DISTRIBUTED SHARED MEMORY

Abstraction and its advantages

Distributed Shared Memory is a resource management component of a distributed operating system that implements the shared memory model in distributed systems, which have no physically shared memory. The shared memory model provides a virtual address space that is shared among all computers in a distributed system.

- It is an abstraction provided to the programmer of a distributed system.
- It gives the impression of a single memory. 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.

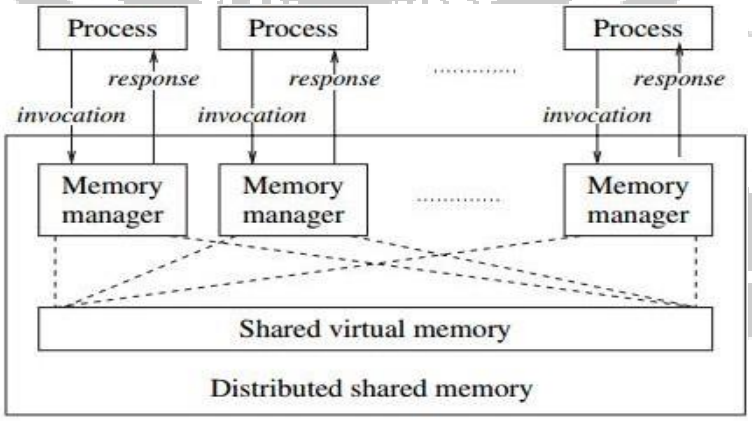


Fig : Abstract view of Distributed Shared MemoryAdvantages of DSM

- 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 economical 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

- When multiple processors wish to access the same data object, a decision about how tohandle concurrent accesses needs to be made. If concurrent access is permitted by different processors to different replicas, the problem of replica consistency needs to be addressed.

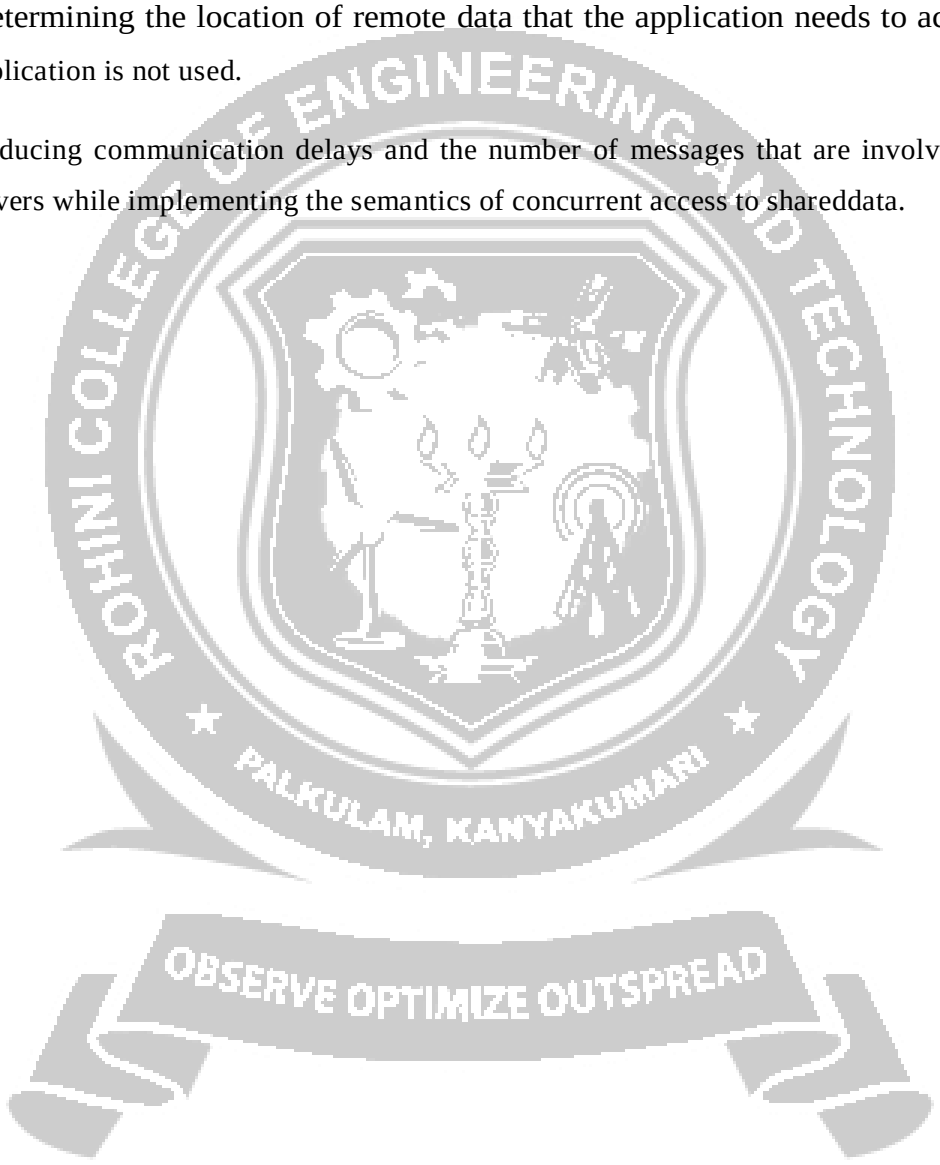
Challenges in implementing replica coherency in DSM systems

