The Design and Implementation of Distributed Shared Memory Based on Scope consistency

Wenlian Yang

A MAJOR REPORT

IN

THE DEPARTMENT

Of

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

November, 2002

0-612-78000-7

ABSTRACT

The Design and Implementation of Distributed Shared Memory

Based on Scope consistency

Wenlian Yang

Distributed Shared Memory (DSM) is one of the main approaches to implement distributed computing. The purpose of this project is to design and implement a prototype DSM that runs on popular TCP/IP networked Window PCs without special compliers and/or linkers. Our DSM uses lock based scope consistency (ScC), and supports central model, multiple server model and fully distributed model. We use object-oriented method to encapsulate these models within one framework. We provide C++ APIs and a DSM engine for C++ programmers to use DSM functions. The project is developed using C++ and Winsock APIs under Windows NT platform.

OUTLINE

This report is organized as follows:

In chapter 1, we briefly discuss the concepts of Distributed Shared Memory (DSM), including memory consistency models, granularity models and previous works. In chapter 2, we present overview of our DSM system. We present an example to show how to use our DSM. In chapter 3 we discuss several DSM implementation protocols, and their relative merits. In chapter 4, we discuss the implementations on both the central server model and the multiple server model. In chapter 5, we discuss how to implement DSM using fully distributed solution. In chapter 6, we discuss the programming issues in DSM development.

Acknowledgements

I wish to thank all those who make the final realization of this report possible. It is not possible to mention all their names, however I would like to express my special gratitude to the following contributors.

I am greatly indebted to my supervisor, Professor H. F. Li, who taught me distributed system course, encouraged and led me to do my major report in the area of distributed shared memory system, for his technical advice, generous attention, critical comments throughout this project.

I am pleased to thank Dr. Dhrubajyoti Goswami as examiner for my work, for his valuable comments on this report.

Finally I would like to express my gratitude to my parents, who take care of my little daughter, and to my wife Juan.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction and Problem Statement

1.1 The Concepts of DSM

Distributed Shared Memory (DSM) is a popular and powerful distributed computing paradigm. A DSM system has the following features:

- Multiple computers.
- Interconnections connecting multiple computers.
- Multiple computers perform together to do useful work.

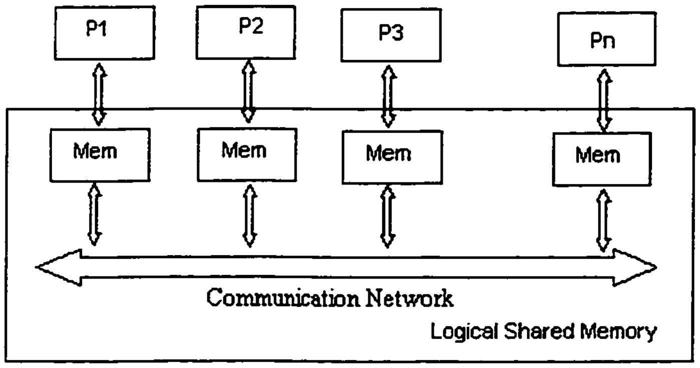


Figure 1.1 DSM Architecture

Figure 1.1 shows the general architecture of DSM, we can see that DSM is logically shared memory. For more detailed introduction of DSM, please refer to [15].

1.2 Benefits and Drawbacks

Compared to another popular distributed computing paradigm, message-passing interface (MPI), DSM has some major benefits:

- Ease of programming: similar to the sequential programming in many aspects, DSM programming is easier to learn and use. DSM provides simpler abstraction for application programmers because DSM is built on top of MPI and hides many complex details of internal data passing mechanism.

- DSM programs are usually short, more readable and understandable than the programs written using MPI APIs. The programmers do not need to know about how to use the complex MPI API functions, how to implement application communication protocol, how to detect communication deadlock and how to implement synchronization.

Compared to MPI, a major drawback of DSM is its efficiency. Usually a program written with MPI is more efficient than a program written with DSM. DSM kernel, because of its consistency requirements, greatly affects the speed of a DSM program. In order to improve performance, DSM usually uses two strategies: locality and weaker consistency models. Locality means there are multiple copies of a shared object. Each distributed process should try to use its local copy whenever possible. Locality generates the problem of how to make these copies consistent. The recent trend is to use weaker consistency models. This means that the programmer shares some responsibilities with DSM kernel to maintain consistency.

1.3 Memory consistency models

1.3.1 Why are they important?

Every DSM uses and implements at least one consistency model. A consistency model is a contract between the programmer and the DSM kernel. It says that if and only if the programmer agrees to obey certain rules, the DSM kernel promises to guarantee the expected results and execution behavior.

Consider the following example. Assume processes P1 and P2 share variables a and b, initially a=b=0. Can the following codes guarantee that only one process is killed? Is it possible for both processes to be killed?

```
P1

a=1;
if (b=0) then
kill(p2);
```

```
P2

b=1;
if (a=0) then
kill(p1);
```

The answer to the above questions depends on which memory consistency model is used. Under sequential consistency, we can guarantee that only one process is killed. But under processor consistency we cannot make the same guarantee.

Memory Consistency model is very important, because it defines the semantics of a DSM program. The same codes may have different execution behavior under different models. The programmer needs to fully understand memory consistency model in order to write a correct DSM program. Generally there are two categories of consistency models: (1) consistency models that do not require synchronization operations such as strict consistency, sequential consistency and processor consistency. (2) Consistency models that require synchronization operations, such as weak consistency and release consistency. For more formal description of consistency models, please refer to [10].

The following discussion summarizes some most commonly used consistency models. The notation O (x) V is used to make the discussion more understandable, where O represents operation R (Read) or W (Write), x represents a memory location, and V is the value read from or written to the location x. R (x) 1 means that 1 is read from location x. W (x) 1 means that 1 is written to location x. A→B means A happens before B.

1.3.2 Consistency Models that do not need synchronization operation

Strict consistency is defined as: any read to a memory location x returns the value written by the most recent write operation to x. Implications of strict consistency are:

- All the processes instantaneously see all the writes.
- All subsequent reads see the new value of the write.

Strict consistency describes the memory behavior of multiple processes run on a uni-processor computer. Strict consistency is too strong to be used in DSM, because it assumes zero transmission delay, which is not practical. No matter how fast the interconnection system is, there is some delay and there is no global clock that can be used in a distributed system.

The following is a valid example under strict consistency: process p1 writes 1 to location x, then process p2 reads 1 from location x

```
P1: W(x)1
P2:         R(x)1
```

The following is an example that is invalid under strict consistency:

```
P1: W(x)1
P2:         R(x)0 R(x)1
```

The invalidity arises, as P2 does not instantaneously see the write by P1.

Sequential Consistency (SC) was defined by Lamport [1]: "the result of any execution is the same as if the reads and writes occurred in some order, and the operations of each individual processor appear in this sequence in the order specified by its program"

The following is a valid example under Sequential Consistency.

```
P1:  W(x)1
P2:          W(x)2
P3:                  R(x)1 R(x)2
P4:                  R(x)1 R(x)2
```

The following is an invalid example under Sequential Consistency.

```
P1:  W(x)1
P2:          W(x)2
P3:                  R(x)1 R(x)2
P4:                  R(x)2 R(x)1
```

The invalidity arises as P3 sees that W(x)1→W(x)2, while P4 sees that W(x)2→W(x)1.

The sequential consistency requires that the execution of each process must follow the program order. But this is not necessarily needed in some cases. Some executions do not obey the program order but still generate SC results. This is because memory access can be divided into two types: competing access and not competing access. Program order should be only enforced where there are competing accesses.

Sequential consistency should be a model for the entire shared memory. Implementation of this model requires that all reads and writes to be broadcast across the network, allowing every node to see the same sequence of actions. For the programmer, programming using this model is the simplest. The drawback of this model is that its realization is inefficient and the performance is poor.

Processor Consistency is defined as writes performed by a process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order.

The following is a scenario that is invalid under sequential consistency but is valid under processor consistency:

```
P1:  W(x)1
P2:          W(x)2
P3:                  R(x)1 R(x)2
P4:                  R(x)2 R(x)1
```

P3 sees that $W(x)1 \rightarrow W(x)2$, while P4 sees that $W(x)2 \rightarrow W(x)1$. Because W(x)1 is performed in P1, W(x)2 is performed in P2, it still conforms to the definition of processor consistency.

The following is an example that violates processor consistency:

```
P1:  W(x)1 W(x)2
P2:                  R(x)2 R(x)1
P3:                  R(x)1 R(x)2
```

P3 sees that W(x)2→W(x)1, while P2 sees that W(x)1→W(x)2. Because in P1 W(x)1→W(x)2. So P2 should not see that W(x)2→W(x)1, thus not conforming to processor consistency.

Processor consistency better reflects the fact that network latency between different nodes can be different. Some problems may not have solutions under processor consistency. So it is seldom used in DSM.

1.3.3 Consistency Models that need synchronization operation

Weak Consistency (WC) is initially proposed by Dubois et all [2] and defined as:

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

The following shows an execution, which is valid under weak consistency:

```
P1: W(x)1 W(x)2         S
P2:        R(x)1 R(x)2  S  R(x)2
P3:        R(x)2 R(x)1  S  R(x)2
```

Where S represents synchronization operation.

Before the synchronization, P3 did not see P1's writes in the order they occurred. But after the synchronization, it sees the final value written to x.

WC allows for very fast performance. There is less communication to be performed under WC. The writes can be buffered locally until synchronization instruction is performed. For the programmers, the programming under WC is more complex than that under SC. They need to use special synchronization instructions to maintain consistency. Hence they are partly responsible for the consistency in sharing.

Release consistency model (RC) is proposed by Gharachorloo and et all [3], which further divides the synchronization operation into acquire and release. To generate SC results, RC requires that the program be properly labeled by acquire/release. They prove that properly labeled RC program can generate SC results.

Release Consistency is defined as:

- Before an ordinary access can be performed, all previous acquire operations must have been completed.
- Before a release access can be performed, all previous accesses must have been completed.
- All synchronization accesses must be FIFO (First In First Out) atomic.

The following shows an example of how to label a program. Process P1, P2 and P3 contain operation W(x). Writing to a same location are considered as data-race operations. In order to keep consistency, W(x) should be separated by synchronization operation Acq(Acquire) and Rls(Release) as follows:

P1: Acq W(x) Rls

P2: Acq W(x) Rls

P3: Acq W(x) Rls

If there is enough synchronization (no data-race), a program based on RC can guarantee SC results. RC has two representative implementations: Eager Release Consistency (ERC) [4] and Lazy Release Consistency (LRC) [5].

Figure 1.2 shows the concepts of ERC and LRC. In ERC, when there is a release it broadcasts data to all nodes. In LRC, data is buffered locally until someone acquires it.

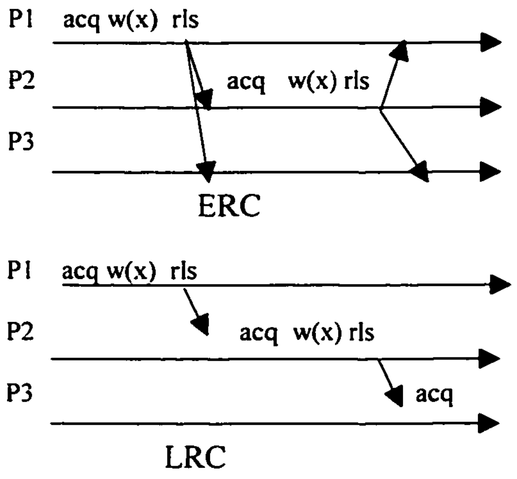


Figure 1.2 ERC and LRC

1.4 Granularity Models

Granularity is another important design issue in DSM. Granularity is closely related to the problem of false sharing. Granularity specifies the size of the minimal data-chunks to be used by the consistency protocols as a unit of sharing.

Figure 1.3 shows the concept of false sharing. Process 1 wants to write location X and process 2 wants to write location Y. If X and Y belong to a same chunk of shared memory, the two processes cannot perform write at the same time.

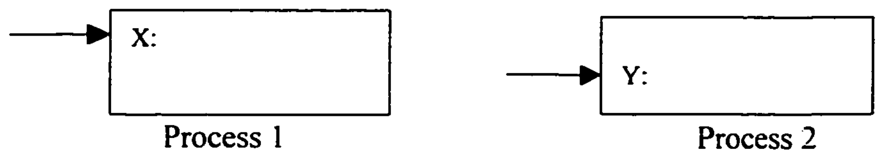


Figure 1.3 False Sharing

False sharing is another factor that may greatly affect the DSM performance. Usually the larger granularity, the more likelihood of false sharing. Performance is decreased when a process has to wait for another process to finish writing different locations in a same block. But if the granularity is too small, synchronization overhead may be considerable. Synchronization is also considered expensive in DSM. Some DSM implementations use fine-grained sharing, which means that granularity is at object level. The granularity size is determined at run time and is usually very small. But due to its implementation complexity, most DSMs still use fixed size page-based granularity.

A good compromise between object-based granularity and page-based granularity is to use scope consistency [6]. Scope consistency improves release consistency in that its granularity is user defined. In contrast, release consistency only allows each acquire/release to operate on fixed length pages of memory chunks.

Another benefit of using scope consistency is that it allows you flexible partitioning of the problem depending on the speeds of the computers. Suppose there are 3 computers: computer 1 and computer 2 have almost the same speed. Computer 3 is 50% slower than the other two. If each computer is assigned the same workload, computer 3 is obviously a bottleneck. Under scope consistency one can assign less workload to computer 3.

1.5 Previous Work

The shared memory concept was first implemented on multiprocessor machine. Due to the scalability limitation on the number of processors, software DSM was proposed to run on a network of computers.

The first ever software DSM, IVY was developed by Kai Li [7] in 1986 at Yale University. IVY uses pages of size 1k and implements sequential consistency. SC is widely considered too strong for DSM implementation.

Munin [4] was developed at Rice University in 1990. Munin uses Eager Release Consistency. It was implemented on SUN workstations connected via Ethernet. In Munin there are nine types of shared data objects. Each type of object has its own default memory coherence strategy. It also allows dynamic system decisions for coherence models.

TreadMarks [5] was developed in 1992 at Rice University. It is the only commercially available DSM. It provides user-level APIs with function such as process creation, process destruction, synchronization, and shared memory allocation on UNIX environments. TreadMarks supports various hardware such as RS-600, IBM, SP1&2, DEC Alpha & DEC Station, HP, SGI, and Sun-SPARC. TreadMarks uses Lazy Release Consistency (LRC) model, and was implemented using UDP/IP protocol.

Midway [8], which was developed at CMU in 1993, uses entry consistency. The system was implemented on a network of DEC stations running MAC 3.0. Midway uses explicit bindings of locks to shared data elements. It reduces network traffic by sending both data and lock updates in a same message, and only to the required processor.

Shrimp [11], which was developed at Princeton University in1994, supports Intel CPU PCs. It runs on NT or Linux and uses a special release consistency model called Scope consistency. Shrimp needs some proprietary hardware to run it.

Shasta [12], which was developed at DEC Inc in 1996, uses a combination of fine-grained and coarse-grained sharing in its implementations. Shasta needs a cluster of workstations or servers to run it.

MultiJav [9], which was developed at Utah State University in 1998, implemented a DSM using Java. MultiJav uses object-based granularity. Their objective is to provide DSM function on multiple Java Virtual Machine (JVM).

We can see that DSM research is still largely associated with academic institutions. It has not moved out into the realm of real business. Most DSMs are developed on unpopular hardware platforms using some special development tools. We believe developing inexpensive, easier to use and high performance DSM on popular PC environment should increase DSM popularity in the future.

1.6 Problem Statement and Accomplishment

In this project we focus on the following main problems:

- Develop a prototype DSM, which runs on popular hardware platforms. We can see that most previous efforts were done on legacy platforms. The platform to use our DSM is based on Windows PCs networking with TCP/IP.
- Provide DSM API functions for C++ programmer at source code level. Many DSMs require special compilers and/or linkers to use their API functions. This is one of the main reasons that limit the popularity of DSM. To use our DSM functions, the programmer only needs to include relevant head files. No special compilers and linkers are needed. The project work will provide a basis for future work to develop middle-ware components to support other programming languages.
- The project focuses on how to map logically shared memory to local physically memory based on scope consistency under different running environments. Our system can run with the support of sever(s), or runs on fully distributed model without any server. We include all DSM functionality in two C++ classes *DsmServer* and *Dsm_Proxy*. *DsmServer* is needed when DSM is running with servers. To facilitate the use on server model, we write an engine program, which uses the *DsmServer* class. *Dsm_Proxy* is needed in all running environments. It contains all the interfaces to complete DSM functionality.
- We focus on the communication and synchronization mechanisms between processes and servers on different running environments. We design and develop the DSM application protocols based on TCP/IP.

Chapter 2 Overview Of Our DSM

2.1 Consistency model used in our DSM

2.1.1 Lock Based Scope Consistency

In this project, we implement a DSM using lock based scope consistency model. Scope consistency model is proposed based on release consistency model. So our discussion begins with release consistency model.

