Design and implementation of page based distributed shared memory in distributed database systems

Padmanabhan Raman

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Design and Implementation of Page based Distributed Shared Memory in Distributed Database Systems

By

Padmanabhan Raman

Project submitted in partial fulfillment of the requirements for the degree of Master of Science in Information Technology

Department of Information Technology
Rochester Institute of Technology

August 1999
© Copyright 1999 Padmanabhan Raman

All Rights Reserved
Student Name: Padmanabhan Raman

Student Number: ________________

Thesis Title: Design and Implementation of Page Based Distributed Shared Memory in Distributed Database Systems

Project Committee

Name: Prof. Rayno Niemi
Signature: ____________________________
Date: 8/24/99
Chair

Name: Prof. Timothy Wells
Signature: ____________________________
Date: 8/24/99
Committee Member

Name: Prof. James Vallino
Signature: ____________________________
Date: 8/24/99
Committee Member
LIST OF FIGURES

Figure 1.1: Page Based Distributed Shared Memory ........................................ 10
Figure 2.1: Page Manager Architecture .......................................................... 11
Figure 2.2: Page Synchronization Manager Architecture ................................. 15
Figure 3.1: A Distributed Shared Memory ....................................................... 18
Figure 3.2: A Distributed Shared Memory with Caching .................................. 19
Figure 4.2.1 - 4.2.5: Cache Ownership Protocol ............................................ 24 - 26
Figure 5.1.1 - 5.1.2: Strict Consistency ......................................................... 31
Figure 5.2.1 - 5.2.3: Sequential Consistency .................................................. 32 - 33
Figure 5.3.1 - 5.3.3: Casual Consistency ....................................................... 35 - 36
Figure 5.4: PRAM Consistency and Processor Consistency ............................... 37
Figure 5.5.1 - 5.5.2: Weak Consistency .......................................................... 41
Figure 5.6: Release Consistency ...................................................................... 44
Figure 6.1: Four Message Protocol .................................................................. 54
Figure 6.2: Three Message Protocol ............................................................... 55
Figure 6.3.1 - 6.3.2: Case 1 - Case 6 ............................................................... 56 - 58
Figure 7.1: Chunk of address space distributed among 2 machines ............... 60
Figure 7.2: Situation after Node 1 references chunk 10 .................................. 61
Figure 7.3: Situation if chunk 10 is read only and replication is used .............. 62
Figure 8.1: Copyset Approach and Page Ownership ....................................... 66
Figure 11.1: Client Architecture ................................................................... 76
Figure 13.1: False sharing of a page containing two unrelated variables .......... 81
Figure 14.1: An Object .................................................................................. 85
LIST OF FIGURES

Figure 14.2: Communication Mechanism on shared objects ....................... 86
LIST OF TABLES

Table 4.1: The Write-Through Cache Consistency Protocol ...................... 21

Table 5.8.1: Consistency Models not using synchronization operations .... 49

Table 5.8.2: Models with synchronization operations............................... 49
Abstract:

This project is the simulation of page based distributed shared memory originally called IVY proposed by Li in 1986\[3\] and then by Li and Hudak in 1989\[4\]. The 'Page Based Distributed Shared Memory System' consists of a collection of clients or workstations connected to a server by a Local Area Network. The server contains a shared memory segment within which the distributed database is located. The shared memory segment is divided in the form of pages and hence the name 'Page Based Distributed Shared Memory System' where each page represents a table within that distributed database.

In the simplest variant, each page is present on exactly one machine. A reference to a local page is done at full memory speed. An attempt to reference a page on a different machine causes a page fault, which is trapped by the software. The software then sends a message to the remote machine, which finds the needed page and sends it to the requesting process. The fault is then restarted and can now complete, which is achieved with the help of Inter Process Communication (IPC) library.

In essence, this design is similar to traditional virtual memory systems: when a process touches a nonresident page, a fault occurs and the operating system fetches the page and maps it in. The difference here is that instead of getting the page from the disk, the software gets it from another processor over the network.
To the user process, however, the system looks very much like a traditional multiprocessor, with multiple processes are free to read and write the shared memory at will. All communication and synchronization is done via the memory, with no communication visible to the user process.

The approach is not to share the entire address space, but only a selected portion of it, namely just those variables or data structures that needs to be used by more than one process. With respect to a distributed database system, and in this model, the shared variables represent the pages or tables within the shared memory segment.

One does not think of each machine as having direct access to an ordinary memory but rather, to a collection of shared variables, giving a higher level of abstraction. This approach greatly reduces the amount of data that must be shared, but in most cases, considerable information about the shared data is available, such as their types, which helps optimize the implementation.

Page-based distributed-shared memory takes a normal linear address space and allows the pages to migrate dynamically over the network on demand. Processes can access all of memory using normal read and write instructions and are not aware of when page faults or network transfers occur. Accesses to remote data are detected and protected by the memory management unit.
In order to facilitate optimization, the shared variables or tables are replicated on multiple machines. Potentially, reads can be done locally without any network traffic, and writes are done using a multicopy update protocol. This protocol is widely used in distributed database system.

The main purpose of this simulation is to discuss the issues in a Distributed Database System and how they can be overcome with the help of a Page Based Distributed Shared Memory System. In a Distributed Database System, multiple clients can read data from or write data to the database. The main issue in such a type of a system is achieving Consistency. Consistency is defined as “Any read to a location within the database returns the value stored by the most recent write operation to that location within the database” [7]. Since multiple clients are trying to perform various operations on the same database, it is difficult to ensure that the requesting client gets the most recent copy of the database.
1. **Introduction:**

Figure 1.1 illustrates the concept of Page Based Distributed Shared Memory in Distributed Database Systems.

![Figure 1.1 "Page Based Distributed Shared Memory"](image)

As shown in the figure above, the server machine or workstation contains a shared memory segment, which is distributed across the network. The shared memory segment contains a database that is divided into pages and hence the name ‘Page Based Distributed Shared Memory’ where each page represents a table within the database. Clients C1, C2 and C3 can perform any operation such as read or write on this database.
2. **Server Architecture:**

The server, who is the heart of this 'Page Based Distributed Shared Memory' software, is divided into various modules or managers to overcome the issues of achieving consistency.

![Page Manager Architecture](image)

Figure 2.1 "Page Manager Architecture"

Figure 2.1 illustrates the architecture of the server, so called the 'Page Manager'.

- **Page Consistency Protocol Manager**

  The 'Page Consistency Protocol Manager' works on the principle of 'Invalidate Protocol' to achieve consistency. Invalidate Protocol states that "In the distributed shared memory system, in case of replicated read-write pages, when a client tried to read a remote page, a local copy is made in order reduce network traffic. As long as all references are reads, everything is fine."
However, if any client attempts to write to a replicated page, a potential consistency problem arises because changing one copy and leaving others alone is unacceptable. Hence when a client on the network sees that one of its page is being updated, it invalidates the page effectively removing it from cache. The final result is that only one cache now holds the modified word, so consistency problems are avoided" [9].

The various approaches to achieve this, advantages, disadvantages, and limitations of each are discussed in more detail in chapter 4.

- **Page Consistency Model Manager.**

  The ‘Page Consistency Model Manager’ works on the principle of ‘Sequential Consistency’ to achieve consistency. Sequential Consistency states that “Any read to a memory location with in the distributed shared memory returns the value stored by the most recent write operation to that memory location” [9]. The various approaches to achieve this, advantages, disadvantages, and limitations of each are discussed in more detail in chapter 5.

- **Page Ownership Manager.**

  For an operation on any page located on the shared memory segment, there has to be one and only one client who owns the ownership for that page in read or write mode. Other client’s will have a copy of the same page in read only mode. The client decides the mode of ownership (read or write). It is important that for any operation by a client on a particular page, either the requesting client needs to be the owner or there has to some other client machine or workstation on the local area network which is the owner of that page.
If not, the requesting client is granted the ownership for that page. Since this module is responsible for managing the page ownership rights, it is called as the 'Page Ownership Manager'. The Page Ownership Manager is not only responsible for transferring the ownership rights from one client to another, but also knows the recent owner for every page.

The basis, on which ownership is transferred, various approaches to achieve this, advantages, disadvantages and limitations of each are discussed in more detail in chapter 6.

- Page Replication Manager.

Every page within the shared memory segment represents a table within the distributed database. When a client requests an operation of a specific page, the client stores a copy of that page in its local cache to reduce the network traffic. Since the client caches a replica of the page, located on a shared memory segment, it is called as 'Page Replication' and hence the name 'Page Replication Manager'.

The Page Replication Manager is responsible to ensure that the page requested by a client is located on a shared memory segment, if so the client is given a replica of that page in Read/Write mode, as requested by the client, otherwise the client's request is denied.

The various approaches to achieve this, advantages, disadvantages, and limitations of each are discussed in more detail in chapter 7.
• Page Copyset Manager.

In distributed databases, when multiple clients are trying to read from and write to a specific table, it is highly important to keep track of the clients which also have a local copy of that page in their cache. Before any client tries to write data to the page, all other clients should delete their copy from the local cache in order to achieve consistency.

