

Distributed Shared Memory: A Survey

J. Silcock

School of Computing and Mathematics,
Deakin University,
Geelong.

Abstract

Distributed Shared Memory is an important topic in distributed system research as it allows programmers to write and use code for shared memory multiprocessors on a system which does not in fact have physically shared memory. In this report I will discuss the literature I have read related to Distributed Shared Memory in an attempt to identify the important research issues in this area. The first section covers the high level design choices and research issues regarding DSM. The second section discusses the implementation issues and some current implementations of DSM.

1 Introduction

Distributed Shared Memory (DSM) has been an active area of research since the mid 1980s. Some of the first work in this area was carried out by Libes [Libes 1985], who implemented a shared variable scheme at user level on a Unix system. [Coulouris et al. 93] describes distributed shared memory as “an abstraction used for sharing data between processes that do not share physical memory”. Figure 1 shows the logical view and physical situation in DSM. The logical view is one of several machines sharing a centralized memory. The physical situation, however, is quite different, the distributed machines have their own local memory and are connected to one another through the interconnection network. With the addition of appropriate software these individual machines are able to directly address memory in another machine’s local address space [Hellwagner 1990].

The main motivation behind the initial implementation of DSM was to emulate the well understood and convenient shared memory programming paradigm. DSM is a promising tool for parallel applications, however its ultimate success depends upon the efficiency of its implementation [Coulouris et al. 93]. The overall objective in DSM is to reduce the access time for non-local memory to as close as possible to the access time of local memory [Hellwagner 1990]. The underlying method of implementation is through the use of message passing and remote procedure calls because the shared memory must be moved or copied or messages sent to a centralized memory server in order to access non-local memory. Thus, the issues and research goals of DSM are similar to those of multiprocessor caches and networked file systems [Tam et al. 90].

The shared “chunks” of memory are, in some implementations, replicated over the network in a form of caching which is often used to improve performance in DSM systems. However, the maintenance of the coherence of this replicated memory in a transparent and efficient way is an important research objective because computer network latency is typically much larger than that of a shared bus [Nitzberg et al. 94], [Tam et al. 90].

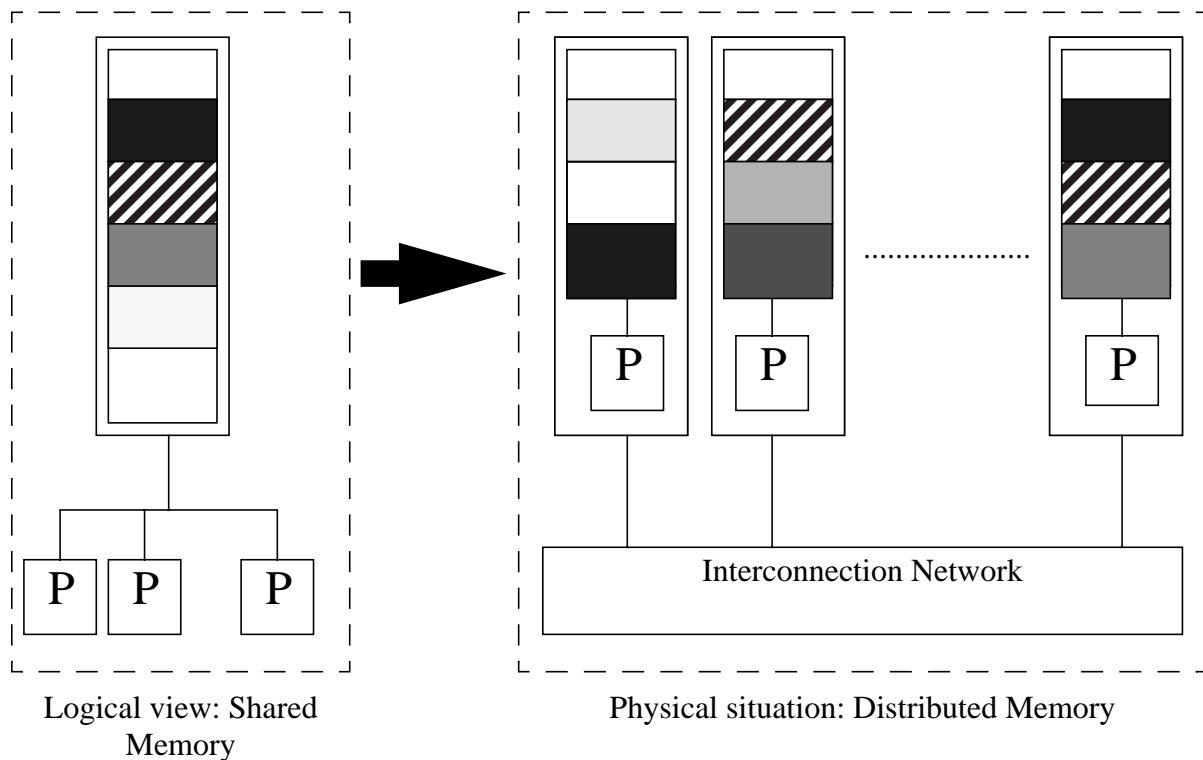


Figure 1. Logical view and physical situation of distributed shared memory [Hellwagner, 1990]

The advantages of DSM [Coulouris et al. 93], [Nitzberg et al. 94], [Levelt et al. 92] are that it:

- Increases the ease of programming by sparing programmers the concerns of message passing;
- Allows the use of algorithms and software written for shared memory multiprocessors on distributed systems;
- Distributed machines scale to much larger numbers than multiprocessors resulting from the absence of hardware bottlenecks; and
- Enables the use of low cost of distributed memory machines.

The goal of this report is to discuss the literature pertinent to Distributed Shared Memory in an attempt to identify the important research issues in this area. The first section will cover the high level design choices and research issues regarding DSM. The second section will discuss the implementation issues for DSM.

