Design Issues in Implementation of Distributed Shared Memory in User Space

Varun Chandola
{chandola@cs.umn.edu}

Abstract ***Distributed Shared Memory (DSM) [1] has become a very popular paradigm in distributed systems. A DSM is essentially a way of seamlessly sharing the physical memories of loosely connected systems. An implementation of a DSM can be categorized as page-based, shared-variable based or object based depending on the granularity of the data being shared. This paper discussed the design and implementation of an object based DSM running in user space. The design discussed the decisions made to achieve two important parameters associated with DSM viz. – performance and consistency. The implementation of the design is done in JAVA and a performance analysis of various consistency schemes is done to help users decide which scheme to use to optimize the balance between the performance of the DSM system and the consistency of the shared objects***

1 Introduction

Sharing data is an essential requirement of any distributed system. Note that when I refer to a distributed system it stands for a multi-computer architecture in which each node is an independent machine connected to each other through a network. Sharing data in a multi-processor architecture is relatively easier, since all of the nodes share the same system bus and hence have a uniform view of the physical memory. On the other hand a multi-computer system does not enjoy such hardware privileges. So sharing data becomes a problem which has to be tackled in the software (either inside the Operating System or as a user-level application) and not in hardware as in multi-processor systems. Traditional methods of data sharing *viz.* message passing via sockets are not appealing from a programmer's perspective, in which he or she has to explicitly take care of the networking issues. A DSM provides an abstraction to the programmer of a uniform shared memory located across different machines.

Since a DSM system involves moving of data from one node to another which are on typical networks, performance is an important criterion in the design of a DSM system. Just as is the case with multi-processor systems, since same copies of data might reside on different nodes, consistency between these copies is also another major issue.

DSM systems can be classified into three broad categories [2]:

- Page-based DSM – in which the unit of data sharing is a memory page.
- Shared variable based DSM – in which the unit of data sharing is a variable.
- An object based DSM – in which the unit of data sharing is an object.

This paper presents design of an object based DSM that runs in the user space[1]. The choice of objects as units of granularity over a page or shared variables is because of the modularity and flexibility offered by objects. Moreover, objects eliminate *false sharing* which will be discussed in the next section. Another reason is that integration of object based DSM with the object oriented languages is easy to achieve. The design also provides flexibility in terms of consistency models used by allowing the user applications to specify under what consistency scheme they want a particular object to be shared.

The rest of the paper is organized as follows. Section 2 addresses various issues involved with

---

[1] The design is not a pure object based design because in typical object based DSM like CLOUDS [1] the user application cannot differentiate between a shared and a normal object. It is the responsibility of the runtime system to access the object from local store or remote location. In my design, the user application is aware of the objects which are to be shared among multi-computers. So it can be thought of as a hybrid between shared variable based and object based DSM.

the design of a DSM. Section 3 describes various types of DSM systems *viz. – page based, shared variable based* and *object based* and how they individually deal with the issues in section 2. The next section briefly discusses the related existing implementations of DSM. Section 5 describes the design of the object based DSM proposed in this paper. Section 6 gives the details of the JAVA implementation of this design. The performance of the system for different consistency schemes is measured in section 7 and the trade-off between performance and consistency is analyzed.

2 Issues in DSM design

Any DSM system has to rely on the underlying message passing technique across the network for data exchange between two computers. The DSM has to present a uniform global view of the entire address space (consisting of physical memories of all the machines in the network) to a program executing on any machine. A DSM manager on a particular machine would capture all the remote data accesses made by any process running on that machine.
A design of a DSM would involve making following choices

- Where, with respect to the Virtual Memory Manager does the DSM operate?
- What kind of consistency model should the system provide?
- What should be the granularity of the shared data?
- What kind of naming scheme has to be used to access remote data?