Copyset is defined as a set of page or table names to which the client has a copy in its local cache. The Copyset also indicates the tables to which the client is an owner and the tables to which the client has a copy in read-only mode. Since this module is responsible for managing the Copyset for each client machine or workstation on the local area network, it is called as ‘Page Copyset Manager’ as each page on a shared memory segment represents a table.

The various approaches to achieve this, the advantages, disadvantages and limitations of these approaches are discussed in more detail in chapter 8.
• Page Synchronization Manager:

![Diagram of Page Synchronization Manager Architecture](image)

Figure 2.2 'Page Synchronization Manager Architecture'

In a Distributed Database System, as multiple clients are trying to perform various operations on a common shared memory segment, in order to achieve consistency, it is necessary for every client to synchronize their actions. The shared memory segment is also called the 'Critical Region', and the processes that monitor the activities on this critical region are called 'Monitors'. Figure 2.2 shows the architecture of the 'Page Synchronization Manager'. The 'Page Synchronization Manager' consists of a Monitor, which constantly monitors the shared memory segment and is responsible to ensure that there is only one client at any given point of time which is performing any operation on the requested page. The various approaches to achieve this, the advantages, disadvantages and limitations of these approaches are discussed in more detail in chapter 9.
• Page Message Queue Manager:

In a distributed application, when there are multiple clients requesting various operations from the server, it is necessary for the server to service every client’s request. Since the server is busy servicing a client’s request, there is a high possibility that the new incoming client’s request may be lost. In order to overcome this limitation, the ‘Page Message Queue Manager’ implements a First In First Out (FIFO) Queue Message System for every page located on the shared memory segment.

For every request of an operation on a specific page located within the shared memory segment, the ‘Page Message Queue Manager’ spawns a new child thread [8] for that page, if a child process doesn’t exist. This child process is responsible for any operation only for the page for which it has been created.

The ‘Page Message Queue Manager’ receives the incoming request from the client for the page specified, and stores it in the queue of the existing child process responsible for the page requested and starts listening for new incoming requests. The child process constantly monitors the queue and satisfies the incoming client’s request. Hence the ‘Page Message Queue Manager’ ensures that no incoming client’s request is lost even though the child process is busy servicing another client’s request.
The various approaches to achieve this, the advantages, disadvantages and limitations of these approaches are discussed in more detail in chapter 10.

A 'Page Manager' is required with most of the commercial distributed databases available today, such as Oracle, SQL Server, Informix, Sybase et cetera. For simulation purposes, files are used to represent tables within the distributed database, which resides in the shared memory segment on the server. Both the client and server workstations use the local disk space as a medium of storage.
3. Distributed Shared Memory:

A simple but practical way to build a distributed shared memory is to base it on a network to which more than one node is connected. Figure 3.1 illustrates a system with three nodes and a memory shared among all of them. When any of the node wants to read a word from the memory, it puts the address of the word it wants on the network and asserts a signal indicating that it wants to read. When the memory has fetched the requested word, it puts the word on the network and asserts another signal to announce that it is ready. The node then reads in the word. Writes work in an analogous way.

![Figure 3.1 A Distributed Shared Memory](image)

To prevent two or more nodes from trying to access the memory at the same time, arbitration mechanisms are used. These following arbitration mechanisms are as described below:

3.1 First Come First Serve Approach[^1]:

A node might first need to request it by asserting a signal. Only after receiving permission would it be allowed to use the network. The granting of this permission can be done in a centralized way, using an arbitration device, or in a decentralized way, with the first requesting node along the network winning any conflict.

The disadvantage of having the 'First Come First Server' approach is that with as few as three or four nodes, the network is likely to become overloaded.
3.2 Snooping Cache Approach\textsuperscript{[12]}:

The usual approach taken to reduce the network load is to equip each node with a \textit{"snooping cache"} \textsuperscript{[12]}, so called because it "snoops" on the network. Caches are shown in Figure 3.2. Snooping is defined as the mechanism by which the node constantly monitors the network.

![Diagram of distributed shared memory with caching]

Figure 3.2 A Distributed Shared Memory with caching

It has three important properties:

1. Consistency is achieved by having all the caches do network snooping.

2. The protocol to achieve consistency is built into the memory management unit or the Distributed shared memory software.

3. The entire algorithm is performed in well under a memory cycle.
4. **Page Consistency Protocol Manager:**

If pages were not replicated, achieving consistency would not have been an issue. There would have been exactly one copy of each page, and it is moved back and forth dynamically as needed. With only one copy of each page, there is no danger that different copies will have different values. If read-only pages are replicated, there is also no problem. The read-only pages are never changed, so all the copies are always identical. Only a single copy is kept of each read-write page, so inconsistencies are avoided.

In the distributed shared memory system, in case of replicated read-write pages, when a process tries to read a remote page, a local copy is made because the system does not know what is on the page or whether it is writeable. Both the local copy and the original page are set up in their respective memory management unit as read only. As long as all references are reads, everything is fine.

However, if any process attempts to write on a replicated page, a potential consistency problem arises because changing one copy and leaving others alone is unacceptable. The Page Consistency Protocol Manager can use various protocols to tackle this problem. The protocols described below follow the least used to most preferred pattern.

4.1 **Write Through Cache Consistency Protocol**[^6]:

One particular simple and common protocol is called “write through”[^6]. When a node first reads a word from memory, that word is fetched over the network and is stored in the cache of the node making the request. If that word is needed again later, the node can take it from the cache without making a request over the network, thus reducing network traffic.
These two cases, read miss (word not cached) and read hit (word cached) are shown in Table 4.1 as the first two lines in the table. In simple systems, only the word requested is cached, but in most, a block of words, is transferred and cached on the initial access and kept for possible future use.

<table>
<thead>
<tr>
<th>Event</th>
<th>Action taken by a cache in response to its own CPU’s operation</th>
<th>Action taken by a cache in response to a remote CPU’s operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Miss</td>
<td>Fetch data from memory and store in cache</td>
<td>(No action)</td>
</tr>
<tr>
<td>Read Hit</td>
<td>Fetch data from Local cache</td>
<td>(No action)</td>
</tr>
<tr>
<td>Write Miss</td>
<td>Update data in memory and store in cache</td>
<td>(No action)</td>
</tr>
<tr>
<td>Write Hit</td>
<td>Update memory and cache</td>
<td>Invalidate cache entry</td>
</tr>
</tbody>
</table>

*Table 4.1: The write-through cache consistency protocol. The entries for hit in the third column mean that the snooping Node has the word in its cache, not that the requesting Node has it.*

The first column lists four basic events that can happen. The second one tells what a cache does in response to its own node’s actions. The third one tells what happens when a cache sees (by snooping) that a different node has had a hit or miss.

The only time cache S (the snooper) must do something is when it sees that another node has written a word that S has cached (a write hit from S’s point of view). The action is for S to delete the word from its cache.
Each node does its caching independent of the others. Consequently, it is possible for a particular word to be cached at two or more nodes at the same time. When a write is done, if no node has the word being written in its cache, the memory is just updated, as if caching were not being used. If the node doing the write has the only copy of the word, its cache is updated and memory is updated over the bus as well.

The advantage of write-through protocol is that, it is simple to understand and implement but has the serious disadvantage that all writes use the network. While the protocol certainly reduces network traffic to some extent, the number of nodes that can attached to a network is still too small to permit large-scale distributed shared memory to be built using it.

4.2 Write Update Protocol[6]:

In a 'Write Update Protocol', when a node wants to write a word that two or more nodes have in their caches, if the word is currently in the cache of the node doing the write, the cache entry is updated. It is also written to the network to update memory. All the other caches see the write (because they are snooping on the bus) and check to see if they are also holding the word being modified. If so, they update their cache entries, so that after the write completes, memory is up-to-date. This is commonly known as the “write update protocol” [6].

With update, the write is allowed to take place locally, but the address of the modified word and its new value are broadcast on the network simultaneously to all the other nodes on the network. Each of its caches holding the word being updated sees that an address it is caching is being modified, so it copies the new value from the network to its cache, overwriting the old value. The final result is that all caches that held the word before its update also hold it afterward, and all acquire the new value.
The disadvantage of this protocol is that updating is slower in most cases, because updating needs to provide the new cache entry as well as the address to be updated. If these two items must be presented on the network consecutively, network load is increased.

4.3 Write Invalidate Protocol [6]:

In a 'Write Invalidate Protocol', when a node wants to write a word that two or more nodes have in their caches, if the word is currently in the cache of the node doing the write, the cache entry is updated. It is also written to the network to update memory.

All the other caches see the write (because they are snooping on the bus) and check to see if they are also holding the word being modified. If so, they invalidate their cache entries, so that after the write completes, memory is up-to-date and only one machine has the word in its cache. This is commonly known as the "write invalidate protocol" [6].

With invalidation, the address of the word being updated is broadcast on the network, but the new value is not. When a node on the network sees that one of its words is being updated, it invalidates the cache block containing the word, effectively removing it from the cache.

The final result with invalidation is that only one cache now holds the modified word, so consistency problems are avoided. If one of the node on the network that now holds an invalid copy of the cache block tries to use it, it will get a cache miss and fetch the block from the one processor holding a valid copy.