1. Programmers are aware of the availability of replica consistency models and from coding their distributed applications according to the semantics of these models.
2. As DSM is implemented as asynchronous message passing, it faces the overhead of asynchronous synchronization.
3. Since the control is given to memory management, the programmers lose the ability to use their own message-passing solutions for accessing shared objects.

Issues in designing a DSM system:

- Determining the semantics to allow for concurrent access to shared objects.

- Determining the best way to implement the semantics of concurrent access to shared data either to use read or write replication.
- Selecting the locations for replication 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 shareddata.



MEMORY CONSISTENCY MODELS

A memory consistency model is a set of rules which specify when a written value by one thread can be read by another thread.

- These rules are essential to write a correct program**.**
- **Memory coherence** is the ability of the system to execute memory operations correctly.
- The problem of ensuring memory coherence is identifying which of theinterleaving are correct, which of course requires a clear definition of correctness.
- The memory consistency model defines the set of allowable memory access orderings.
- In DSM system, the programmers write their programs keeping in mind the allowable interleaving permitted by that specific memory consistency model.
- A program written for one model may not work correctly on a DSM system that enforces a different model.
- The model can thus be viewed as a contract between the DSM system and the programmer using that system.
- The memory consistency model affects:
  i) System implementation: hardware, OS, languages, compilers
  ii) Programming correctness
  iii) Performance

Strict consistency, atomic consistency, linearizability

- According to Von Neumann architecture/ uniprocessor machine, any Read operation to a location should return the value or variable written by the most recent Write to that location or a variable.
- The system built over the above principle is called strict or atomic consistency model.

- The features of the atomic consistency model area:

i)Common global time axis is implicitly available in a uniprocessor system

ii) The write operation is immediately visible to all processes

Atomic Consistency Model:

i) Any Read to a location is required to return the value written by the most recent Write to that location in accordance with global time reference. For non overlapping operations, with respect to the global time reference, the specification is clear. For overlapping operations the following further specifications are necessary.

ii) All operations appear to be executed atomically and sequentially.

iii) All processors see the same ordering of events, which is equivalent to the global-time occurrence of non-overlapping events.