2.1 Relation with Virtual Memory Manager

The issue that arises here is from the fact that the virtual memory on a particular machine and the distributed shared memory (which is nothing but an abstraction) are inherently different. But the aim is to provide a uniform view of both to the user applications. A DSM can be implemented parallel to the VM so that in case of a page fault the VM can decide if the missing page is available locally or requires a remote access. For remote accesses the VM can ask the DSM manager to fetch the page. Moreover the VM also has to perform book-keeping operations like maintaining the page table entries for the remote pages. Such kind of design is closest to the shared memory in multi-processor systems.
An alternative design would be in which the DSM manager is built on top of the VM such that the VM is not aware of different kinds of memory accesses. The language provides capabilities such that certain data segments are fetched from remote addresses while rest are accessed locally. Performance-wise, the former approach is better because it eliminates one step of writing the data into the memory buffers.

2.2 Choice of consistency model

Consistency is a very important parameter in a design of DSM. As discussed earlier, the possible replication of shared data across multiple nodes mean that different machines have different copies of the data. Thus the DSM has to maintain consistency in the state of these copies of the same data. This problem is similar to cache coherency in multi processors. Several solutions which are also prevalent in other scenarios are proposed for this which can be broadly classified as

- **Write-all scheme:** In this scheme, if a data is modified at some node, the updated copy of the data is shipped to all the nodes which are currently holding a copy of that data. This scheme is simple to implement and most reliable, but it is expensive in terms of network traffic because the whole data has to be shipped across multiple clients.
- **Write-invalidate scheme:** In this scheme, if a data is modified at some node, instead of sending the entire data across, the host sends an invalidate message to all other machine holding a copy of that data. Thus this scheme puts lesser load on the network and hence performs better. Whenever a machine tries to access an invalid data, the DSM manager fetches the valid data from the host.
- **Lock-based synchronization:** The above two schemes enforce "weak" consistency in the sense that they do not guarantee that

events are globally ordered. With locks or some similar synchronization primitive, a release consistency can be enforced. This means that a node accessing some data can acquire a lock such that no other node can access it at that time. Once the node comes out of the critical section, it updates all the nodes which have the copy of that data.

Most of the implementations use the first two schemes and usually keep synchronization separate from consistency model. In my design also I have allowed the first two schemes to be implemented. It is important to note that the choice of the consistency protocol will affect consistency of the system as well as the performance of the system. Unfortunately both these parameters act opposite to each other so there is a trade-off between performance and the consistency of the system.

2.3 Granularity

Granularity refers to the size of a data unit in which it exists in the distributed shared memory. This is an important decision which essentially governs the design of a DSM. The immediate successor of shared memory from multiprocessor world would have a page as the unit for data transfer. But it has its own disadvantages which are discussed in section 3.

2.4 Naming scheme

When a process wants to access remote data it has to know on which machine does the data reside and fetch it from there. Since all data (or at least the shared one) is visible to all the machines, there has to be a unique naming mechanism to avoid any conflicts.
One possible solution is to have a logical global address space. The VM manager at each node performs the translation of the logical address to get the location of the data segment on a remote machine. But such an approach would not be useful if the granularity of shared data is less than a page. In such a case the calling process will have to possess explicit knowledge of the remote location of the data which it wants to access.

3 Types of DSM

Based on the granularity of the data we can divide DSM into three categories *viz. page-based, shared-variable based* and *object-based.* There are other criterion based on which we can categorize the designs proposed so far. These can be based on the consistency model or user-space vs. kernel-space etc.

3.1 Page based

As the name suggests, this DSM has a memory page as the unit of data sharing. Page based DSM closely emulate the shared memory in multiprocessor realm. The entire address space (spread over all the computers) is divided into pages. Whenever the virtual memory manager (VMM) finds a request to an address space which is not local, it asks the DSM manager to fetch that page from the remote machine. Such kind of page fault handling is simple and similar to what is done for local page faults.

To improve performance, most implementations do *replication* of the pages so that same page does not have to be transferred again and again. This especially improves performance for read only pages. Keeping multiple copies of same page lead to the issues of consistency between these copies. If pages are not replicated, achieving sequential consistency is not difficult. But for replicated copies page based DSM generally follow the same scheme as cache coherency schemes for shared memory in multiprocessors.