If a node using the word could somehow be given temporary ownership of the word, it could avoid having to update memory on subsequent writes until a different node exhibited interest in the word.
This protocol manages cache blocks, each of which can be in one of the following states:

1. **INVALID** - This cache block does not contain valid data.

2. **CLEAN** - Memory is up-to-date; the block may be in other caches.

3. **DIRTY** - Memory is incorrect; no other cache holds the block.

The basic idea is that a word that is being read by multiple nodes is allowed to be present in all their caches. A word that is being heavily written by only one machine is kept in its cache and not written back to memory on every write to reduce network traffic. The operation of the protocol is as explained below:

For simulation purposes, each cache block consists of a single page, B has a cached copy of the word at address W, as illustrated in Figure 4.2.1. The value is W1. The memory also has a valid copy. In Figure 4.2.2, A requests and gets a copy of W from the memory. Although B sees the read request go by, it does not respond to it.
Memory

A reads word W and gets W1. B does not respond to the read, but the memory does.

Now A writes a new value, W2 to W. B sees the write request and responds by invalidating its cache entry. A's state is changed to DIRTY, as shown in Figure 4.2.3. The DIRTY State means that A has the only cached copy of W and that memory is out-of-date for W.

At this point, A overwrites the word again, as shown in Figure 4.2.4. The write is done locally, in the cache, with no network traffic. All subsequent writes also avoid updating memory.
Sooner or later, some other node, C as shown in Figure 4.2.5, accesses the word. A sees the request on the network and asserts a signal that inhibits memory from responding. Instead, A provides the needed word and invalidates its own entry. C sees that the word is coming from another cache, not from memory, and that it is in DIRTY State, so it marks the entry accordingly. C is now the owner, which means that it has the responsibility of watching out for other nodes that request the word, and servicing them itself. The word remains in DIRTY State until it is purged from the cache it is currently residing in for lack of space. At that time it disappears from all caches and is written back to memory.

**Figure 4.2.1 – 4.2.5: An example of how a cache ownership protocol works.**
If Node A accesses the word, Node C sees the request on the network and asserts a signal that inhibits memory from responding. Instead, C provides the needed word and invalidates its own entry. A sees that the word is coming from another cache, not from memory, and that it is in DIRTY state, so it marks the entry accordingly. A is now the owner, which means that it has the responsibility of watching out for other nodes that request the word and servicing them itself. The word remains in DIRTY State until it is purged from the cache it is currently residing in for lack of space. At that time it disappears from all caches and is written back to memory. This protocol is also called “cache consistency protocol” [1].

In a distributed shared memory system, the software does not know which word is to be written or what the new value will be. To find out, it could make a secret copy of the page about to be changed, make the page writable and broadcast a short packet giving the address and new value on the network. The nodes receiving this packet could then check to see if they have the page in question, and if so, update it.

The amount of work here is enormous, but worse yet, the scheme is also not foolproof. If several updates, originating on different processors, take place simultaneously, different nodes may see them in a different order, so the memory will not be sequentially consistent.

Another issue here is that a node may make thousands of consecutive writes to the same page because many programs exhibit locality of reference. Having to cache all these updates and pass them to remote machines is an expensive overload.
The advantage of Invalidate protocol is that invalidating requires supplying just the address to be invalidated and hence the network load is reduced. Secondly invalidating is faster than updating as network load is reduced. For these reasons, the ‘Page Consistency Protocol Manager’ uses an invalidation protocol instead of an update protocol.

4.4 Implementation of Page Consistency Protocol Manager:

The ‘Page Consistency Protocol Manager’ works on the principle of ‘Invalidation Protocol. The ‘Page Consistency Protocol Manager’ works in co-ordination with ‘Page Copyset Manager’. The ‘Page Copyset Manager’ implements a self-referential and nested structure as a medium of data storage to store all the information about a page such as the owner of a page along with its mode (Read/Write) and also the list of clients or workstations who have a copy of the page in Read Only mode. Self-Referential and nested structures were used as a medium of data storage because Inter Process Communication Library was used in the implementation of this project. Inter Process Communication Library has the drawback of not being able to support marshalling and unmarshalling of objects.

When ever a client wants to perform a write operation on any page, the ‘Page Consistency Manager’ checks the status of the page of find if the page is currently being used by another client or not. If the page is currently being used, the client’s request is stored in a First In First Out message queue implemented by the ‘Page Message Queue Manager’. As soon as all the pending operations on the requesting page is completed, the client who wants to perform a write operation is granted service.
The 'Page Consistency Protocol Manager' uses the copyset of the requested page from the 'Page Copyset Manager'. The 'Page Consistency Manager' sends an invalidate message to all the clients in the copyset who has a copy of the page other than the requested client. As soon as the client receives an invalidate message from the page manager, the client deletes the copy of the requested page from its cache and sends an acknowledgement message to the 'Page Consistency Protocol Manager'. The 'Page Consistency Protocol Manager' waits for an acknowledgement from all the clients. The moment it receives an acknowledgement from the client, it compares the client name with the copyset and deletes the client information from the copyset.

Once it has received an acknowledgement from all the clients, the ownership is transferred to the requesting client. The 'Page Consistency Protocol Manager' informs the 'Page Copyset Manager' of the new updates and the copyset for the requested page is updated accordingly by the 'Page Copyset Manager'. The 'Page Consistency Protocol Manager' then sends a grant message to the requesting client.
5. **Page Consistency Model Manager:**

In a distributed shared memory system, there are one or more copies of each read-only page and one copy of each writeable page. When a remote machine references a writeable page, a page fault occurs and the page is fetched. However, if some writeable pages are heavily shared, having only a single copy of each one results in a serious performance bottleneck.

Allowing multiple copies eases the performance problem, since it is then sufficient to update any copy, but doing so introduces a new problem: how to keep all the copies consistent. Maintaining perfect consistency is especially painful when the various copies are on different machines that can only communicate by sending messages over a slow network. Hence a consistency model is essentially a contract between the software and the memory. If the software agrees to obey certain rules, the memory promises to work correctly. If the software violates these rules, the correctness of memory operation is no longer guaranteed.

The Page Consistency Model Manager can use various consistency models in order to achieve consistency. The consistency models described below follow the strictest to the least strict pattern.
5.1 Strict Consistency [6]:

The most stringent consistency model is called strict consistency. It is defined by the following condition: *Any read to a memory location x returns the value stored by the most recent write operation to x.*

![Figure 5.1.1](image1.png)  
![Figure 5.1.2](image2.png)

*Behavior of two processes. The horizontal axis is time. Figure 5.1.1: Strictly consistent memory. Figure 5.1.2: Memory that is not strictly consistent.*

For example: P1 and P2 are processes at different heights as shown in Figure 5.1.1. The operations done by each process are shown horizontally, with time increasing to the right. Straight line separates the processes. The symbols W (x) a and R (y) b mean that a write to x with the value a and read from y returning b have been done respectively. The initial value of all variables is assumed to be 0. In Figure 5.1.1, P1 does a write to location x, storing the value 1. Later, P2 reads x and sees the 1. This behavior is correct for a strict consistent memory.

In contrast, in Figure 5.1.2, P2 does a read after the write and gets 0. A subsequent read gives 1. Such behavior is incorrect for a strict consistent memory. When a memory is strictly consistent, all writes are instantaneously visible to all processes and an absolute global time order is maintained. If a memory location is changed, all subsequent reads from that location see the new value, no matter how soon after the change the reads are done and no matter which processes are doing the reading and where they are located. Similarly, if a read is done, it gets the then - current value, no matter how quickly the next write is done.
5.2 Sequential Consistency \cite{6}:

While strict consistency is the ideal programming model, it is nearly impossible to implement in a distributed system. Sequential consistency is a slightly weaker model than strict consistency. A sequentially consistent memory is one that satisfies the following condition:

*The result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

When processes run in parallel on different machines or on a timesharing system, any valid interleaving sequence of read or writes is acceptable behavior, but all processes must see the same sequence of memory references. A memory in which one process (or processor) sees one interleaving and another process sees a different one is not a sequentially consistent memory. The time does not play a role as shown in Figure 5.2.1. A memory behaving as shown in Figure 5.2.1 is sequentially consistent even though the first read done by P2 returns the initial value of 0 instead of the new value of 1.

<table>
<thead>
<tr>
<th>P1: W(x)1</th>
<th>P1: W(x)1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2: R(x)0</td>
<td>R(x)1</td>
</tr>
<tr>
<td>P2: R(x)1</td>
<td>R(x)0</td>
</tr>
</tbody>
</table>

*Figure 5.2.1 – 5.2.2: Two possible results of running the same program.*
Sequentially consistent memory does not guarantee that a read returns the value written by another process a nanosecond earlier, or a microsecond earlier, or even a minute earlier. It merely guarantees that all processes see all memory references in the same order.

If the program that generated Figure 5.2.1 is run again, it might give the result of Figure 5.2.2. The results are not deterministic. Running a program again may not give the same result in the absence of explicit synchronization operations. A program that works for some of the results and not for others violates the contract with the memory and is incorrect.