2 Design and Research Issues

There are several design choices that have to be made when implementing Distributed Shared Memory [Nitzberg et al. 94], [Tam et al. 90], [Tanenbaum 95], [Coulouris et al. 93]. These are:

- Structure and Granularity of the Shared Memory;
- Coherence Protocols and Consistency Models;
- Synchronization;

- Data location and access;
- Heterogeneity;
- Scalability;
- Replacement Strategy; and
- Thrashing.

2.1 Structure and Granularity of the Shared Memory

These two issues are closely related. The shared memory can take the form of an unstructured linear array of words or the structured forms of objects, language types or an associative memory [Nitzberg et al. 94]. The granularity relates to the size of the chunks of the data being shared. A decision has to be made whether it should be fine or coarse grained and whether data should be shared at the bit, word, complex data structure or page level. The authors of the paper [Tam et al. 90] say the coarse grained solution, page-based distributed memory, is an attempt to implement a virtual memory model where paging takes place over the network instead of to disk. It offers a model which is similar to the shared memory model and is familiar to programmers, with sequential consistency at the cost of performance.

2.2 Coherence Protocols and Consistency Models

[Nitzberg et al. 94] states, “Although consistency and coherence are used somewhat interchangeably in the literature, we use coherence as the general term for the semantics of memory operations and consistency to refer to a specific kind of memory coherence”. The strongest form of coherence is strict consistency which requires the serialization of data accesses. Unfortunately, the latter causes a bottleneck and removes the possibility of parallel execution in DSM systems. The replication of data and removal of the strict consistency requirement increases parallelism by allowing reads of the same data to be executed in parallel. However, parallel writes would result in inconsistent data. Thus some mechanism is required to ensure the serialization of write operations and that any subsequent reads or writes access the updated data. This mechanism is the coherence protocol. There are two coherence protocols:

- *write-invalidate*. Many copies of read only data are allowed but only one writable piece. All other data is invalidated before a write can proceed.
- *write-update*. When data is written all other copies of the data are updated before any further accesses to the data are allowed. Subsequent research shows that in an appropriate hardware environment write-update can be implemented efficiently [Nitzberg et al. 94]. However, Li and Hudak [Li et al. 89] reject write-update protocols, because the high cost of network latency makes them impractical.

If the write-update coherence protocol is used the problem then becomes one of maintaining the consistency of the replicated data [Tanenbaum 95]. A consistency model according to [Adve et al. 91] is a contract between the software and the hardware which says that if the software agrees to some formally specified constraints then the hardware will appear to be consistent.

The cache coherence protocols of tightly coupled multiprocessors are a well researched topic [Mosberger 94], however, many of these protocols are thought to be unsuitable for distributed systems because the strict consistency models used cause too much network traffic [Nitzberg et al. 94]. Consistency models determine the conditions under which memory updates will be propagated through the system. These models can be divided into those with and without synchronization operations. The models without synchronization operations are:

- atomic or strict;
- sequential;
- causal; and
- PRAM consistency models.

The models with synchronization operations include:

- weak;
- release; and
- entry consistency models.

2.2.1 Atomic or Strict Consistency

Tanenbaum describes strict consistency as that in which, “any read to a memory location x returns the value stored by the most recent write operation to x .” [Tanenbaum 95]

This model is what most programmers intuitively expect and have in fact observed on uniprocessors, any memory operation is seen immediately across the network [Nitzberg et al. 94]. This model requires a unified notion of time across all machines, since this is not possible in a distributed system this model is rejected. However, it serves as a base model for the evaluation of memory consistency model performance [Mosberger 94].

2.2.2 Sequential Consistency

Lamport defines sequential consistency as that in which, “the result of any operation is the same as if the operations of all the machines were executed in some sequential order, and the operations of each individual machine appear in the same sequence in the order specified by its program” [Lamport 79].

Sequential consistency is a slightly weaker model than strict consistency. It can be achieved on a distributed system since time does not play a role rather the sequence of operations [Tanenbaum 95]. In sequential consistency all processes have to agree on the order in which observed effects take place [Mosberger 94]. Thus, the results appear as though some interleaving of the operations on separate machines has taken place [Nitzberg et al. 94].

2.2.3 Causal Consistency

“A memory is causally consistent if all machines agree on the order of causally related events. Causally unrelated events (concurrent events) can be observed in different orders” [Mosberger 94].

Events are causally related if the result of one event will affect the result of another. Events which are not causally related are said to be concurrent events. Concurrent writes can be seen in a different order on different machines but causally related ones must be seen in the same order by all machines.

2.2.4 Pipelined RAM (PRAM) Consistency

“...all processors (machines) observe the writes from a single processor (machine) in the same order while they may disagree on the writes by different processors (machines).” [Mosberger 94].

Writes from a single process are pipelined and the writing process does not have to wait for each one to complete before starting the next one. A read results in the local value being returned however a write causes the local copy to be updated and a broadcast of the update to

all machines holding a copy of the data. Thus, two or more updates from the same source will be pipelined in the same order by all machines [Mosberger 94], [Tanenbaum 95].

2.2.5 Weak Consistency

“A memory system is weakly consistent if it enforces the following restrictions:

1. accesses to synchronization variables are sequentially consistent and
2. no access to a synchronization variable is issued in a processor (machine) before all previous accesses have been performed and
3. no access is issued by a processor (machine) before a previous access to a synchronization variable has been performed.” [Mosberger 94].

The term “previous” used above refers to program order. Consistency is enforced using synchronization operators, the data is guaranteed to be sequentially consistent [Nitzberg et al. 94]. Access of a synchronization variable forces all previous writes to complete, when synchronization access is completed all writes are also guaranteed to be completed, forcing all existing copies to be updated.