Naming issues in page based DSM are usually addresses by having a global address space and the VM manager identifying page faults as local or remote. The VM manager itself or a separate DSM manager then fetches the remote page by translating the logical address.

The disadvantage of a page based DSM is that it is not very efficient performance-wise. Most of the time the actual data being shared is much smaller than the page size hence there is lot of wastage in terms of shared space. Moreover it also leads to *false sharing.* Most of the programming languages organize data into smaller units like a package or object or some data structure. Hence it would

make more sense to share data in these units which is the idea behind the next two types of implementations.

3.2 Shared variable based

Shared variable based DSM present a more structured approach towards sharing of data between multiple machines. The basic idea is to let the applications decide as to which variables are to be shared. The DSM manager will maintain a database of the shared variables present in the logical shared memory. The primary difference from the page based approach is that each shared variable is individually managed thereby eliminating the possibility of *false sharing*.

This approach expects the programmer to explicitly declare which variables are to be shared globally. These variables are then stored by the DSM manager of that machine. Any process on a different machine can access this variable by requesting through that machine's DSM manager.

The advantage which the shared variable DSM holds is that it provides a more structured way of sharing data between multiple computers. The consistency of the multiple copies of a variable is ensured using any one of the schemes discussed in section 2.2. Most of the implementations favor the write invalidate scheme because it ensures weak consistency without affecting the performance too much.

3.3 Object based

The final type of DSM is the object-based DSM in which an object is the unit of data sharing. The primary difference of this approach from the shared variable based approach is that in object based DSM all data is encapsulated as objects. Thus the programmer does not have to explicitly declare the objects which he/she wants to share globally. Moreover, since the data is accessible only through the methods defined, this scheme provides protection for the data which is not found in the shared variable approach.

Each object contains the actual data and methods which can be invoked to obtain or set the data. Note that the direct access of data is not allowed. Instead the process can invoke the methods available for that object. Depending on the location of the object with reference to the calling process, the method can be invoked locally or remotely. The runtime and/or the Operating System take care of invoking the method without the application being aware of the object's location.

This approach enjoys the advantages of the shared variable approach in terms of managing each shared object separately. The consistency schemes can be applied similarly to the shared variable DSM.

The disadvantage of this approach is that all the data is forced to be encapsulated with objects, thereby putting an extra overhead in invoking methods to get or set the actual data. This overhead would be completely unnecessary for data which is not being shared. Hence this approach can be thought of as a trade-off between ease of programming and performance. One thing to note here is that object based DSM and shared variable based DSM are language provided features whereas the page based DSM requires the support of Operating System.

4 Related Works

Ivy [3] is a page based distributed shared memory system implemented on Apollo workstations which run across a typical Ethernet. Each process has a local and a shared address space. The process can map a remote address in its shared address space and access remote memory through a DSM manager which runs on every machine. Ivy uses a write-invalidate type of coherence scheme for ensuring consistency.

Aurora [4] is a distributed shared memory system based on a standard C++ class library and run-time system. It provides a shared data programming abstraction on distributed memory hardware. The system does not contain any language extensions and it does not require special hardware support.

Midway [5] is a DSM system that is based on sharing individual data structures. The consistency is maintained by explicitly synchronizing each access of shared variables i.e. the accesses are put in a critical section so that no other process can modify the data.

Other popular DSM implementations are Clouds[3] (page based), Munin [6] (shared variable based), Linda [2] and Orca [2](both object based).

5 Design of Object based DSM in user space

Fig 1 illustrates the way data is shared across different nodes in a multi-computer architecture.

5.1 Granularity

Object based DSM encapsulates the data in objects which are globally accessible.
Any process which wants to access the data invokes the methods provided by these objects. The reason behind choosing objects as unit of data access is that they provide a very structured and modular view of the shared memory. Moreover, they are performance wise better than page based systems. Shared variable approach also enjoys the same kind of benefits in terms of performance. But objects provide a layer of protection over the data since the data can be accessed only by invoking the methods provided by the object.

