A Discussion of Distributed Shared Memory

Tad Hunt

Rochester Institute of Technology

tad@nick.csh.rit.edu

ABSTRACT

The purpose of this paper is to discuss the topic of distributed shared memory. I introduce the concept of distributed shared memory, then move into a discussion on its relative strengths and weaknesses. From there, I move on to a discussion of a few various implementations for homogenous systems, showing how the strengths and weaknesses that I discussed previously shaped the design of the various implementations. Next I discuss its relative usefulness to the distributed applications programmer. Finally, I discuss heterogeneous distributed shared memory.

4 February 1995

Chapter 1
Shared Memory Definitions

1.1. Traditional Shared Memory

To understand what distributed shared memory is, it is useful to understand traditional, or *physical* shared memory. Physical shared memory is named as such because the memory shared between processes is physically the same memory, even if the processes are running on different processors. As shown in *figure 1*, the memory that is being shared is all part of the same contiguous block of physical memory. Each shared memory variable that is accessible to a process is mapped into that process's address space, where it acts like a normal variable. This technique allows an arbitrary number of processes using the same physical memory to access that shared memory.

Traditional shared memory is used for sharing data and variables between processes. An example of something that uses shared memory is the UNIX® and *X-Windows* based multi-player game, *netrek*. The server is set up to spawn off a copy of itself for each user in the game. Each copy of the server talks to the others through shared memory. Each server uses the shared memory for data such as where the other players are located in relation to you, how much damage they have, etc. Message passing would generate a lot of overhead since every server process would have to be notified of every change in the data. Not only would this be a lot of communication overhead, but the overhead to store the redundant data would be high as well. Shared memory, in this case is the optimum solution.

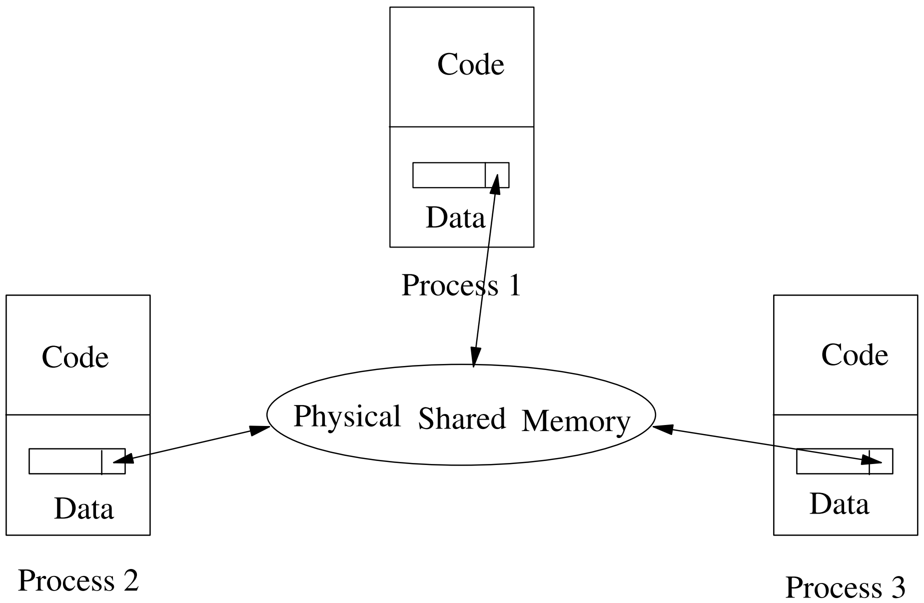

Figure 1: *Physical Shared Memory*

1.2. Distributed Shared Memory

Distributed shared memory expands upon the theme of traditional shared memory by moving it from using a single physical memory to sharing memory across any number of processes in a distributed system.

To understand distributed shared memory, first start with the concept of traditional shared memory as described above. Now imagine the processes in *figure 1* running on separate networked workstations, such as in *figure 2*. Each process still needs to reference the shared memory segment, however they can't because they no longer share any physical memory. Therefore every workstation running a process that wants access to a shared memory segment must take care of mapping the shared memory segment into its own virtual memory, and also make sure that the data in the locally held copy is the same as everyone elses copy of it. Since it would be nice to hide this mapping process from the client process, it is the job of the operating system to take care of this behind the scenes work.

