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  $P_j$ , but the receive event has been undone at process  $P_i$ . Process  $P_j$  will not resend  $D$  without an additional mechanism.
- **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.
- 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.**
- **Overlapping failures further complicate the recovery process.** If overlapping failures are to be tolerated, a mechanism must be introduced to deal with amnesia and the resulting inconsistencies.

## # 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.
- Checkpoint-based rollback-recovery techniques can be classified into three categories: *uncoordinated checkpointing*, *coordinated checkpointing*, and *communication-induced checkpointing*

### @Uncoordinated checkpointing

- In uncoordinated checkpointing, each process has autonomy in deciding when to take checkpoints.
- This eliminates the synchronization overhead as there is no need for coordination between processes and it allows processes to take checkpoints when it is most convenient or efficient.
- Autonomy in taking checkpoints also allows each process to select appropriate checkpoints positions.

#### Advantages

- Lower runtime overhead during normal execution, because no coordination among processes is necessary.

#### Disadvantages

- Possibility of the domino effect during a recovery, which may cause the loss of a large amount of useful work.
- Recovery from a failure is slow because processes need to iterate to find a consistent set of checkpoints.
- Uncoordinated checkpointing forces each process to maintain multiple checkpoints, and to periodically invoke a garbage collection algorithm to reclaim the checkpoints.
- Not suitable for application with frequent output commits

In order to determine a consistent global checkpoint during recovery, the processes record the dependencies among their checkpoints caused by message exchange during failure-free operation. The following direct dependency tracking technique is commonly used in uncoordinated checkpointing.

**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 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  $C_{j,y}$ .

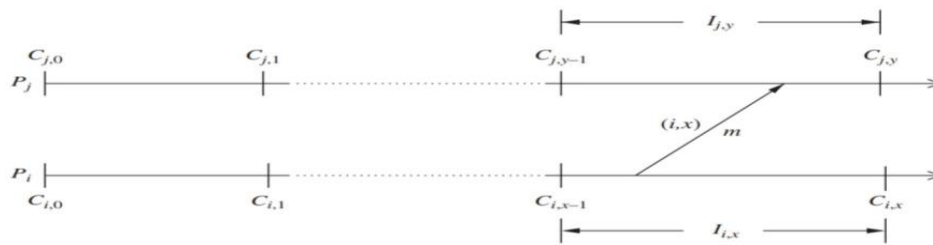


Fig: Checkpoint index and checkpoint interval

- 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.

**@Coordinated checkpointing**

- In coordinated checkpointing, processes orchestrate their checkpointing activities so that all local checkpoints form a consistent global state

**Advantages**

- It is not susceptible to the domino effect, since every process always restarts from its most recent checkpoint.
- Coordinated checkpointing requires each process to maintain only one checkpoint on the stable storage, reducing the storage overhead and eliminating the need for garbage collection.

**Disadvantages**

- Large latency is involved in committing output, as a global checkpoint is needed before a message is sent to the OWP.
- Delays and overhead are involved everytime a new global checkpoint is taken.

The following approaches are used to guarantee checkpoint consistency in coordinated checkpointing: either the sending of messages is blocked for the duration of the protocol, or checkpoint indices are piggybacked to avoid blocking.

**Blocking coordinated checkpointing:**

A straightforward approach to coordinated checkpointing is 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. 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 and therefore, non-blocking checkpointing schemes are preferable.

#### Non-blocking checkpoint coordination:

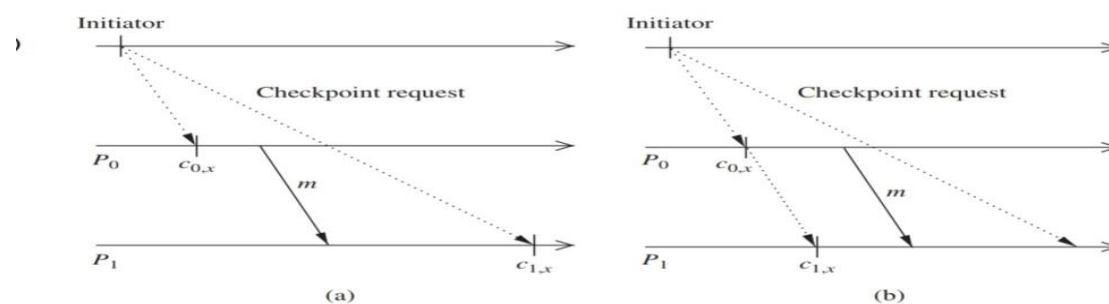
The processes need not stop their execution while taking checkpoints. A fundamental problem in coordinated checkpointing is to prevent a process from receiving application messages that could make the checkpoint inconsistent.