Release Consistency Model

Release Consistency is first proposed by Gharachorloo et all [3]. They propose to categorize and label shared memory accesses as shown in figure 2.1.

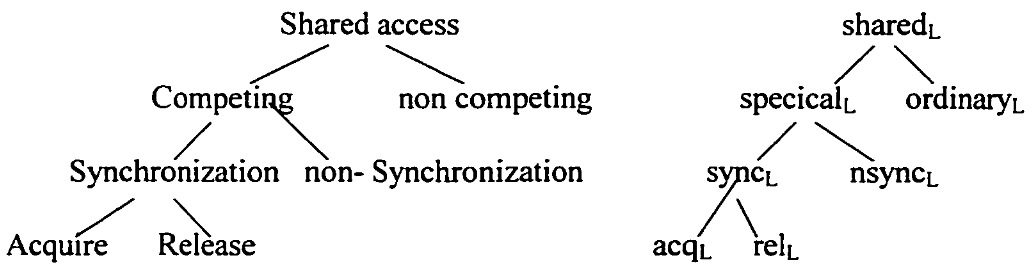


Figure 2.1 Shared access categorization

The following is quoted from [3]:

"Release consistency is an extension of weak consistency that exploits the information about acquire, release, and non-synchronization accesses. The following gives the conditions for ensuring release consistency.

(A) Before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous acquire accesses must be performed, and

(B) Before a release access is allowed to perform with respect to any other processor, all previous ordinary load and store accesses must be performed, and

(C) Special accesses are processor consistent with respect to one another."

Compared to sequential consistency (SC), release consistency (RC) allows more overlap as show in figure 2.2. Under SC, every access is serialized. Under RC, read/write accesses between acquire (ACQ) and release (REL) are allowed to overlap, and are not required to obey the program order. In addition, there is overlap between acquire (ACQ) and release (REL) of different processes. Obviously RC shows more performance potentials than SC. RC has two implementations: eager release consistency (ERC) [4] and lazy release consistency (LRC) [5]. LRC further decreases the number of messages exchanged. Hence the principle of lazy update is also used in our implementations.

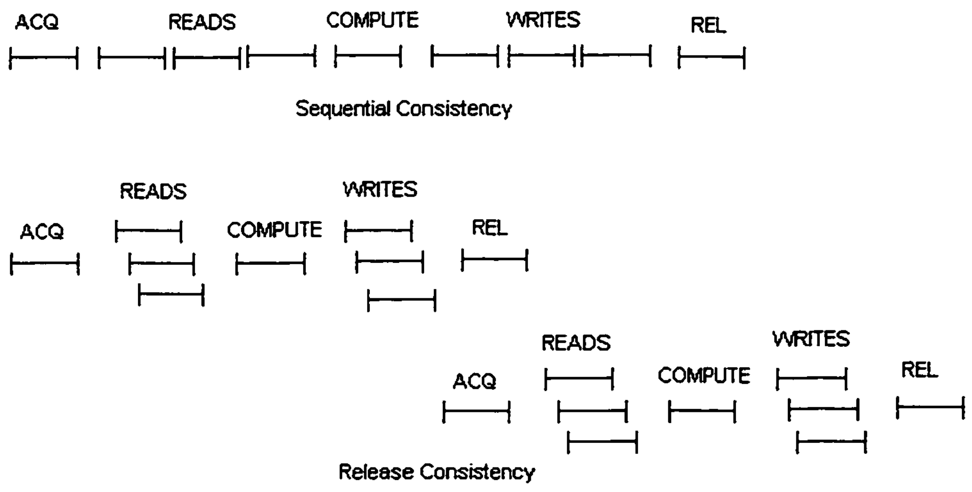


Figure 2.2 Overlap comparisons of SC and RC

Scope Consistency model

Scope consistency model is proposed by Iftode et all [6]. Consistency scope is defined as: "A consistency scope is a scope with respect to which memory references are performed. That is, modifications to data performed within a scope are only guaranteed to be visible within that scope."

The more formal definition of scope consistency model needs two more definitions: *A reference being performed with respect to a consistency scope* and *a reference being performed with respect to a processor.*

The following definitions are from [6]:

"A write that occurs in a consistency scope is *performed with respect to that scope* when the current session of that scope closes.

A write *is performed with respect to a processor P* if a subsequent read issued by P returns the value stored by that write."

The scope consistency rules are:

"1. Before a new session of a consistency scope is allowed to open at processor P, any write previously performed with respect to that consistency scope must be performed with respect to P.

2. A memory access issued by processor P is allowed to perform only after all consistency scope sessions previously opened by P have been successfully opened. "

To make the above abstract definitions more clear, let us review some examples to see what is scope consistency.

Example: Lock-based Consistency Scopes.

Figure 2.3 illustrates scope consistency with an example. When Process 1 acquires lock 1 it also enters the scope defined by that lock. Process 1 writes A while in scope 1 and writes B while in scopes 1 and 2. Then process 2 acquires lock 2, thus opening scope 2, and reads both A and B. Under scope consistency, process 2 is guaranteed to see Process 1 's write to B assuming that process 2 acquires lock 2 after process 1 releases it. However there is no guarantee that process 2 will see process 1's write to A because the scope 1 was not opened at process 2 when it reads A. Thus, process 1 does not need to propagate the modification of A to process 2.

```
Process 1                   Process 2

Acquire(1)              .......................... Open scope 1
    A=1
    Acquire(2)          .......................... Open  scope 2
        B=1
    Release(2)          .......................... Close scope 2
Release(1)              .......................... Close scope 1
                            Acquire(2)  ......Open scope 2
                              X=A       ...... A is not guaranteed to be 1
                              Y=B       ......B is 1
                            Release(2)  ......Close scope 2
```

Figure 2.3 Lock based scopes

The above also shows an example of *implicit scope;* to implement it requires the compiler to know which variables are in which scopes. Because this project is supposed to provide directive support, implicit scope is not assumed. Instead, explicit scope is used. To specify explicit scope, the programmer needs to use binding directive to associate data segmentation with synchronization variables. Thus we assume scope to be specified by the programmer as special directives.

The above example also raises the question "Is it allowable for one variable to belong to several consistency scopes?" Suppose in process 2 Acquire(2) is changed to Acquire(1) and Release(2) to Release(1), what values are read for A and B. Under implicit scope, we should get 1 for both A and B, because one variable is allowed to belong to several consistency scopes. This is not supported in our DSM, because our work is not based on compiler level.

There are some slight differences between RC and ScC when implicit scope is used. Figure 2.4 shows an example. Suppose that A and B is in the same data page. Under ScC process 2 can only see that B is 1, because A's write is not within the scope, but under LRC process 2 can see both A and B is 1 because they are located in the same page. LRC uses fix length data page and propagates the entire data page when other process acquires.

Under explicit scope and single assignment of variables to a scope, scope consistency reduces to become release consistency. Hence our implementation actually conveys to LRC in this round.

```
Process 1            Process 2
 A=1
 Acquire(1)
     B=1
Release(1)
                     Acquire(1)
                        X=A        .......under LRC A and B is 1
                        Y=B        .......under  ScC only B is guaranteed to be 1
                     Release(1)
```

Figure 2.4 Scope consistency versus LRC

2.1.2 How to use locks under scope consistency?

In scope consistency, shared memory is divided into several segments. Each segment has a lock to protect its consistency. A lock is allowed to protect the consistency of several segments. At any given time only one process is allowed to access a given segment, and be granted the lock. Other processes are blocked until the lock is released. Different processes can access different segments at the same time if they belong to different locks.

In figure 2.5, process 1 tries to access segment S1. Simultaneously process 2 tries to access segment S3, and process 3 tries to access segment S2. Because Lock1 protects S1 and S2, so when process 1 and process 2 send acquire requests at the same time, only one of them is granted the lock. Lock2 protects S2 and S4. When process 3 tries to acquire it, there is no competition. Thus it is granted Lock 2 and can access segment 2.

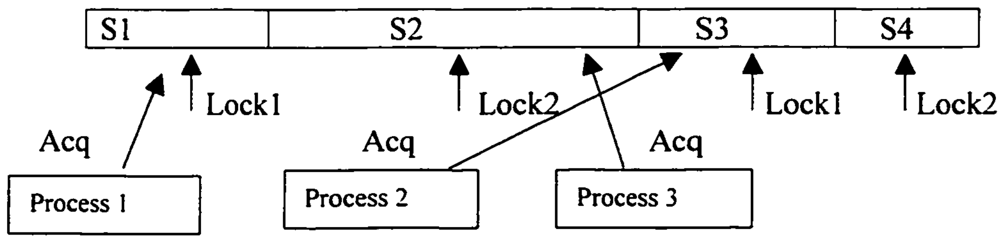


Figure 2.5 Segments and Locks.

To achieve better performance, the DSM programmer needs to review his/her program carefully in order to reduce the possibilities that different processes simultaneously access the same lock. Consider the following code segments to be run on two machines:

Program1:

```
for (I=1; I<=6; I++) {
    Acquire(I)
    //do some computations
```

```
        Release(I)  }
Program2:

    if (pid==1) {
        for (I=1; I<=5; I+=2) {
            Acquire(I)
            //do some computations
            Release(I)  }
        for (I=2; I<=6; I+=2) {
            Acquire(I)
            //do some computations
            Release(I)  }
    }
    else {
        for (I=2; I<=6; I+=2) {
            Acquire(I)
            //do some computations
            Release(I)  }
        for (I=1; I<=5; I+=2) {
            Acquire(I)
            //do some computations
            Release(I)  }
    }
```

According to program 2, process 1 access locks in the order: 1 3 5 2 4 6, process 2 access locks in the order 2,4,6,1,3,5. In program 1, both processes access the locks in the same order 1,2,3,4,5,6. So the possibility to access same lock for program 1 is higher than that for program 2. Of the two programs, program 2 shows a better design, because it is less likely to access a same lock.

2.2 Running Environments of our DSM

Three running environments are supported (figure 2.6) in our DSM :(a) central server model, (b) multiple server model and (c) fully distributed model. In model (a) or (b) there is at least one server to serve the requests from distributed processes. There is no direct communication channel between any two processes. In model (c), there is no central coordinator between processes, each process plays the same role as others. Each process acts as both server and client.

In our DSM system, we identify two types of servers. One type is called master server, and the other type is called peer server. In model (a), there is only one master server. In model (b) there is only one master server and one or more peer servers.

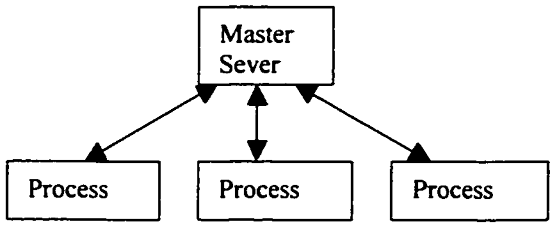
The roles of master server:

- Register all distributed processes. In model (b), master server also registers peer server(s).
- Support declarative directives through the interfaces of *DSM_Proxy* class, as standard C++ does not support any DSM syntax to declare a shared memory.
- Process acquire/release requests to keep the shared memory consistency.

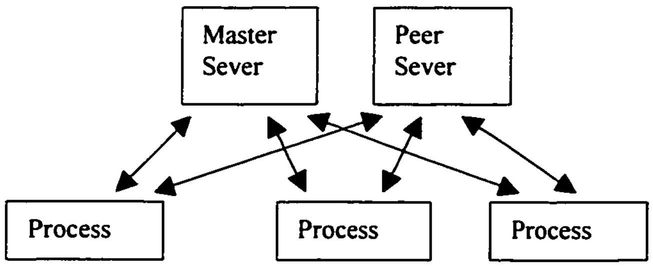
The roles of Peer server:

- Process and respond to acquire/release requests to keep the shared memory consistency.

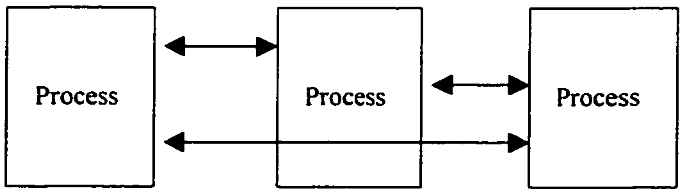
In model (a), only master server is keeping consistency of shared memory. In model (b), both master server and peer servers are keeping the memory consistency. So each server receives fewer acquire/release requests and thus enhances concurrency.



(a) Central Server Model



(b) Multiple Server Model



(c) Fully distributed model

Figure 2.6 Different Running Environments for DSM

2.3 How to use our DSM functions

In our system, DSM functions are provided through the use of *Dsm_Proxy* class. *Dsm_Proxy* class is designed to support distributed computing on different running environments. *DsmServer* class is only used by the engine we provide. Programmers do not need to know its details. Right now the DSM functions are only available to C++

programmers. To use the DSM functions, the programmer needs to include the head file *Dsm_proxy1.h*, which has the definitions of the *Dsm_Proxy* class. At the top of his/her C++ program, the following line should appear:

```
#include "Dsm_Proxy1.h"
```

The first step to use *Dsm_Proxy* class is to specify its proxy mode. Proxy mode determines which running environment is used. The following describes show how to specify proxy mode.

- ❑ For model (a) or (b) construct a *Dsm_Proxy* object by using parameter *PROXYSERVER*. The following shows an example:

  ```
  Dsm_Proxy dsm_proxy(PROXYSERVER);
  ```

- ❑ *PROXYSERVER* is supported by default. Hence the following statement has the same effect:

  ```
  Dsm_Proxy dsm_proxy;
  ```

- ❑ For model (c), use *NOPROXY* parameter to construct a *Dsm_Proxy* object:

  ```
  Dsm_Proxy dsm_proxy(NOPROXY);
  ```

Initialization of *Dsm_Proxy* object is carried out as follows:

- ❑ *Dsm_Proxy* needs Winsock API support. The initialization for Winsock is done by the *Dsm_Proxy's Initialize* function. If the initialization is successful, it returns 1 otherwise it returns 0.The following code illustrates its use:

  ```
  if (dsm_proxy.Initialize())   {//call other Dsm_proxy functions}
  ```

Identification of the processes:

- Distributed processes work together cooperatively. They need to be identified. In model (a) or (b), the first process connected to the master server is assigned process id 1. The process id is assigned according to the order in which the process connects to the master server.
- In model (c), the program needs to generate its own process ids. This can be accomplished by identifying a primary process with process id of 1. The process id of other processes is assigned according to the order it connects to the primary process.

To simplify the design DSM kernel, it is required that only process with process id 1 can create and initialize the shared memory. The same applies to the use of create lock and bind lock directives.

2.4 Shared Memory Representation in our DSM

How shared memory is represented internally?

- In our system, shared memory is represented internally by using memory id; a memory id is a string identifier that uniquely identifies logically shared memory. Internally each shared memory is represented by the following structure:

Memory Id	Number Of Elements	Type Size	Pointer to the buffer

- The above structure is first created and stored in the master server in model (a) and (b). In model (b), the peer server gets replicated copy of the structure from the master server afterward. In model (c), the above structure is first created and stored in the primary process. Then it is replicated to other processes.

In our system, two shared memory types are supported. One is called *ReadOnly* shared memory, and the other is called *ReadWrite* shared memory. For *ReadOnly memory*, at its creation, it should be initialized. *ReadOnly* memory is unchanged after initialization. The other processes need to provide local address and memory id to get a copy of its values. For *ReadWrite* shared memory, after initialization its values may be changed by processes simultaneously. All processes need to use acquire/release synchronization directives to ensure consistency.

How to make connections between local memory and shared memory?

For *ReadOnly* memory, use Dsm_Proxy's *CreateReadOnly* function in process 1, and use *GetReadOnly* function in all other processes.

The following shows an example of how to use *CreateReadOnly* and *GetReadOnly*:

```
        if (dsm_proxy.Initialize()) {  // Check is Winsock is installed
            printf("input server name \n"); scanf("%s",srv);
            dsm_proxy.Connect(srv,5000);  //Connect to the Server on TCP port 5000
            pid=dsm_proxy.GetPId() ;  // Get process id from the server
           if (pid==1)  {
              // If it's the primary process call CreateReadOnly to create
              // read-only memory on the server
              dsm_proxy.CreateReadOnly("lock",(char *)lock[0],6,sizeof(int));
              //........... }
          else {
              // Other processes use GetReadOnly to get values of
              // read-only shared memory.
              dsm_proxy.GetReadOnly("lock", &lock[0]);
              //...........                 }
      }
```

For *ReadWrite* memory, use *CreateReadWrite* function call in process 1, and use *RegisterReadWrite* function in other processes. The following is an example:

```
if (dsm_proxy.Initialize()) {  // Check is Winsock is installed
      printf("input server name \n"); scanf("%s",srv);
      dsm_proxy.Connect(srv,5000); //Connect to the Server on TCP port 5000
      pid=dsm_proxy.GetPId() ); // Get process id from the server
      if (pid==1)  {
         int sort[200];
         //initialize sort array
         //......
        // If it's the primary process call CreateReadWrite to create
         // read-write memory on the server
         dsm_proxy.CreateReadWrite("sort",(char *)lock[0],6,sizeof(int));
         //......
      }
      else {
         int sort[200];
         //Use RegisterReadWrite will fill any values in sort array.
         dsm_proxy.RegisterReadWrite("sort",(char *)sort[0]);
      }
}
```

2.5 Initial Synchronization in Our DSM.

Initial synchronization is needed to make sure that all processes have consistent initial states and all processes begin simultaneously. The initial synchronization is slightly different depending on the running environments.