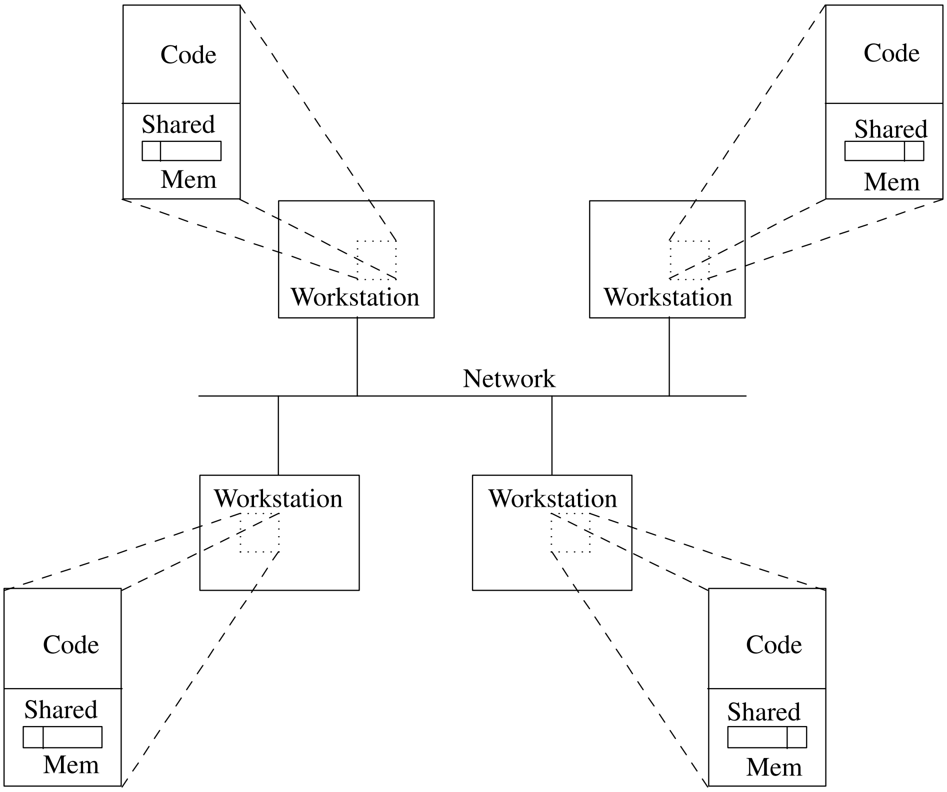

Figure 2: *Distributed Shared Memory*

Chapter 2
Applications

2. Distributed Systems

Distributed systems in general become useful when a task needs to be done that would overwhelm a single system. An example of such a task is a large database that needs real-time updating. A single system would be able to handle it, only as long as the database remained small and relatively uncomplex. As the database grows, the more beneficial offloading the task to a distributed system becomes, because then each node in the system can work on a small part of the database.

2.1. Distributed Applications Without Distributed Shared Memory

Before introducing distributed systems using distributed shared memory, it will be helpful to describe in some detail a distributed system that has been designed without the concept of distributed shared memory in mind. At *Computer Science House*, there is a group of people who have designed and are working on implementing a distributed system on top of Unix. The goal of this system is to create a virtual reality system for visualization and simulation.

2.1.1. Virtual Reality as a Distributed System

A virtual reality is very computation intensive. It has to track and notify all objects of collisions with other objects, as well as keep the client objects notified of what is happening around them. Collectively, all of the interconnected nodes in our distributed system are referred to as the **Virtual Environment Server (VES)**. Since a virtual world is very computation intensive, distributing the computations across a network of systems will make for better response time. In a virtual environment, response time is of extreme importance.

2.1.2. Brief Overview of the Computer Science House VES

The VES begins as four different daemons, known as the *Database, Network, Collision, and IO* daemons. Since the VES is based on top of Unix, we use internet domain sockets for communication. When the *base*, or local world, is initialized, the database daemon notifies the collision daemon of all the objects that it needs to keep track of and notify in case of collisions. Inside the virtual world, there are constructs called *regions*. Regions are protected areas, much like home directories under unix. An object can enter a region only if it has the proper permissions. The root region is the region that is controlled by the initial collision daemon.

Each region has its own collision daemon keeping track of the objects contained in it. When a user enters a region, control is passed from the parent region's collision daemon to the collision daemon in charge of the region. In this way, a virtual world consisting of any number of nodes can be established. (VRLSD)

2.2. Distributed System Applications Using Shared Memory

Many applications can benefit from using distributed shared memory. In Levelt's Paper, *A Comparison of Two Paradigms for Distributed Shared Memory*, he mentions several problems used to test the implementations of distributed shared memory they wrote about. I'm not going to go into any great detail of these problems because they are traditional computer science problems that most readers should already know about.