A sequentially consistent memory can be implemented on a distributed shared memory that replicates writable pages by ensuring that no memory operation is started until all previous ones have been completed. An efficient, totally-ordered reliable broadcast mechanism states that all shared variables could be grouped together on one or more pages, and operation to the shared pages could be broadcast since the exact order in which the operations are interleaved does not matter as long as all processes agree on the order of all operations on the shared memory.

\[
P1: \quad W(x)1
\]
\[
P2: \quad R(x)0 \quad R(x)1
\]

Figure 5.2.3

For example: A sequence of read and writes operations of process \( i \) is designated by \( H_i \) (the history of \( P_i \)). Figure 5.2.3 shows two such sequences. \( H_1 \) and \( H_2 \) for \( P_1 \) and \( P_2 \), respectively, as follows:
The set of all such sequences is called H. To get the relative order in which the operations appears to be executed, the operation strings in H are merged into a single string, S, in which each operation appears in S once. Intuitively, S gives the order that the operations would be carried out had there been a single centralized memory. All legal values for S must obey two constraints:

1. Program order must be maintained.

2. Memory coherence must be respected.

The first constraint means that if a read or write access, A, appears before another access, B, in one of the strings in H, A must also appear before B in S. If this constraint is true for all pairs of operations, the resulting S will not show any operations in an order that violates any of the programs.

The second constraint, called memory coherence, means that a read to some location, x, must always return the value most recently written to x; that is, the value v written by the most recent W (x) v before the R (x). Memory coherence examines in isolation each location and the sequence of operations on it, without regard to other locations. Consistency, in contrast, deals with writes to different locations and their ordering.

There is only one legal value of S for this example: S = R (x) 0 W (x) 1 R (x) 1,

The behavior of a program is said to be correct if its operation sequence corresponds to some legal value of S.
The casual consistency represents a weakening of sequential consistency in that it makes a distinction between events that are potentially causally related and those that are not.

For example: Suppose that process P1 writes a variable x. Then P2 reads x and writes y. Here the reading of x and the writing of y are potentially causally related because the computation of y may have depended on the value of x read by P2 (i.e., the value written by P1). On the other hand, if two processes spontaneously write two variables, these are not casually related. When there is a read followed later by a write, the two events are potentially causally related. Similarly, a read is casually related to the write that provided the data the read got. Operations that are not causally related are said to be concurrent.

For a memory to considered causally consistent, it is necessary that the memory obey the following condition:

All processes in the same order must see writes that are potentially casually related. Concurrent writes may be seen in a different order on different machines.

\[
\begin{array}{c|c|c|c}
 & W(x)1 & W(x)3 \\
P1: & R(x)1 & W(x)2 \\
P2: & R(x)1 & R(x)3 & R(x)2 \\
P3: & R(x)1 & R(x)2 & R(x)3 \\
P4: & & & \\
\end{array}
\]

*Figure 5.3.1: This sequence is allowed with causally consistent memory, but not with sequentially consistent memory or strictly consistent memory.*
Let's consider an example as shown in Figure 5.3.1. There is an event sequence that is allowed with a causally consistent memory, but which is forbidden with a sequentially consistent memory or a strictly consistent memory. The writes \( W(x)_2 \) and \( W(x)_3 \) are concurrent since they are not casually related and there is no read followed immediately by a write of \( W(x)_2 \), so it is not required that all processes see them in the same order. If the software fails when different processes see concurrent events in a different order, it has violated the memory contract offered by causal memory.

In Figure 5.3.2, \( W(x)_2 \) potentially depends on \( W(x)_1 \) because the two may be a result of a computation involving the value read by \( R(x)_1 \). The two writes are casually related, so all processes must see them in the same order. Therefore it is incorrect.

In Figure 5.3.3, the read has been removed, so \( W(x)_1 \) and \( W(x)_2 \) are now concurrent writes. Casual memory does not require concurrent write to be globally ordered, so figure 5.3.3 is correct. Implementing casual consistency requires keeping track of which processes have seen the writes. It effectively means that a dependency graph of which operation is dependent on which other operation must be constructed and maintained which involves some overhead.
5.4 PRAM Consistency and Processor Consistency:

In casual consistency, it is permitted that concurrent writes be seen in a different order on different machines, although casually related ones must be seen in the same order by all machines. PRAM Consistency is subjected to the condition:

*W*rites done by a single process are received by all processes in the same order in which they were issued, but writes from different processes may be seen in a different order by different processes.

![Figure 5.4: A valid sequence of events for PRAM consistency.](image)

PRAM stands for Pipelined RAM, because writes by a single process can be pipelined, that is, the process does not have to stall waiting for each one to complete before starting the next one. PRAM consistency is contrasted with casual consistency in the Figure 5.4 above. The sequence of events shown here is allowed with PRAM consistent memory but not with any of the models discussed above.

PRAM consistency is easy to implement. In effect, there are no guarantees about the order in which different processes see writes, except that two or more writes from a single source must arrive in order, as though they were in a pipeline. In this model all writes generated by different processes are concurrent.
Processor Consistency is similar to PRAM consistency. In Processor consistency, there is an additional condition imposed on Processor consistent memory, namely called memory coherence. For every memory location, x, there be global agreement about the order of writes to x.
5.5 Weak Consistency [6]:

Although PRAM consistency and processor consistency can give a better performance than the stronger models, they are still unnecessary restrictive for many applications because they require that writes originating in a single process be seen everywhere in order. Not all applications require even seeing all writes, let alone seeing them in order. ‘Critical Section’ is defined as a common shared memory segment to which multiple processes are trying to perform a read or write operation. For example, a process inside a critical section reading and writing some variables in a tight loop. Even though other processes are not supposed to touch the variables until the first process has left its critical section, the memory has no way of knowing when a process is in a critical section and when it is not, so it has to propagate all writes to all memories in the usual way.

The ideal solution would be to let the process finish its critical section and then make sure that the final results were sent everywhere, not worrying too much whether all intermediate results has also been propagated to all memories in order, or even at all. Introducing a new kind of variable, a synchronization variable, which is used for synchronization purposes, can do this. The operations on it are used to synchronize memory. When synchronization completes, net effect of all writes done on that machine is propagated outward and net effect of all writes done on other machines are bought in. In other words, all of shared memory is synchronized.

*Weak consistency has three properties:*

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

The first point states that all processes see all accesses to synchronization variables in the same order. Effectively, when a synchronization variable is accessed, this fact is broadcast to the world, and no other synchronization variable can be accessed by any other process until the synchronization variable currently being accessed is released.

The second point states that accessing a synchronization variable flushes the pipeline. When the synchronization access is done, all previous writes are guaranteed to be done as well. By doing a synchronization after updating shared data, a process can force the new values out to all other memories.

The third point states that when ordinary (i.e., not synchronization) variables are accesses, either for reading or writing, all previous synchronization has been performed. By doing a synchronization before reading shared data, a process can be sure of getting the most recent values.

A read is said to have been performed when the process has read the value written by the most recent write. A write is said to have been performed at the instant when all subsequent reads return the value written by the most recent write. A synchronization is said to have been performed when all shared variables have been updated.

From an implementation standpoint, when the contract between the software and the memory says that memory only has to be brought up to date when a synchronization variable is accessed, a new write can be started before the previous ones have been completed, and in some cases writes can be avoided altogether. Of course, this contract puts a greater burden on the programmer, but the potential gain is better performance.
Compared to the other memory models discussed above, this model enforces consistency on a group of operations, not on individual reads and writes. This model is most useful when isolated accesses to shared variables are rare, with most coming in clusters (many accesses in a short period, then none for a long time).

\begin{figure}[h]
\centering
\begin{tabular}{c}
\text{P1:} \\
W(x)1 & W(x)2 & S \\
R(x)1 & R(x)2 & S \\
\text{P2:} \\
\end{tabular}
\caption{Figure (5.5.1): A valid sequence of events for weak consistency.}
\end{figure}

\begin{figure}[h]
\centering
\begin{tabular}{c}
\text{P1:} \\
W(x)1 & W(x)2 & S \\
S & R(x)1 \\
\text{P2:} \\
\end{tabular}
\caption{Figure (5.5.2): An invalid sequence for weak consistency.}
\end{figure}

For example: As shown in Figure 5.5.1, process P1 does two writes to an ordinary variable, and then synchronizes (indicated by the letter S). If P2 has yet not been synchronized, no guarantees are given about what they see, so this sequence of events is valid. Figure 5.5.2 is different. Here P2 has been synchronized, which means that its memory is bought up to date. When it reads x, it must get the value 2. Getting 1, as shown in the Figure 5.5.2, is not permitted with weak consistency.
Weak consistency has the problem that when a synchronization variable is accessed, the memory does not know whether this is being done because the process is finished writing the shared variables or about to start reading them. Consequently, it must take the actions required in both cases, namely making sure that all locally initiated writes have been completed (i.e., propagated to all other machines), as well as gathering in all writes from other machines. If the memory could tell the difference between entering a critical region and leaving one, a more efficient implementation might be possible. To provide this information, two kinds of synchronization variables or operations are needed instead of one.