2.2.6 Release Consistency

A Distributed Shared Memory is release consistent if these rules are obeyed:

- “1. Before an ordinary access to a shared variable is performed, all previous acquires done by the process must have completed successfully.
2. Before a release is allowed to be performed, all previous reads and writes done by the process must have completed.
3. The acquire and release accesses must be processor consistent (sequential consistency is not required).” [Tanenbaum 95]

Thus, release consistency is weak consistency with two types of synchronization operators, *acquire* and *release*. The memory is made coherent only before a release is performed, the synchronization operators themselves are guaranteed to be processor consistent [Nitzberg et al. 94].

2.2.7 Entry Consistency

Distributed Shared Memory is entry consistent if it follows the following rules:

- “1. An acquire access of a synchronization variable is not allowed to perform with respect to that process until all updates to the guarded shared data have been performed with respect to that process.
2. Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in non-exclusive mode.
3. After an exclusive mode access to a synchronization variable has been performed, any other process’ next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable’s owner.” [Tanenbaum 95]

Entry consistency is weaker consistency than release consistency [Bershad et al. 91]. When using entry consistency, processes synchronize using locks and barriers, the memory only becomes consistent on entry to the critical section [Bershad et al. 93], [Zekauskas et al. 94].

There is a weakening of the consistency models from strict to entry consistency. Weaker models reduce the amount of network traffic hence the performance of the system improves. Thus, weaker consistency models have been used in an attempt to achieve better performance in distributed systems. However, this makes the programming model more complicated and

makes weaker consistency the concern of operating systems and language designers [Mosberger 94].

2.3 Synchronization

Synchronization can be managed by a synchronization manager, as in the case of page-based systems, or it can be made the responsibility of the application programmer, using explicit synchronization primitives, as in the shared variable implementation. Finally, it can be made the responsibility of the system developer, as in object based implementations, with synchronization being implicit at application level [Nitzberg et al. 94].

2.4 Data location and access

When a process requires a piece of non-local data the system must include a mechanism to find and retrieve this data. If the data is not migrated or replicated, this is trivial since the data exists only in the central server or remains fixed. However, if the data is allowed to migrate or is replicated there are several possible solutions to the location problem. In their paper Li and Hudak [Li, et al. 89] give several possible solutions. They subdivide the solutions into centralized and distributed manager approaches as follows:

- Centralized approaches; and
- Distributed manager approaches.

2.4.1 Centralized approaches

In these approaches there is a single centralized manager for the whole shared memory.

- *Monitor-like centralized manager approach*, where a central memory manager acts like a monitor. It synchronizes all access to each piece of data, keeps track of all replicated copies of the data through the copy set information and has information about the owner of any page. Any machine requiring access to a page sends a request to the manager. The owner of a page is the machine that has write privileges on that page and the copy set is the information regarding the location of all replicated pages on the network.
- *Improved centralized manager approach*, where, as opposed to the monitor-like approach, the access to data is no longer synchronized by the central manager. The central manager still maintains the copy set of the replicated pages and ownership information. Thus, any machine requiring access to a page still sends a request to the manager which has the information regarding the owner of that page.

2.4.2 Distributed manager approaches

The centralized approaches can cause a potential bottleneck. The following approaches provide a means to distribute the management tasks among the machines.

- *Fixed distributed manager approach*, where each machine is given a predetermined subset of the pages to manage. A mapping function, say a hashing function provides the mapping between pages and machines. A page fault causes the faulting machine to apply the mapping function to locate the machine on which the manager resides. The faulting machine can then get the location of the true page owner from the manager of that page.
- *Broadcast distributed manager approach*, where each machine manages the pages it owns and read and write requests cause a broadcast message to be sent to locate the owner of the required page. A write broadcast request results in the owner invalidating all pages in its copy set and itself and sending the page to the requesting machine. A read

request causes the owner to send a copy of the page to the requesting machine and to add to its copy set.

- *Dynamic distributed manager approach*, where a write request results in the ownership being transferred to the requesting machine. The copy set information moves with the ownership. Each machine maintains a table with a variable *probowner*, the probable owner of each page, which provides a hint to the actual owner of the page. If *probowner* is not the actual owner it provides the start of a sequence through which the true owner can be found. The *probowner* field is updated whenever an invalidation request is received through a broadcast message.
- *Improvement using fewer broadcasts*, where a reduced number of broadcasts is required compared to the previous two distributed approaches. The latter required a broadcast to be issued for every invalidation updating the owner of the page. This approach still uses the *probowner* variable but only enforces a broadcast message updating *probowner* after every M page faults.
- *Distribution of copy sets*, where copy sets are maintained on all machines which have a valid copy of the data. A read request can be satisfied by any machine with a valid copy which then adds the requesting machine to its copy set. Invalidation messages are propagated in waves through the network starting at the owner which sends invalidation messages to its copy set which in turn send invalidation messages to their copy sets.

2.5 Heterogeneity

Sharing data between heterogeneous machines is an important problem for distributed shared memory designers [Nitzberg et al. 94]. Data shared at the page level is not typed, hence accommodating different data representations of different machines, languages or operating systems is a very difficult problem. The Mermaid approach mentioned in [Li et al. 88] is to only allow one type of data on an appropriately tagged page. The overhead of converting the data might be too high to make DSM on a heterogeneous system worth implementing [Tam et al. 90].

2.6 Scalability

One of the benefits of distributed systems [Nitzberg et al. 94], [Tanenbaum 95] is that they scale better than many tightly-coupled, shared-memory multiprocessors. However this advantage can be lost if the scalability is limited by bottlenecks. Just as the buses in tightly-coupled multiprocessor systems limit their scalability so too do operations which require global information or distribute information globally in distributed systems such as broadcast messages [Nitzberg et al. 94].