2.2.1. Traditional Applications

They implement four different problems to test the capabilities of their distributed shared memory system. They implement the *traveling salesman, alpha-beta, matrix multiplication, and all pairs shortest path* problems. With each application, the distributed shared memory met or exceeded the execution time of physical shared memory. This proves that in applications where distributed systems are used, distributed shared memory can help to improve even further the efficiency of the distributed system

2.2.2. VES Application

Distributed shared memory could be applied to the VES discussed earlier to achieve greater efficiency. When an object requests to enter a new region, it must pass the data for the object to a different collision daemon. If the daemons used used distributed shared memory, all it would have to do is pass the object id to the new collision daemon. this would increase the efficiency of the distributed system.

The shared memory would not just make the collision daemon more efficient, it could also be used between the other three daemons as well. Instead of the daemons passing all the data for every object back and forth, all they would have to do is pass an index into the shared memory back and forth, which would lower network overhead, thereby increasing the throughput of the distributed system.

2.3. Ease Of Applications Programming

If the complexity of programming a distributed shared memory application is sufficiently high, then the benefits gained will not outweigh the programming overhead. Therefore an implementation of distributed shared memory must be reasonably simple to interface to, and relatively efficient as well. An optimum solution would be to entirely hide the distributed shared memory aspect from the client, so what they see looks like regular shared memory.

If the fact that the distributed shared memory is transparent to the applications programmer, then efficiency of the programmer and of the application will increase. The best way to hide the details from the application programmer would be put the operating system in charge of doing all the dirty work, this just happens to be the topic of the next chapter!

Chapter 3
Distributed Shared Memory at the Operating System Level

3. Distributed Shared Memory Options

At the operating system level, there are a couple of different options for distributed shared memory. On the low level, there is distributed shared virtual memory (DSVM), and on the high level there is distributed shared data objects (DSDO). Each option has different strengths and weaknesses depending on the application. There is a third option as well, which encompasses portions of the second. That is a distributed object orientated operating system. By its very nature, processes and shared memory are slightly different than in conventional systems.

3.1. Distributed Shared Virtual Memory (DSVM)

Distributed shared virtual memory is based upon paging and swapping. Basically, the DSVM looks the same on all the systems accessing it. Its just a fixed size page of memory that has to be passed as a whole that is replicated on every node in the distributed system. Since DSVM is an extension of paged virtual memory, all of the problems entailed in it are propagated to the distributed level.

One of the problems with DSVM is when a program has two or more variables on the same page, or a single variable spanning multiple pages. In the latter case, that means that each time that shared variable is modified, two pages must be sent out to each node. In the first case, each time one of the variables gets modified, the same page must be resent to all of the nodes, which will cause a lot of overhead because of page faults on that one page constantly being generated. The processes that want to read the variable on the page will be waiting most of the time for the page to be unlocked so that they can read it.

High transmission overhead is also involved if the variable the program accesses is small compared to the size of the page. For example, assuming a page size of 8k (very common) and a variable of 32 bytes, a significant amount of useless will end up wasting network bandwith, thereby lowering the efficiency and slowing down the distributed system.

One of the pros to using DSVM is that it is simple to implement because it is just an extension of virtual memory. Also, it will be entirely hidden from the program, as in it will look just like conventional shared memory to them.

3.2. Distributed Shared Data Objects (DSDO)

Distributed shared data objects are much more efficient in network memory overhead because each object is only as big as it needs to be, and each data object is separate from the other data objects. Therefore, if a process needs to read a shared data object, all the operating system has to do is find out if it has been changed since last time the process read it, and if it has, get the new copy, then let the process read the data.

There really aren't any drawbacks to DSDO other than more complexity added to the operating system in order to hide from the client the extra processing. This is reasonably complex, but not utterly so. The operating system has to keep track of locations and sizes of data objects so that when they are referenced, it can do the appropriate thing.

