KTU NOTES
The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

🌐 Website: www.ktunotes.in

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.
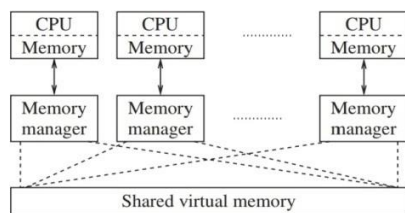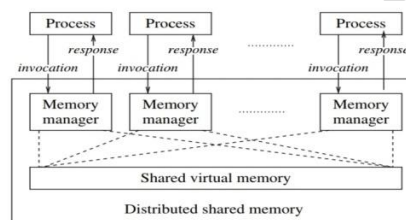

Fig 1- Abstract view of DSM

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] ⟵ 1;
(1b)   timestamp[i] ⟵ max_{k∈[1...n]}(timestamp[k]) + 1;
(1c)   choosing[i] ⟵ 0;
(1d)   for count = 1 to n do
(1e)       while choosing[count] do no-op;
(1f)       while timestamp[count] ≠ 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] ⟵ 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 Pi, $1 \leq i \leq n$.

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 check pointing.
Techniques that avoid domino effect
1. Coordinated check pointing 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, 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.
- 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.
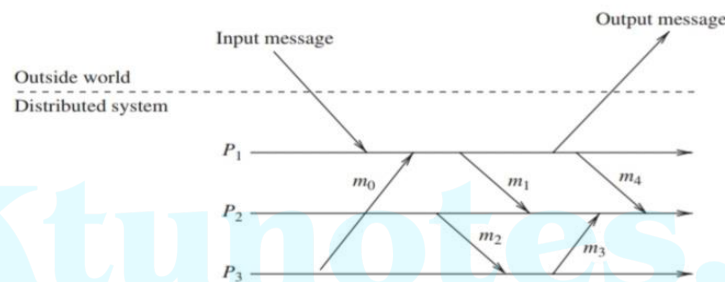

FIG: An example of a distributed system with three processes

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 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 "|".

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
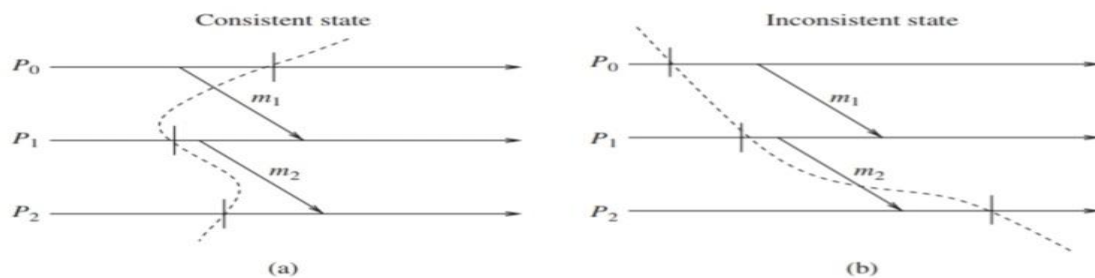
Consistent global state

- 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 m1 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 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.