One optimization which I have made in my design is to create a hybrid of both shared variable as well as object based approach. So there will be following entities for any user application

- *Local variables, objects and data structures* – these will be local to the machine and will not be accessible to any other process.
- *Shared objects* – these are the globally shared objects which will be either located on one node or replicated across several nodes as per the consistency scheme.

5.2 Consistency scheme

As seen in earlier sections, the choice of the consistency scheme to be used governs the performance of the system to a great extent. Thus while a lock based scheme or a write all policy provides a greater level of consistency, it also has poor performance in comparison to a weaker consistency scheme such as write invalidate policy. In this design I provide the application to specify the type of consistency scheme they want to apply to a particular object. The three types of consistency schemes currently supported are

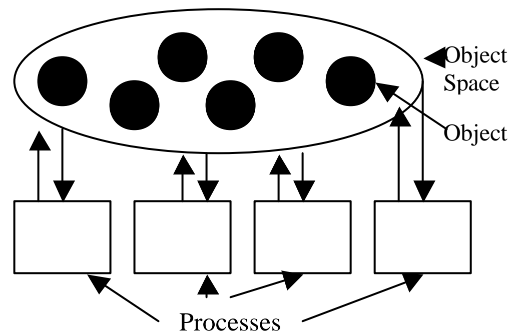

Fig 1 – The objects are present in a global object space and can be accessed by the processes by invoking the methods of that object

- *Write all*
- *Write invalidate*
- *Read only*

So any process can create an object and specify the consistency scheme which should be used by the DSM manager. Figure 2 shows the architecture of the design for two machines. This can be generalized to several machines.

5.3 DSM Manager

The DSM manager manages the shared objects in each node. At each node, the DSM manager maintains a table of all the objects whose copies are present in that node. There can be two types of shared objects in the table

- Objects created at that node. Thus these objects are the master copies.

- Objects replicated from remote nodes for a local process.

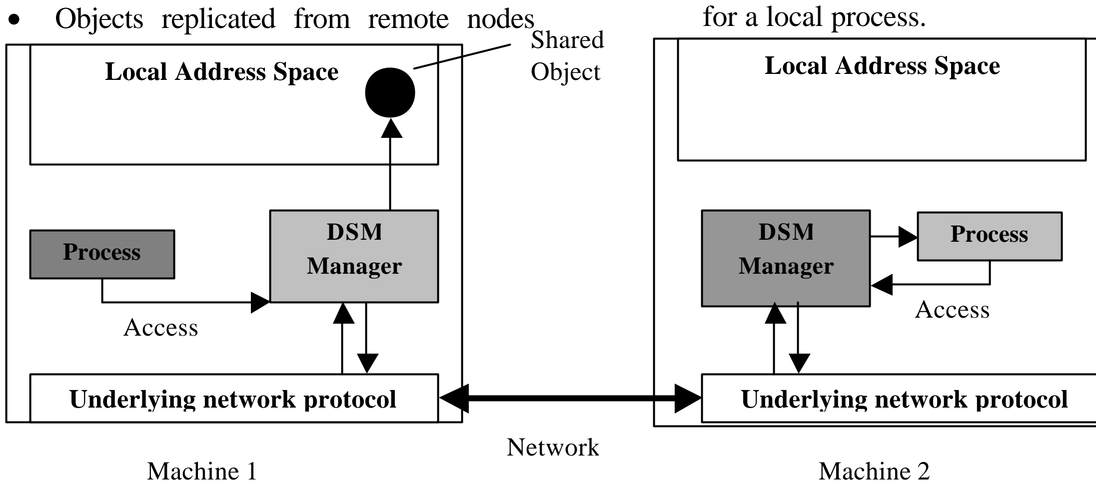

Fig 2 – Architectural overview of the DSM. An object to be shared globally is managed by the DSM manager at that node. Any process which wants to access a remote object requests its DSM manager. The DSM manager in turn contacts the DSM manager of the node which has the object and requests for the object.