2.7 Replacement Strategy

Replacement strategies are required to decide which blocks on a machine should be replaced by blocks of data being copied or migrated to that machine. The replacement strategies used most often in DSM implementations are similar to the replacement strategies used in caching:

- least recently used. The block which has been the least recently used is replaced; or
- random replacement strategies [Nitzberg et al. 94], [Tanenbaum 95].

2.8 Thrashing

This is a problem when non-replicated data is required by more than one process or replicated data is written often by one process and read often by other processes. Strategies to reduce

thrashing, such as allowing replication have to be implemented [Nitzberg et al. 94], [Tam et al. 90].

3 Implementation

This section covers implementation issues, the approaches to implementation and the categories of current implementations. It covers, in more detail, the three categories Tanenbaum discusses in his book [Tanenbaum 95] and the implementations given as examples in his book.

3.1 Basic Schemes for Implementing DSM

There are four basic approaches for the implementation of DSM. Stumm and Zhou describe them as [Stumm et al. 1990] follows:

- Central Server;
- Migration;
- Read Replication; and
- Full Replication Schemes.

3.1.1 Central Server Scheme

This is the simplest scheme for implementing DSM, depicted in Figure 2. The central server maintains the only copy of the data and controls all accesses to the data. In fact, the central server carries out all operations on the data. Thus, a request to perform an operation upon a piece of data is sent to the central server which receives the request, accesses the data and sends a response to the requesting machine. The advantages of this scheme are that it is easy to implement, controls all synchronization, avoids all consistency related problems but it can introduce a considerable bottleneck to the system.

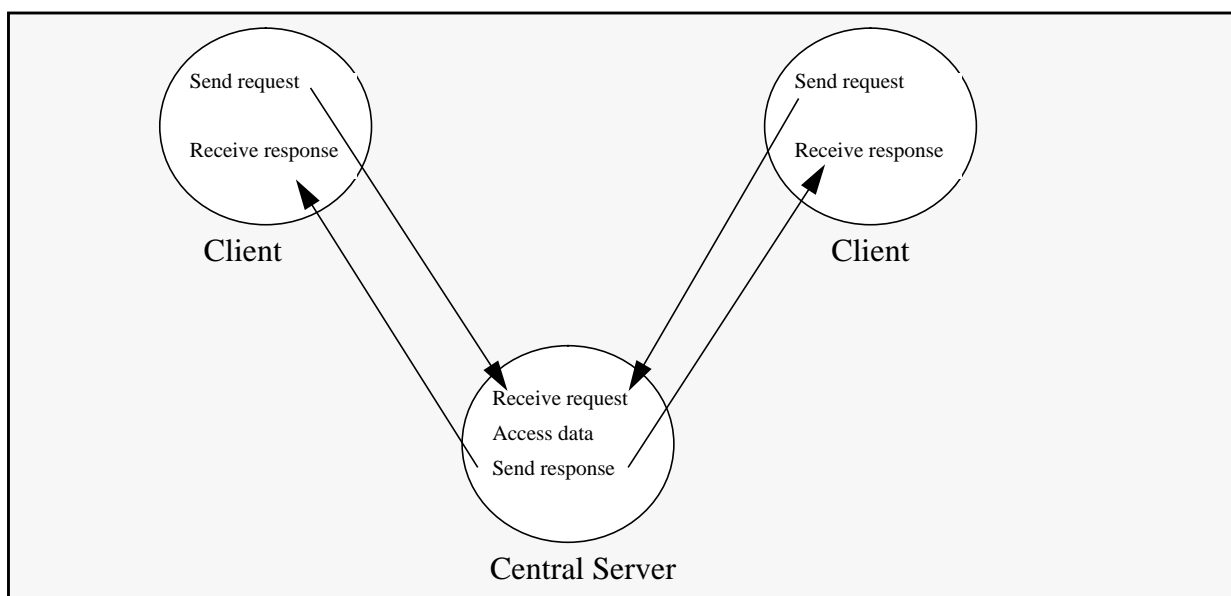


Figure 2. Central Server scheme

3.1.2 Migration Scheme

In the Migration Scheme, as in the Central Server Scheme (Figure 3), only one copy of the data is maintained on the network, however, control of the memory and the memory itself are now distributed across the network. A process requiring access to a non-local piece of data sends a request to the machine holding that piece of data, the machine sends a copy of the data to the requesting machine and makes its copy of the data invalid.

This scheme has the advantage of being able to be incorporated into the existing virtual memory system of the local operating system, like the central server scheme it has no consistency problems and synchronized access is implicit. However, it has the disadvantage of possibly causing thrashing when more than one process requires the same piece of data.

