Distributed Shared Memory

By

Dr. Prabhat Ranjan

Central University of South Bihar
Distributed Shared Memory

- The two basic paradigms for interprocess communication are:
  - (i) Shared-memory approach
  - (ii) Message-passing approach

- Message-passing systems or systems supporting RPCs adhere to the message-passing approach paradigm.

- This paradigm consists of two basic primitives for interprocess communication:
  - (i) Send (recipient, data)
  - (ii) Receive (data)

- The basic communication paradigm remains the same (Message-passing systems or systems supporting RPCs) because the communicating processes directly interact with each other for exchanging the shared data.

- However, in case of shared-memory paradigm processes in distributed system are provided with a shared address space.

- Processes use this address space in the same way they use normal local memory.

- Processes access data in the shared address space through the following two basic primitives:
  - (i) \( \text{data} = \text{Read} \ (\text{address}) \)
  - (ii) \( \text{Write} \ (\text{address}, \ \text{data}) \)

- \text{Read} returns the data item referenced by address, and \text{write} sets the contents referenced by address to the value of data.
Distributed Shared Memory

- For loosely coupled distributed memory systems, no physically shared memory is available to support the shared memory paradigm for interprocess communication.
- But some recent loosely coupled distributed-memory systems have implemented a **software layer** on top of the message-passing communication system to provide a **shared-memory abstraction** to the programmers.
- The **shared-memory abstraction** gives these systems the **illusion** of physically shared memory and allows programmers to use the shared-memory paradigm.
- The software layer **can be implemented in an operating system kernel** with proper system kernel support.

The Distributed Shared Memory (DSM) refers to the shared-memory paradigm applied to loosely coupled distributed-memory systems.
DSM provides a virtual address space shared among processes on loosely coupled processors.

DSM is basically an abstraction that integrates the local memory of different machines in a network environment into a single logical entity shared by cooperating processes executing on multiple sites.

DSM is sometimes also referred to as Distributed Shared Virtual Memory (DSVM).
Distributed Shared Memory

NOTE

- A software memory-mapping manager routine in each node maps the local memory onto the shared virtual memory.
- To facilitate the mapping operation, the shared-memory space is partitioned into blocks.
- The idea of data caching is used in DSM systems to reduce network latency.
- The main memory of individual nodes is used to cache pieces of the shared-memory space.
Distributed Shared Memory

Procedure

1. When a process on a node accesses some data from a memory block of the shared memory space,
   ➢ The local memory-mapping manager takes charge of its request.

2. If the memory block containing the accessed data is resident in the local memory, the request is satisfied by supplying the accessed data from the local memory.

3. Otherwise, a network block fault is generated and the control is passed to the operating system.

4. The operating system then sends a message to the node on which the desired memory block is located to get the block.

5. The missing block is migrated from the remote node to the client process's node and the operating system maps it into the application's address space.

6. The faulting instruction is then restarted and can now complete.
   ➢ Here, the scenario is that data blocks keep migrating from one node to another on demand but no communication is visible to the user processes.

Note

DSM system allows replication and/or migration of shared-memory data blocks depending on the implementations.
Issues involved in the design and implementation of DSM systems are:

1. **Granularity.**
2. **Structure of shared-memory space.**
3. **Memory coherence and access synchronization (Memory Consistency).**
4. **Data location and access**
5. **Replacement strategy.**
6. **Thrashing.**
7. **Heterogeneity.**
Granularity.

- Granularity refers to the block size of a DSM system,
  - Possibilities block size are the word, block (a few words), page, or segment (multiple pages).
- Selecting proper block size is an important part of the design of a DSM system
  - As block size is usually a measure of the amount of network traffic generated by network block faults.

Factors influencing block size selection

First:

- In a loosely coupled multiprocessor system, sending large packets of data (for example, 1024 kilobytes) is not much more expensive than sending small ones (for example, 256 bytes).
  - The biggest advantage is that because the startup time for a network transfer is substantial, it does not take much longer to transfer 1024 bytes than it does to transfer 512 bytes.
By transferring data in large units (large block size), when a large piece of address space has to be moved, the number of transfers may often be reduced.