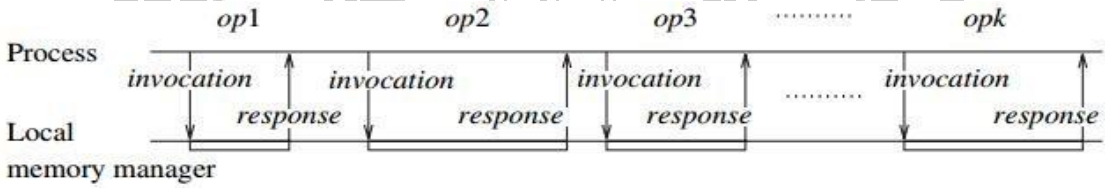


Fig : Invocations and responses in sequential system

The invocation and the response to each invocation can be viewed as being atomic events. An execution sequence in global time is viewed as a sequence Seq of such invocationsand responses. The Seq must satisfy the following conditions:

- **Liveness:** Each invocation must have a corresponding response.
- **Correctness:** The projection of Seq on any processor i, denoted Seqi, must be a sequence of alternating invocations and responses if pipelining is disallowed.

A linearizable execution needs to generate an equivalent global order on the eventsthat is a permutation of Seq, satisfying the semantics of linearizability.

Linearizable property:

A sequence Seq of invocations and responses is linearizable (LIN) if there is a permutation Seq' of adjacent pairs of corresponding (invoc, resp) events satisfying:

1. ***For every variable v, the projection of Seq' on v, denoted Seqv' , is such that every Read (adjacent (invoc, resp) event pair) returns the most recent Write (adjacent, (nvoc, resp) event pair) that immediately preceded it.***
2. ***If the response op1(resp) of operation op1 occurred before the invocation op2(invoc) of operation op2 in Seq, then op1 (adjacent (invoc, resp)event pair) occurs before op2 (adjacent (invoc, resp) event pair) in Seq.***

- Linearizability is a guarantee about single operations on single objects.
- It provides a real- guarantee on the behavior of a set of single operations on a single object.

Linearizability requires that each operation appears to occur atomically at some point between its invocation and completion. This point is called the linearization point.

Implementation of Linearizability

- Implementing linearizability is expensive because a global time scale needs to besimulated.
- As all processors need to agree on a common order, the implementation needs touse total order.
- For simplicity, the algorithm described here assumes full replication of each dataitem at all the processors.
- This demands the total ordering to be combined with a broadcast.
- The memory manager software is placed between the application above it and thetotal order broadcast layer below it.

(shared var)

int: x:

(1) When the memory manager receives a Read or Write from application:

(1a) **total_order_broadcast** the Read or Write request to all processors;(1b) **await** own request that was broadcast;

(1c) **perform** pending response to the application as follows

(1d) **case** Read: return value from local replica;

(1e) **case** Write: write to local replica and return ack to application.

(2) When the memory manager receives a **total_order_broadcast**(Write, x, val) from network;

(2a) **write** val to local replica of x.

(3) When the memory manager receives a **total_order_broadcast**(Read, x) from network;

(3a) **no operation**,

Fig : Implementing Linearizability

The algorithm in Fig. ensures total order broadcast such that all processors follow the same order:

1. For two non-overlapping operations at different processors, the response to the former operation precedes the invocation of the latter in global time.
2. For two overlapping operations, the total order ensures a common view by all processors.

Sequential Consistency

Sequential consistency requires that a shared memory multiprocessor appear to be a multiprogramming uniprocessor system to any program running on it.

Sequential consistency requires that:

1. All instructions are executed in order.
2. Every write operation becomes instantaneously visible throughout the system.

The main motivation behind sequential consistency is that the atomic consistency is very difficult to implement since the it is very difficult for a system to synchronize to global clock. Sequential consistency is specified as follows:

- The result of any execution is the same as if all operations of the processorswere executed in some sequential order.
- The operations of each individual processor appear in this sequence in the localprogram order.

Sequential Consistency:

A sequence Seq of invocation and response events is sequentially consistent if there is a permutation Seq' of adjacent pairs of corresponding (invoc, resp) events satisfying:

1. ***For every variable v, the projection of Seq' on v, denoted Seqv' , is such that every Read (adjacent, (invoc,resp) event pair) returns the most recent Write (adjacent, (invoc, resp) event pair) that immediately preceded it.***
2. ***If the response op1(resp) of operation op1 at process Pi occurred before the invocation op2(invoc) of operation op2 by process Pi in Seq, then op1 (adjacent (invoc, resp) event pair) occurs before op2 (adjacent (invoc,resp) event pair) in Seq.***

Implementation of Sequential Consistency

All processors are required to see the same global order, but global time ordering neednot be preserved across processes. So it is sufficient to use total order broadcasts for the Write operations only. In the simplified algorithm, no total order broadcast is required for Read operations, because:

1. all consecutive operations by the same processor are ordered in the same orderbecause pipelining is not used.
2. Read operations by different processors are independent of each other and need tobe ordered only with respect to the Write operations in the execution.