5.4 Creation of a new object

A process encapsulates the data which it wants to share in a remote data object. This object internally contacts the DSM manager of that host and registers itself with it. The user process also has to specify the name of the object for identification. Globally the name resolution is done using the name of the object and the hostname where it was created. Hence the name specified by the process should be unique among all the objects created in that host machine.
The process has to specify the permissions for this object. This can be one of the following:

- *Read-only* – In this case the object is shared as read-only. Only the owner process can modify the object
- *Read and Write* – In this case any process can modify the data in the object.

The process also has to specify the consistency scheme to be used

- *Write all* – Under this scheme, whenever the object is modified, the updated copy of the object is sent to all the machines which have a copy of this object.
- *Write invalidate* – Under this scheme, whenever the object is modified, all copies existing on other machines are invalidated.

On finding a request for creation of a new object, the DSM creates a new remote object[2] and stores it in an object table which it maintains.

5.5 Reading a shared object

If a process has to read a shared object, it invokes the method on the object to get the data. The method in turn connects to the local DSM manager. The DSM manager checks if there exists a valid copy of the object in its object table. If it does, the data is returned to the caller. Otherwise the object is fetched from the host machine and stored locally.

5.6 Modifying a shared object

A process can modify a shared object either if the object is not read-only or else the

[2] This remote object is different from the object which the process possesses. The one which the object possesses does not actually contain the data but methods which fetch the data from the DSM manager. The remote object stored with the DSM manager is the actual shared object which migrates or is replicated across the network.

process is the owner of the object. To modify an object the process requests the DSM manager which checks if it has a *non-dirty* copy of the object in its object table. By non-dirty I mean that the previous change to this copy has been updated to other nodes also. If there is a *non-dirty* copy, the manager updates the data and then propagates the change to the other nodes sharing the same object according to the associated consistency scheme.
If the object has write-all consistency, then the DSM manager ships the updated copy to all the nodes which share the object. If the object has write-invalidate consistency, the DSM manager sends an invalidate signal to all nodes sharing the object. On receiving the invalidate signal, the managers set the invalid bit of the object to 1.
From the time of updating the local copy till updating all the nodes sharing the object, the *dirty* bit of the object is set to 1. It is reset only once the manager has propagated the change across the network.
In case the manager finds the object to be *dirty*, it waits for the object to become *non-dirty* again before the next update is made.

6 Implementation of the design

The above design has been implemented in JAVA. *Table 1* describes the various classes and the methods and fields for each class.
The communication mechanism used is message passing using JAVA sockets. A DSM manager runs at every node and listens for requests from local processes and remote DSM managers. For each request it forks a thread and deals with the request.

A user application encapsulates the data which it wants to share globally in **RemoteObject.** Currently the data types which can be shared are integer, character, float and String.

7 Performance Analyses of Consistency Schemes

The proposed design allows a user application to specify the consistency policy to be used with the object. Two policies are supported *viz. write-all* and *write-invalidate*. I conducted experiments to illustrate the performance of both the schemes.
The experiment was run on five computers of Computer Science Dept., University of Minnesota. All of these systems had SunOS running on sun4u sparc machines.
Two types of test codes were run over the DSM
a). One node performed a computation and modified a shared object which was replicated on all machines.
b). One node performed a computation and modified a shared object which was replicated on all machines. All other nodes access the replicated object and perform some more computation.

The two measurements which I focus on are – (a) total elapsed time for the experiment to run (et) and (b) sum of the time taken by each processor to finish its job (st).
The results for the two test codes are given in *Table 2(a) and (b).*

The results show that using any replication scheme is better than not using any, for distributed programs in which a shared object is accessed by many remote processes.

For the first test case, the elapsed time is slightly more for the write-all scheme than the write-invalidate scheme. This is because the object is replicated on all the sharing nodes for write-all. But for write-invalidate only the invalid bit is transferred. [3] The difference is more pronounced for the total time (st) measurement because here we are adding up the time taken at each node. For the second test case both *ttr* and *st* measurements are more for write-invalidate scheme than the write-all scheme.
This would look counter-intuitive after the first set of results.

[3] The size of the object was made large (a 1000 character string) so that difference between shipping the entire object versus shipping the invalidate bit only is more pronounced.