This property is especially important because many programs exhibit locality of reference, meaning that if a program has referenced one word on a page, it is likely to reference other words on the same page in the immediate future.

On the other hand, the network will be tied up longer with a larger transfer, blocking other faults caused by other processes.

Second: False sharing

Too large an effective page size introduces a new problem, called false sharing, illustrated in Figure.

Here in figure it can be noticed that, a page contains two unrelated shared variables, A and B.
Second: **False sharing**

- Processor 1 makes heavy use of $A$, reading and writing it and similarly, process 2 uses $B$.
- Under these circumstances, the page containing both variables will constantly be traveling back and forth between the two machines.
- The problem here is that although the variables are unrelated, since they appear by accident on the same page
  - When a process uses one of them, it also gets the other.
  - The larger the effective page size, the more often false sharing will occur, and conversely, the smaller the effective page size, the less often it will occur.
Third: **Thrashing**

- The problem of thrashing may occur when data items in the same data block are being updated by multiple nodes at the same time, causing large numbers of data block transfers among the nodes *without much progress in the execution of the application*.

- While a thrashing problem may occur with any block size, it is more likely with larger block sizes, as different regions in the same block may be updated by processes on different nodes, causing data block transfers, that are not necessary with smaller block sizes.

![Diagram showing thrashing in DSM](image-url)
CONSISTENCY MODELS

- Consistency model refers to the degree of consistency that has to be maintained for the shared-memory data for the memory to work correctly for a certain set of applications.
- It is defined as a set of rules that applications must obey if they want the DSM system to provide the degree of consistency guaranteed by the consistency model.
- Several consistency models have been proposed in the literature and the main ones are described as follows:

Strict Consistency Model

- The strict consistency model is the strongest form of memory coherence, having the most stringent consistency requirement.
- A shared-memory system is said to support the strict consistency model if the \texttt{.value} returned by a read operation on a memory address is always the same as the value written by the most recent write operation to that address, irrespective of the locations of the processes performing the read and write operations.
  - Any read to a memory location \( x \) returns the value stored by the most recent write operation to \( x \).
  - That is, all writes instantaneously become visible to all processes.
Strict Consistency Model: contd.

Example

- The initial value of all variables in these diagrams is assumed to be 0.
- In Fig. (a) \( P_1 \) does a write to location \( x \), storing the value 1.
- Later, \( P_2 \) reads \( x \) and sees the 1. This behavior is correct for a strictly consistent memory.
- In Fig. (b) \( P_2 \) does a read after the write (possibly only a nanosecond after it, but still after it), and gets 0.
- A subsequent read gives 1. Such behavior is incorrect for a strictly consistent memory.
Sequential Consistency Model

- The sequential consistency model was proposed by Lamport [1979].
- A shared-memory system is said to support the sequential consistency model if all processes see the same order of all memory access operations on the shared memory.
- The exact order in which the memory access operations are interleaved does not matter.
- That is, if the three operations read \( (r_1) \), write \( (w_1)' \) read \( (r_2) \) are performed on a memory address in that order, any of the orderings \( (r_1, w_1, r_2) \), \( (r_1, r_2, w_1) \), \( (w_1, r_1, r_2) \), \( (w_1, r_2, r_1) \), \( (r_2, r_1, w_1) \), \( (r_2, w_1, r_1) \) of the three operations is acceptable provided all processes see the same ordering.
- If one process sees one of the orderings of the three operations and another process sees a different one, the memory is not a sequentially consistent memory.
Sequential Consistency Model

Example:

A memory behaving as shown in Fig. (a) is sequentially consistent even though the first read done by P2 returns the initial value of 0 instead of the new value of 1.

Sequentially consistent memory does not guarantee that a read returns the value written by another process a nanosecond earlier, or a microsecond earlier, or even a minute earlier.

It guarantees that all processes see all memory references in the same order.

