# TOPIC 5

# DISTRIBUTED SHARED MEMORY

## **Introduction**

Different systems, developed on the basis of remote procedure calls, have different IPC requirements. Based on the needs of different systems, several communication protocols have been proposed for use in RPCs

## **Structure:**

5.1 Consistency Models

5.3 Sequential Consistency Model

5.4 Causal Consistency Model

5.5 Reduction

5.6 Conventional

5.7 Replacement Strategy

5.8 Which Block to Replace

5.9 Where to place a replaced Block

5.10 Thrashing

5.11 Advantages of DSM

5.12 Simpler Abstraction

5.13 Better Portability of Distributed Application Programs

5.14 Better Performance of some applications

5.15 Flexible Communication Environment

**Learning Activity 5.0**

1. Explain Thrashing

2. Discuss DSM

**5.1 CONSISTENCY MODELS**

Consistency requirements vary from application to application. A consistency model basically 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 to DSM system to provide the degree of consistency guaranteed by the consistency model. Several consistency models have been proposed in the literature. Of these, the main ones are described below.

It may be noted here that the investigation of new consistency models is currently an active area of research. The basic idea is to invent a consistency model that can allow consistency requirements to be related to a greater degree than existing consistency models, with the relaxation done in such a way that a set of applications can function correctly. This helps in improving the performance of these applications because better concurrency can be achieved by relaxing the consistency requirement. However, applications that depend on a stronger consistency model may not perform correctly if executed in a system that supports only a weaker consistency model. This is because if a system supports the stronger consistency model, then the weaker consistency model is automatically supported but the converse is not true.

**5.2 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 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. That is, all writes instantaneously become visible to all processes.

Implementation of the strict consistency model requires the existence of an absolute global time so that memory read / write operations can be correctly ordered to make the meaning of “most recent” clear. However, absolute synchronization of clocks of all the nodes of a distributed system is not possible. Therefore, the existence of an absolute global time in a distributed system is also not possible. Consequently, implementation of the strict consistency model for a DSM system is practically impossible.

**5.3 SEQUENTIAL CONSISTENCY MODEL**

The sequential consistency model was proposed 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 r1 , write w1 ,, read (r2 ) are preformed on a memory address in that order,

any of the ordering (r1, w1, r2), (r1 , r2 , w1 ), (w1 , r1 , r2) , (w1 , r2 , r1), ( r2 , r1 , w1),

(r2 , w1 , r1 ) 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. Note here that the only acceptable ordering for a strictly consistent memory is (r1 , w1 , r2 ) .

**56**

The consistency requirement of the sequential consistency model is weaker than that of the strict consistency model because the sequential consistency model does not guarantee that a read operation on a particular memory address always return the same value as written by the most recent write operation to that address. As a consequence, with a sequentially consistent memory, running a program twice may not give the same result in the absence of explicit synchronization operations. This problem does not exist in a strictly consistent memory.

A DSM system supporting the sequential consistency model can be implemented by ensuring that no memory operation is started until all the previous ones have been completed. A sequentially consistent memory provides one-copy / single-copy semantics because all the process sharing a memory location always see exactly the same contents stored in it. This is the most intuitively expected semantics for memory coherence. Therefore, sequential consistency is acceptable by most applications.

**5.4 CAUSAL CONSISTENCY MODEL**

The causal consistency model, relaxes the requirement of the sequential consistency model for better concurrency. Unlike the sequential consistency model, in the causal consistency model, all processes see only those memory reference operations in the same (correct) order that are potentially causally related. Memory reference operations that are not potentially causally related may be seen by different processes in different orders. A memory reference operation (Read / write) is said to be potentially causally related to another memory reference operation if the first one might have been influenced in any way by the second one. For example, if a process performs a read operation followed by a write operation, the write operation is potentially causally related to the read operation because the computation of the value written may have depended in some way on the value obtained by the read operation. On the other hand, a write operation performed by one process is not causally related to a write operation performed by another process if the first process has not read either the value written by the second process or any memory variable that was directly or indirectly derived from the value written by the second process.

A shared memory system is said to support the causal consistency model if all write operations that are potentially causally related are seen by all processes in the same (correct) order. Write operations that are not potentially causally related may be seen by different processes in different orders. Note that “correct order”

**57**

means that if a write operation w2 is causally related to another write operation w1 , the acceptable order is w1 , w2 because the value written by w2 might have been influenced in some way by the value written by w1 . Therefore, w2 , w1 is not an acceptable order.