Under model (a) or (b) the first process is in charge of creating and binding locks, the programmer should use *StartAllProcess* function to notify master server the end of declarative directives. Upon this, the master server may proceed to send the necessary information to all processes. These include the number of locks, the number of processes, details of each lock and details of shared memory. Under model (b), peer servers are transparent to programmers. A process only needs to connect to the master server.

Under model (c), each process plays the same role as other processes except for the primary process. In order to register all the processes, the programmer should use *SetPrimaryProcess* function to set a primary process. Then the primary process builds a process list to be distributed to all processes for channel creation among all pairs of the processes. Figure 2.7 and 2.8 show the details of the initialization procedures.

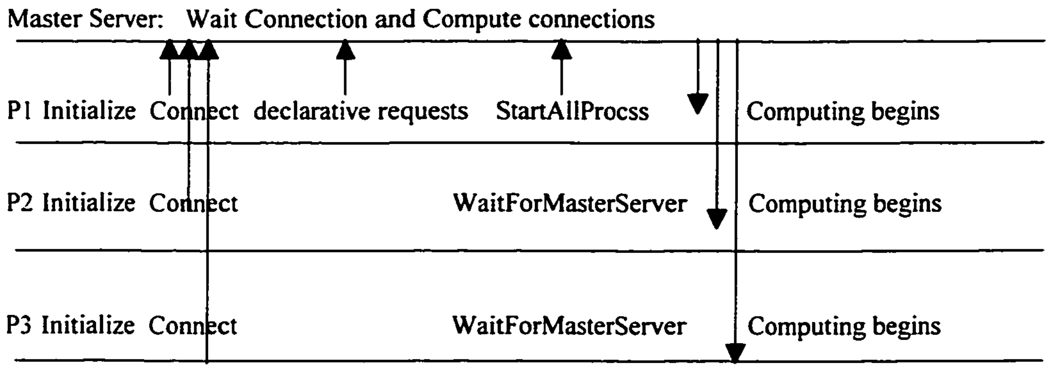


Figure 2.7 Initial Synchronization under Server models.

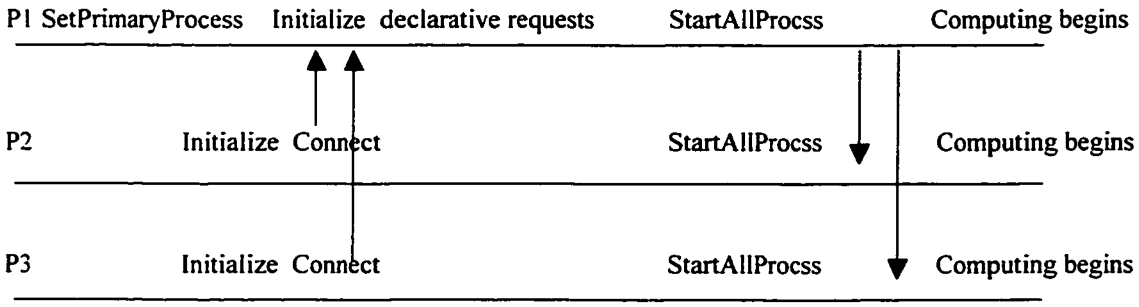


Figure 2.8 Initial Synchronization under Fully Distributed Model.

2.6 Performance Issues

Some DSMs claim that they implement object-based (also called fine-grained) granularity model. This raises the question "Is smaller granularity definitely better than bigger granularity?" Our opinion is that decreasing granularity to a certain point will not increase DSM performance any further. The reason is that most communication protocols use internal buffer as the minimal communication unit. This means that if the object size is smaller than the internal buffer size, transferring an integer may cost the same as transferring a bigger complex object. When granularity is smaller than internal communication buffer size, there is no further performance gain.

The next question is "For an application using DSM programming, how big should the granularity be to get speed up?" We think there is no general answer to this question, it depends on many factors and it varies from application to application. But the following are some key factors we need to consider when we choose the granularity:

- The nature of application problem: For example, a smaller partition is more suitable for a matrix addition problem than a sorting problem. Because for matrix addition, only one pass processing is needed while for sorting problem multiple passes are needed.

- The speed of the communication protocol: DSM programming requires the coordination and cooperation of many processes. Each process only does a part of the work. The efficiency of communication protocol affects sharing of results.

To illustrate these factors, consider a one-pass problem with complexity O(n) that is partitioned equally between two similar computers. We assume the processing time for one partition is T_P. Suppose the time to use acquire/release to get a partition is T_C. If the problem is running on one computer, the time to solve the problem is $2^* T_P$. If problem is running on two computers, the time is $T_P + T_C$. To get the speed up requires that $T_P > T_C$. Suppose computer 1 is faster than computer 2. The time to process one partition for computer 1 is T_{P1}. The time to process one partition for computer 2 is T_{P2}. So computer1 has to wait Tp2-Tp1 before results are transferred from computer 2. The total time for computer 1 is $T_{P1} + T_{P2} - T_{P1} + T_C$. So if $T_{P2} - T_{P1} > T_{P1}$, then there will be no speed up no matter how fast the communication protocol is.

The following guideline may be useful in deciding on granularity:

- The granularity should be bigger than internal communication buffer.
- The granularity should be big enough to ensure $T_P > T_C$.

Test acquire protocol on two nodes:

This test is conducted on two computers: computer 1 is a Pentium-III 450, 256M RAM computer running Windows NT 4.0, computer 2 is Pentium-II MMX 233 192M RAM computer running Windows NT 4.0. We use the central server model. Computer 1 runs the server engine and one of the processes. The process runs on computer 1 does not compete for the lock used by computer 2. Computer 2 runs a process that repeatedly acquires and releases the lock for 100 times. The purpose of this test is get information on the time it takes to acquire a remote copy when there is no competing process.

The following shows the codes that runs in computer 2:

```
Avg=0.0;
for (i=1;i<=100; i++) {
    tic1=GetTickCount();
    dsm_proxy.Acquire(1);
    tic2=GetTickCount();
    dsm_proxy.Release(1);
    Avg+=tic2-tic1;
}
Avg/=100.0;
```

To measure the time elapsed, we use Window NT API function *GetTickCount*, which retrieves the number of milliseconds that have elapsed since Windows was started. In the above codes, *Avg* stores the average time to perform an acquire.

Table 2.1 shows the test results: From the table we can see that when the granularity is 2k or 4k, acquire time is almost the same. But when the granularity is increased to 8k acquire time increases sharply.

Granularity	Acquire Time Range For 5 tests	Average Acquire Time For 5 tests
2k	14.12~19.13 ms	17.48 ms
4k	14.92~17.92 ms	16.84 ms
8K	65.30~94.13 ms	83.42ms

Table 2.1 Results for acquire test on two nodes

2.7 An example to use our DSM

The following is a sample program to illustrate parallel merge sort completed by two processes under central server model. The array contains 200 integers and is initialized with descending order integers from 200 to 1. Each process is supposed to sort the shared memory array in ascending order. The array is partitioned into 6 segments. Each segment is protected by a lock. The segment sizes are showed as follows:

20	40	30	30	50	30

In the program, Process 1 first sorts segment 1, 3, 5. Process 2 first sorts segment 2, 4, 6. Then both processes uses acquire/release to obtain the remaining segments.

At last both processes sort the whole array.

```
#include "Dsm_Proxy.h"
int compare( const void *arg1, const void *arg2 ){
    int *a; int *b;  a=(int *)arg1; b=(int *)arg2;
    return *a-*b;  };
int main(int argc, char* argv[])
{      Dsm_Proxy dsm_proxy(PROXYSERVER); //Run DSM on server modes
       int pid;         /*process id */          char srv[30];        /*server name */
       if (dsm_proxy.Initialize()) {    //check if initialize is success
           scanf("%s",srv); // Input master server name
           dsm_proxy.Connect(srv,5000); //connect to master server
           pid=dsm_proxy.GetPId()
           if (pid==1)  {  //if it's first process
                int lock[6;  int ary[200];
                for (int i=0; i<200; i++)  ary[i]=200-i; //inited  to descending order
                //store created locks in lock array
                lock[0]=dsm_proxy.CreateLock();
                lock[1]=dsm_proxy.CreateLock();
                lock[2]=dsm_proxy.CreateLock();
                lock[3]=dsm_proxy.CreateLock();
                lock[4]=dsm_proxy.CreateLock();
```

```
lock[5]=dsm_proxy.CreateLock();
//create lock array as read-only shared memory
dsm_proxy.CreateReadOnly("lock",(char *)lock[0],6,sizeof(int));
//create ary as read-write memory
dsm_proxy.CreateReadWrite("ary",(char *)ary,200,sizeof(int));
//define segments
dsm_proxy.BindLock(1,"ary",0,20);
dsm_proxy.BindLock(2,"ary",20,40);
dsm_proxy.BindLock(3,"ary",60,30);
dsm_proxy.BindLock(4,"ary",90,30);
dsm_proxy.BindLock(5,"ary",120,50);
dsm_proxy.BindLock(6,"ary",170,30);
dsm_proxy.StartAllProcess(); //Notify master server to move on
dsm_proxy.StartMonitor() ;  //Start monitor master server response
dsm_proxy.Acquire(lock[0]);
qsort(ary,20,sizeof(int),compare);
dsm_proxy.Release(lock[0]);
dsm_proxy.Acquire(lock[2]);
qsort(&ary[60],30,sizeof(int),compare);
dsm_proxy.Release(lock[2]);
dsm_proxy.Acquire(lock[4]);
qsort(&ary[120],50,sizeof(int),compare);
dsm_proxy.Release(lock[4]);
//bring results from process 2
dsm_proxy.Acquire(lock[1]);
dsm_proxy.Release(lock[1]);
dsm_proxy.Acquire(lock[3]);
dsm_proxy.Release(lock[3]);
dsm_proxy.Acquire(lock[5]);
dsm_proxy.Release(lock[5]);
qsort(ary,200,sizeof(int),compare);
```

```
        }
        else {    int lock[6];   int ary[200];
                  dsm_proxy.WaitForMasterProcess();  //Wait for all processes
                  //Get read-only shared memory
                  dsm_proxy.GetReadOnly("lock",lock[0]);
                   //Register "ary" read-wirte shared memory
                  dsm_proxy.RegisterReadWrite("ary",(char *)ary);
                  dsm_proxy.StartMonitor();
                  dsm_proxy.Acquire(lock[1]);
                  qsort(&ary[20],40,sizeof(int),compare);
                  dsm_proxy.Release(lock[1]);
                  dsm_proxy.Acquire(lock[3]);
                  qsort(&ary[90],30,sizeof(int),compare);
                  dsm_proxy.Release(lock[3]);
                  dsm_proxy.Acquire(lock[5]);
                  qsort(&ary[170],30,sizeof(int),compare);
                  dsm_proxy.Release(lock[5]);
                  dsm_proxy.Acquire(lock[0]);
                  dsm_proxy.Release(lock[0]);
                  dsm_proxy.Acquire(lock[2]);
                  dsm_proxy.Release(lock[2]);
                  dsm_proxy.Acquire(lock[4]);
                  dsm_proxy.Release(lock[4]);
                  qsort(ary,200,sizeof(int),compare);
         }
    }
    return 0;
}
```

Chapter 3 DSM Implementation Protocols

3.1 Comparison of ERC and LRC

DSM computers are connected by relatively low speed communication network. In order to improve DSM performance, it is important to avoid broadcast of large amount of data whenever possible. Figure 3.1 shows the broadcast under ERC. Assume p1, p2 and p3 in turn acquire the same lock. Under ERC when there is a release operation, it broadcasts the new page to all other processes. Under release consistency, only one process is allowed to acquire the same lock at any given time, most broadcast pages are actually wasted. As shown in the figure 3.1, at time t1 process P1 broadcast the new page to P2 and P3, but only P2 uses the page, the page sent to P3 is wasted. Similarly at time t2 the page sent to P1 is wasted. Under release consistency, the page size is usually big (1-2k), and the broadcast is expensive.

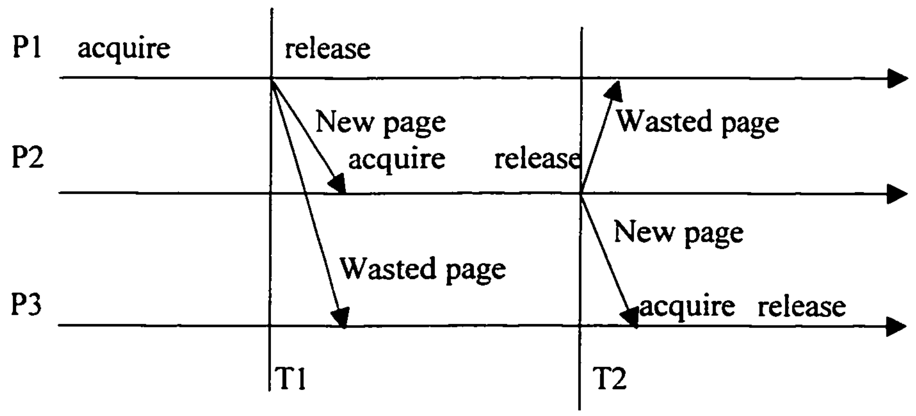


Figure 3.1 Network traffic under ERC

Can we avoid the releasing broadcast and only send the new page to the process that needs (acquires) it? The answer to this question led to LRC. Under LRC, release is buffered locally and the process delays sending new page until another process acquires it. This split-phased transaction avoids unnecessary transmission of data, As shown in figure 3.2, suppose that at the beginning all processes buffer a same set of shared values. Assume p1, p2 and p3 in turn acquire the same lock. At time T1 process P2 executes an acquire. It broadcasts its version to all other process. Because the message size of this broadcast is much smaller than a data page, it is not expensive compared to broadcast a new page. All other processes compare the version they received with their local version. Process P1 has a newer version, so it responds to P2 by sending a new page to P2. This is repeated similarly at time T2. We can see that broadcast exists in both ERC and LRC, but under LRC the actual data broadcast is avoided.

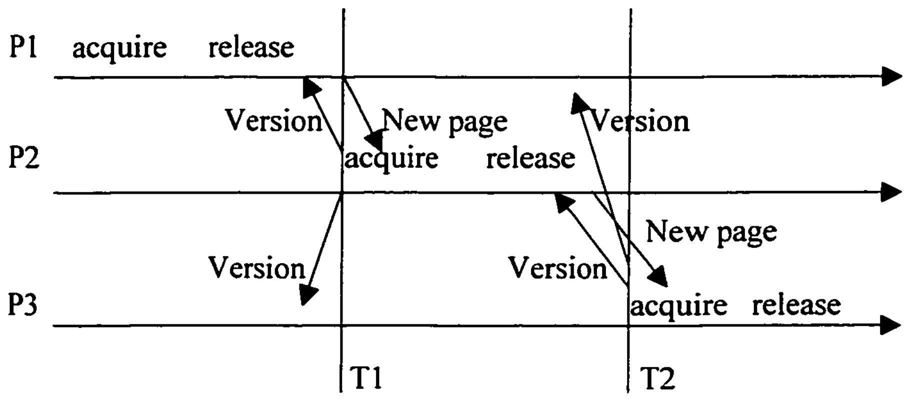


Figure 3.2 Network traffic under LRC

3.2 DSM Implementation models

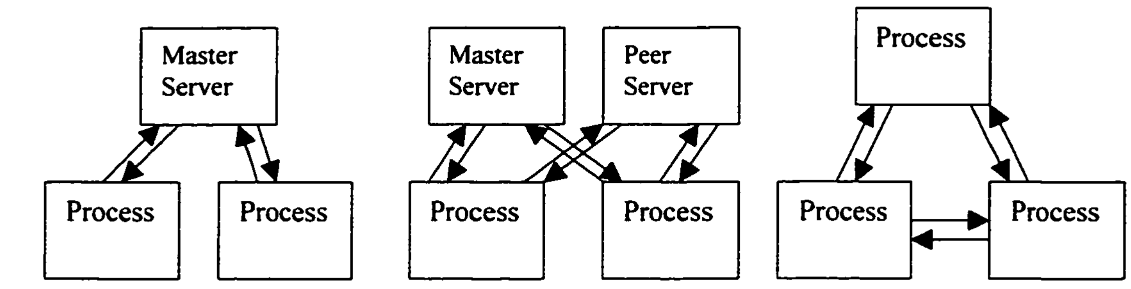
DSM can be implemented in different ways. They can roughly be divided into three categories: central server model, multiple server model and fully distributed model.