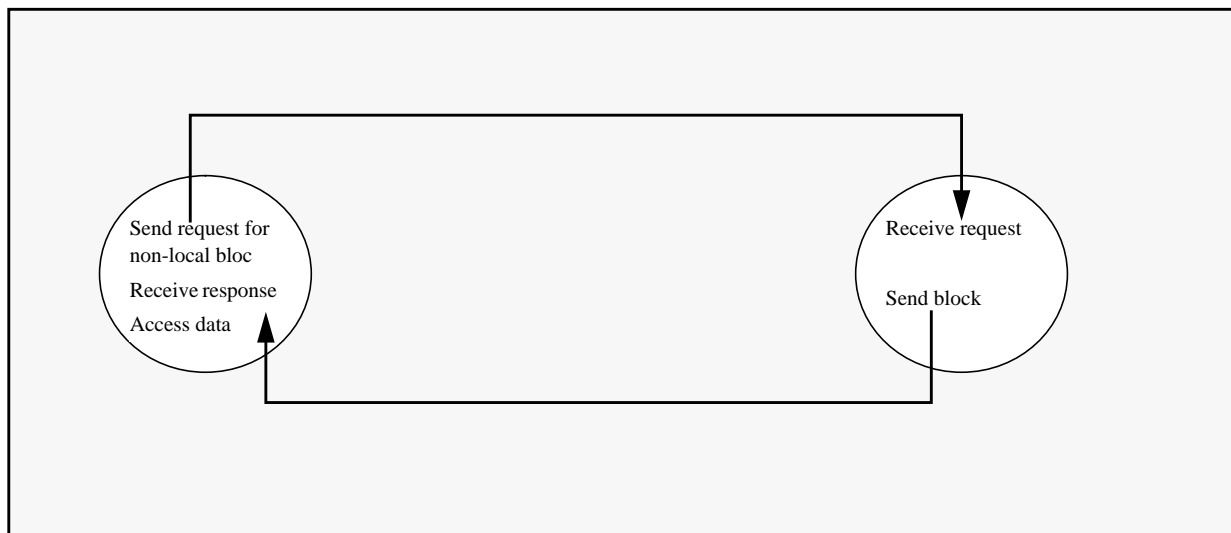


Figure 3. Migration scheme

3.1.3 Read Replication Scheme

The read replication scheme (Figure 4) allows multiple read-only copies of a piece of data and a single read/write copy of the data over the network. A process requiring write access to a piece of data sends a request to the process which currently has write access to the data. All existing read-only copies of the data are invalidated before the access is granted and the requesting process can alter the data.

The advantage that this scheme offers is that multiple processes can now have read access to the same piece of data, making read operations less expensive, however it can increase the cost of write operations since multiple copies have to be invalidated. Thus, this scheme would be indicated in an application where the number of reads far exceeds the number of writes.

3.1.4 Full Replication Scheme

The full replication scheme (Figure 5) allows multiple readers and writers. One method of keeping the multiple copies of the data consistent is to implement a global sequencer which attaches a sequence number to each write operation which allows the system to maintain sequential consistency.

This scheme reduces the cost of data migration and invalidation, when a write is requested, but introduces the problem of maintaining consistency

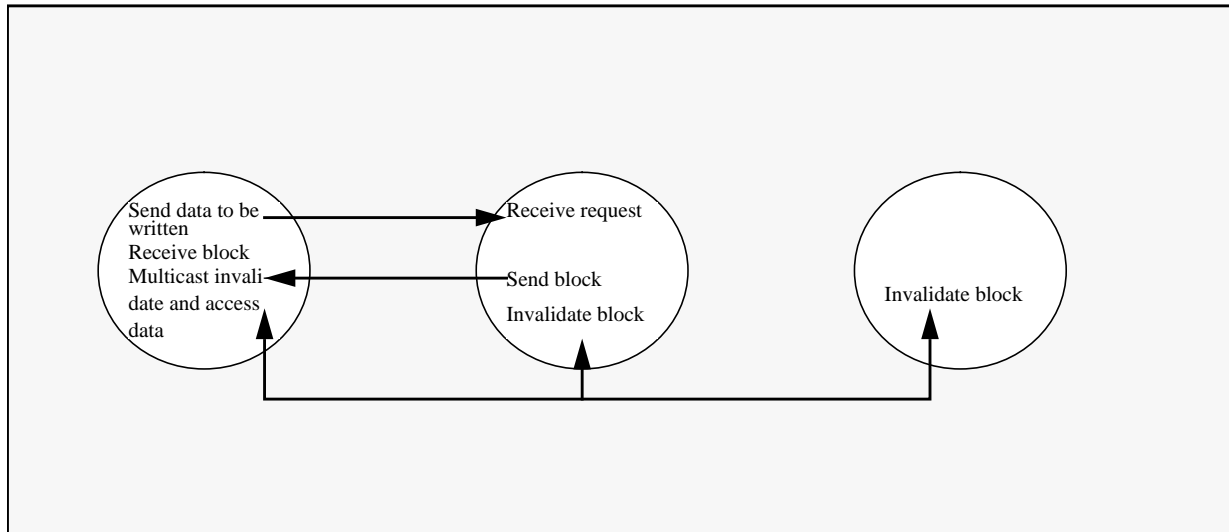


Figure 3. Read Replication scheme

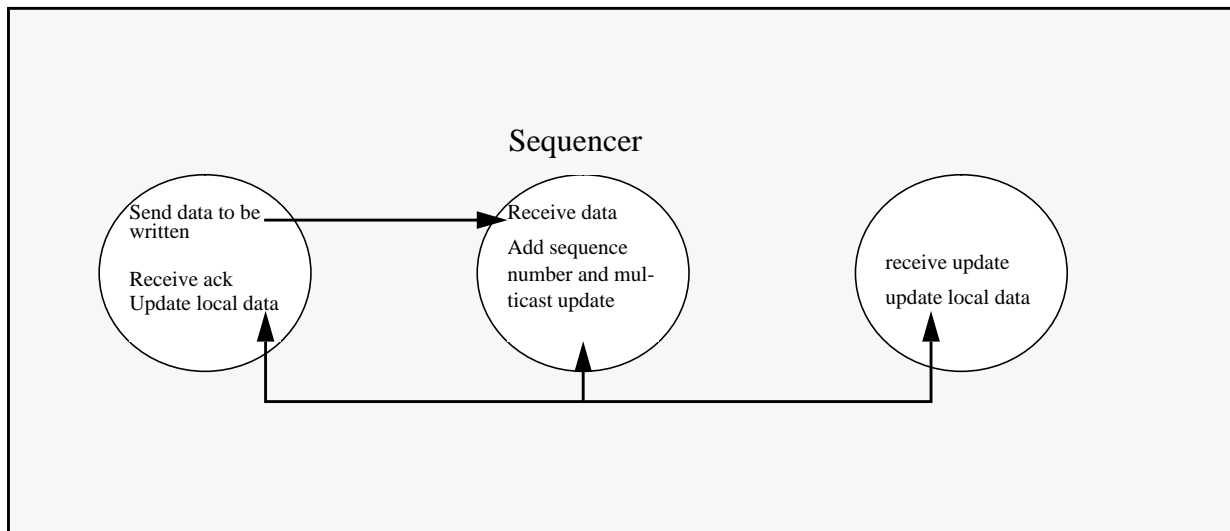


Figure 5. Full Replication scheme.