**5.5 REDUCTION**

Shared variables that must be automatically modified may be annotated to be reduction type. For example, in a parallel computation application, a global minimum must be atomically fetched and modified if it is greater than the local minimum. In Munin, a reduction variable is always modified by being locked (acquire lock), read, updated, and unlocked (release lock). For better performance, a reduction variable is stored at a fixed owner that receives updates to the variable from other processes, synchronizes the updates received from different processes, performs the updates on the variable, and propagates the updated variable to its replica locations.

**5.6 CONVENTIONAL**

Shared variables that are not annotated as one of the above types are conventional variables. The already described release consistency protocol of Munin is used to maintain the consistency of replicated conventional variables. The write invalidation protocol is used in this case of ensure that no process ever reads a stale version of a conventional variable. The page containing a conventional variable is dynamically moved to the location of a process that wants to perform a write operation on the variable.

Experience with Munin has shown that read-only migratory, and write shared annotation types are very useful because variables of these types are frequently used, but producer, consumer, result and reduction annotation types are of little use because variable of these types are less frequently used.

**5.7 REPLACEMENT STRATEGY**

In DSM systems that allow shared memory blocks to be dynamically migrated / replicated, the following issues must be addressed when the available space for caching shared data fills up at a node :

**58**

1. Which block should be replaced to make space for a newly required block?
2. Where should the replaced block be placed?

**5.8 WHICH BLOCK TO REPLACE**

The problem of replacement has been studied extensively for paged main memories and shared memory multiprocessor systems. The usual classification of replacement algorithms group them into the following categories [Smith 1982] :

1. Usage based versus non-usage based : Usage based algorithms keep track of the history of usage of a cache line (or page) and use this information to make replacement decisions. That is, the reuse of a cache line normally improves the replacement status of that line. Least recently used (LRU) is an example of this type of algorithm. Conversely, non-usage-based algorithms do not take the record of use of cache lines into account when doing replacement. First in, first out (FIFO) and Rand (Random or pseudorandom) belong to this class.
2. Fixed space versus variable space : Fixed space algorithms assume that the cache size is fixed while variable space algorithms are based on the assumption that the cache size can be changed dynamically depending on the need. Therefore, replacement in fixed space algorithms simply involves the selection of a selection of a specific cache line. On the other hand, in a variable space algorithm, a fetch does not imply a replacement, and a swap out can take place without a corresponding fetch.

Variable space algorithms are not suitable for a DSM system became each node’s memory that acts as cache for the virtually shared memory is fixed in size. Moreover, as compared to non usage based algorithms, usage based algorithms are more suitable for DSM systems because they allow to take advantage of the data access locality feature. However, unlike most caching systems, which use a simple LRU policy for replacement, most DSM systems differentiate the status of data items and use a priority spechanism. As an example, the replacement policy used by the DSM system of IVY [LL 1986, 1988] is presented here. In the DSM system of IVY, each memory block of a node is classified into one of the following five types.

1. Unused : A free memory block that is not currently being used.
2. Nil : A block that has been invalidated.

**59**

1. Read-only : A block for which the node has only read access right.
2. Read owned : A block for which the node has only read access right but is also the owner of the block.
3. Writable : A block for which the node has write access permission, Obviously, the node is the owner of the block because IVY uses the write invalidate protocol.

Based on this classification of blocks, the following replacement priority is used.

1. Both unused and nil blocks have the highest replacement priority. That is, they will be replaced first if a block is needed. It is obvious for an usused block to have the highest replacement priority. A nil block also has the same replacement priority because it is no longer useful and future access to the block would cause a network fault to occur. Notice that a nil block may be a recently referenced block, and this is exactly why a simple LRU policy is not adequate.
2. The read only blocks have the next replacement priority. This is because a copy of a read only block is available with its owner, and therefore it is possible to simply discard that block. When the node again requires that block in the future, the block has to be brought from its owner node at that time.
3. Read owned and writable blocks for which replica(s) exist on some other node(s) have the next replacement priority because it is sufficient to pass ownership to one of the replica nodes. The block itself need not be sent, resulting in a smaller message.
4. Read owned and writable blocks for which only this node has a copy have the lowest replacement priority because replacement of such a block involves transfer of the block’s ownership as well as the block from the current node to some other nodes. An LRU policy is used to select a block for replacement when all the blocks in the local cache have the same priority.