Central server model: Each distributed process is considered as a client. The central server is responsible for registering all the processes, processing all declarative directives and processing acquire/release requests. There is no direct communication channel between any two processes. This model is considered the simplest way to implement DSM.

Multiple server model: Each distributed process is also considered as a client. However, there are two types of servers: master server and peer servers. The master server has the same role as in central server model. The peer server is only responsible for processing acquire/release requests. The performance of each server improves because each server potentially processes fewer requests. Similarly, there is no direct communication channel between any two processes. However each process is connected to all the servers.

Fully distributed model: There is no central server. Instead, the first process (primary process) is to take charge of initialization activities, include registering processes and processing declarative directives. After initialization, each process plays the same role. Each process is in charge of processing acquire/release requests and synchronizing the requests. A local copy of the shared segment is maintained in each node. Direct interprocess communication is needed to support the synchronized update activities. Figure 3.3 illustrates these models.

In this project, we first developed a DSM running under the central server model before moving on to the multiple server model and the fully distributed solution. That result in a design that includes three models in the same framework.



(a) Central Server model (b) Multiple Server model (c) Fully distributed model

Figure 3.3 DSM implementation models

3.3 Detailed Comparison of Implementation models

Performance is a key concern in DSM. Different implementations have different advantages and disadvantages. We compare them by considering the following aspects:

- The speed of server
- The speed of process
- Replication of internal structures
- The need to broadcast.
- Resources used.

3.3.1 Central server model

In this model, the master server keeps a thread per process to monitor client requests. On the client side, each process is assigned a thread to handle the responses from the master server. Each process sends requests to the server and waits for its response. Figure 3.4 shows how processes and master server interact.

From figure 3.4, we can see that the master server maintains 6 threads for six distributed processes. Because all the acquire/release requests are sent to the master server, the master server can became a bottleneck. On the other hand, the overhead on the client side is the least as compared to the other models. Hence client process moves fastest compared with its counterpart in the other models. This model also uses the least resources compared to other models.

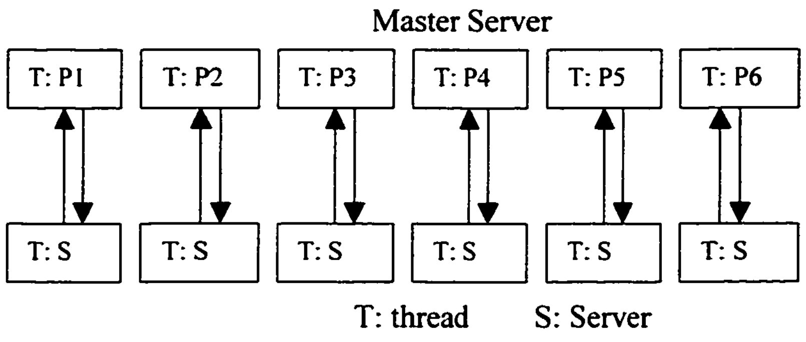


Figure 3.4 Communication under central server model

In this model, the central server always keeps the most recent version of data. When a process executes an acquire operation, it sends a request to the server. The server responds by sending the most recent data segments. When a process releases a lock, it sends the updated to the server. There is no broadcast on both acquire and release operations. Figure 3.5 shows the acquire/release operation under central sever model.

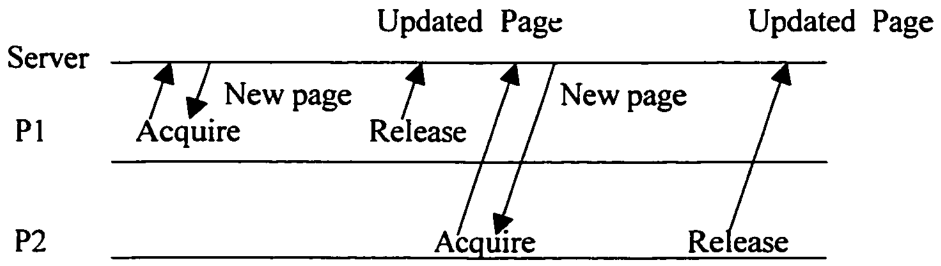


Figure 3.5 Acquire/Release under central server model

3.3.2 Multiple server model

In this model, there are multiple servers in charge of maintaining memory consistency. A lock protects chunks of shared read-write memory not to be accessed by different processes at the same time. For one specific lock, only one server is in charge of maintaining the consistency of its acquire/release operation.

In figure 3.6, all the acquire/release requests to lock 1 are routed to server 1. Similarly, all requests to lock 2 are routed to server 2. In our implementation, each process knows which server to send requests. Thus there is no need to broadcast requests to all the servers. In our system, locks are evenly distributed among multiple servers. Thus each server increases its performance as it receives fewer requests.

In this model, each server has a thread for each process, master server has additional thread(s) for peer server(s). Each process has a thread per server to monitor its response. The client speed is slightly slower as it has to interact with multiple servers.

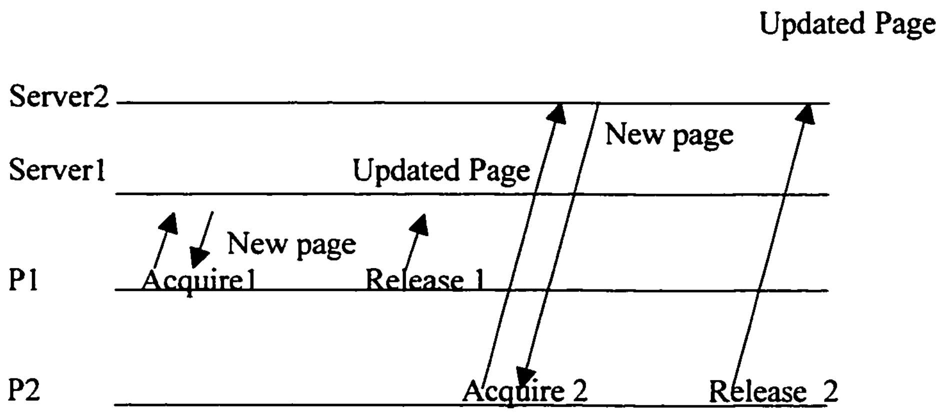
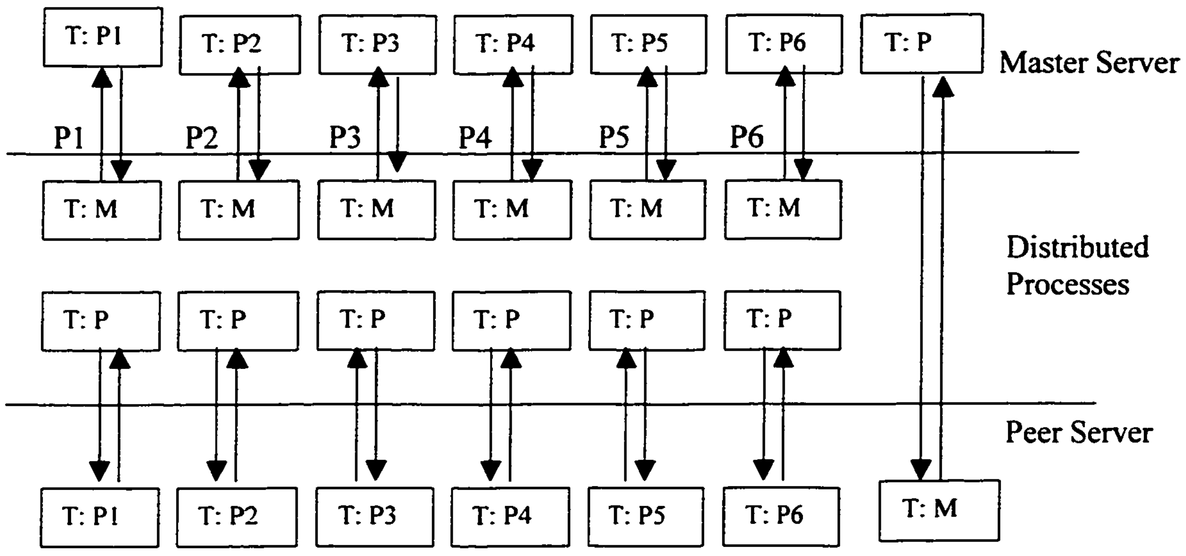


Figure 3.6 Acquire/Release under multiple server model

Figure 3.7 shows communication details under multiple server model.



T: thread M: master server P: Peer Server

Figure 3.7 Communication under Multiple Server Model

3.3.3 Fully distributed model

In this model, there is no central place to store the most recent version of data, some broadcast for synchronization is inevitable, either releasing broadcast or acquiring broadcast must be used. Figure 3.8 shows the two different methods:

- In release operation, a process broadcasts process id and version to all processes. Thus each process becomes aware where the most recent copy can be found.
- In acquire operation, a process broadcasts version to all processes, only the process that has most recent version will respond. Each release generates a new version.

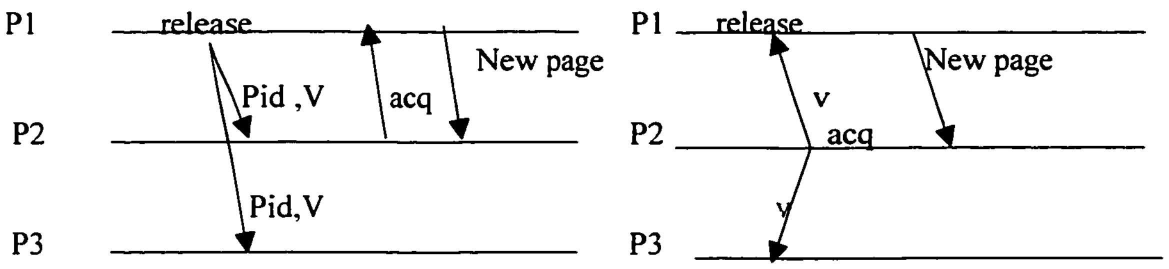


Figure 3.8 Broadcast under Fully Distributed Model

In this model, there is a direct communication channel between any two processes. Inside each process, there is a thread to handle interaction with remaining other processes, as illustrated figure 3.9.

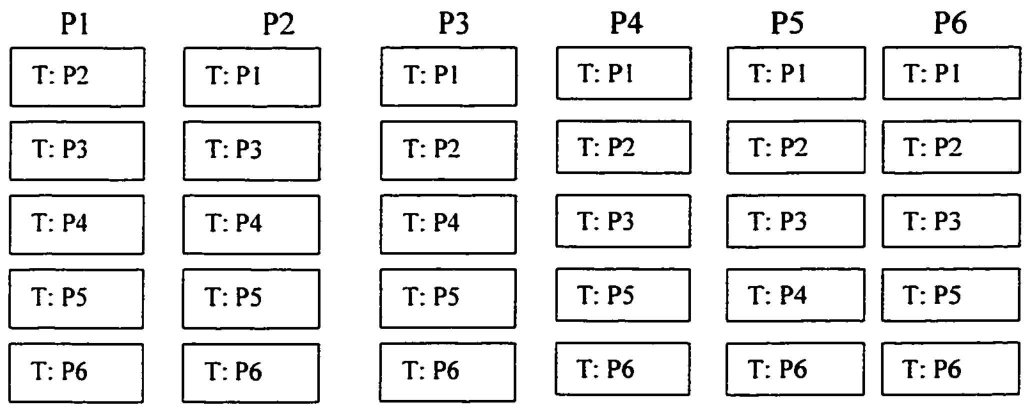


Figure 3.9 Resources used under Fully distributed Model

Due to these threads, the original process can be slowed down. However, whether these threads produce pooper performance would depend on how often they are active. From the message passing perspective that should not be a drawback.

Finally, we summarize comparisons of three implementation models using table 3.1.

Model Name	Server Speed	Client Speed	Broadcast	Replication	Developing Difficulty
Central Server	Low	High	Not Needed	Each process Master Server	Less Difficult
Multiple Server	High	High	Not Needed	Each Process Peer Server Master Server	More Difficult
Fully distributed	N/A	Low	Needed	Each Process	Most Difficult

Table 3.1 comparisons of three implementation models

Chapter 4 Developing DSM Under Server models

4.1 DSM architecture under server models

Under the server models, the server functionality is encapsulated in *DsmServer* class. *Dsm_Proxy* class is used to provide DSM interfaces for programmers. The main tasks of *Dsm_Proxy* are to send requests, wait for responses, assemble received data locally, and send local memory segments to servers. *DsmServer* does the actual complex DSM tasks. The *DsmServer* and *Dsm_Proxy* objects are connected through TCP sockets. Figure 4.1 shows the general DSM architecture under the server models.

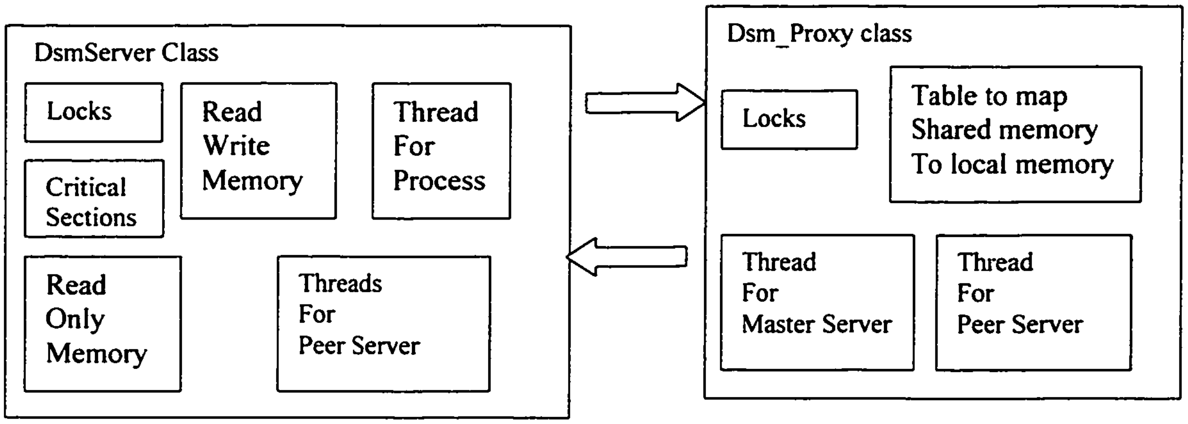


Figure 4.1 DSM architecture

Inside the *DsmServer* class, there are some important data structures such as locks, read-only shared memory and read-write shared memory. It also has many threads to monitor processes and peer servers. *Dsm_Proxy* class is simpler. It keeps the same lock structure as the servers. It has a table that contains information to map shared memory to local memory, and also has threads to monitor server responses.

4.2 Communication under the server models.

In our DSM, we use TCP/IP as the underlying protocol between processes and servers. We choose TCP because of its reliability and its relative ease in programming as compared to the UDP protocol. Each server uses one TCP port for listening requests and another port for sending responses. Figure 4.2 shows a scenario involving one master server, one peer server and three distributed processes.

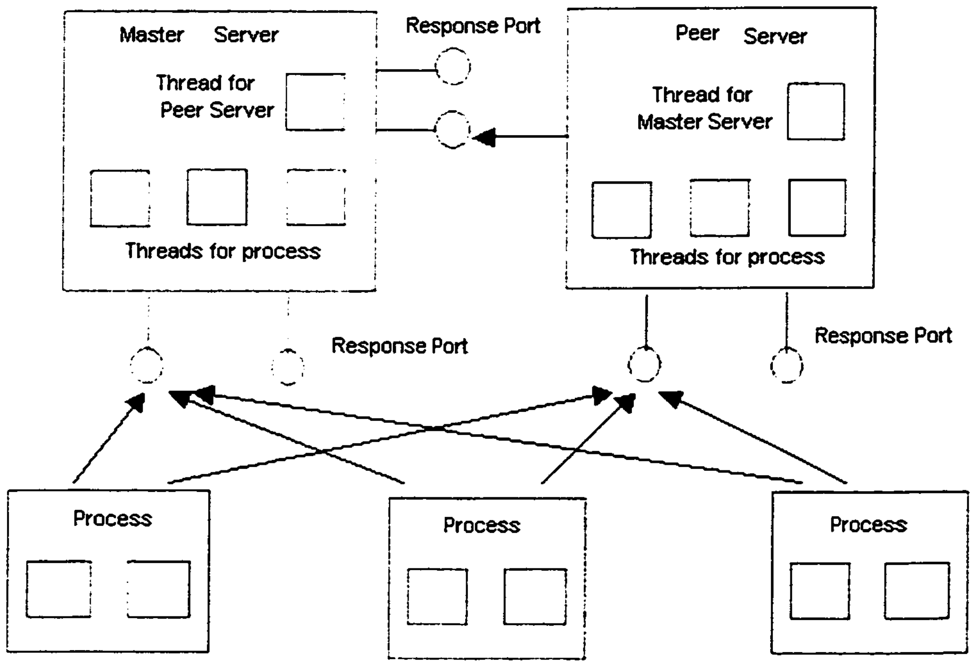


Figure 4.2 Dual port design.

Using two ports has several advantages:

- Application protocol is easier, as there is no sender/receiver switch.
- Efficiency: Both requests and responses can be pipelined.
- Programming is easier as the protocols are simpler.