3.2 Implementation Categories

Generally DSM implementations can be divided into various categories. Nitzberg and Lo [Nitzberg et al. 94] categorize implementations into:

- *Hardware Implementations*, “extend traditional caching techniques to scalable architectures”, i.e. implement DSM with the addition of specialized hardware to the system;
- *Operating System and Library Based Implementations*, “achieve sharing and coherence through virtual-memory-management mechanisms”; and
- *Compiler Implementations*, “shared accesses are automatically converted into synchronization and coherence primitives”.

These implementation categories are similar to those used by [Coulouris et al. 93]:

- *Hardware-based Implementations*. Which “rely on specialized hardware to handle load and store instructions applied to addresses in DSM, and communicate with remote memory modules”;
- *Page-based Implementations*. Which “implement DSM as a region of virtual memory occupying the same address range in the address space of every participating process”;
- and
- *Library-based Implementations*. Where DSM is implemented “through communication between instances of the language run-time.”.

Tanenbaum [Tanenbaum 95] includes a figure showing all shared memory machines in relation to one another as follows in Figure 6. The three types of DSM shown here, managed by MMU (Memory Management Unit, a hardware device), managed by OS, and managed by language runtime system match the categories given by both [Nitzberg et al. 94] and [Coulouris et al. 93]. However, in his book [Tanenbaum 95] Tanenbaum concentrates on software based systems since he considers that hardware based solutions do not constitute true DSM.

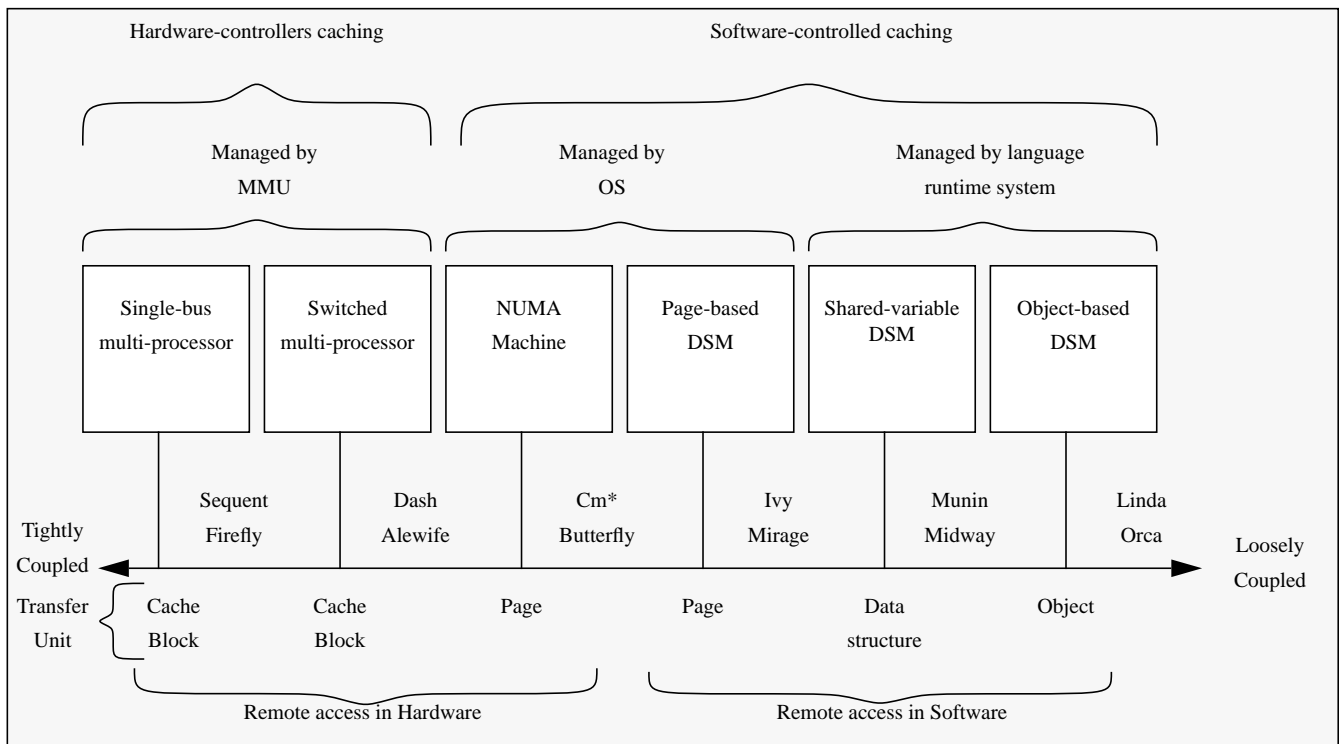


Figure 6. The spectrum of shared memory machines [Tanenbaum, 95]

The software based solutions discussed in [Tanenbaum 95] are essentially divided into the same areas as the software implementations listed by the other authors above with the addition of shared variable techniques. These are an extension of the page based solutions where a suitably annotated section containing the shared variables is shared. Tanenbaum’s categories follow:

- Page-Based Techniques;
- Shared Variable Techniques; and
- Object Based Techniques;

3.2.1 Page-Based Techniques (IVY type, Dash)

The syntax of memory access in this type of DSM is the same as that in a shared memory multiprocessor. Variables are directly referenced unless they are shared by more than one process in which case they would have to be protected explicitly by locks.

Page-based DSM is an attempt to emulate multiprocessor cache. The total address space is subdivided into equal sized chunks which are spread over the machines in the system. A request by a process to access a non-local piece of memory results in a page fault, a trap occurs and the DSM software fetches the required page of memory and restarts the instruction.