**5.9 WHERE TO PLACE A REPLACED BLOCK**

Once a memory block has been selected for replacement, it should be ensured that if there is some useful information in the block, it should not be lost. For example, simply discarding a block having unused, Nil, or read only status does not lead to any loss of data. Similarly, discarding a read owned of a writable block for which replica(s) exist on some other node(s) is also harmless. However, discarding a read owned or a writable block for which there is no replica on any other node may lead to loss of useful data. Therefore, care must be taken to store them somewhere before discarding. The two commonly used approaches for storing a useful block at the time of its replacement are as follows:

1. **Using secondary store :** In this method, the block is simplytransferred on to a local disk. The advantage of this method is that it does not waste any memory space and if the node wants to access the same block again, it can get the block locally without a need for network access.
2. **Using the memory space of other nodes :** Sometimes it maybe faster to transfer a block over the network than to transfer it to a local disk. Therefore, another method for storing a useful block is to keep track of free memory space at all nodes in the system and to simply transfer the replaced block to the memory or a node with available space. This method requires each node to maintain a table of free memory space in all other nodes. This table may be updated by having each node piggyback its memory status information during normal traffic.

**5.10 THRASHING**

Thrashing is said to occur when the system spends a large amount of time transferring shared data blocks from one node to another, compared to the time spent doing the useful work of executing application processes. It is a serious performance problem with DSM systems that allow data blocks to migrate from one node to another. Thrashing may occur in the following situations:

1. When interleaved data accesses made by processes on two or more nodes causes a data block to move back and forth fro one node to another in quick succession (a ping – pong effect)
2. When blocks with read only permissions are repeatedly invalidated soon after they are replicated.

Such situations indicate poor (node) locality in references. If not properly handled, thrashing degrades system performance considerably. Therefore, steps must be taken to solve this problem. The following methods may be used to solve the thrashing problem in DSM systems :

**61**

1. Providing application – controlled locks : Locking data to prevent other nodes from accessing that for a short period of time can reduce thrashing. An application controlled lock can be associated with each data block to implement this method.
2. Nailing a block to a node for a minimum amount of time : Another method to reduce thrashing is to disallow a block to be taken away from a node until a minimum amount of time t elapses after its allocation to that node. The time t can either be fixed statically or be turned dynamically on the basis of access patterns. For example, Mirage [Fleisch and Popek 1989] employs this method to reduce thrashing and dynamically determines the minimum amount of time for which a block will be available at a node on the basis of access patterns.

The main drawback of this scheme is that it is very difficult to choose the appropriate value for the time. If the value is fixed statically, it is liable to be inappropriate in many cases. For example, if a process accesses a block for writing to it only once, other processes will be prevented from accessing the block until time t elapses. On the other hand, it a process accesses a block for performing several write operations on it, time t may elapse before the process has finished using the block and the system may grant permission to another process for accessing the block. Therefore, tuning the value of t dynamically is the preferred approach in this case, the value of t for a block can be decided based on past access patterns of the block. The MMU’s reference bits may be used for this purpose. Another factor that may be used for deciding the value of t for a block is the length of a the queue of processes waiting for their turn to access the block.

1. Tailoring the coherence algorithm to the shared data usage pattern Thrashing can also minimized by using different coherence protocols for shared data having different characteristics. For example, the coherence protocol used in Munin for write shared variables avoids the false sharing problem, which ultimately results in the avoidance of thrashing.

Notice from the description above that complete transparency of distributed shared memory is compromised somewhat while trying to minimize thrashing. This is because most of the approaches described above require the programmer’s assistance. For example, in the method of application controlled locks, the use of locks needs to be directed towards a particular shared memory algorithm and hence the shared memory

abstraction can no longer be transparent. Moreover, the application must be aware of the shared data it is accessing and its

**62**

shared access patterns. Similarly, Munin requires programmers to annotate shared variables with standard annotation types, which makes the shared memory abstraction nontransparent.

1. **Smallest page size algorithm :** In this method, the DSM blocksize is taken as the smallest VM page size of all machines. If a page fault occurs on a node with a large page size, multiple blocks (whose total size is equal to the page size of the faulting node) are moved to satisfy the page fault. Although this algorithm reduces data contention, it suffers from the increased communication and block table management overheads associated with small sized blocks.
2. **Intermediate page size algorithm :** To balance between theproblems of large and small sized blocks, a heterogeneous DSM system may select to choose a block size somewhere in between the largest VM page size and the smallest VM page size of all machines.

**5.11 ADVANTAGES OF DSM**

Distributed Shared Memory is a high level mechanism for interprocess communication in loosely coupled distributed systems. It is receiving increased attention because of the advantages it has over the message passing mechanisms. These advantages are discussed below.

**5.12 SIMPLER ABSTRUCTION**

By now it is widely recognized that directly programming loosely coupled distributed memory machines using message passing models is tedious and error phone. The main reason is that the message passing models force programmers to be conscious of data movement between processes at all times, since processes must explicitly use communication primitives and channels or ports. To alleviate this burden, RPC was introduced to provide a procedure call interface. However, even in RPC, since the procedure call is performed in an address space different from that of the caller’s address space, it is difficult for the caller to pass context related data or complex data structures; that is, parameters must be passed by value. In the message passing model, the programming task is further complicated by the fact that data structures passed between processes in the front of messages must be packed and unpacked. The shared memory programming paradigm shields the application programmers from many such low-level concerns. Therefore, the primary advantage of DSM is the

**63**

simpler abstraction it provides to the application programmers of loosely coupled distributed memory machines.