Table 4.1 lists all the internal messages used between peer server and master server.

Table 4.2 lists all the internal messages used between the servers and the processes.

Name	Usage
SVR_GETSERVERID	Sent by Peer Server to Master Server to get server id
SVR_GETCLIENTPORT	Sent by Peer Server to Master Server to get which port to listen on for client requests
SVR_GETMAXPROCESS	Sent by Peer Server to Master Server to get processes will join distributed computing.
SVR_CREATELOCK	Sent by Master to Peer Servers, to notify Peer Servers is new lock is created
SVR_BINDLOCK	Sent by Master to Peer Servers, to notify Peer Servers to update lock information.

Table 4.1 Internal messages (servers)

Name	Usage
CLI_CREATEREADONLY	Sent by process to master server to create a read-only shared memory on server
CLI_GETREADONLY	Sent by process to master server to get read only shared memory copy.
CLI_CREATEREADWRITE	Sent by process to master server to create a read-write shared memory on server
CLI_REGISTERREADWRITE	Sent by process to master server to get initial information of read write shared memory.
CLI_CREATELOCK	Sent by process to master server to create a lock
CLI_BINDLOCK	Sent by process to master server to define a segment
CLI_STARTALLPROCESS	Sent by master server to all process to begin computing
CLI_ACQUIRE	Sent by process to servers to acquire a lock Also used by server to acknowledge the acquire requests
CLI_RELEASE	Sent by process to servers to release a lock

Table 4.2 Internal messages (processes to servers)

4.3 The design of DsmServer.

4.3.1 DsmServer class examples

In our system, the server functions are defined and implemented in *DsmServer* class. *DsmServer* class encapsulates both master server and peer server under one framework. *DsmServer* is designed as a Unix daemon. Its main tasks are monitoring and processing of acquire/release requests from processes. The master server has additional responsibilities in supporting declarative directives. *DsmServer* can be used to construct a master server object without peer server, a master server object with at least one peer server or a peer server object.

Example of master server with one peer server

```
DsmServer *pDsm=NULL;
//pDsm points to a master server object, which listens for process requests at port 5000
// There are two distributed processes
// The master server also listens for peer server requests at port 9000.
pDsm=new DsmServer(5000,2,9000,1);
```

Example of master server with no peer server:

```
DsmServer *pDsm=NULL;
//pDsm points to a master server object, which listens for process requests at port 6000
// There are four distributed processes
pDsm=new DsmServer(6000,4);
```

The server program needs to call function *Initialize* to check if Winsock API is installed. If *Initialize* is successful it returns 1. Otherwise it returns 0. Then it calls *Run* function to start the daemon. The following codes show how to run a server:

```
if (pDsm->Initialize())
        pDsm->Run();
```

Example of peer server

Peer server has no knowledge of the number of processes and the port assignments. So it cannot run independently. Instead, a peer server has to connect to the master server to get such information. Suppose master server "Jul" listens on port 9000. The following code illustrates a peer server:

```
pDsm=new DsmServer();
if (pDsm->Initialize())  {
        pDsm->SetMasterServer("Jul",9000);
        if (pDsm->ConnectToMasterServer())
                pDsm->Run();
}
```

4.3.2 Public Interfaces

The public interfaces of *DsmServer* class are shown as follows:

```
class DsmServer
{ public :
        DsmServer(
              unsigned int ClientPort=0,          long NumOfProcess=0,
        unsigned ServerPort=0,          long NumOfPeerServers=0);
    ~DsmServer();
    int ConnectToMasterServer();
    void SetMasterServer(char *Name,unsigned int Port);
    int Initialize();
    void Run();
    friend DWORD WINAPI MonitorThreadForClient (LPVOID p);
    friend DWORD WINAPI MonitorSvrReqThread (LPVOID p);
    friend DWORD WINAPI MonitorSvrRpsThread (LPVOID p);
private:
    //More details
};
```

We can observe that there are three thread functions in the class. Thread function *MonitorThreadForClient* is used to monitor the requests from processes. The master server uses *MonitorSvrReqThread* to monitor peer server requests. The peer server uses *MonitorSvrRpsThread* to monitor master server requests.

To make easy use of *DsmServer* class, we provide an engine program for programmers, which is written using *DsmServer* class. Thus they do not need to know any inside details of *DsmServer*. All they need to do is to call *Dsm_Proxy's Connect* function to connect to servers.

4.3.3 Internal data structures of DsmServer

The internal data structures of *DsmServer* are shown as follows:

```
class DsmServer
{  private:
          int m_Role; //the role of the server master server or peer server
          long m_MaxProcess; //max num of process
          long m_MaxPeerServer; //the max num of PeerServers
          //port for listening Peer Server request used by master server
          unsigned short m_PortServerRequest;
          //port for sending response to Peer Server use by master server
          unsigned short m_PortServerResponse;
          unsigned short m_PortClientRequest;   //port for listening client requests
          unsigned short m_PortClientResponse;   //port for sending client response
         // The dynamic array to listen process requests
         struct ClientSockType *m_PtrClientRequest;
        // The dynamic array to send responses to process
         struct ClientSockType *m_PtrClientResponse;
        // The dynamic array to listen Peer Server requests used by master server
        struct ServerSockType *m_PtrServerRequest;
        // The dynamic array to send responses to Peer Server used by master server
        struct ServerSockType *m_PtrServerResponse;
        // Used by peer server to send requests to master server
        struct ServerSockType m_MSerSockRequest;
        //Used by peer server to receive responses from master serve
        struct ServerSockType m_MSerSockResponse;
        //The read only shared memory
        struct ReadOnlyMem m_RdMem[MAX_RDONLY];
        //how many read only shared memory
        int m_RdMemIndex;
        //The read write shared memory
        struct ReadWriteMem m_RWMem[MAX_RW];
        //how many read write shared memory
        int m_RWMemIndex;
        struct LockType m_Lock[MAX_LOCK]; //The lock structure
        int m_LockIndex; //How many lock
        //Each lock has a critical section to protect it.
        CRITICAL_SECTION m_Acqcs[MAX_LOCK];
}
```

Identification of processes and servers

In distributed computing, it is very important to identify each process and server. In our system this is performed through the use of the following two structures: *ClientSockType* and *ServerSockType*. *ClientSockType* is used to identify processes and *ServerSockType* is used to identify peer servers. The following shows their details:

```
struct ClientSockType {
          SOCKET sock;
          int pid;                    //process id
};
struct ServerSockType {
          SOCKET sock;
          int ServerID;                      //Server id
          char ServerName[MAX_LENGTH]; //Server name
};
```

ServerSockType is used between the peer server and master server. Master Server assigns server id when it accepts a peer server connection. A process has no knowledge of peer servers, thus the master server has to keep their IP addresses in *ServerName* field.

The shared memory structure:

In our system, read-only and read-write shared memory have similar structure. Their structures are shown as follows.

```
struct ReadOnlyMem {
  char MemId[MAX_LENGTH];        char *MemBuf;
  int  TypeSize;                 int  NumOfElements;    };
struct ReadWriteMem {
  char MemId[MAX_LENGTH];        char *MemBuf;
  int  TypeSize;                 int  NumOfElements;    }
```

MemId field is a string used to uniquely identify a shared memory. *MemBuf* is a pointer to the memory chunk that actually stores the shared memory. *TypeSize* is a field that identifies type size of shared memory elements. *NumOfElements* field identifies the number of elements stored in the shared memory. *TypeSize* and *NumOfElements* allow us to support various data types.

The lock structure:

In our system, the lock structure is the key structure to implement acquire/release protocol and scope consistency. The lock structure differs slightly from the one used by fully distributed model. The lock structure used in this model is shown as follows:

```
struct LockType {
    int LockId;        int ServerId;   int NumOfSeg;
    struct ReadWriteSegment Segment[MAX_SEG]; };
```

Segment structure:

```
struct ReadWriteSegment {
    int RWIndex;                char MemId[MAX_LENGTH];
    int StartIndex;             int NumOfElements;
    int TypeSize;};
```

Segment structure is also used to implement scope consistency. *Rwindex* field is the index for the read-write shared memory array. *MemId* field uniquely identifies the read-write shared memory. *StartIndex* field identifies the beginning of a segment. *NumOfElements* field identifies the size of segment. *TypeSize* identifies the element type size.

Lock Synchronization

Under acquire/release protocol, it is very important that at any given time only one process is granted a same lock. Under the server models, this is implemented by using Window NT's *CRITICAL_SECTION* structure. Each lock is associated with a critical section. When one process acquires the lock it enters its own critical section, when it releases the lock it leaves the critical section.

4.4 The design of Dsm_Proxy class

4.4.1 Public interfaces

Dsm_Proxy class is provided to the programmers to use DSM functions. The following shows its public interfaces:

```
class Dsm_Proxy
{public:
        void SetProxyType(int proType);
        int Release(int LockId);
        int Acquire(int LockId);
        int BindLock(int LockID, char *MemId, int StartIndex, int NumSeg);
        void StartMonitor();
        int CreateLock();
        void StartAllProcess();
        int RegisterReadWrite(char *MemId,char *localbuf);
        int CreateReadWrite(char *MemId, char *localbuf, int NumOfElements,
                              int TypeSize);
        int GetReadOnly(char *MemId,void *localbuf);
        int CreateReadOnly(char *MemId, char *localbuf, int NumOfElements,
                              int TypeSize);
        int GetProcessId();
        int Connect(char *Server, int Port);
        int Initialize();
        Dsm_Proxy(int ProxyType=PROXYSERVER);
        virtual ~Dsm_Proxy();
        friend  DWORD WINAPI MonitorSvrRpsThread (LPVOID p);
        friend DWORD WINAPI MonitorPeerSvrRpsThread (LPVOID p);
};
```

Functions: *CreateLock*, *BindLock*, *CreateReadOnly*, *GetReadOnly*, *CreateReadWrite*, *RegisterReadWrite*, *GetProcessId*, *SetProxyType*, *Initialize*, *Connect*, *Acquire* and *Release* are the same for all models.

The following thread functions are only used under the server models:

MonitorSvrRpsThread is used to monitor responses from master server.

MonitorPeerSvrRpsThread is used to monitor responses from peer server(s).

4.4.2 Internal Data structures of Dsm_Proxy

```
class Dsm_Proxy
{private:
        int m_ProxyType; //indicate if server is needed to provide DSM functions
        //Socket type to send requests to master server
        struct ClientSockType m_MSvrSockReq;
        //Socket type to receive responses from master server
        struct ClientSockType m_MSvrSockRps;
        long m_NumPSvrs; //Num of Peer Servers
        //Dynamic socket type array used to send requests to peer server
        struct ClientSockType *m_PSvrSockReq;
        //Dynamic socket type array used to receive responses from peer server
        struct ClientSockType *m_PSvrSockRps;
        long m_ProcessId; //The process id
        struct PeerServerList m_PSvrList[MAX_PSVR]; //Peer Server List
        struct LockType m_Lock[MAX_LOCK]; //Lock structure
        int m_LockNum; //Num of locks
        //m_LocalMem is used to build a mapping between
        //local memory and shared memory
        struct RWMemLocalMem m_LocalMem[MAX_LOCAL];
        int m_LocalMemIndex;
}
```

In the server models, each process uses *m_MsvrSockReq* and *m_MsvrSockRps* to communicate with master server. Dynamic array *m_PsvrSockReq* and *m_PsvrSockRps* are used to communicate to peer server(s).

Each process has the same lock structure as the one used in servers. *m_LocalMem* is used to build the link between local memory and shared memory. *m_Lock* and *m_LocalMem* are used to implement acquire/release protocol and scope consistency.

RWMemLocalMem structure is shown as follows:

```
struct RWMemLocalMem {
        char MemId[MAX_LENGTH];      //the unique shared memory id
        void *pLocal;    //the pointer to local buffer that store shared memory
        int NumOfElements;  //redundant for ease of programming
        int TypeSize;            //redundant for ease of programming
};
```

4.5 An example of internal data structures

To show the internal structures of servers and processes, we use the following codes:

```
        int ary[200];  int lock[6];
        for (int i=0; i<200; i++)  ary[i]=200-i;  //Init ary to descending order
        //Create 6 locks
        lock[0]=dsm_proxy.CreateLock();
        lock[1]=dsm_proxy.CreateLock();
        lock[2]=dsm_proxy.CreateLock();
        lock[3]=dsm_proxy.CreateLock();
        lock[4]=dsm_proxy.CreateLock();
        lock[5]=dsm_proxy.CreateLock();
```

```
//Create read only memory "lock"

dsm_proxy.CreateReadOnly("lock",(char *)lock[0],6,sizeof(int));

//Create read write memory "ary"

dsm_proxy.CreateReadWrite("ary",(char *)ary,200,sizeof(int));

//define segments and bind to locks

dsm_proxy.BindLock(1,"ary",0,20);

dsm_proxy.BindLock(2,"ary",20,40);

dsm_proxy.BindLock(3,"ary",60,30);

dsm_proxy.BindLock(4,"ary",90,30);

dsm_proxy.BindLock(5,"ary",120,50);

dsm_proxy.BindLock(6,"ary",170,30);
```

Master Server/Peer Server side:

M_Lock:

LockId	ServerID	NumOfSeg	{MemId, start,Num,Type}
1	0	1	"ary" 0 20 4
2	0	1	"ary" 20 40 4
3	0	1	"ary" 60 30 4
4	0	1	"ary" 90 30 4
5	0	1	"ary" 120 50 4
6	0	1	"ary" 170 30 4

m_Acqcs (critical section for Acquire)

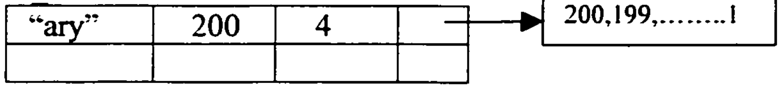
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized

m_RdMem

MemID	Num	Type	
"lock"	6	4	$\rightarrow$ 1,2,3,4,,5,6

M_RdMemIndex=1

m_RWMem



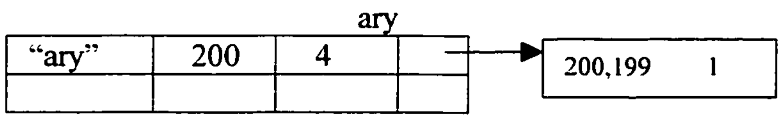
M_RWMemIndex=1

Process side:

M_Lock

LockId	ServerID	NumOfSeg	{MemId, start,Num,Type}
1	0	1	"ary" 0 20 4
2	0	1	"ary" 20 40 4
3	0	1	"ary" 60 30 4
4	0	1	"ary" 90 30 4
5	0	1	"ary" 120 50 4
6	0	1	"ary" 170 30 4

m_LocalMem



4.6 Supporting declarative directives under server models

In our system the following declarative functions are supported under the server models: *GetProcessId, CreateReadOnly, GetReadOnly, CreateReadWrite, RegisterReadWrite, CreateLock* and *BindLock*. The following shows their implementation details.

Function *GetProcessId* sends a request to the master server. Then the master server sends back the assigned process id. Process id reflects the order in which processes connect to the master server. Figure 4.3 shows its details.

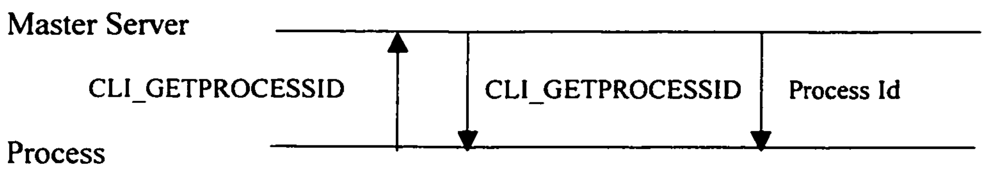


Figure 4.3 GetProcessId

The first process connected to the master server uses function *CreateReadOnly* to create read-only shared memory on the server. Figure 4.4 shows its details.

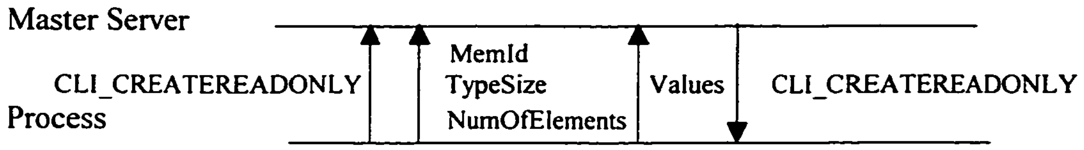


Figure 4.4 CreateReadOnly

All other processes use *GetReadOnly* function to get the values set by the first process. Figure 4.5 shows its details.