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 P1 fails at the point indicated and the whole system recovers to the state indicated by the recovery line; that is, to global state $\{C_{1,8}, C_{2,9}, C_{3,8}, C_{4,8}\}$.
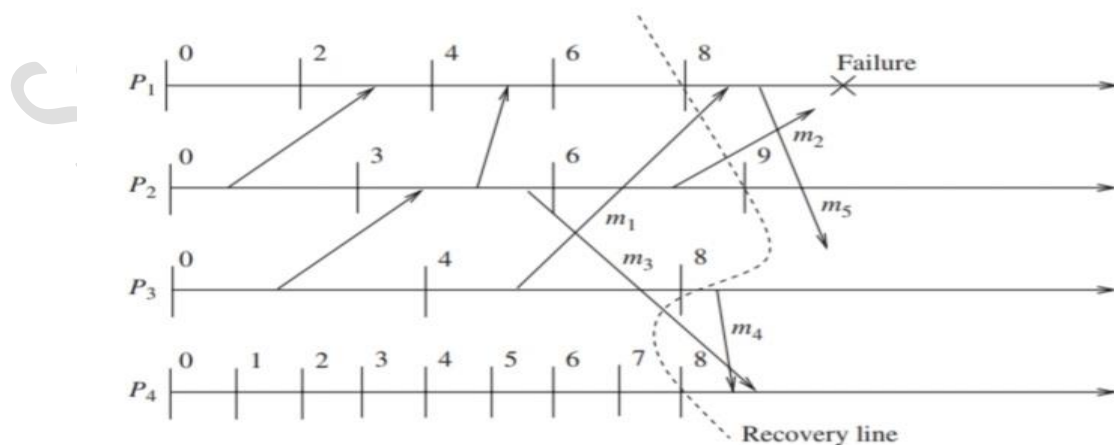
@ In-transit messages

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 $m1$ has been sent but not yet received. We call such a message an *in-transit* message. Message $m2$ 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 m1 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 m2 and m5 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 m4 was sent and received before the rollback. However, due to the rollback of process P4 to C4,8 and process P3 to C3,8, both send and receipt of message m4 are undone. When process P3 restarts from C3,8, it will resend message m4. Therefore, P4 should not replay message m4 from its log. If P4 replays message m4, then message m4 is called a *duplicate* message.

Message m5 is an excellent example of a duplicate message. No matter what, the receiver of m5 will receive a duplicate m5 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.
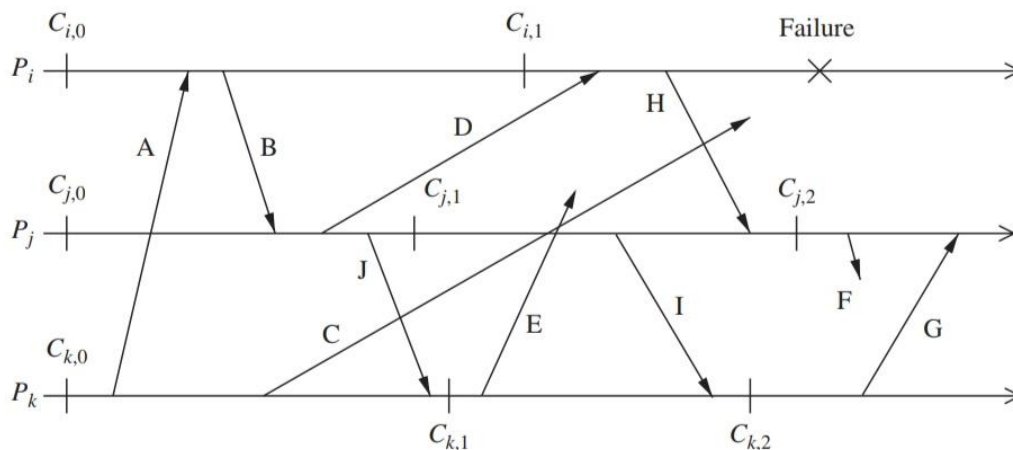
- The computation comprises of three processes P_i, P_j , and P_k, connected through a communication network. The processes communicate solely by exchanging messages over fault free, FIFO communication channels.
- Processes P_i, P_j , and P_k, have taken checkpoints $\{C_{i,0}, C_{i,1}\}$, $\{C_{j,0}, C_{j,1}, C_{j,2}\}$, and $\{C_{k,0}, C_{k,1}\}$, respectively, and these processes have exchanged messages A to J

- Suppose process 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 $C_{i,1}$. To restore the system to a consistent state, the process Pj rolls back to checkpoint $C_{j,1}$ because the rollback of process Pi to checkpoint $C_{i,1}$ created an orphan message H (the receive event of H is recorded at process Pj while the send event of H has been undone at process Pi).
- Pj 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 Pj to checkpoint $C_{j,1}$. To eliminate this orphan message, process Pk 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 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.
- 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.