If the program that generated Fig (a) is run again, it might give the result of Fig. (b).

The results are not deterministic.

Running a program again may not give the same result.
Causal Consistency Model

- The causal consistency model (Hutto and Ahamad, 1990) represents a weakening of sequential consistency in that it makes a distinction between events that are potentially **causally related** and those that are not.
- Suppose that process \( P_1 \) writes a variable \( x \), then \( P_2 \) reads \( x \) and writes \( y \).
- Here the reading of \( x \) and the writing of \( y \) are potentially causally related
  - Because the computation of \( y \) may have depended on the value of \( x \) read by \( P_2 \) (i.e., the value written by \( P_1 \))
- On the other hand, if two processes spontaneously and simultaneously write two variables, these are not causally related.
- When there is a read followed later by a write, the two events are potentially causally related
  - A read is causally related to the write that provided the data the read got
- Operations that are not causally related are said to be **concurrent**.
Causal Consistency Model: Contd.

For a memory to be considered causally consistent, it is necessary that the memory obey the following condition:

i. Writes that are potentially causally related must be seen by all processes in the same order.

ii. Concurrent writes may be seen in a different order on different machines.

In Fig. (a) we have \( W(x)2 \) potentially depending on \( W(x)1 \) because the 2 may be a result of a computation involving the value read by \( R(x)1 \).

The two writes are causally related, so all processes must see them in the same order, thus Fig. (a) is incorrect.

In Fig. (b) the read has been removed, so \( W(x)1 \) and \( W(x)2 \) are now concurrent writes. Causal memory does not require concurrent writes to be globally ordered, so Fig. (b) is correct.
Pipelined Random-Access Memory Consistency Model.

- The pipelined random-access memory (PRAM) consistency model, proposed by Lipton and Sandberg [1988],

**Conditions**

1. It ensures that all write operations performed by a single process are seen by all other processes in the order in which they were performed
   - That is the write operations performed by a single process are in a pipeline.
2. Write operations performed by different processes may be seen by different processes in different orders.

**Examples**

- If \( W_{11} \) and \( W_{12} \) are two write operations performed by a process \( P_1 \) in that order, and \( W_{21} \) and \( W_{22} \) are two write operations performed by a process \( P_2 \) in that order.
- A process \( P_3 \) may see them in the order \([W_{11}, W_{12}], (W_{21}, W_{22})\] and another process \( P_4 \) may see them in the order \([W_{21}, W_{22}], (W_{11}, W_{12})\].
Processor Consistency Model

- The processor consistency model, proposed by Goodman [1989],
- A processor consistent memory is both coherent and adheres to the PRAM consistency model.
- Memory coherence means that for any memory location all processes agree on the same order of all write operations to that location.
- Processor consistency ensures that all write operations performed on the same memory location (no matter by which process they are performed) are seen by all processes in the same order.
- If w12 and w22 are write operations for writing to the same memory location x, all processes must see them in the same order - w12 before w22 or w22 before w12.
- If W11 and W12 are two write operations performed by a process P1 in that order, and W21 and W22 are two write operations performed by a process P2 in that order.
- This means that for processor consistency both processes P3 and P4 must see the write operations in the same order, which may be either [(W11, W12), (W21, W22)] or [(W21, W22), (W11, W12)].
Weak Consistency Model

- Although PRAM consistency and processor consistency can give better performance than the stronger models, they are still unnecessarily restrictive for many applications
  - Because they require that writes originating in a single process be seen everywhere in order.
- For all application it is not necessary to show the change in memory done by every write operation to other processes.
- Consider the case of a process inside a critical section reading and writing some variables in a tight loop.
- Even though other processes are not supposed to touch the variables until the first process has left its critical section,
  - The memory has no way of knowing when a process is in a critical section and when it is not,
  - So it has to propagate all writes to all memories in the usual way.
Weak Consistency Model