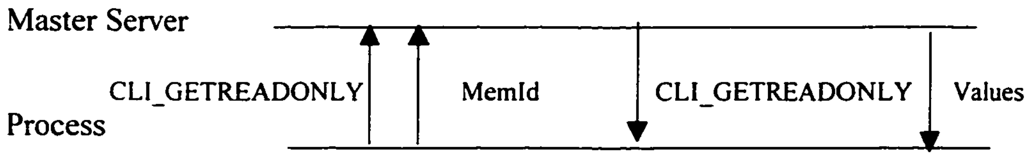


Figure 4.5 GetReadOnly

The first process connected to the master server uses *CreateReadWrite* to create read-write shared memory on the server. Figure 4.6 shows its details.

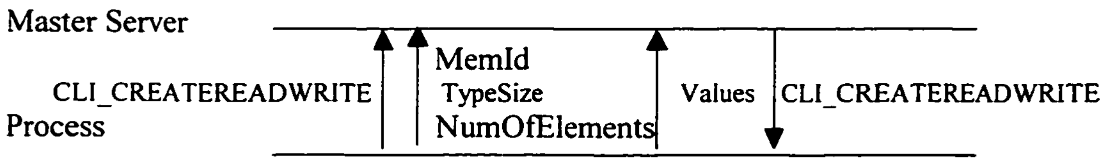


Figure 4.6 CreateReadWrite

All other processes use *RegisterReadWrite* function to get the initial descriptions set by the first process. *RegisterReadWrite* is used to build a link between local memory and read-write shared memory. Figure 4.7 shows its details.

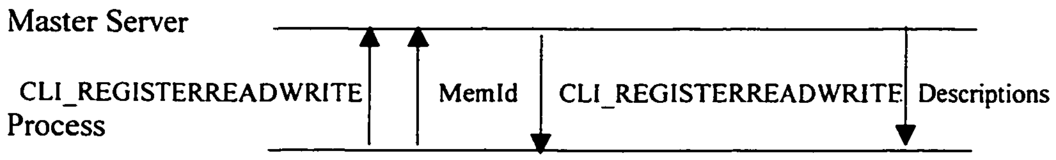


Figure 4.7 RegisterReadWrite

The first process connected to the master server uses *CreateLock* to get a unique lock id from the master server. Lock id begins from 1 and is increased by 1 when a new lock is created. Figure 4.8 shows its details.

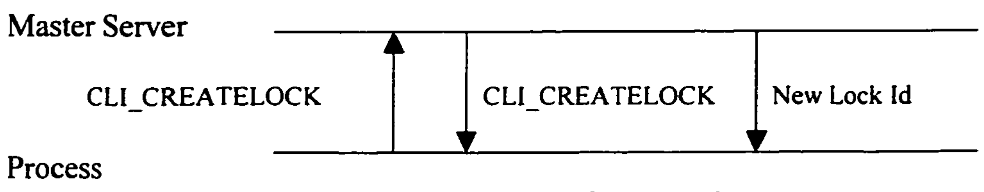


Figure 4.8 CreateLock

The first process connected to the master server uses *BindLock* to define data segments protected by a lock. Figure 4.9 shows its details.

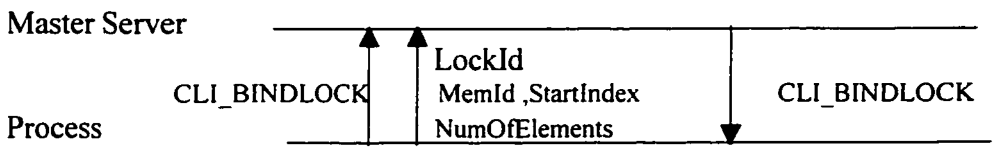


Figure 4.9 BindLock

4.7 Initialization under server models

Initialization for Master Server: If a master server has more than one peer server, it waits for all peer servers to connect to it before it accepts any process connection. Then it waits for the connection of all the processes.

Initialization for Peer Server: After peer server connects to the master server, it requests the master server for its server id, TCP port to listen on and the number of processes. Then it allocates resources accordingly and starts a thread to monitor information sent from the master server.

Initializations for Process: After a process connects to the master server, it requests the master server for its process id and the number of peer servers. If the master server has more than one peer server, the process asks the master server to send back the peer server list and connects to each peer server.

Replications: Under the server models, lock information needs to be replicated. Before the first process connects to the master server, all peer servers have already connected to the master server. So replication to peer server happens right after a process sends declarative requests to the master server. Client requests *CLI_CREATEREADWRITE*, *CLI_CREATELOCK* and *CLI_BINDLOCK* are sent by the master server to peer servers as *SVR_CREATEREADWRITE*, *SVR_CREATELOCK* and *SVR_BINDLOCK*. Thus the peer servers can create similar structures.

Replicating lock information to all processes is more complex. When the first process sends declarative requests, other processes may not have yet connected to the master server. Thus replication only happens after all processes have connected to the master server and there is no more declarative requests. To satisfy these conditions, special synchronization functions need to be called.

The first process is responsible to call *StartAllProcess* to tell the master server that there is no more declarative requests. All remaining processes need to call function *WaitForMasterProcess* to wait for the primary process to complete declarative requests. The functions *GetReadOnly* and *RegisterReadWrite* can only be called after function *WaitForMasterProcess* is called. If the master server knows that all connections are made and there are no more declarative requests, it replicates lock information to all processes. Then all distributed processes can move on.

4.8 Acquire/Release Protocol under the server models

Under the server models, the general application protocol between server and client is shown as follows:

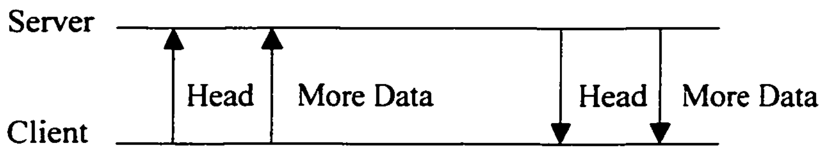


Figure 4.10 General application protocol

The head uses the following *SvrCmdPack* structure:

```
struct SvrCmdPack {
        int CmdType;            // command type  tell server what task to perform
        long MoreRemains;    // indicate if there are more data        };
```

The client first sends head to the server. If more data is needed, the client sends more necessary packets to the server. After the server processes the command, it acknowledges the client using a head that has the same command type. If there are more data sent from the server, the client receives them as well.

Under the server models, acquire/release protocol needs the coordination between the server and processes, with the server performing most of the tasks.

Client side acquire protocol is shown as the following:

(1)Prepare acquire command.

(2)According to *m_Lock* determine which server to send request to.

(3)Send acquire command to specific server.

(4)Send *LockId* to the same server.

(5)Wait server to send back acquire command.

(6)Wait for lockid and seg.

(7)Wait for segment description.

(8)Wait for segment data.

(9)According to *m_LocalMem* copy received data to local memory.

In the central server model, step (2) is not needed.

Server side acquire protocol is shown as the following:

(1)Wait for acquire command.

(2)Wait for lockid.

(3)Servers try to enter critical section according lockid.

(4)Acknowledge client with acquire command.

(5)Send lockid and num of segments back to client.

(6)Send each segment's description to client.

(7) Send each segment's data to client.

The following shows acquire protocol details between server and client.

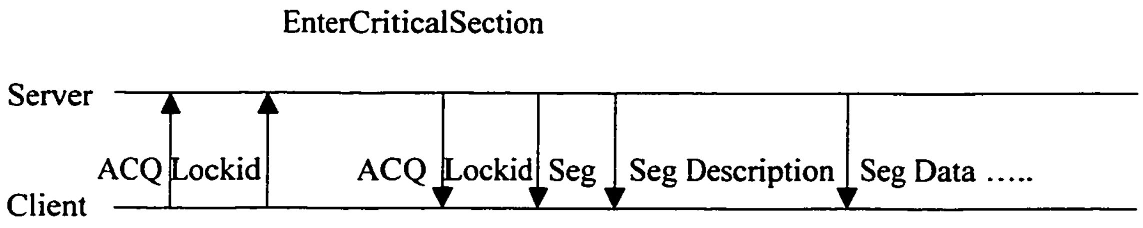


Figure 4.11 Acquire Protocol

Client side release protocol is shown as the following:

(1)Prepare release command.

(2)According to *m_Lock* find which server to send request to.

(3)Send release command to specific server.

(4)Send *LockId* to the same server.

(5)Send segment description.

(6)Assemble segment data.

(7)Send segment data.

(8)Repeat (5)-(7) for all segments.

Server side release protocol is shown as the following:

(1)Wait for release command.

(2) Wait for *LockId.*

(3) Wait for segment description.

(4) Wait for segment data.

(5) Update the copy on the server side.

(6) Repeat (3)-(6) for all segments.

(7) LeaveCriticalSection.

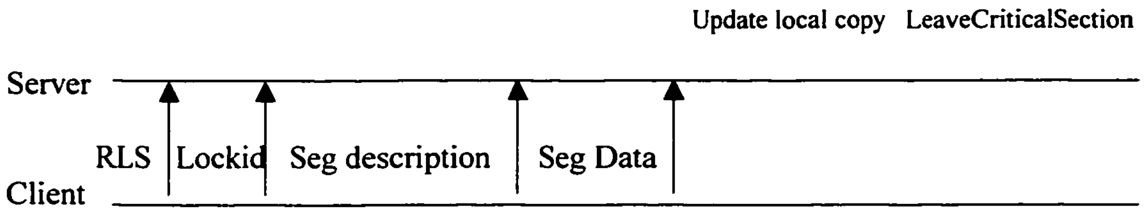


Figure 4.12 Release Protocol

Chapter 5 Developing DSM Under Fully Distributed Model

5.1 DSM architecture under fully distributed model

In this model, *Dsm_Proxy* class is not design as a thin client. Complex DSM tasks previously performed in *DsmServer* class are now moved to *Dsm_Proxy* class. Figure 5.1 shows the DSM architecture under fully distributed model. In order to support declarative directives, the structure for *ReadOnly* memory is added to the *Dsm_Proxy* class. Critical sections are not used, as synchronization is realized through the acquire/release protocol.

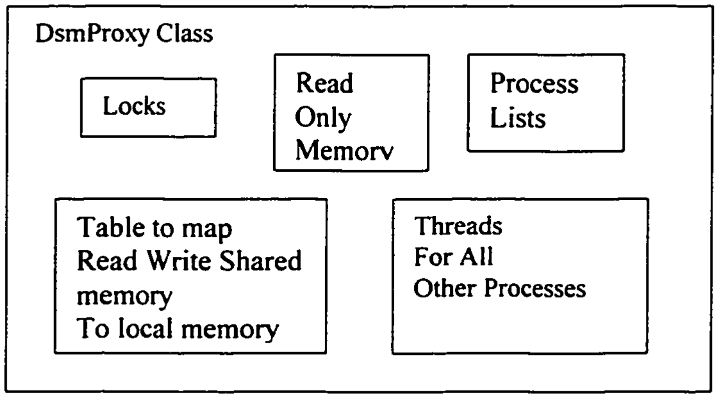

DsmProxy Class
Locks
Read Only Memorv
Process Lists
Table to map Read Write Shared memory To local memory
Threads For All Other Processes

Figure 5.1 DSM architecture under fully distributed model

Additional fields in the Lock structure are used. They are:

- *CurrentOwnerId* identifies the process that currently owns the lock.
- *LastWriteProcess* identifies the process that most recently wrote.
- *NewVersion* identifies the most recent version of a data segment.
- *CurVersion* identifies the version of a local data segment.

One interface function *SetPrimaryProcess* is added to allow a programmer to set the first process. Function *SetPrimaryProcess* accepts two parameters:(1) the number of processes, and (2) the port at which the process listens to other processes.

SetPrimaryProcess should be called before Initialize is called. The following shows an example:

```
//There are two 2 processes, each listen on TCP port 5000.
Dsm_Proxy dsm_proxy(NOPROXY);
dsm_proxy.SetPrimaryProcess(2,5000);
if (dsm_proxy.Initialize() ) { // more codes here };
```

To make DSM programming easier, each process only needs to know the computer name or IP address of the primary process. The primary process keeps a computer list of all processes connected to it and broadcast this list to all other processes. Then each process can build the communication channels as shown in figure 5.2

5.2 Communication under fully distributed model

In the fully distributed model, the most recent data may exist in any of processes. Each process is involved in the synchronization. Direct communication channel is used between any pairs of processes.

Figure 5.2 shows the communication among three distributed processes. Each process listens on the same port for the other processes. If there are n processes, a process has n-1 sockets for listening to requests from the other processes. It also has n-1 sockets for sending to the other processes. A process has an internal dynamic socket array *Req* for listening to all other processes, and an internal dynamic socket array *Rps* for connecting to all other processes.

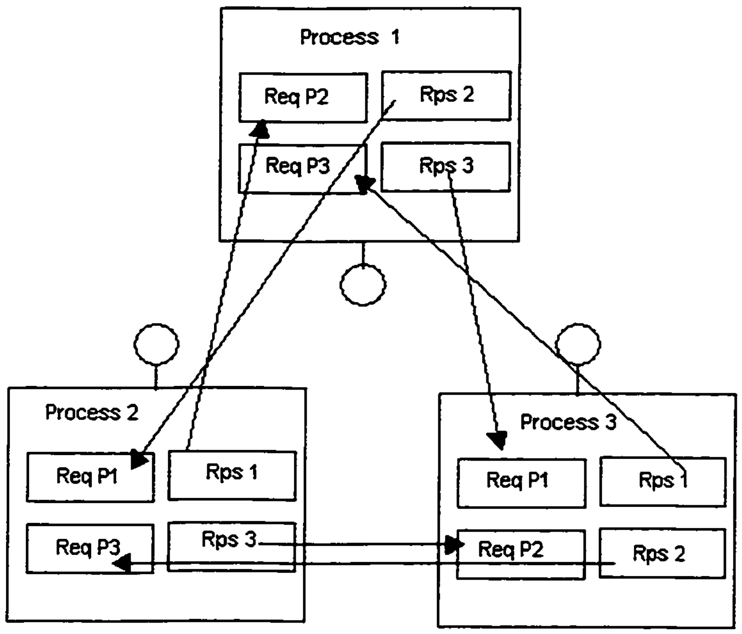


Figure 5.2 Communication example under fully distributed model.

Table 5.1 lists all internal messages exchanged between processes. We can see that fewer internal messages are used under fully distributed model. This is because declarative messages are not sent. Declarative operation is processed locally by the first process.

Name	Usage
CLI_ACQUIRE	Sent to the process who last acquire the lock
CLI_ACQACK	Sent back to process who acquires it
CLI_RELEASE	Sent to every process that there is a new version
CLI_LOCKOWNER	Sent to every process who currently acquires the lock
CLI_LOCKRELEASE	Sent to every process that lock is released.

Table 5.1 Internal messages exchanged among processes

5.3 The design of Dsm_Proxy Class

5.3.1 Public interfaces

The public interfaces of *Dsm_Proxy* used under fully distributed model are shown as follows:

```
class Dsm_Proxy
{public:
        void SetPrimaryProcess(int NumOfProcess,int ListenPort);
        int Release(int LockId);
        int Acquire(int LockId);
        int BindLock(int LockID, char *MemId, int StartIndex, int NumSeg);
        int CreateLock();
        void StartAllProcess();
        void WaitForMasterProcess();
        int RegisterReadWrite(char *MemId, char *localbuf);
        int CreateReadWrite (char *MemId, char *localbuf, int NumOfElements,
                              int TypeSize);
        int GetReadOnly(char *MemId, void *localbuf);
        int CreateReadOnly(char *MemId, char *localbuf, int NumOfElements,
                            int TypeSize);
        int GetProcessId();
        int Connect(char *Server, int Port);
        int Initialize();
        Dsm_Proxy(int ProxyType=PROXYSERVER);
        friend DWORD WINAPI MonitorProcessThread (LPVOID p);
        friend DWORD WINAPI WaitConnectThread (LPVOID p);
}
```

Functions: *CreateLock, BindLock, CreateReadOnly, GetReadOnly, CreateReadWrite, RegisterReadWrite, GetProcessId, Acquire* and *Release* are the same as the server models. There are some minor differences in initialization functions used in different models. *SetPrimaryProcess* is only used under fully distributed model to set which process is the primary process. The primary process does not use the *Connect* function. Function *StartMonitor* is not used. The primary process uses *StartAllProcess* function. All other processes use *WaitForMasterProcess* for initialization.

Under fully distributed model, except in the primary process, all other process use thread function *WaitConnectThread* to wait for other processes to connect. Thread function *MonitorProcessThread* is used to process acquire/release related requests an can process more internal messages such as *CLI_LOCKOWNER*, *CLI_ACQACK* and *CLI_LOCKRELEASE*.