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



#### Impossibility of min-process non-blocking checkpointing:

- A min-process, non-blocking checkpointing algorithm is one that forces only a minimum number of processes to take a new checkpoint, and at the same time it does not force any process to suspend its computation.

#### Algorithm

- The algorithm consists of two phases. During the first phase, the checkpoint initiator identifies all processes with which it has communicated since the last checkpoint and sends them a request.
- Upon receiving the request, each process in turn identifies all processes it has communicated with since the last checkpoint and sends them a request, and so on, until no more processes can be identified
- During the second phase, all processes identified in the first phase take a checkpoint. The result is a consistent checkpoint that involves only the participating processes.
- In this protocol, after a process takes a checkpoint, it cannot send any message until the second phase terminates successfully, although receiving a message after the checkpoint has been taken is allowable.

#### @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 (over and above their autonomous checkpoints), and thus process independence is constrained to guarantee the eventual progress of the recovery line.
- Communication-induced checkpointing reduces or completely eliminates the useless checkpoints.
- 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
- 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 check pointing, no special coordination messages are exchanged

Two types of communication-induced checkpointing

### 1. Model-based checkpointing

- Model-based checkpointing prevents patterns of communications and checkpoints that could result in inconsistent states among the existing checkpoints.
- A process detects the potential for inconsistent checkpoints and independently forces local checkpoints to prevent the formation of undesirable patterns.
- No control messages are exchanged among the processes during normal operation. All information necessary to execute the protocol is piggybacked on application messages. The decision to take a forced checkpoint is done locally using the information available.
- There are several domino-effect-free checkpoint and communication model.
- The MRS (mark, send, and receive) model of Russell avoids the domino effect by ensuring that within every checkpoint interval all message receiving events precede all message-sending events.

### 2. Index-based checkpointing

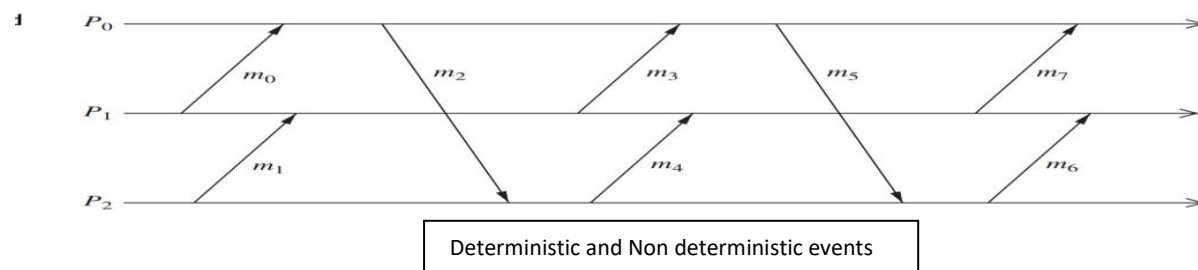
- Index-based communication-induced checkpointing assigns monotonically increasing indexes to checkpoints, such that the checkpoints having the same index at different processes form a consistent state.
- Inconsistency between checkpoints of the same index can be avoided in a lazy fashion if indexes are piggybacked on application messages to help receivers decide when they should take a forced a checkpoint.

## # 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. Note that a message send event is *not* a non-deterministic event.
- For example, in Figure, the execution of process  $P_0$  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  $m_0$ ,  $m_3$ , and  $m_7$ , respectively.
- Send event of message  $m_2$  is uniquely determined by the initial state of  $P_0$  and by the receipt of message  $m_0$ , 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.
- Determinant: the information need to “replay” the occurrence of a non-deterministic event (e.g., message reception).
- 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.
- Because execution within each deterministic interval depends only on the sequence of non-deterministic events that preceded the interval’s beginning, the pre-failure execution of a failed process can be reconstructed during recovery up to the first non-deterministic event whose determinant is not logged.



### 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.

Suppose a set of processes  $\Psi$  crashes. A process  $p$  in  $\Psi$  becomes an orphan when  $p$  itself does not fail and  $p$ 's state depends on the execution of a nondeterministic event  $e$  whose determinant cannot be recovered from the stable storage or from the volatile memory of a surviving process. Formally, it can be stated as follows:

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

This property is called the **always-no-orphans condition**. It states that if any surviving process depends on an event  $e$ , then either event  $e$  is logged on the stable storage, or the process has a copy of the determinant of event  $e$ . If neither condition is true, then the process is an orphan because it depends on an event  $e$  that cannot be generated during recovery since its determinant is lost.

Log-based rollback-recovery protocols guarantee that upon recovery of all failed processes, the system does not contain any orphan process, i.e., a process whose state depends on a non-deterministic event that cannot be reproduced during recovery.