Release consistency provides these two kinds. Acquire accesses are used to tell the memory system that a critical region is about to be entered. Release accesses say that a critical region has just been exited. These accesses can be implemented either as ordinary operations on special variables or as special operations. In either case, the programmer is responsible for putting explicit code in the program telling when to do them, for example, by calling library procedures such as acquire and release or procedures such as enter_critical_region or leave_critical_region.

It is also possible to use barriers instead of critical regions with release consistency. A barrier is a synchronization mechanism that prevents any process from starting phase n+1 of a program until all processes have finished phase n. When a process arrives at a barrier, it must wait until all processes get there too. When the last one arrives, all shared variables are synchronized and then all process is resumed. Departure from the barrier is acquire and arrival is release.
In addition to these synchronization accesses, reading and writing shared variables is also possible. Acquire and release does not have to apply to all of memory. Instead, they may only guard specific shared variables, in which case only those variables are kept consistent. The shared variables that are kept consistent are said to be protected.

The contract between the memory and the software says that when the software does an acquire, the memory will make sure that all the local copies of the protected variables are brought up to date to be consistent with the remote ones. When a release is done, protected variables that have been changed are propagated out to other machines. Doing an acquire does not guarantee that locally made changes will be sent to other machines immediately. Similarly, doing a release does not necessarily import changes from other machines.

In the context of distributed-shared memory, to do acquire process sends messages to synchronization manager requesting acquire on a particular lock for a shared variable. In no other process currently owns the lock for the requested shared variable, the requesting process is granted the lock and acquire completes. Then an arbitrary sequence of reads and writes to the shared variable take place locally.

None of these are propagated to other machines. When the release is done, the modified data are sent to the other machines that use them. After each machine has acknowledged receipt of the data, the synchronization manager is informed of the release. In this way, an arbitrary number of reads and writes on shared variables can be done with a fixed amount of overhead. Acquires and releases on different locks occur independently of one another.
Figure 5.6: A valid event sequence for release consistency.

Figure 5.6 depicts a valid sequence of events for release consistency. Process P1 does an acquire, changes a shared variable twice, and then does a release. Process P2 does an acquire, and reads x. It is guaranteed to get the value x had at the time of the release, namely 2 (unless P2’s acquire performs before P1’s acquire). If the acquire had been done before P1 did the release, the acquire would have been delayed until the release had occurred. Since P3 does not do an acquire before reading a shared variable, the memory has no obligation to give it the current value if x, so returning 1 is allowed.

A distributed-shared memory is release consistent if it obeys the following rules:

1. Before an ordinary access to a shared variable is performed, all previous acquires done by the process must have completed successfully.

2. Before a release is allowed to be performed, all previous reads and writes done by the process must have completed.

3. The acquire and release accesses must be processor consistent (sequential consistency is not required).
If all the above conditions are met and processes use acquire and release properly (i.e., in acquire - release pairs), the results of any execution will be no different than they would have been on a sequentially consistent memory. In effect, blocks of accesses to shared variables are made atomic by the acquire and release primitives to prevent interleaving.

A different implementation of release consistency is “lazy release consistency” \[6\]. Normal release consistency is also called “eager release consistency” \[6\]. In eager release consistency, when a release is done, the processor doing the release pushes out all the modified data to all other processors that already have a cached copy and thus might potentially need it. There is no way to tell if they actually will need it, so to be safe, all of them get everything that has changed.

Although pushing all the data out this way is straightforward, it is generally inefficient. In lazy release consistency, at the time of a release, nothing is sent anywhere. Instead, when an acquire is done, the processor trying to do the acquire has to get the most recent values of the variables from the machine or machines holding them. A timestamp protocol \[5\] can be used to determine which variables have to be transferred.

In many programs, a critical region is located inside a loop. With eager release consistency, one each pass through the loop a release is done, and all the modified data have to be pushed out to all the processors maintaining copies of them. This algorithm wastes bandwidth and introduces needless delay. With lazy release consistency, at the release nothing is done. At the next acquire, the processor determines that it already has all the data it needs, so no messages are generated here either. The net result is that with lazy release consistency no network traffic is generated at all until another processor does an acquire. Repeated acquire-release pairs done by the same processor in the absence of competition from the outside are free.
5.7 Entry Consistency [6]:

In lazy release consistency and eager release consistency, it requires the programmer to use acquire and release at the start and end of each critical section, respectively. However, unlike release consistency, entry consistency requires each ordinary shared variable to be associated with some synchronization variable such as a lock or barrier. If it is desired that elements of an array be accessed independently in parallel, then different array elements must be associated with different locks.

When an acquire is done on a synchronization variable, only those ordinary shared variables guarded by that synchronization variable are made consistent. Entry consistency differs from lazy release consistency in that the latter does not associate shared variables with locks or barriers and at acquire time has to determine empirically which variable it needs.

Associating with each synchronization variable a list of shared variables reduces the overhead associated with acquiring and releasing a synchronization variable, since only a few shared variables have to be synchronized. It also allows multiple critical sections involving disjoint-shared variables to execute simultaneously, increasing the amount of parallelism. The disadvantage is extra overhead and complexity of associating every shared data variable with some synchronization variable. Programming in this manner is more complicated and error prone.

Each synchronization variable has a current owner, namely, the process that last acquired it. The owner may enter and exit critical regions repeatedly without having to send any messages on the network. A process not currently owning a synchronization variable but wanting to acquire it must sent a message to the current owner asking for ownership and the current values of the associated variables.
Nonexclusive Mode means that processes have a copy of the shared variable in Read Only Mode, whereas Exclusive Mode means that the owner has a copy of the shared variable in Read/Write mode. It is also possible for several processes, simultaneously to own a synchronization variable in nonexclusive mode, meaning that they can read, but not write, the associated data variable.

A memory exhibits entry consistency if it meets all the following conditions:

1. An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.

2. Before an exclusive mode access to synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.

3. After an exclusive mode access to a synchronization variable has been performed, any other processes next nonexclusive mode access to that synchronization variable may not be performed until it has checked with the owner of the synchronization variable.

The first condition says that when a process does an acquire, the acquire may not complete (i.e., return control to the next statement) until all the guarded shared variables have been brought up to date. In other words, at acquire, all remote changes to the guarded data must be made visible.

The second condition says that before updating a shared variable, a process must enter a critical region in exclusive mode to make sure that no other process is trying to update it at the same time.
The third condition says that if a process wants to enter a critical region in nonexclusive mode, it must first check with the owner of the synchronization variable guarding the critical region to fetch the most recent copies of the guarded shared variables.
5.8 Summary of Consistency Models [6]:

Consistency models differ in how restrictive they are, how complex their implementations are, their ease of programming, and their performance. Strict consistency is the most restrictive, but because its implementation in a distributed shared memory system is essentially impossible, it is never used.

Sequential consistency is feasible, popular with programmers, and widely used. It has the problem of poor performance, however. The way to get around this result is to relax the consistently model. Some of the possibilities are shown in Table 5.8.1, roughly in the order of decreasing restrictiveness.

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict</td>
<td>Absolute time ordering of all shared accesses matters.</td>
</tr>
<tr>
<td>Sequential</td>
<td>All processes see all shared accesses in the same order.</td>
</tr>
<tr>
<td>Casual</td>
<td>All processes see all casually related shared accesses in the same order.</td>
</tr>
<tr>
<td>Processor</td>
<td>PRAM consistency + memory coherence.</td>
</tr>
<tr>
<td>PRAM</td>
<td>All processes see writes from each processor in the order they were issued. Writes from different processors may not always be seen in the same order.</td>
</tr>
</tbody>
</table>

Table 5.8.1: Consistency models not using synchronization operations.

| Weak        | Shared data can only be counted on to be consistent after a synchronization is done. |
| Release     | Shared data are made consistent when a critical region is exited. |
| Entry       | Shared data pertaining to a critical region are made consistent when a critical region is entered. |

Table 5.8.2: Models with synchronization operations.
Casual consistency, processor consistency, and PRAM consistency all represent weakening in which there is no longer a globally agreed upon view of which operations appeared in which order. Different processes may see different sequences of operations. These three differ in terms of which sequences are allowed and which are not, but in all cases, it is up to the programmer to avoid things that work only if memory is sequentially consistent.

A different approach is to introduce explicit synchronization variables, as weak consistency, release consistency, and entry consistency do. These three are summarized as shown in Table 5.8.2. When a process performs an operation on an ordinary shared variable; no guarantees are given about when they will be visible to other processes.

Only when a synchronization variable is accesses are changes propagated. The three models differ in how synchronization works, but in all cases a process can perform multiple reads and writes in a critical section without invoking any data transport. When the critical section has been completed, the final result is either propagated to the other processes or made ready for propagation when any other process makes a request.

In short, weak consistency, release consistency, and entry consistency require additional programming constructs that, when used as directed, allow programmers to pretend that memory is sequentially consistent, when in fact, it is not. In principle, these three models using explicit synchronization should be able to offer the best performance, but it is likely that different applications will give quite different results.