5.3.2 Internal data structures

The internal data structures of Dsm_Proxy class used under the fully distributed model are shown as follows:

```
class Dsm_Proxy  {
   private:
         int m_ProxyType; //indicate if extra server is needed
         long m_ProcessId; //The process id
         struct LockType m_Lock[MAX_LOCK]; //Lock structure
          int m_LockNum; //How many lock
         //The table to map Read-Write memory to local memory
         struct RWMemLocalMem m_LocalMem[MAX_LOCAL];
         int m_LocalMemIndex;
         // The Read Only shared memory structure
         struct RWMemLocalMem m_ReadOnly[MAX_LOCAL];
         int m_ReadOnlyIndex;
```

```
        //How many processes will join the distributed processes
        int m_MaxProcess;
        struct ProcessList *m_ProcessList; //The process list structure
        //Dynamic socket array used for listen other processes to connect
        struct ClientSockType *m_ProReq;
        //Dynamoc socket array used to connect other process
        struct ClientSockType *m_ProRps;
        //Which port is used to listen requests
        unsigned short m_PortReq;
};
```

ProcessList is the only new data structure used under fully distributed model. The following shows its details:

```
struct ProcessList {
  int Pid;
  char ComputerName[MAX_LENGTH];
};
```

5.4 Supporting declarative directives under fully distributed model

Similar to the server models, the following declarative functions are supported: *GetProcessId, CreateReadOnly, GetReadOnly, CreateReadWrite, RegisterReadWrite, CreateLock* and *BindLocK.*

Only the primary process is allowed to use function *CreateReadOnly* to create read only shared memory. *CreateReadOnly* involves only local operations. It fills a new element in *m_ReadOnly* array and increases its index. Read-only shared memory structure is replicated to other processes when they connect to the primary process. All other processes use *GetReadOnly* function to get the values set by the primary process.

Function *GetReadOnly* looks up the replicated *m_ReadOnly* structure to get the values. Hence it does not generate network traffic.

Only the primary process is allowed to use function *CreateReadWrite* to create read-write shared memory. *CreateReadWrite* builds a mapping between read-write shared memory and local memory. The structure *m_LocalMem* built by *CreateReadWrite* is replicated to all other processes. All other processes use *RegisterReadWrite* function to build a link between local memory and read-write shared memory. The replicated *m_LocalMem* structure contains a local pointer, which cannot be directly used and must be allocated locally.

Only the primary process is allowed to use *CreateLock* to get a unique lock id, lock id begins with 1 and is increased by 1 when a new lock is created.

Only the primary process is allowed to use *BindLock* to define segments protected by the lock. *BindLock* initializes the lock structure fields *NewVersion, CurVersion*, and *LastWriteProcess*. Lock structure is replicated to all other processes. But some fields are changed locally. For example, fields *NewVersion, CurVersion* are both set 1 for primary process. When the primary process initially acquires a lock, it knows it already has a local copy. For other processes *NewVersion* is set 1, and *CurVersion* is set 0. This means that there is a most recent copy elsewhere.

5.5 Initialization under fully distributed model

The initialization under fully distributed model are quite different from that under the server models. The initialization details of primary process are shown as follows:

(1) Listen on local port.

(2) Accept a connection

(3) Send pid, max_process, port back to connected process

(4) Send lock information back to connected process

(5) Send read-only shared memory information back to connected process

(6) Send read-write shared memory information back to connected process

(7) Add the new connected process to its process list

(8) Wait for connected process to start its thread that listen on its own port

(9) Connect to that Process

(10) Send back Acknowledge message

(11) Start a thread to monitor connected process

(12) Repeat (2) to (11) until all other processes have connected

(13) Broadcast process list

(14) Wait remaining processes to connect each other.

(15) Broadcast message to move on

The following shows initialization details of other processes:

(1) Try to connect to primary process.

(2) Wait for pid, m_maxprocess, and port.

(3) Wait for lock information.

(4) Wait for read-only shared memory information.

(5) Wait for read-write shared memory information.

(6) Allocate resources according to m_MaxProcess.

(7) Create a thread to allow other processes to connect and notify primary process to connect.

(8) Wait for acknowledge from the primary process to connect this process.

(9) Create a thread to monitor process requests.

(10) Wait for process list.

(11) Try to connect all remaining process according to process list.

(12) Send acknowledging message to primary process.

(13) Wait for primary process to move on.

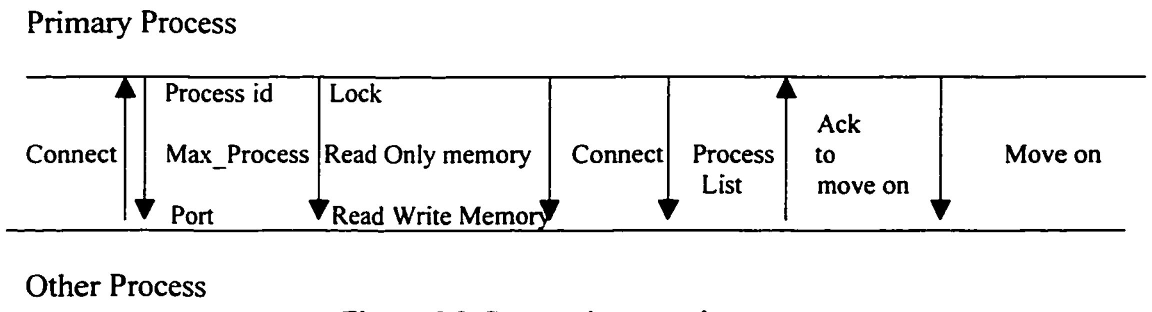


Figure 5.3 Connection to primary process

Figure 5.3 shows how a process connects to the primary process. From figure 5.3 we can see that *Max_Process*, *Port*, *Lock*, *ReadOnly* memory, *ReadWrite* memory and process list are replicate to all other processes from primary process.

5.6 Acquire/Release protocol under fully distributed model

Acquire/release is performed at each process. Some new internal messages are used: *CLI_LOCKOWNER* informs other processes the owner of the lock. *CLI_ACQACK* acknowledges for *CLI_ACQUIRE*. *CLI_LOCKRELEASE* releases a lock.

The details of acquire protocol are shown as follows:

(1) Check *CurrentOwenerId* field of lock structure. Acquire is blocked until *CurrentOwenerId* is empty.

(2) If *CurVersion* equals *NewVersion*, broadcast *CLI_LOCKOWNER* and *LockID* message to other processes. Lock is granted and local copy is used.

(3) If the *CurVersion* is not equal to *NewVersion*, *LastWriteProcess* field identifies which process to send *CLI_ACQUIRE* request.

(4) For the process that receives *CLI_ACQUIRE* requests, send back its data page and acknowledge it with *CLI_ACQACK* message.

(5) Wait for *CLI_ACQACK* response from that process.

(6) Assemble data from that process.

(7) Broadcast *CLI_LOCKOWNER* and *LockID* message to other processes as lock is granted

From the above, it can be observed that acquire request is not sent to others if the lock is already granted. The process waits until no one owns the lock. If no one owns the lock, it checks if it already has the most recent page. If so, the local copy is used. Then the lock is granted by broadcasting to the world that it is the lock owner. If it does not have the most recent one, then a remote copy is used. It sends a *CLI_ACQUIRE* request to the process that last wrote on it. Upon the receipt of *CLI_ACQACK*, it assembles the received data. Then lock is granted by broadcasting that it is the lock owner to all other processes.

The details of release protocol are shown as follows:

(1) Increase *NewVersion* by one.

(2) Assign *NewVersion* to *CurVersion*.

(3) Broadcast *CLI_LOCKRELEASE*, lock id, process id and new version to every process.

(4) For other processes, upon receiving *CLI_LOCKRELEASE* message, they receive lock id, process id and new version. Then set the lock's *CurrentOwenerId* field to be empty, and reset *NewVersion* and *LastWriteProcess* fields according to the message received.

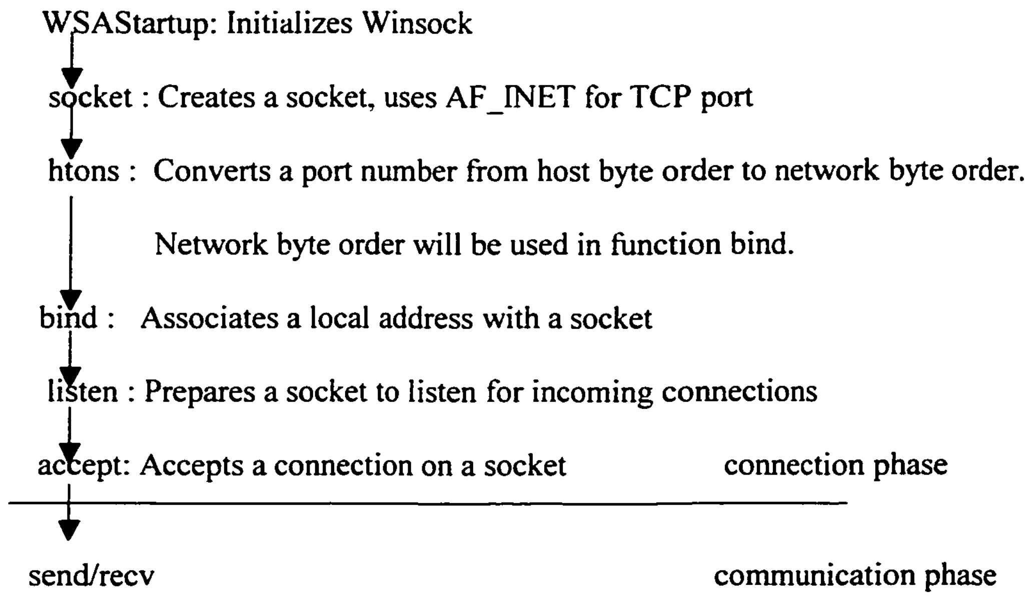
Chapter 6 Programming issues when implementing DSM

6.1 How to use TCP/IP multithreaded programming in DSM development

6.1.1 System calls used in TCP/IP programming

TCP/IP is used in our DSM as the underlying communication protocol. We use a lot of TCP/IP programming in our development. For more detailed information on TCP/IP programming please refer to [14]

The following diagram shows Winsock APIs call sequence in a server application:



WSAStartup: Initializes Winsock

socket : Creates a socket, uses AF_INET for TCP port

htons : Converts a port number from host byte order to network byte order. Network byte order will be used in function bind.

bind : Associates a local address with a socket

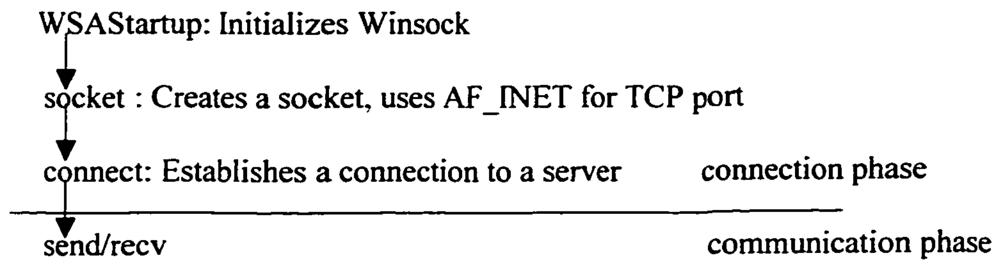
listen : Prepares a socket to listen for incoming connections

accept: Accepts a connection on a socket — connection phase

send/recv — communication phase

After a server program creates a socket, it calls *htons* to convert TCP port. Then it uses *bind* to specify which port it listens and uses *listen* for incoming connections. Finally it calls *accept* to wait until a connection is made. For multi-clients server program, a static/dynamic socket array structure should be used to keep multiple connections. Then a server communicates with clients using *send/recv*.

The following diagram shows Winsock API call sequence in a client program:


WSAStartup: Initializes Winsock
socket : Creates a socket, uses AF_INET for TCP port
connect: Establishes a connection to a server
connection phase
send/recv
communication phase

After a client program creates a socket, it calls *connect* to connect to the server, and interacts with the server using *send/recv* commands.

Both the server program and the client program can be divided into two phases:

- Connection phase: In this phase server and client use some APIs to establish connections. After connections are made, these APIs are not used again.
- Communication phase: In this phase the server and the client use *send/receive* and other APIs to communicate. These APIs are repeatedly used when needed.

6.1.2 Methods to serve multiple clients

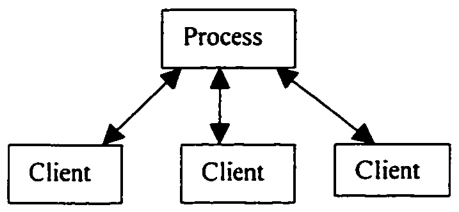
Generally there are three methods to design multi-client server program:

(a) Use one process by using *select* function call

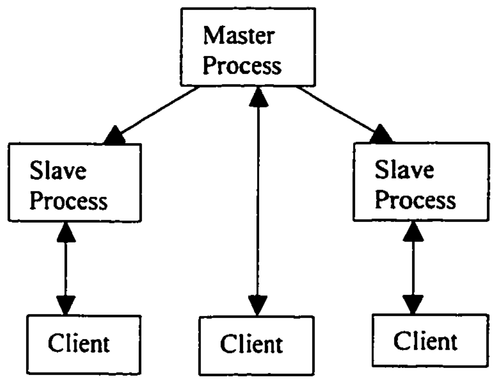
(b) Use multiple processes by using *fork.*

(c) Use multiple threads in one process

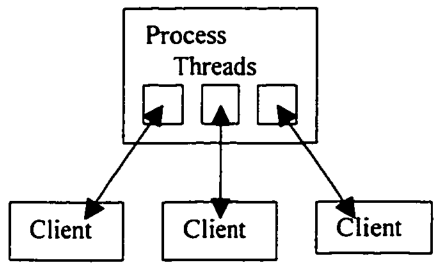
Method (a) uses function *FD_CLR, FD_SET* to add socket descriptors to a set. Then it uses *select* to wait until descriptor to become ready. *Select* is blocking until the descriptor is ready. Then we can use *FD_ISSSET* to test which descriptor is ready. Method (b) uses Unix system call *fork* to generate identical processes. Each process services a user connection. The disadvantage of this method is that it is resource expensive, and there are no shared states among these processes. Figure 6.1 shows the ideas of these three methods.



(a) One process multiple clients



(b) Multiple processes multiple clients



(c) Multiple threads for multiple clients

Figure 6.1 server programming methods

Both methods (a) and (b) are not used in our implementation because the main process is blocking. Under fully distributed model the main process is used to implement the application. Thus blocking is not acceptable. Instead, we use method (c). Threads are running in the background and are parallel to main process. The other benefit of using threads is that the main process and threads uses and shares the same address space, internal data sharing is easier.

Figure 6.2 shows the differences between thread and process. A process owns all the necessary resources to run an application. Two different processes do not shared resources such as global data and heap. A thread is running separately within the process. So they shared the global data and heap. It's much quicker and more efficient for thread switching than process switching.

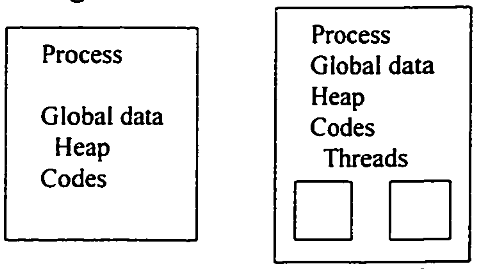


Figure 6.2 Process and thread

6.1.3 Using threads in a class

Under Windows NT, the thread function must have the following prototype:

DWORD WINAPI FuncName (LPVIOP p);

The function returns a *DWORD*, which is used to provide return status. Usually we use API function *CreateThread* to create a thread in the same process. However we should know that thread APIs are C style and not C++ APIs. You should be very careful when you use them. The prototype of *CreateThread* is shown as follows:

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,  //Security
          DWORD dwStackSize,                            //thread stack
          LPTHREAD_START_ROUTINE lpStartAddress,    //address of thread function
          LPVOID lpParameter, // pointer to parameters
         DWORD dwCreationFlags, //thread creation flags
         LPDWORD lpThreadId //thread id);
```

The first two parameters are usually set to 0. This enables the use of default values.

Thread APIs are platform dependent. It's rather tricky in using them in C++ classes.

There are two ways to use thread in a C++ class:

(1) Specify a thread function as friend function to the class.

(2) Specify a thread function as static function to the class.

The following is an example for method (1):

```
DWORD WINAPI ThreadFunc (LPVOID p);
class Example1 {
    public :  void Start();
              friend  DWORD WINAPI ThreadFunc (LPVOID p);
    private:    //internal data structures and functions
 }
void Example1::Start()
{     DWORD tId;                    //pass this allow thread to access Example1 class
      HANDLE hThread;                                        ↓
        hThread=CreateThread(NULL,0, ThreadFunc,(LPVOID)this,0,&tId);
}
DWORD WINAPI ThreadFunc (LPVOID p){
      //Thread codes
      //We must cast ptr to Example1 class pointer
      Example1 *ptr = (Example1 *) p;
      // Then we can use ptr-> to access all Example1's class interfaces.
}
```

An example for method (2) is shown as follows:

```
class Example2 {
    public :  void Start();
    private:  static DWORD WINAPI ThreadFunc (LPVOID p);
}
```