In page-based DSM a decision has to be made whether to replicate pages or maintain only one copy of any page and move it around the network. In the latter case a situation can arise where a page is moved backwards and forwards between two or more machines which share data on the same page and are accessing it often, this can drastically increase network traffic and reduce performance. Replication of pages can reduce the traffic, however, consistency must then be maintained between the replicated pages.

Some of the consistency models based on those used for cache consistency can be used in distributed systems using page-based DSM. The weakening of consistency models can improve performance

The granularity of the pages has to be decided before implementation. A substantial overhead in the transportation of data across the network is the setup time, hence the larger the page the cheaper it is to transport it across the network [Tanenbaum 95]. Another advantage of large pages is that processes are less likely to require more than one page. However, large pages can cause false sharing, where two processes use two unrelated variables on the same page [Nitzberg et al. 94]. This can result in the page moving backwards and forwards between the two machines unnecessarily. A solution for this is for the compiler to anticipate false sharing and locate the variables appropriately in the address space. Smaller pages may prevent false sharing but increase the chance that more than one process will require the pages.

Page-based distributed shared memory is a simple, familiar, well understood model which is easily implemented and can run existing multiprocessor programs. However, the biggest issue in this implementation is that it can exhibit poor performance, because of the network traffic generated by false sharing and strict consistency protocols [Tanenbaum 95].

Tanenbaum gives IVY type implementations as examples of page-based solutions. IVY as described in [Hellwagner 1990] and [Li et al. 89] is regarded as the pioneering work on DSM. Memory mapping managers are used to map the addresses, local memories and the distributed memory. The unit of granularity is the page. The memory coherence protocol used in IVY is derived from the Berkley cache coherence protocol [Tam et al. 90]. It uses page ownership and invalidation. IVY was developed in an attempt to explore whether DSM was feasible. Hellwagner [Hellwagner 1990] states that this was shown even when the DSM was implemented in software.

3.2.2 Shared Variable Techniques (Munin, Midway)

In shared variable techniques only the variables and data structures required by more than one process are shared. The problems associated with this technique are very similar to those of maintaining a distributed database which only contains shared variables [Tanenbaum 95].

In the current implementations of the shared-variable technique the shared variables are identified as type shared [Tanenbaum 95]. Synchronization for mutual exclusion is achieved using special synchronization variables. This makes synchronization the responsibility of the programmer.

The replication of the shared variables brings with it the problem of how to maintain consistency. While updating a page required the rewriting of the whole page, in the shared

variable implementation an update algorithm can be used to update individually controlled variables [Tanenbaum 95]. Nevertheless, a consistency protocol has to be decided upon when the system is being implemented.

This approach is a step towards ordering shared memory in a more structured way than page based systems [Tanenbaum 95]. However programmers must still provide information about which variables are shared and control access to shared variables through semaphores and locks which makes programming more difficult and it is possible for the programmer to compromise consistency.

Munin is given by Tanenbaum as an example of shared-variable based DSM. It is described in [Bennett et al. 90] and [Hellwagner 1990]. The address space in each machine is divided into shared and private address space. The latter contains the runtime memory coherence structures and the global shared memory map. The memory on each machine is viewed as a separate segment. Each data item in Munin is provided with a memory coherence mechanism suitable for the access it requires. The type of each data item is to be supplied for each item by either the user or a smart compiler. A memory fault will cause the runtime system to check the object's type and call a suitable mechanism to handle that object type.

Midway [Bershad et al. 93] is another example, given by Tanenbaum, of the shared variable technique. Bershad, et al. write that Midway supports multiple consistency models in each program, which may all be active at the same time, i.e. processor consistency, release consistency, or entry consistency. The implementation of Midway is made up of three main components [Bershad et al. 93]:

- a set of keywords and function calls used to annotate a parallel program;
- a compiler which will generate and maintain reference information; and
- a runtime system to implement several consistency models.

Tanenbaum [Tanenbaum 95] says “Programs in Midway are basically conventional programs written in C, C++ or ML, with certain additional information provided by the programmer.”

3.2.3 Object Based Techniques (Orca, Linda)

An object can be defined as “a programmer-defined encapsulated data structure” which comprises data and methods [Tanenbaum 95]. In object-based DSM memory can be conceptualized as an abstract space filled with objects. Processes on multiple machines share these objects. The management and location of objects are handled at the level of the object operation by the operating system. Thus the programmer does not have to worry about synchronization as this is handled implicitly in the object definition.

One of the issues to be decided before implementation is whether to allow replication. If it is not allowed the data can be thought of as immutable and all access must be through the methods associated with the only copy of each object. This may lead to poor performance. Objects can instead be allowed to migrate or be replicated. Migration can improve performance by moving objects to where they are needed. Replication can improve performance even further but consistency has to be taken into consideration.

Tanenbaum [Tanenbaum 95] cites Orca and Linda as examples of object based distributed shared memory. The shared data in Orca are encapsulated in data objects, which are instances of user-defined data types. The authors of the Orca language say “the key idea in Orca is to access shared data structures through higher level operations”. Thus, the programmer is able to define operations to manipulate the data structures. Only these operations can be carried out upon the encapsulated data objects. The parallel processes are created through the explicit creation of sequential processes that execute in parallel with one

another [Bal et al. 92].

Linda [Carriero et al. 86] is a language which consists of four simple operators which are added to a host language to turn it into a parallel programming language. Linda processes communicate through a “globally shared collection of ordered tuples” these tuples exist in a global abstract tuple space. Despite Tanenbaum’s apparent enthusiasm for the object-based model, the authors of Linda say the object-based model is “powerful and attractive, but utterly irrelevant to parallelism” [Carriero et al. 86], [Tanenbaum 95].