The 'Page Consistency Model Manager' works on the principle of Sequential Consistency.
5.9 Implementation of Page Consistency Model Manager:

The ‘Page Consistency Model Manager’ works on the principle of Sequential Consistency. In co-ordination to achieve consistency, the ‘Page Consistency Model Manager’ works in co-ordination with the ‘Page Synchronization Manager’ and the ‘Page Message Queue Manager’. Sequential Consistency Model states that all processes should see accesses to the shared memory segment in the same order. In order to satisfy this principle, the Page Consistency Model Manager ensures that all requests to a specific page in the shared memory segment are processed in the order in which they are received.

When a client makes a request for any operation on a specific page, the ‘Page Consistency Model Manager’ checks with the ‘Page Synchronization Manager’ to find out the status of the page. The ‘Page Synchronization Manager’ checks to see the status of the requested page, such as if another client is currently using the page or the requested page is free. If the requested page is currently being used by another process, the ‘Page Consistency Model Manager’ checks if already a ‘Page Message Queue Manager’ exists for the requested page. Every page has its own ‘Page Message Queue Manager’.

If a ‘Page Message Queue Manager’ doesn’t exist, the ‘Page Consistency Model Manager’ creates a new ‘Page Message Queue Manager’ for the requested page and the requesting client’s request is stored in the queue. The ‘Page Consistency Model Manager’ then starts listening for new incoming clients’ request. The ‘Page Message Queue Manager’ then checks the status of the requested page with the ‘Page Synchronization Manager’. The moment all the pending requests on that page is completed and the status of the requested page is marked as free, the ‘Page Message Queue Manager’ starts servicing the client’s request in the queue.
Even in the first place when the 'Page Consistency Model Manager' receives the incoming client's request on a specific page and the status of the requested page is marked as free, the 'Page Consistency Model Manager' checks to see if already a 'Page Message Queue Manager' exists for the requested page.

If a 'Page Message Queue Manager' doesn't exist, the 'Page Consistency Model Manager' creates a new 'Page Message Queue Manager' for the requested page and the requesting client's request is stored in the queue. The 'Page Consistency Model Manager' then starts listening for new incoming clients' request. The 'Page Message Queue Manager' starts servicing the incoming client's request.

The advantage from the implementation point of view is that, every page has its own 'Page Message Queue Manager. The 'Page Message Queue Manager' is responsible for any operation on the page for which it is created. Implementation is very easy in this approach and code is easy to develop and maintain because roles and responsibilities are well defined and the client's request is processed in the order in which they are received by which all client's see accesses to the shared memory segment in the same order.
6. Page Ownership Manager:

In ‘Write Invalidate Protocol’, at any instant of time, each page is either in R (read-only) or W (read and write) state. The state a page is in may change as execution progresses. Each page has an owner, namely the process that most recently wrote on the page. When a page is in W State, only one copy exists, mapped into the owner’s address space in read-write mode. When a page is in R state, the owner has a copy (mapped read only), but other processes may have copies too in read only mode.

The ‘Page Ownership Manager’ is responsible to find the owner of a specific page on the network. The following patterns mentioned below can be used to achieve this purpose. The patterns are described are in the least used to highly preferred order.

6.1 Broadcast Protocol: 

The simplest solution on the network is by doing a broadcast, asking the owner of the specified page to respond. Once the owner has been located this way, the protocol can proceed as above. An obvious optimization is not just to ask who the owner is, but also to tell whether the sender wants to read or write and say whether it needs a copy of the page. The owner can then send a single message transferring ownership and he page as well, if needed.

Broadcasting has the disadvantage of interrupting each processor, forcing it to inspect the request packet. For all the processors except the owner’s handling the interrupt is essentially wasted time. Secondly broadcasting may use up considerable network bandwidth, depending on the hardware.
Another approach is called the ‘Four Message Protocol’, in which one process is designated as the page manager which is a centralized manager. It is the job of the manager to keep track of who owns each page. When a process, P, wants to read a page it does not have or wants to write a page it does not own, it send a message to the page manager telling which operation it wants to perform and on which page.

The manager then send back a message telling who the owner is. P now contacts the owner to get the page and/or the ownership, as required. Four messages are needed in this protocol as shown in Figure 6.1.

![Diagram of Four Message Protocol]

Figure 6.1: Ownership location using a central manager. Four-message protocol.

An optimization of this ownership location protocol is overcome with the help of ‘Three Message Protocol’ discussed in more detail in 6.3
6.3 Three Message Protocol:

Three Messages are needed in this protocol as shown in Figure 6.2. In this approach, the page manager forwards the request directly to the owner, which then replies directly back to P, saving one message. The page manager uses the incoming requests not only to provide replies but also to keep track of changes in ownership. When a process says that it wants to write on a page, the manager records that process as the new owner.

![Diagram](file.png)

*Figure 6.2: Ownership location using a central manager. Three-message protocol.*

The problem with this approach of having a three-message protocol centralized page manager is the potentially heavy load on the page manager, handling all incoming requests. Having multiple page managers instead of just one can reduce this problem. Splitting the work over multiple managers introduces a new problem, however - finding the right manager.

A simple solution is to use the low-order bits of the page manager as an index into a table of managers. Thus with eight page managers, all pages that end with 000 are handled by manager 0, all pages that end with 001 are handled by manager 1 and so on.
A different mapping, for example by using a hash function, is also possible. Another possible algorithm is having each process keep track of the probable owner, which forwards them if ownership has changed. If ownership has changed several times, the request message will also have to be forwarded several times. At the start of the execution and every n times ownership changes, the location of the new owner should be broadcast, to allow all processors to update their tables of probable owners. The ‘Page Ownership Manager’ was implemented using the ‘Three Message Protocol’ with one centralized page manager.

6.4 Implementation of ‘Page Ownership Manager’:

The ‘Page Ownership Manager’ uses the ‘Three Message Protocol’ and decides the ownership on the basis of six patterns or case described below. In all the cases in the figures shown below, process P on processor 1 wants to read (marked as R) or write (marked as W) a page. The cases differ in terms of whether P is the owner, whether P has a copy, whether other processes have copies, and the current of the state of the page, as shown.

**Case 1:**

- Process P reads
  - Processor 1
    - Owner
    - P
    - W
  - Processor 2
    - Page

- Process P writes
  - Processor 1
    - Owner
    - P
    - W
  - Processor 2
    - Page

1. Do read
1. Do write
Case 2:

1. Do read

Case 3:

1. Do Read

1. Invalidate copies
2. Mark page as W
3. Do write

Case 4:

1. Do read

1. Ask for validation
2. Ask for ownership
3. Mark page as W
4. Do write
Case 5:

```
1. Ask for copy
2. Mark page as R
3. Do read
```

Case 6:

```
1. Ask for invalidation
2. Ask for ownership
3. Mark page as W
4. Do write
```

Figure 6.3.1

Figure (6.3.1) Process P wants to read a page.

Figure 6.3.2

Figure (6.3.2) Process P wants to write a page.

In case (1) through (4) as of Figure 6.3.1, P just does the read. In all four cases the page is mapped into its address space, so the read is done and no page fault occurs. In case (5) and (6), the page is not mapped in, so a page fault occurs and the software gets control.

The software then sends a message to the owner asking for a copy. When the copy comes back, the page is mapped in and it is restarted. If the owner had the page in W State, it must degrade to R State, but may keep the page. In this protocol, the other process keeps ownership, but in a slightly different case that is also transferred.
Writes are handled differently, as depicted in Figure 6.3.2. In the first case, the write just happens, without a fault, since the page is mapped in read-write mode. In the second case (no other copies), the page is changed to W State and written. In the third case there are other copies, so they must first be invalidated before the write can take place.

In case (4) through (6), some other process is the owner at the time P does the write. In all three cases, P must ask the current owner to invalidate any existing copies, pass ownership to P, and send a copy of the page unless P already has a copy. Only then may the write take place. In all three cases, P ends up with the only copy of the page, which is in W State.

In all case (1) through case (6), before a write is performed the protocol guarantees that only one copy of the page exists, namely in the address space of the process about to do the write.
7. Page Replication Manager:

The idea behind 'Page Based Distributed Shared Memory' is to emulate the cache of a multiprocessor using the memory management unit and the operating system software. In a distributed shared memory system, the address space is divided up into chunks, with the chunks being spread all over the processors in the system. When a processor references an address that is not local, a trap occurs and the distributed shared memory software fetches the chunk containing the address and restarts the faulting instruction, which now completes successfully. This concept is illustrated in Figure 7.1 as shown below for an address space with eleven chunks and two processors. The shared global address space is basically the shared memory segment, which is divided in the form of pages where each page represents a shared variable. With respect to a Distributed Database System, each page represents a table with in the distributed database that resides in the shared global address space.

![Diagram of address space distribution](image)

*Figure 7.1: Chunks of address space distributed among 2 machines.*
If processor 1 references instructions or data in chunks 0, 2, 5 or 9, the references are done locally. References to other chunks cause traps. For example, a reference to an address in chunk 10 will cause a trap to the distributed shared memory software, which then moves chunk 10 from machine 2 to machine 1, as shown in Figure 7.2.