Log-based rollback-recovery protocols are of three types: pessimistic logging, optimistic logging, and causal logging protocols.

### @ Pessimistic logging

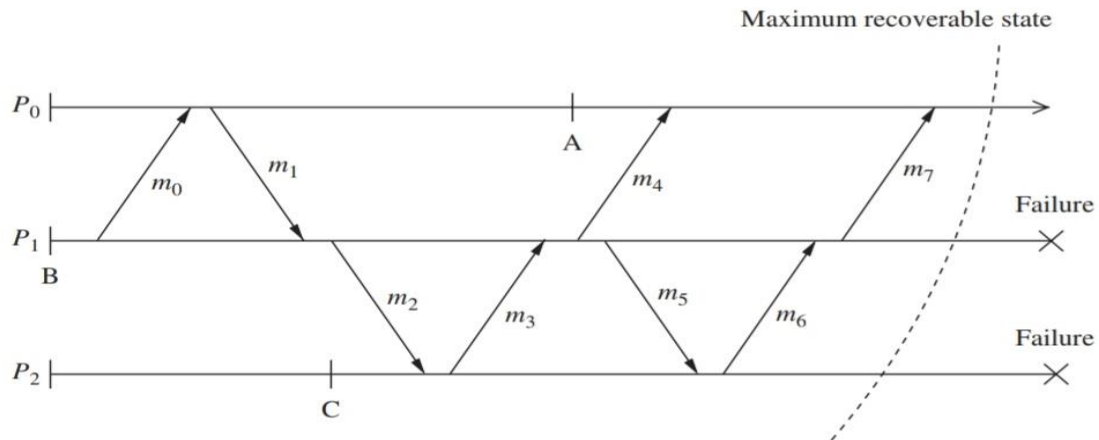
Pessimistic logging protocols assume that a failure can occur after any non-deterministic event in the computation.

This assumption is "pessimistic" since in reality failures are rare. Pessimistic protocols implement the following property, often referred to as *synchronous logging*, which is a stronger than the always-no-orphans condition:

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

That is, if an event has not been logged on the stable storage, then no process can depend on it.

Consider the example in Figure. During failure-free operation the logs of processes  $P_0$ ,  $P_1$ , and  $P_2$  contain the determinants needed to replay messages  $m_0$ ,  $m_4$ ,  $m_7$ ,  $m_1$ ,  $m_3$ ,  $m_6$ , and  $m_2$ ,  $m_5$ , respectively. 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. This guarantees that  $P_1$  and  $P_2$  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  $P_0$  that includes the receipt of message  $m_7$  from  $P_1$ . In a pessimistic logging system, the observable state of each process is always recoverable.



Disadvantage: performance penalty for synchronous logging

Advantages:

- immediate output commit
- restart from most recent checkpoint
- recovery limited to failed process(es)
- simple garbage collection

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.

The **sender-based message logging (SBML) protocol**

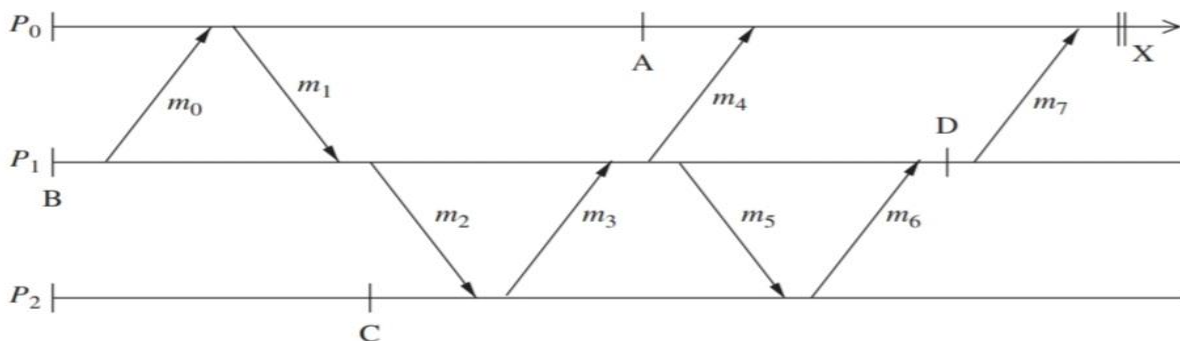
Two steps.

1. First, before sending  $m$ , the sender logs its content in volatile memory.
2. Then, when the receiver of  $m$  responds with an acknowledgment that includes the order in which the message was delivered, the sender adds to the determinant the ordering information.

SBML avoids the overhead of accessing stable storage but tolerates only one failure and cannot handle non-deterministic events internal to a process.