(shared var)

int: x:

(1) When the memory manager receives a Read or Write from application:

(1a) **case** Read: **return** value from local replica;

(1b) **case** Write(x, val): **total_order_broadcast**$_i$(Write(x, val)) to all processors including itself.

(2) When the memory manager at Pi receives a **total_order_broadcasts**$_j$(write, x, val) from network;

(2a) **Write** val to local replica of x;

(2b) **if** i=j **then return** acknowledgement to application.

Fig : Sequential Consistency using Local Read algorithm

Local-read algorithm

- A Read operation completes atomically, whereas a Write operation does not.
- Between the invocation of a Write by Pi (line 1b) and its c knowledgement (lines 2a, 2b), there may be multiple Write operations initiated by other processors that take effect at Pi (line 2a).
- Thus, a Write issued locally has its completion locally delayed. Such an algorithmis acceptable for Read intensive programs.

Local-write algorithm

- This does not delay acknowledgement of Writes.
- For Write intensive programs, it is desirable that a locally issued Write gets acknowledged immediately even though the total order broadcast for the Write, and the actual update for the Write may not go into effect by updating the variableat the same time.
- The algorithm achieves this at the cost of delaying a Read operation by a processor until all previously issued local Write operations by that same processor have locally gone into effect.
- The variable counter is used to track the number of Write operations that have beenlocally initiated but not completed at any time.
- A Read operation completes only if there are no prior locally initiated Write operations that have not written to their variables.
- Else, a Read operation is delayed until after all previously initiated Write operations have written to their local variables, which happens after the total orderbroadcasts associated with the Write have delivered the broadcast message locally.

(shared var)int:x;

(1) When the memory manager at P_i receives a Read(x) from application:

(1a) if counter = 0 then

(1b) return x

(1c) else keep the Read pending

(2) When the memory manager at P_i receives a Write(x, val) from application:

(2a) count ← counter + 1;

(2b) total_order_broadcast$_i$ Write(x, val)

(2c) return acknowledgement to the application.

(3) When the memory manager at P_i receives a total_order_broadcast$_j$ Write(x, val) from network:

(3a) write val to local replica of x;

(3b) if i=j then

(3c) counter ← counter – 1;

(3d) if (counter = 0 and any Reads are pending) then

(3e) perform pending responses for the Reads to the application.

Fig : Sequential Consistency using local write algorithm

Casual Consistency

- The causal consistency model represents a weakening of sequential consistency inthat it makes a distinction between events that are potentially causally related andthose that are not.

According to casual consistency model, only that Writes that are causally related must be seen in that same order by all processors, whereas concurrent Writes may be seen by different processors in different orders.

The causality relation is defined as follows:

- **Local order:** At a processor, the serial order of the events defines the local causalorder.
- **Inter-process order:** A Write operation causally precedes a Read operation issued by another processor if the Read returns a value written by the Write.

- **Transitive closure:** The transitive closure of the above two relations defines the (global) causal order.

Pipelined RAM (PRAM) or Processor Consistency

- In causal consistency, the concurrent writes be seen in a different order on different machines, although causally-related ones must be seen in the same order by all machines.
- PRAM consistency or Pipelined RAM states that Writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.
- This is a weaker form of consistency requires only that Write operations issued by the same processor are seen by all other processors in the same order that they were issued, but Write operations issued by different processors may be seen in different orders by different processors.
- The local causality relation, as defined by the local order of Write operations, needs to be seen by other processors. Hence, this form of consistency is termed processor consistency.
- An equivalent name for this consistency model is **pipelined RAM (PRAM**), to capture the behavior that all operations issued by any processor appear to the otherprocessors in a FIFO pipelined sequence.