**5.13 BETTER PORTABILITY OF DISTRIBUTED APPLICATION PROGRAMS**

The access protocol used in case of DSM is consistent with the way sequential applications access data. This allows for a more natural transition from sequential to distributed applications. In principle, distributed application programs written for a shared memory multiprocessor system can be executed on a distributed shared memory system without change. Therefore, it is easier to port an existing distributed application program to a distributed memory system with DSM facility than to a distributed men system without this facility.

**5.14** **BETTER PERFORMANCE OF SOME APPLICATIONS**

The layer of software that provides DSM abstraction is implemented on top of a message passing system and uses the services of the underlying message passing communication system. Therefore, in principle, the performance of applications that the DSM is expected to be worse than if they use message passing directly. However, this is not always true, and it has been found that some applications using DSM can even outperform their message passing counterparts. This is possible for three reasons [Stumm and Zhou 1990] :

1. **Locality of data:** The communication model of DSM is to makethe data more accessible by moving it around. DSM algorithms normally move data between nodes in large blocks. Therefore, in those applications that exhibit a reasonable degree of locality in their data accesses, communication overhead is amortized over multiple memory accesses. This ultimately results in reduced overall communication cost for such applications.
2. **On demand data movement:** The computation model of DSMalso facilitates on demand movement of data as they are being accessed. On the other hand, there are several distributed applications that execute in phase, where each computation phase is preceded by a data exchange phase. The time needed for the data exchange phase is often dictated by the throughput of existing communication bottlenecks. Therefore, in such applications, the on-demand data movement facility provided by DSM eliminates the data exchange phase, spreads the communication load over a longer period of time, and allows for a greater degree of concurrency.
3. **Larger memory space:** With DSM facility, the total memorysize is the sum of the memory sizes of all the nodes in the system. Thus, paging and swapping activities, which involve disk access, are greatly reduced.

**5.15 FLEXIBLE COMMUNICATION ENVIRONMENT**

The message passing paradigm requires recipient identification and coexistence of the sender and receiver processes. That is, the sender process of a piece of data must know the names of its receiver processes (except in multicast communication), and the receivers of the data must exist at the time the data is sent and, in a state, that they can (or eventually can) receive the data. Otherwise, the data is undeliverable. In contrast, the shared memory paradigm of DSM provides a more flexible communication environment in which the sender process need not specify the identity of the receiver processes of the data. It simply places the data in the shared memory and the receivers access it directly from the shared memory. Therefore, the coexistence of the sender and receiver processes is also not necessary in the shared memory paradigm. In fact, the lifetime of the shared data is independent of the lifetime of any of its receiver processes.

**Revision Exercises:**

5.1 The distributed shared memory abstraction is implemented by using the services of the underlying message passing communication system. Therefore, in principle, the performance of applications that the use DSM is expected to be worse than if they use message passing directly. In spite of this fact, why do some distributed operating system designers support the DSM abstraction in their systems? Are there any applications that can have better performance in a system with DSM facility than in a system that has only message passing facility? If yes, give the types of such applications. If no, explain why.

5.2 Discuss the relative advantages and disadvantages of using large block size and small block size in the design of a block-based DSM system. Why do most DSM system designers prefer to use the typical page size used in a conventional virtual memory implementation as the block size of the DSM system?

**65**

5.3 It is often said that the structure of the shared memory space and the granularity of data sharing in a DSM system are closely related. Explain why?

5.4 What is false sharing? When is it likely to occur? Can this problem lead to any other problem in a DSM system? Give reasons for your answer.

5.5 What should be done to minimize the false sharing problem? Can this problem be complete eliminated? What other problems may occur if one tries to completely eliminate the false sharing problem?

**REFERENCES:**

Ghosh, Sukumar (2007), Distributed Systems – An Algorithmic Approach, Chapman & Hall/CRC, [*ISBN*](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [*978-1-58488-564-1*](https://en.wikipedia.org/wiki/Special:BookSources/978-1-58488-564-1).

[*Lynch, Nancy A.*](https://en.wikipedia.org/wiki/Nancy_Lynch) (1996), Distributed Algorithms, [*Morgan Kaufmann*](https://en.wikipedia.org/wiki/Morgan_Kaufmann_Publishers), [*ISBN*](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [*1-55860-348-4*](https://en.wikipedia.org/wiki/Special:BookSources/1-55860-348-4).