### @ 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.
- Thus, optimistic logging does not require the application to block waiting for the determinants to be written to the stable storage, and therefore incurs much less overhead during failure-free execution.
- Optimistic logging protocols do not implement the always-no-orphans condition.
- The protocols allow the temporary creation of orphan processes which are eventually eliminated. The *always-no-orphans* condition holds after the recovery is complete. This is achieved by rolling back orphan processes until their states do not depend on any message whose determinant has been lost.
- To perform rollbacks correctly, optimistic logging protocols track causal dependencies during failure free execution.
- Optimistic logging protocols require a non-trivial garbage collection 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



Consider the example shown in Figure. Suppose process  $P_2$  fails before the determinant for  $m_5$  is logged to the stable storage. Process  $P_1$  then becomes an orphan process and must roll back to undo the effects of receiving the orphan message  $m_6$ . The rollback of  $P_1$  further forces  $P_0$  to roll back to undo the effects of receiving message  $m_7$ .

Advantage: better performance in failure-free execution

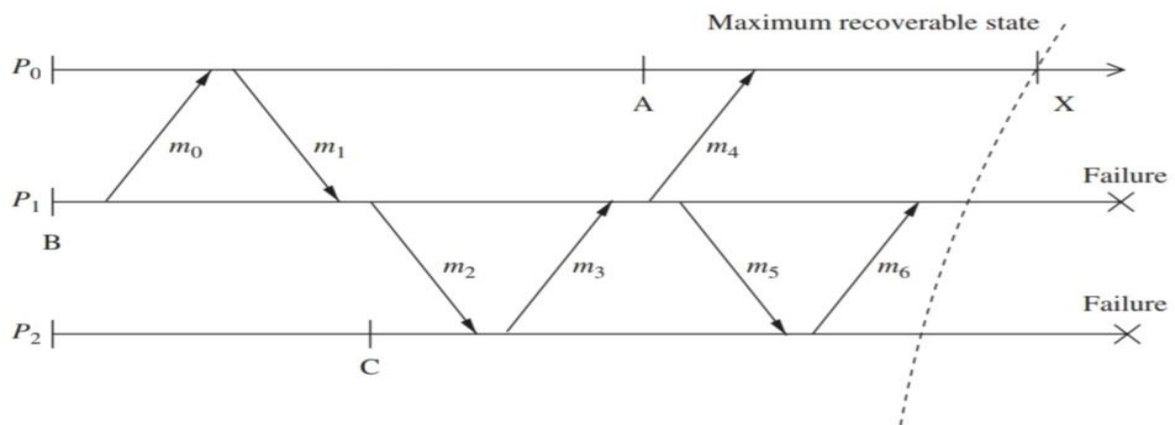
Disadvantages:

- coordination required on output commit
- more complex garbage collection

Since determinants are logged asynchronously, output commit in optimistic logging protocols requires a guarantee that no failure scenario can revoke the output. For example, if process P<sub>0</sub> needs to commit output at state X, it must log messages m<sub>4</sub> and m<sub>7</sub> to the stable storage and ask P<sub>2</sub> to log m<sub>2</sub> and m<sub>5</sub>. In this case, if any process fails, the computation can be reconstructed up to state X.

### @ 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 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.
- Each process maintains information about all the events that have causally affected its state
- 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.



- Consider the example in Figure. Messages m<sub>5</sub> and m<sub>6</sub> are likely to be lost on the failures of P<sub>1</sub> and P<sub>2</sub> at the indicated instants. Process
- P<sub>0</sub> 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 m<sub>0</sub>, m<sub>1</sub>, m<sub>2</sub>, m<sub>3</sub>, and m<sub>4</sub>.
- 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 P<sub>0</sub>.
- 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 P<sub>0</sub> will be able to "guide" the recovery of P<sub>1</sub> and P<sub>2</sub> since it knows the order in which P<sub>1</sub> should replay messages m<sub>1</sub> and m<sub>3</sub> to reach the state from which P<sub>1</sub> sent message m<sub>4</sub>.
- Similarly, P<sub>0</sub> has the order in which P<sub>2</sub> should replay message m<sub>2</sub> to be consistent with both P<sub>0</sub> and P<sub>1</sub>.
- The content of these messages is obtained from the sender log of P<sub>0</sub> or regenerated deterministically during the recovery of P<sub>1</sub> and P<sub>2</sub>.
- Note that information about messages m<sub>5</sub> and m<sub>6</sub> is lost due to failures. These messages may be resent after recovery possibly in a different order.
- However, since they did not causally affect the surviving process or the outside world, the resulting state is consistent.
- Each process maintains information about all the events that have causally affected its state.

\*\*\*\*\*

Savion Manuel @EduSmash