Slow Memory

- The use of weakly consistent memories or slow memory in distributed shared memory systems to combat unacceptable network delay and to allow such systemsto scale is proposed.
- Slow memory is presented as a memory that allows the effects of writes to propagate slowly through the system, eliminating the need for costly consistency maintenance protocols that limit concurrency.
- Slow memory processes a valuable locality property and supports a reduction

from traditional atomic memory. Thus slow memory is as expressive as atomic memory.

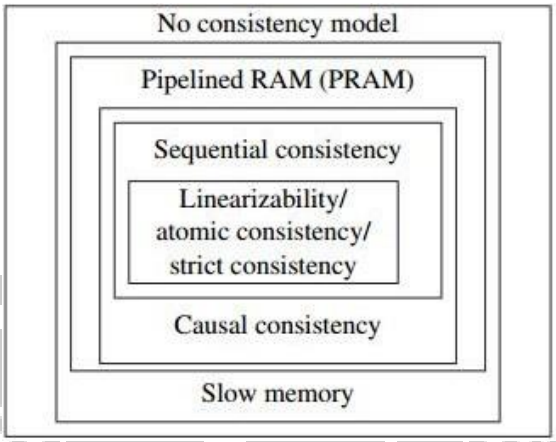


Fig : Hierarchy of memory consistency models

Models based on synchronization instructions

Synchronization instructions are like run-time library. The synchronization statements across the various processors must satisfy the consistency conditions; other program statements between synchronization statements may be executed by the different processors without any conditions.

i) Weak Consistency

The protocol is said to support weak consistency if:

- All accesses to synchronization variables are seen by all processes (or nodes, processors) in the same order (sequentially) - these are synchronization operations. Accesses to critical sections are seen sequentially.
- All other accesses may be seen in different order on different processes (or nodes,processors).
- The set of both read and write operations in between different synchronization operations is the same in each process.

Drawbacks:

When a synchronization variable is accessed, the memory does not know

whether this is being done because the process is finished writing the shared variables or about to begin reading them.

ii) Release Consistency

The drawbacks of weak consistency are overcome by:

1. Ensuring that all locally initiated Writes have been completed, i.e., propagated to all other processes.
2. Ensuring that all Writes from other machines have been locally reflected

To differentiate the entering and leaving of CS, release consistency providesacquire and release operations.

Acquire:

- Acquire accesses are used to tell the memory system that a critical region is about to be entered.
- The actions for case 2 need to be performed to ensure that local replicas of variables are made consistent with remote ones.

Release:

- This accesses say that a critical region has just been exited.
- Hence, the actions for case 1 need to be performed to ensure that remote replicas of variables are made consistent with the local ones that have been updated.

The Acquire and Release operations can be defined to apply to a subset of the variables. The accesses themselves can be implemented either as ordinary operationson special variables or as special operations. If the semantics of a CS is not associated with the Acquire and Release operations, then the operations effectively provide for barrier synchronization.

The barrier synchronization states that until all processes complete the previous phase, none can enter the next phase.

This is implemented through protected variables which follows the given rules:

- All previously initiated Acquire operations must complete successfully before a process can access a protected shared variable.
- All accesses to a protected shared variable must complete before a Releaseoperation can be performed.
- The Acquire and Release operations effectively follow the PRAM consistency model.

The lazy release consistency model is relaxation of the release consistency model in which rather than propagating the updated values throughout the system as soon as a process leaves a critical region, the updated values are propagated to the rest of the system only on demand, i.e., only when they are needed.

iii) Entry Consistency

- Entry consistency requires the programmer to use Acquire and Release at the startand end of each CS, respectively.
- Entry consistency requires each ordinary shared variable to be associated with some synchronization variable such as a lock or barrier.
- When an Acquire is performed on a synchronization variable, only access to those ordinary shared variables that are guarded by that synchronization variable is regulated.