4 Conclusion

Since one of the main goals of Distributed Systems is transparency, the achievement of this when DSM is implemented is only possible if the use of the shared memory is completely invisible. Hellwagner [Hellwagner 1990] says that the overall objective in DSM is to reduce the access time for non-local memory to as close as possible to the access time of local memory. Thus, the major research push appears to be in the area of the reduction of access times for the distributed memory. This has been looked at from the perspective of various consistency models, data location and access methods and the granularity and structure of the shared data. All of these are related to a reduction in the number of messages being sent between the distributed machines since this is seen as the major overhead. Centralized systems monitoring the shared data and controlling the access have been implemented and have been largely rejected because they can cause bottlenecks and require a large number of messages between the nodes and the central server. The move to distributed control of the shared memory and the replication of shared data has brought with it many problems related to the maintenance of coherence. The relaxation of consistency models has led to a reduction in the number of messages but a complication of the programming model, particularly with the addition of explicit synchronization primitives to identify and control access to shared data. Further research could be done in this area particularly with reference to the use of distributed synchronization algorithms and the use of “lazy” synchronization techniques. In the latter a lock on a critical region is not released by a process until requested by another process.

5 Acknowledgements

I would like to thank my supervisor Professor Andrzej Goscinski and all the members of the RHODOS group for their assistance in this paper.

6 Bibliography

- [Adve et al. 91] Sarita Adve, Mark Hill, Barton Miller, Robert Netzer, *Detecting Data Races on Weak Memory Systems.*, Proceedings 18th Annual International Symposium on Computer Architecture, May 1991.
- [Bal et al. 92] H. Bal, M. Kaashoek, A. Tanenbaum, Orca: A Language for Parallel Programming of Distributed Systems., IEEE Transactions on Software Engineering, Vol. 18, No. 3, March 1992.
- [Bennett et al. 90] J. Bennett, J. Carter, W. Zwaenepoel, *Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence.*, Proceedings Second ACM Symposium on Principles and Practice of Parallel Programming, ACM, pp. 596-615, Dec. 1984.
- [Bershad et al. 91] Brian N. Bershad, Matthew J. Zekauskas, *Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors*, CMU Technical Report CMU-CS-91-170, September 1991.
- [Bershad et al. 93] B. Bershad, M. Zekauskas, W. Sawdon, *The Midway Distributed Shared Memory System*, Proceedings IEEE COMPCON Conference., IEEE, pp 528-537,

- 1993.
- [Carriero et al. 86] Nicholas Carriero, David Gelernter, *The S/Net's Linda Kernel.*, ACM Transactions on Computer Systems, Vol. 4, No. 2, May 1986, Pages 110-129.
- [Coulouris et al.93] G. F. Coulouris, J. Dollimore, T. Kindberg, *Distributed Systems. Concepts and Design.* Addison-Wesley Publishing Company.
- [Hellwagner 1990] H. Hellwagner, *A Survey of Virtually Shared Memory Systems*, TUM-19056 Technische Universitat Munchen, Dec. 1990.
- [Jul et al. 88] E. Jul, H. Levy, N. Hutchinson, A. Black, *Fine-Grained Mobility in the Emerald System.*, ACM Transactions on Computer Systems, Vol.6, No. 1, February 1988, pp 109-133.
- [Lamport 79] L. Lamport, *How to make a multi processor computer that correctly executes Multiprocess programs.*, IEEE Transactions on Computers, Vol. C-28, September 1979, pp 690-691.
- [Levelt et al. 92] Levelt, W.G., Kaashoek, M.F. Bal, H.E., and Tanenbaum, A.S.: *A Comparison of Two Paradigms for Distributed Shared Memory*, Software--Practice and Experience, vol. 22, Nov. 1992, pp. 985-1010.
- [Li et al. 89] K. Li, P. Hudak, *Memory Coherence in Shared Virtual Memory Systems*, ACM Transactions on Computer Systems, Vol. 7, No. 4, pp 321-359, November 1989.
- [Li et al. 88] Kai Li, Michael Stumm, David Wortman., *Shared Virtual Memory accommodating Heterogeneity*, Technical Report CS-TR-210-89, February 1989.
- [Libes 85] Don Libes, *User-Level Shared Variables*, Proceedings, Tenth USENIX Conference, Summer 1985.
- [Mosberger 94] D. Mosberger, *Memory Consistency Models*, Tech., Report TR 93/11, Dept. of Computer Science, Univ. of Arizona, 1993.
- [Nitzberg et al. 94] B. Nitzberg, V. Lo, *Distributed Shared Memory: A Survey of Issues and Algorithms*, In Casavant T.L. and Singal M. (eds), Readings in Distributed Computing Systems, IEEE Press, 1994, pp 375-386.
- [Ramachandran et al. 91] U. Ramachandran, M. Khalidi, *An Implementation of Distributed Shared Memory*, Software - Practice and Experience, Vol.21(5), John Wiley and Sons, Ltd., May 1991.
- [Sinha 93] Himansu Shekhar Sinha, *Mermera: Non-Coherent Distributed Shared Memory for Parallel Computing.*, PhD Thesis, Boston University, Boston., April 1993.
- [Stumm et al. 1990] M. Stumm S. Zhou, *Algorithms Implementing Distributed Shared Memory*, IEEE Computer, Vol. 23, No 5, May 1990, pp 54-64.
- [Tam et al. 90] M. Tam, J. Smith, D. Farber, *A Survey of Distributed Shared Memory Systems.*, ACM SIGOPS, June 1990.
- [Tanenbaum 95] A. Tanenbaum, *Distributed Operating Systems.*, Prentice Hall, 1995.
- [Zekauskas et al. 94] Matthew J. Zekauskas, Wayne A. Sawdon and Brian N. Bershad, *Software Write Detection for a Distributed Shared Memory*, Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI), 1994.