```
void Example2::Start()
{     DWORD tId;                    // This complex type cast is needed
      HANDLE hThread;                              |
      hThread=CreateThread(                        ↓
     NULL,0, (unsigned long (__stdcall *)(void *))ThreadFunc, NULL,0,&tId);
}
DWORD WINAPI Example2::ThreadFunc (LPVOID p){
     //The ThreadFunc is a member function of Example2
     //No extra codes are needed to access it.
}
```

In our implementation, we use method (1).

6.1.4 How to implement no-byte-loss application protocol in DSM

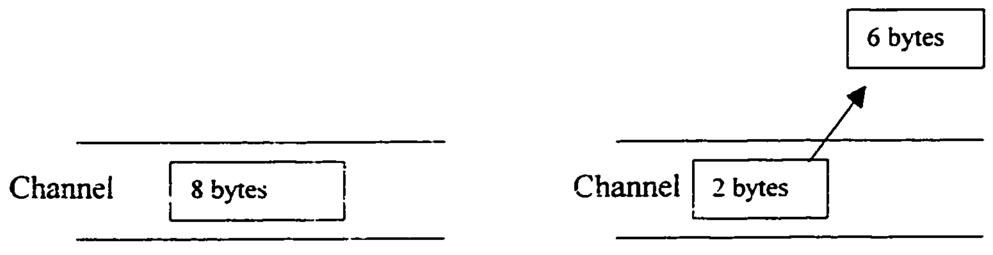
Implementing DSM requires that a protocol between a server and a client is byte-loss-free. This means even single byte loss is not acceptable. For example, if the server sends an eight-byte head to a client, the client must guarantee to receive 8 bytes. If we try to achieve this goal by simply using *recv* function call, we will have problems. The following sample code illustrates this:

```
     char head[8];
     memset(head,0,8);          //clear buffer before receive
     recv(sock, (char *)head,8,0); //the recv can return even if it receives less bytes than 8
```

The recv function receives data from a socket, and has the following prototype:

int recv (SOCKET *s*, char **buf*, int *len*, int *flags*);

Under Windows NT, the fourth parameter is not supported and is always set to zero. The problem of *recv* is that under Windows NT, it may return even if it does not fill the full length of buffer. Figure 6.3 shows the case. Before the recv is called the channel has 8 bytes. After *recv* is called, it may just read 6 bytes from the channel. This causes misinterpretation of data and failure of application protocol.



(1) Before recv (2) after recv
Figure 6.3 recv function

To implement byte-loss-free protocol, we use our self-defined *BlockRecv* function to receive incoming data stream. Function *BlockRecv* receives the same parameters as *recv*. But unlike *recv*, it is blocking until it fills the data buffer. We use *ioctlsocket* to test if the channel has "len" bytes in its buffer. The following code illustrates *BlockRecv*:

```
void BlockRecv ( SOCKET s, char *buf, int len, int flags ){
        long rec; unsigned int rcvb; int total;
        do {     //Before recv we test if the channel has more bytes than len
          ioctlsocket(s,FIONREAD,(unsigned long *)&rec);
        } while (rec<len);
        total=0;  //we will compute many bytes received in total
        do {
          rcvb=recv(s, (char *)buf+total,len-total,0); //rcvb is the bytes actually received
          if (rcvb>0)  total=total+rcvb;  //Add rcvb  to toal
          else break;
        }  while (total<len);  //loop until full len bytes are received
};
```

6.2 Programming techniques used in DSM synchronization

6.2.1 Using critical section to synchronize acquire requests

Under the server models, acquire requests are sent to a specific server. Because at any time at most one process is allowed to get the lock, we need a mechanism to solve the problem when the server simultaneously receives acquire requests from several processes for the same lock. One possible solution is to queue the acquire requests. When there is no owner of the lock or the lock is released, the process on top of the queue is removed and granted the lock. We do not use this method because it needs additional coding. Instead, critical section is used in our implementation. Critical section is a simpler solution when the threads of a single process need mutual-exclusion synchronization. We should know that there is no guarantee about the order in which threads will enter the critical section. However, the operating system guarantees fair selection of all threads. In our case the order is not important. The following shows the related codes:

```
class DsmServer {
        //Each lock has its own critical sections.
        private:    CRITICAL_SECTION m_Acqcs[MAX_LOCK];
}
DsmServer::DsmServer( unsigned int ClientPort, long NumOfProcess,
          unsigned int ServerPort,long NumOfPeerServers)
{         //In the constructor critical sections are initialized
           for (int i=0; i<MAX_LOCK; i++)   InitializeCriticalSection(&m_Acqcs[i]);
}
```

The following function is used by the server to process acquire requests

```
void DsmServer::ProcessAcquire(int ProcessId)
{       struct SvrStdDataPack data;
        struct SvrCmdPack cmd;
        //receive which lock id the process wants to acquire
        BlockRecv(m_PtrClientRequest[ProcessId].sock,(char *)&data,sizeof(data),0);
        //......
        EnterCriticalSection(&m_Acqcs[LockId-1]);
        //Other codes
}
void DsmServer::ProcessRelease(int ProcessId)
{       struct SvrStdDataPack data;
        BlockRecv(m_PtrClientRequest[ProcessId].sock,(char *)&data,sizeof(data),0);
        //Codes for receiving newest data page from process
        //And assemble them in the server are omitted
        LeaveCriticalSection(&m_Acqcs[LockId-1]);
}
```

6.2.1 Using Named Event for synchronization

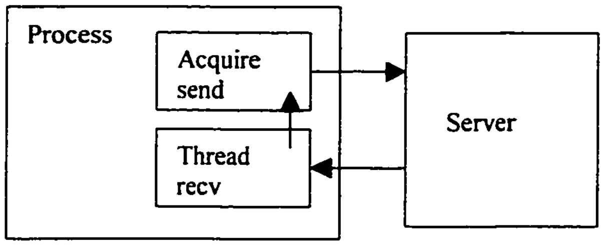
When we use multi-thread methods to implement DSM, we should avoid using *recv* function in different thread functions running concurrently. The following example demonstrates some undesirable outcome.

```
Thread 1{            Thread 2{

    recv()               recv()

}                    }
```

Both thread 1 and thread2 use *recv* function to wait for some data. Suppose thread 1 is to wait for 8 bytes head, thread 2 is to wait for 256 bytes data. The protocol used is to send the head followed by variable size data. Under the above arrangement, if thread 2 is running before thread 1, then the protocol may fail. Because thread 2 consumes the head that is needed by thread 1. So in our implementations, *recv* only happens in the thread functions or in the functions called by thread functions. Therefore there is need for thread functions and other functions to synchronize. Consider the following acquire example:



Under the server models, acquire is implemented in this way: acquire sends a request to the server, and waits for response. As *recv* is only used in a thread, when the server sends back a response, the thread function needs to notify the acquire function. But how ? One simple method is that acquire function and thread function share a flag. After acquire sends a request, it periodically polls the flag. Thread function sets the flag until it receives server response. In our implementations, we use named event to avoid the polling. It works like this: after acquire sends a request, it creates a named event and waits for it. Thread will set the name event until it receives the server response. The following example shows the details.

```
int Dsm_Proxy::Acquire(int LockId) {
   if (m_Lock[LockId-1].LastWriteProcess>0) {
        cmd.CmdType=CLI_ACQUIRE;
        cmd.MoreRemains=LockId;
        int pid=m_Lock[LockId-1].LastWriteProcess;
        sd=send(m_ProRps[pid-1].sock,(char *)&cmd,sizeof(cmd),0 );
        m_HAcq=CreateEvent(NULL,FALSE,FALSE,_T("Acq"));
        assert(m_HAcq!=NULL);
        WaitForSingleObject(m_HAcq,INFINITE);
        CloseHandle(m_HAcq);
        //More codes are omitted  here
}
```

The acquire acknowledge is supported by.

```
void Dsm_Proxy::ProcessAcqAck(int LockId , int pid)
{       struct SvrStdDataPack data;
        //codes to assemble received data are omitted here
        SetEvent(m_HAcq);
}
```

6.3 Key code examples

The examples in this section are to show the use of methodologies discussed in the previous sections. The following codes are all used under the fully distributed model. It shows how the primary process registers processes and how information is replicated:

```
void Dsm_Proxy::WaitProcessConnect()
{       SOCKET SockListenRequest;
        struct SvrStdDataPack data;
        struct sockaddr_in local1,from;
        int fromlen;
        int socket_type =SOCK_STREAM;
        int pid=2;          int res;
        fromlen =sizeof(from);
```

```
local1.sin_family = AF_INET;
local1.sin_addr.s_addr = INADDR_ANY;
local1.sin_port = htons(m_PortReq );
SockListenRequest= socket(AF_INET, socket_type,0); // TCP socket
assert(SockListenRequest!= INVALID_SOCKET);
res=bind(SockListenRequest,(struct sockaddr*)&local1,sizeof(local1));
assert(res!= SOCKET_ERROR);
res=listen(SockListenRequest,5);
assert(res!= SOCKET_ERROR);
while (pid<=this->m_MaxProcess ) {
    m_ProReq[pid-1].sock =
        accept(SockListenRequest,(struct sockaddr*)&from, &fromlen);
    assert(m_ProReq[pid-1].sock!=INVALID_SOCKET);
    m_ProReq[pid-1].pid=pid;
    //Send pid,max_process,port to connected process
    data.pData.Prm1=pid;
    data.pData.Prm2=m_MaxProcess;
    data.pData.Prm3=m_PortReq;
    send(m_ProReq[pid-1].sock,(char *)&data,sizeof(data),0);
     //Send lock information
    data.pData.Prm1=m_LockNum;
    send(m_ProReq[pid-1].sock,(char *)&data,sizeof(data),0);
    send(m_ProReq[pid-1].sock,
          (char*)&m_Lock[0],sizeof(m_Lock[0])*m_LockNum,0 );
    //send read-only shared memory information
    data.pData.Prm1=m_ReadOnlyIndex;
     send(m_ProReq[pid-1].sock,(char *)&data,sizeof(data),0);
     if ( m_ReadOnlyIndex>=1) {
        send(m_ProReq[pid-1].sock, (char *)&m_ReadOnly[0],
                     sizeof(m_ReadOnly[0])*m_ReadOnlyIndex,0);
        int blocksize;
```

```
          for (int i=0;i<m_ReadOnlyIndex;i++) {
              blocksize=
                  m_ReadOnly[i].TypeSize*m_ReadOnly[i].NumOfElements;
                  send(m_ProReq[pid-1].sock,(char *)m_ReadOnly[i].pLocal,
                          blocksize,0);
          }
    }
   //send read-write shared memory information
    data.pData.Prm1=m_LocalMemIndex;
    send(m_ProReq[pid-1].sock,(char *)&data,sizeof(data),0);
    if ( m_LocalMemIndex>=1) {
          send(m_ProReq[pid-1].sock,(char *)&m_LocalMem[0],
                   sizeof(m_LocalMem[0])*m_LocalMemIndex,0);   }
   //Add a new process to process list
   strcpy(m_ProcessList[pid-1].ComputerName,inet_ntoa(from.sin_addr));
   m_ProcessList[pid-1].Pid=pid;
  //wait for other process to start thread
  BlockRecv(m_ProReq[pid-1].sock,(char *)&data, sizeof(data),0);
  //Call ConnectProcess will let primary process connect to other process
  ConnectProcess(pid);
  //Send  ack to other process notify connect success
  send(m_ProReq[pid-1].sock,(char *)&data, sizeof(data),0);
  //start a thread to monitor process
  DWORD tId; HANDLE hThread;
   m_CurProcess=pid;
   hThread=CreateThread(NULL,0,MonitorProcessThread ,
                                  (LPVOID)this,0,&tId);
   Sleep(1000);      pid+=1;
 }
}
```

The follow thread is used to implement acquire/release protocol:

```
DWORD WINAPI MonitorProcessThread (LPVOID p)
{        Dsm_Proxy *ptr = (Dsm_Proxy *) p;
        struct SvrStdDataPack data;  struct SvrCmdPack cmd;
        int index=ptr->m_CurProcess-1;      int lockid;
        while (1) {
          //Block until receive command head
          BlockRecv(ptr->m_ProReq[index].sock,(char *)&cmd,sizeof(cmd),0);
          switch(cmd.CmdType) { //According head type, do some further process
               case CLI_ACQUIRE:
                   ptr->ProcessAcqReq(cmd.MoreRemains,index+1);  break;
               case CLI_ACQACK:
                    ptr->ProcessAcqAck(cmd.MoreRemains,index+1);   break;
               case CLI_LOCKRELEASE:
                    lockid=cmd.MoreRemains;
                    BlockRecv(ptr->m_ProReq[index].sock,(char *)&data,sizeof(data),0);
                    ptr->m_Lock[index].CurrentOwnerId=0;
                    ptr->m_Lock[index].LastWriteProcess=data.pData.Prm1;
                    ptr->m_Lock[index].NewVersion=data.pData.Prm2 ;break;
               case CLI_LOCKOWNER:
                    lockid=cmd.MoreRemains;
                    BlockRecv(ptr->m_ProReq[index].sock,(char *)&data,sizeof(data),0);
                    ptr->m_Lock[index].CurrentOwnerId=data.pShort;  break;
          }
        }
        return 1;
}
```

Conclusions & Future works

The following conclusions are drawn from this project:

- DSM can be implemented using different models (algorithms). But for the programmers their usage is almost the same. We show how to encapsulate different DSM models under one framework by using object-oriented method.

- Our developing experiences show that using multi-thread programming is more efficient. We believe multi-thread programming plus object-oriented programming is the best combination when developing DSM.

- The most important aspect of developing DSM is to handle communication among processes and servers. The communication model we propose uses dual TCP ports. This simplifies both designs and programming.

- We started our design on master server model, later we moved on to multiple server model and fully distributed solution. Our experiences demonstrate that although the algorithms are quite different under various models, most of the internal data structures and program structures are reusable.

- We use some platform dependent programming techniques in our development, such as critical section structures and named events. The use of these programming techniques considerably simplifies the coding.

The following are recommended for future improvements

- Compilation of a process into a multithreaded process is necessary in order to relax its program order in software distributed shared memory. This is a step omitted in the reported implementation. This should be dealt with before significant benefits can be obtained from the relaxed consistency model.

- Supporting DSM on Unix/Linux platforms. The existing codes use Winsock APIs. It is easy to port them to Unix/Linux using little effort.

- One major limitation of our system is that it only supports C++ at source code level. To solve this problem, we think that middle-ware components such as DCOM and CORBA are the best solution. Minor code rewriting is needed to transform the existing DSM to middle-ware components to support other commercial languages.

- Optimize existing application protocol to decrease network traffic is possible. The existing acquire/release protocol sends or receives the whole data segment. This is not needed when only a few data items are changed in the segment.

- Additional performance analysis is needed to address issues such as the scalability of DSM, the refinements of consistency models to be used and allocation of application processes to various nodes in the network.

Bibliography

[1] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocessor programs. IEEE Transactions on Computers, C-28(9): 241-248, September, 1979.

[2] Dubois, M., Scheurich, C., and Briggs, F. Memory access buffering in multiprocessors. In Proceeding of the 13th Annual International Symposium on Computer Architecture, p. 434-442, 1986.

[3] Gharachollo, K, Lenoski, D., Laudon, J., Gibbs, P., Gupta, A., and Hennessy, J., memory consistency and event ordering in scalable shared-memory multiprocessors.

[4] Carter, J.B., Bennett, J. K., and Zwaenepoel, W. Implementation and Performance of Munin. In Proceedings of the 13th ACM Symposium on Operating Systems Principles, pp 152-164, 1991.

[5] Keleher, P., Cox, A. L., and Zwaenepoel, W, Lazy release consistence for software distributed shared memory, In Proceedings of the 15th ACM Symposium on Operating Systems Principles, pp 1-10, 1993.

[6] Liviu Iftode, Jaswinder Pal Singh and Kai Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency (1996) Department of Computer Science. Proc. of the 8th ACM Annual Symp on Parallel Algorithms and Architectures

[7] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. ACM Transactions on Computer Systems, 7(4): 321-- 359, November 1989.

[8] B. N. Bershad, M. J. Zekauskas, W. A. Sawdon. The Midway distributed shared memory system. *COMPCON*, spring 1993.

[9] X. Chen and V.H. Allan *MultiJav: A Distributed Shared Memory System Based on Multiple Java Virtual Machines*, The 1998 International Conference on Parallel and Distributed Processing Technique and Applications (PDPTA'98) July 13-16, 1998, Las Vegas, Nevada, USA.

[10] S.V. Adve and M.D. Hill. Weak Ordering--A New Definition. In Proceedings of the 17th Annual Symposium on Computer
Architecture, pages 2--14, June 1990.

[11] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In Proceedings of the 21st Annual Symposium on Computer Architecture, pages 142-- 153, April 1994.

[12] Shasta: a System for Supporting Fine-Grain Shared Memory across Clusters. Daniel J. Scales and Kourosh Gharachorloo. Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing. March, 1997.

[13] Julian Templeman , Beginning Windows NT Programming, Published by Wrox Press Ltd. 1998

[14] D.E.Comer, Internetworking with TCP/IP Volume III: Client-Server programming and applications BSD sockets Version Second Edition. Published by Prentice Hall 1996.

[15] Jelica Protic et all, Distributed Shared Memory Concepts and Systems. Published IEEE Computer Society Press, 1998.