- A better solution would be to let the process finish its critical section and then make sure that the final results were sent everywhere.
- This can be done by introducing a new kind of variable, a synchronization variable.
  - It is used for synchronization purposes.
- The operations on synchronization variable are used to synchronize memory.
- That is, when a synchronization variable is accessed by a process, the entire (shared) memory is synchronized by making all the changes to the memory made by all processes visible to all other processes.
- When a synchronization completes, all writes done on that machine are propagated outward and all writes done on other machines are brought in.
  - In this way all of (shared) memory is synchronized.
Weak Consistency Model

- Process P1 does two writes to an ordinary variable, and then synchronizes (indicating by the letter S).
- If P2 and P3 have not yet been synchronized, no guarantees are given about what they see, so this sequence of events is valid.
- In case of Fig. (b), P2 has been synchronized, which means that its memory is brought up to date.
- When it reads \( x \), it must get the value 2. Getting 1 as shown in the figure (b), is not permitted with weak consistency.
Weak Consistency Model

Dubois et al. (1986) define this model, called weak consistency, by saying that it has three properties:

1. **Accesses to synchronization variables are sequentially consistent.**
2. **No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.**
3. **No data access (read or write) is allowed to be performed until all previous accesses to synchronization variables have been performed.**

**Point 3**
- when ordinary (i.e., not synchronization) variables are accessed, either for reading or writing, all previous synchronizations have been performed.
- By doing a synchronization before reading shared data, a process can be sure of getting the most recent values.
Release Consistency Model

- In the weak consistency model the entire (shared) memory is synchronized when a synchronization variable is accessed by a process,
- Memory synchronization basically involves the following operations:
  1. All changes made to the memory by the process are propagated to other nodes.
  2. All changes made to the memory by other processes are propagated from other nodes to the process's node.

- A closer observation shows that executing above two operation each time is not necessary, that is:
  1. The first operation need only be performed when the process exits from a critical section
  2. The second operation need only be performed when the process enters a critical section.

**NOTE:**
Since a single synchronization variable is used in the weak consistency model, the system cannot know whether a process accessing a synchronization variable is entering a critical section or exiting from a critical section.
For better performance, the release consistency model provides a mechanism to clearly tell the system whether a process is **entering a critical section** or **exiting from a critical section**

Then the system can decide and perform only either the first or the second operation when a synchronization variable is accessed by a process.

This is achieved by using two synchronization variables called **acquire** and **release**.

**Acquire** is used by a process to tell the system that it is about to **enter a critical section**, so that the system performs only the second operation when this variable is accessed.

**Release** is used by a process to tell the system that it has just **exited a critical section**, so that the system performs only the first operation when this variable is accessed.
Release Consistency Model

- For better performance, the release consistency model provides a mechanism to clearly tell the system whether a process is entering a critical section or exiting from a critical section.
  - Then the system can decide and perform only either the first or the second operation when a synchronization variable is accessed by a process.
- This is achieved by using two synchronization variables called acquire and release.
- **Acquire** is used by a process to tell the system that it is about to enter a critical section, so that the system performs only the second operation when this variable is accessed.
- **Release** is used by a process to tell the system that it has just exited a critical section, so that the system performs only the first operation when this variable is accessed.
Data location and access
- To share data in a DSM system, it should be possible to locate and retrieve the data accessed by a process.
- A DSM system must implement some form of **data block locating mechanism** in order to service network data block faults.

Replacement strategy
- If the **local memory of a node is full**, a cache miss at that node implies not only a fetch of the accessed data block from a remote node but also a replacement.
- A data block of the local memory must be replaced by the new data block.
- A **cache replacement strategy** is necessary in the design of a DSM system.

Thrashing
- In a DSM system, data blocks migrate between nodes on demand.
- If two nodes compete for write access to a single data item, the corresponding data block may be transferred back and forth at such a high rate that no real work can get done.
- A DSM system must use a policy to avoid this situation (usually known as thrashing).
Heterogeneity.

- The DSM systems built for homogeneous systems need not address the heterogeneity issue.
- If the underlying system environment is heterogeneous, the DSM system must be designed to take care of heterogeneity so that it functions properly with machines having different architectures.