Class Name	Fields	Methods
RemoteObject – *This class represents the object in a process. It does not contain the actual data, but methods to access the actual remote objects.*	**data :** *The actual data being shared* **dataType :** *int/float/char/String* **accessType :** *write-all / write-invalidate* **readWrite :** *read-only/read & write* **name :** *A locally unique name of object* **ownerID :** *A key for owner identification*	**getData() :** *fetches the data from the shared object by requesting the DSM Manager* **setData() :** *modifies the value of a remote object through the DSM Manager*
DSMObject – *This class represents the actual object being shared across the network.*	**data :** *The actual data being shared* **dataType :** *int/float/char/String* **accessType :** *write-all / write-invalidate* **readWrite :** *read-only/read & write* **name :** *A locally unique name of object* **ownerID :** *A key for owner identification* **hostname:** *Address of the machine possessing the master copy* **dirtyBit :** *A bit denoting if the object has been modified but not updated yet* **validBit :** *A bit denoting if the object has been invalidated under the write-invalidate policy* **users:** *A vector containing addresses of all the nodes having a copy of the object.*	**getData() :** *returns the data of the shared object* **setData() :** *modifies the value of the remote object*
Manager – *This class represents the DSM manager which manages the shared objects at each node.*	**objectTable:** *A table containing all the shared objects at a particular node.*	**local() :** *Called when a local process creates a new object or requests for a remote object* **remote() :** *Called when a remote DSM manager accesses a shared object or sends an updated copy of an object or an invalidation message.*

Table 1: Various classes and their fields and methods

	ttr (msec)	***st (msec)***
Write-invalidate	66	294
Write-all	70	309

Table 2(a). Performance results for test case (a)

	ttr (msec)	***st (msec)***
Write-invalidate	140	664
Write-all	91	455

Table 2(b). Performance results for test case (b)

But the reason is that in this test case, all the nodes access the modified object again. Since for write-all the modified object is already present in the local DSM manager, the time is less. For write-invalidate the object has to be fetched for each node since the local copy is invalid.

7 Conclusions

The comparison of various types of DSM implementations reveals that page based DSM are the closest in design to the shared memory for multi-processor systems. But shared variable and object based DSM provide a more structured approach to the design and are more efficient in terms of performance as well as ensuring consistency. The proposed design enjoys the modularity and structure of the object based approach but it also enhances performance by not enforcing sharing on all the objects. Instead the user application is allowed to specify which objects it wants to share globally.
Consistency between the shared objects is an important issue in DSM design. In the proposed design the application is given the flexibility to choose the consistency scheme. As shown in the experimental results, *replication* gives better performance. Moreover, if the shared object is being modified and accessed by several processes simultaneously, then *write-all* scheme gives better performance. But if the modified object is accessed only by few processes, then *write-invalidate* scheme works better.

8 References

1. Kai Li, Paul Hudak, *Memory coherence in shared virtual memory systems*, Proceedings of the fifth annual ACM symposium on Principles of distributed computing, p.229-239, August 11-13, 1986, Calgary, Alberta, Canada.
2. Andrew S Tanenbaum, *Distributed Shared Memory,* Distributed Operating Systems, Ch 6, p289-p372.
3. Ajay Mohindra and Umakishore Ramachandran. *A survey of shared distributed memory in loosely coupled systems*. Technical Report GIT-CC-91/01, Georgia Institute of Technology, Atlanta, GA, USA, January 1991
4. Paul Lu. Aurora: Scoped Behavior for Per-Context Optimized Distributed Data Sharing, *11th International Parallel Processing Symposium (IPPS)*, Geneva, Switzerland, April 1-5, 1997, pp. 467-473.
5. Brian N. Bershad and Matthew J. Zekauskas. *Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors*. CMU-CS 91-170, Carnegie Mellon University, September 1991.
6. J. K. Bennett, J. B. Carter, W. Zwaenepoel, *Munin: distributed shared memory based on type-specific memory coherence*, Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming, p.168-176, March 14-16, 1990, Seattle, Washington, United States