![Network Diagram](image)

*Figure 7.2: Situation after Node 1 references chunks 10.*

One improvement to the system that can improve performance considerably is to replicate chunks that are read only. For example, if chunk 10 in figure 1 is a section of program text, its use by processor 1 can result in a copy being sent to processor 1, without the original in processor 2's memory being distributed, as shown in Figure 7.3. In this way, processors 1 and 2 can both reference chunk 10 as often as needed without causing traps to fetch missing memory.
Another possibility is to replicate not only read-only chunks, but also all chunks. As long as reads are being done, there is effectively no difference between replicating a read-only chunk and replicating a read-write chunk. However, if a replicated chunk is suddenly modified, special action is taken in order to prevent having multiple, inconsistent copies in existence.

7.1 Implementation of ‘Page replication Manager’:


Whenever client requests a copy of a page in read only mode, the ‘Page Replication Manager’ checks with the ‘Page Ownership Manager’ if a copy of the page can be provided to the requesting client. The ‘Page Ownership Manager’ performs all the validation check’s based on Case (1) through Case (6) as discussed in the ‘Page Ownership Manager’.
If the request is granted by the ‘Page Ownership Manager’, the ‘Page Replication Manager’ provides a copy of the requested page in read-only mode to the requesting client. Once the client has successfully received a copy of the page, the ‘Page Replication Manager’ informs the ‘Page Copyset Manager’ of the new changes. The ‘Page Copyset Manager’ adds the client name and the mode of the page (Read Only) in the copyset of the requested page.

Similarly, whenever client requests an ownership of a page, the ‘Page Replication Manager’ checks with the ‘Page Ownership Manager’ if ownership of the page can be provided to the requesting client. The ‘Page Ownership Manager’ performs all the validation check’s based on Case (1) through Case (6) as discussed in the ‘Page Ownership Manager’.

If there are currently any operations pending on the requested page or there already exists an owner for the requested page, the ownership is not granted by the ‘Page Ownership Manager’. The incoming request is forwarded to the ‘Page Consistency Model Manager’.

The ‘Page Consistency Model Manager’ creates a new ‘Page Message Queue Manager’ for the requested page if there doesn’t exist one and adds the incoming client’s request in the queue for the page requested else adds the incoming client’s request in the queue for the page requested.

Every page in the shared memory segment has its own ‘Page Message Queue Manager’, which stores the incoming client’s request in the order in which they are received. Once the status of the page is set to free the ‘Page Message Queue Manager’ starts servicing the client’s request in the queue.
The 'Page Message Queue Manager' then sends a message to the 'Page Consistency Model Manager' and 'Page Consistency Protocol Manager' that handle invalidation of copies of the requested page stored locally by other clients. The 'Page Message Queue Manager' then informs the 'Page Replication Manager' to provide the ownership of the page and a replica of the requested page is provided to the owner in Read-Write Mode.

Once the invalidation is complete and the requesting client has successfully received a copy of the requested page, the 'Page Copyset Manager' is informed of the update. The 'Page Copyset Manager' updates its copyset accordingly. The copyset for the requested page now contains only the name of the requested client as the owner along with its mode (Read-Write).

Secondly, when the client requests an ownership of a page, if there are currently no operations pending on the requested page or there doesn’t exist an owner for the requested page, the 'Page Ownership Manager' performs all the validation check’s based on Case (1) through Case (6) as discussed in the 'Page Ownership Manager' and the requested client is granted ownership.

Once the requesting client receives a replica of the requested page in Read-Write mode, the 'Page Copyset Manager' is informed of the new updates to the requested page. The 'Page Copyset Manager' updates its copyset accordingly. The copyset for the requested page now contains only the name of the requested client as the owner along with its mode (Read-Write).
8. **Page Copyset Manager:**

In a 'Page Based Distributed Shared Memory', since there are multiple clients which have a copy of the same page, it is necessary to keep a track of the clients which have a copy of the page. The Page Copyset Manager is responsible to maintain a database which stores all the information pertaining to a page such as the owner and mode of ownership (read/write), name of the clients which have a copy of the same page in read only mode. The 'Page Consistency Protocol Manager' uses this database during page invalidation purposes.

There are two possibilities to implement the 'Page Copyset Manager':

8.1 **Broadcast Approach**\(^2\):

In the broadcast approach, a message giving the page number is broadcast over the network, asking for all nodes holding the page to invalidate it. The disadvantage of this approach is that it works only if broadcast messages are totally reliable and can never be lost.

8.2 **Copyset Approach**\(^2\):

In the Copyset Approach, the owner or page manager maintain a list or copyset indicating which processors hold which pages, as shown in the figure below. The advantage of this approach is that, the server maintains all the information pertaining to a page such as the owner and the client's which has a copy in read-only mode in the form of a database. The client is freed from the overhead to maintain his own database. Since the database is stored on the centralized server, it is easy to maintain and update the database.
Figure 8.1: The owner of each page maintains a copyset telling which other nodes are sharing that page. The double boxes indicate Page ownership.

8.3 Implementation of Page Copyset Manager:

The Page Copyset Manager is implemented using the Copyset Approach. For simulation purposes nested and self-referential structures were used instead of data structures such as linked lists or binary tree as a database to store all the client information pertaining to a page. Since Inter Process Communication Library was used in the implementation of this project, Inter Process Communication Library has the drawback of not supporting marshalling and unmarshalling of objects and data structures.

As shown in Figure 8.1, a node 1, as indicated by the double box around the 4 owns page 4. The copyset consists of 2 and 4, because copies of page 4 can be found on those machines. When a page must be invalidated, 'Page Consistency Protocol Manager' uses this database to send a message to each processor holding the page and waits for an acknowledgment.
When each message has been acknowledged, the invalidation is complete and the 'Page Copyset Manager' is informed of the new changes. The 'Page Copyset Manager' updates the database to reflect the new changes. The copyset for the requested page now only contains one entry of the requesting process as the owner along with the mode of ownership (Read/Write).

Similarly when a copy of the page is requested by another process other than the owner in read-only mode, the 'Page Replication Manager' provides a copy of the page in read-only mode and informs the 'Page Copyset Manager' of the new changes. The 'Page Copyset Manager' then adds an entry on the page for the requested client along with the mode (read-only).
9. **Page Synchronization Manager:**

In a distributed shared memory system, processes often need to synchronize their actions. When multiple clients are trying to read from or write to a common segment in the shared memory region, the shared memory segment has to ensure that there is only one client in the shared memory segment working on that specific location. In terms of Distributed Database Systems, the shared memory segment represents the table within the database that is located inside the shared memory region located on the server.

The most common approach to achieve this mechanism is as discussed below:

9.1 **Mutual Exclusion Approach**[^10]:

A common example is mutual exclusion, in which only one process at a time may execute a certain part of the code. In normal use, a variable is set to 0 when no process is in the critical section and to 1 when one process is. If the value read is 1, the process just keeps repeating the code until the process in the critical region has exited and set the variable to 0.

The advantage of this mechanism is that, it is easy to implement. The disadvantage of this mechanism is that it can result in deadlock. Deadlock is defined as When Process A has acquired a lock on variable X and Process B has acquired a lock on variable Y. For example, the value of X is dependent on the value of Y and vice-versa. The result is that Process A is waiting for Process B to release a lock on variable Y and Process B is waiting for Process A to release a lock on variable X. End result is that both processes are waiting in an infinite loop for each other. Deadlock prevention algorithms can be integrated with Mutual Exclusion Approach to prevent deadlocks.
9.2 Implementation of Page Synchronization Manager:

The Page Synchronization Manager was implemented using the Mutual Exclusion Approach. The mutual exclusion approach has the disadvantage of resulting in deadlocks. In order to avoid a situation of deadlocks, the 'Page Synchronization Manager' ensures that one process is inside the critical region at any given point of time.

For example, if Process A wants to change the value of shared variable X which is dependant on the value of Y, Process A enters the critical region, acquires lock on X and Y, calculates the value of X and updates it. Once the update is done locks on both X and Y is released. Secondly now, when Process B wants to enter the critical region and change the value of Y that is dependent on the value of X. Process B enters the critical region, acquires the lock on both shared variables X and Y, calculates the value of Y and updates it. Once the update is done locks on both X and Y is released. Hence deadlock is prevented.

The 'Page Synchronization Manager' also implements semaphores that act as guardians to the shared memory segment. The semaphores ensure that only process is allowed access to the shared memory at any given point of time. When a process has entered a critical region, and any other process wants to enter the critical region, in order to ensure that the incoming clients request is not lost, the 'Page Synchronization Manager' co-ordinates with the 'Page Message Queue Manager'.

Under such circumstances, the incoming clients’ request is placed in a First In First Out (FIFO) message queue implemented by the 'Page Message Queue Manager'. All incoming requests are then forwarded to the 'Page Message Queue Manager'. The 'Page Message Queue Manager' constantly monitors the shared memory segment.
The moment the lock is released and no process is inside the shared memory segment, the ‘Page Message Queue Manager’ starts servicing the requests of the clients in the queue in the manner in which they are stored. Thus ensuring that no incoming requests from clients are lost.

Restricting only one process to enter in a distributed shared memory system means, if one process, A, is inside the critical region and another process, B, (on a different machine) wants to enter it, B will sit in a tight loop testing the variable, waiting for it to go to zero. The page containing the variable will remain on B’s machine. When A exits the critical region and tries to write 0 to the variable, it will get a page fault and pull in the page containing the variable.