Basically, its a more complex abstraction on top of virtual memory. (DSM)

3.3. CLOUDS -- a Distributed Object-Orientated Operating System

CLOUDS was designed from the ground up as an object orientated distributed system. As a brief overview, the concept of a *process* has been mutated into a concept of an object. An object is equivalent to an abstract data structure in a high level language, with a hook to a *thread*. A *thread* is code being executed. So every object has a data segment and a thread associated with it. The objects are location independent for data and processing, so in the course of execution, a thread may call an object on another node in the distributed system, and in turn that object may call another object somewhere else, etc. (CLOUDS)

Chapter 4
Discussion of Heterogeneous Distributed Shared Memory

4. Concept of Heterogeneous Distributed Shared Memory

The idea of heterogeneous distributed shared memory is that a distributed system composed of different types of hardware can share memory. Since the machines are heterogeneous, much care must be taken to insure that the data is translated the same on all the systems. Problems like byte ordering on various machines must be taken into account, decisions have to be made as to what *network byte order* will be, so that those machines that differ can compensate for the ordering, and the type of distributed shared memory must be compatible among all the systems in the distributed system.

4.1. Distributed Shared Virtual Memory in a Heterogeneous Environment

I discussed some of the problems facing distributed shared virtual memory earlier in this paper. By moving to a heterogeneous distributed system more problems are added. There is no set standard for page size among memory management units, therefore what might be a page on one node, may be a half or two pages on another node. In order to solve this problem, so much data about the system would have to be sent, that the overhead would seriously degrade system performance, probably to the point where the system would spend more time paging memory across the network than actually processing the information.

4.2. Distributed Shared Data Objects in a Heterogeneous Environment

I mentioned before that there weren't any serious drawbacks to distributed shared data objects other than the added complexity to the operating system to deal with them. When moved to a heterogeneous environment, that statement still stands. The degree of complexity for implementation of doesn't increase much at all. The only thing that needs to be watched for is byte order. All data that is received on at a node must be in the correct byte order for the machine, and still be the valid data. This can be accomplished very simply by running a little function on the incoming network byte order data. Also, on transmission, the machine must be sure to put the data into network byte order. (HETERO)

Chapter 5
Conclusion

5. Distributed Shared Memory

In this paper I have discussed several different types and implementations of distributed shared memory. Among those types discussed were distributed shared virtual memory, distributed shared data objects, and finally distributed objects. Each type of distributed shared memory has its strengths and weaknesses. However, distributed shared data objects come out on the top any way they are judged. DSDO generates the least amount of network overhead other than the CLOUDS implementation of distributed objects. Distributed shared memory is reasonable tool for solving problems in a distributed system because performance is at least as good, if not better than physical shared memory.

5.1. Scalability of Distributed Shared Memory

Scalability, that is improvement by adding or subtracting nodes in the distributed system is an important factor that hasn't been discussed yet. Basically, the optimum amount of nodes is that which balances network load with processing speed. As an example, if a distributed system is based on top of a 10 Mb/s ethernet, and it has 100 nodes to share that 10 Mb/s ethernet link, then each node can only have a 100 k/s portion of the bandwith or else collisions will begin to occur. That means that if the nodes can put out more than 100 k/s of data, then even though they are capable of outputting that much data that quickly, the network is a bottleneck, and most of the time the nodes will be waiting for the network to clear.

In a distributed shared memory environment this is especially important because each shared memory segment, be it a virtual page, or a data object, has to be sent to every node that is accessing that segment, so if it is a popular segment, that means many copies of it must be sent, which increases the amount of network traffic, and decreases the amount of free network time remaining.

Therefore scalability is determined by the processing power of the nodes of the distributed system, and the speed of the network connecting those nodes. The faster the network, the more nodes it can support.

Biblography

DSM:
A Comparison of Two Paradigrims for Distributed Shared Memory, Willem G Levelt, M. Frans Kaashoek, Henri E. Bal, Andrew S. Tanenbaum, Dept of Mathematics and Computer Science, Vrije Vniversiteit, Amsterdam, The Netherlands. Journal of Software--Practice and Experience, Nov 1992.

CLOUDS:
An Implementation of Distributed Shared Memory, Ramachandran U., Khalidi M.Y.A., Software Practice & Experience, May 1991

HETERO:
Heterogeneous Distributed Shared Memory, Songnian Zhou, Michael Stumm, David Wortman, Computer Systems Research Institute, University of Toronto, Kai Li, Dept of Computer Science, Princeton University, Princeton, NJ. IEEE Transactions of Parallel and Distributed Systems, Sept 1992.

VRLSD:
VR-LSD Project Description, Eric Van Hensbergen; Author, Pat Fleckenstein, Tad Hunt; Contributors, Computer Science House, Rochester Institute of Technology, November 5, 1992