Immediately thereafter, B will also get a page fault, pulling the page back. This performance is acceptable. The problem of synchronization occurs when several other processes are also trying to enter the critical region, but that disadvantage is overcome by allowing only one process to enter the critical region. The Page Synchronization Manager that accepts messages asking to enter and leave critical regions, lock and unlock the shared variables, and so on, sending back replies when the work is done.

When a region cannot be entered or a variable cannot be locked, no reply is sent back immediately, causing the sender to block. When the region becomes available or the variable can be locked, a message is sent back to the requesting process. In this way, synchronization is done with a minimum of network traffic, but at the expense of centralizing control per lock.
10. Page Message Queue Manager:

When there are multiple clients requesting various operations from the server, it is necessary for the server to service every client’s request. Since the server is busy servicing a client’s request, there is a high possibility that the new incoming client’s request may be lost. Hence the ‘Page Message Queue Manager’ ensures that no incoming clients request is lost.

In order to overcome this limitation, the ‘Page Message Queue Manager’ implements a First In First Out (FIFO) Queue Message System for every page located on the shared memory segment. For every request of operation on a specific page located within the shared memory segment, the ‘Page Message Queue Manager’ spawns a new child thread for that page, if a child process doesn’t exist. This child process is responsible for any operation requested by the clients only for the page for which it has been created.

The ‘Page Message Queue Manager’ receives the incoming request from the client for the page specified, and stores it in the queue of the existing child process responsible for the page requested and starts listening for new incoming requests. The child process constantly monitors the queue and satisfies the incoming client’s request. Hence the ‘Page Message Queue Manager’ ensures that no incoming client’s request is lost even though the child process is busy servicing another client’s request.
10.1 Implementation of the ‘Page Message Queue Manager’:

The message queue structure implemented by the ‘Page Message Queue Manager’ follows the First In First Out (FIFO) pattern. ‘First In First Out’ pattern states that 'The queue is emptied in the order in which it was filled'. The queue is implemented with the help of self referential and nested structures rather than a binary tree, linked lists or any other data structure.

This project was implemented with the help of Inter Process Communication Library. Inter Process Communication Library has the disadvantage of not being able to support marshalling and unmarshalling of objects and any other data structure other than queue’s. The structure within the queue contains the requesting client’s name, mode of request (Read/Write), if the requesting client is currently the owner and the page on which the operation is currently being requested.

The ‘Page Message Queue Manager’ works in conjunction with ‘Page Synchronization Manager’. In order to achieve sequential consistency and synchronization, it is necessary to ensure that only one child process is in the critical section or shared memory segment at any given point of time.

The child process constantly monitors its queue to check if the queue is empty or contains any entry. If the queue is empty it waits till there is a request for that page from any client in its queue. If the queue contains any pending requests for the page for which the queue is created, the child process picks up the request from its queue and verifies the status of the shared memory segment with the ‘Page Synchronization Manager’.
If the shared memory segment is free and there are currently no child processes inside the shared memory or critical section, the client's request in the queue is serviced. If the shared memory is occupied or there is currently a child process inside the shared memory, the child process constantly checks the status of shared memory at regular intervals also called as "Pooling"[^2]. The moment the shared memory segment is free, the client's request in the queue is serviced.

Hence, the 'Page Message Queue Manager' ensures that no incoming client's requests to the requested page are lost. Clients are serviced in the order or fashion in which they are received and sequential consistency and synchronization is maintained.
11. Client Architecture:

11.1 Introduction:

The client machine or workstation is responsible for establishing a connection to the 'Page Manager'. It is also responsible for making requests to the Page Manager for any operation such as read or write on a specific page. After making a request for an operation on a specific page, the client can expect a reply either from the page manager or from any other client.

For example, if the client has requested a copy of a page or shared variable in read only mode, the Page Manager checks the status of the requested page. If no client is performing a write operation on that page, a copy of the requested page is granted to the requesting client in read-only mode else the requesting client waits till the process working on the page releases the lock on the requested page. In this case, the requesting client receives the reply from the page manager.

In another scenario, if the client has requested the ownership of a page or shared variable in write mode, the 'Page Manager' checks the status of the page and uses the 'Three Message Protocol Approach' as discussed above. The client who owns the ownership of the requested page sends a reply back to the requesting client and the ownership is transferred and the requesting client receives a copy of the page in write mode. In this case, the requesting client receives the reply from the client who was previously the owner of the requested page.
The client should also act as a server and service any requesting client’s request for a specific page for which the client is an owner. Hence the same client is responsible for the various operations discussed above at any given point of time. Consider a situation that a client is performing a read or write operation for page X. The page manager sends a delete message to the client who is currently involved with page X to delete a copy of page Y from its local cache, as some other client machine or workstation wants to perform a write operation on page Y. The page manager's request will be lost because the client is currently busy.

Similarly, let’s say C1 is a client that is performing a read operation for page X. C1 is also the ownership for Page Y. Another client machine or workstation on the local area network, lets say C2 wants to perform a write operation on page Y. The ‘Page Manager’ using the ‘Three Message Protocol Approach’ sends a message to C1 to transfer the ownership of Page Y to C2.

The Page Manager’s request is lost by C1 because it is currently busy working with Page X and as a result C2 keeps waiting till C1 has completed its current operation on page X and starts listening for messages from Page Manager. Hence at any given point of time, the client should be able to perform its own operation such as read/write on any page, also reply and acknowledge messages sent by the page manager and also service any incoming client’s request also.

The following architecture discussed below explains how the client is able to perform its own operation such as read/write on any page, also reply and acknowledge messages sent by the page manager and also service any incoming client’s request at the same time. All operations are synchronized and execute in parallel with the help of Inter Process Communication Library functions.
11.2 Implementation of the Client Architecture:

Figure 11.1 illustrates the client architecture. The client architecture is responsible for all communication to the Page Manager as well as other client machine or workstations connected by a local area network. It is also responsible to ensure that no incoming request, reply or acknowledgement messages are lost. In order to achieve this, the client architecture consists of a primary client process that acts as the parent and three child processes.

The primary client process is responsible for establishing a connection to the 'Page Manager' and starts the parallel execution of the three child processes. Since the child process inherits a copy of the parent, the child processes also get connected to the 'Page Manager'.

The first child process is responsible for making an operation request to the 'Page Manager' indicating the page and the mode of operation (Read/Write). It is also responsible for making an information request to the 'Page Manager', incase the requesting client wants to claim ownership for a specific page in the distributed shared memory segment.
During an operation request, the client is actually interested in during an operation on the requested page, whereas during an information request the client is only interested in either having a local copy of the requested page in read-only mode or it can also claim ownership of the requested page. The first process is also responsible for handling all reads and writes related to a page.

The second child process is responsible for receiving the reply and acknowledgement messages from either the Page Manager or the requesting client. For example, before a requesting client claims ownership of the requested page, the page manager broadcasts a delete message to every client who has a copy of the requested page on the basis of the copyset for the requested page. The second child process receives the delete message and deletes a copy of the requested page from its local cache and sends an acknowledgement message to the page manager.

Similarly, when the first process is informing the Page Manager about its ownership for a specific page, the Page Manager verifies to check if there already exists any other client machine or workstation, which owns an ownership for the requested page.

If there doesn’t exist any other client machine or workstation which is already the owner for the requested page, the Page Manager grants the ownership else the request is denied and the requesting client has to make an operation request. Thus the ‘Page Manager’ ensures that at any given point of time, there is one and only one owner who has the ownership of the requested page.

In either case, the second process receives the reply from the page manager and informs the first process if the request was granted or denied. Communication between different processes is achieved with the help of Inter Process Communication Library functions.
In another scenario, when the client has requested the ownership of the page or a copy of the page in read-only mode, the page manager verifies if the requested page exists in the shared memory segment. If the requested page is not a part of the shared memory segment, the page manager sends a denial message to the requesting client. On receiving, the denial message from the Page Manager, the second process informs the first process that its request has been denied.

Similarly, when the client has requested the ownership of a page or shared variable in write mode, the ‘Page Manager’ checks the status of the page and uses the ‘Three Message Protocol Approach’ as discussed above. The client who owns the ownership of the requested page sends a reply back to the requesting client and the ownership is transferred and the requesting client receives a copy of the page in write mode.

The requesting client then receives a reply from the client who was previously the owner of the requested page. Once the ownership is successfully transferred, the requesting client sends an acknowledgement message to the client who had serviced its request.

The third child process is responsible for servicing any other requesting client or workstations request. For example, when a client needs the ownership of a specific page, the client makes a request to the page manager. The Page Manager uses the ‘Three Message Protocol Approach’ and the copyset of the specified page from the ‘Page Copyset Manager’ to find the owner of the requested page.

Before the ownership is transferred, the ‘Page Manager’ sends a message to all other clients from the copyset who have a copy of the requested page. All the other clients excluding the requested client and the present owner delete a copy of the requested page from their local cache and send an acknowledgement message to the ‘Page Manager’.
Once the Page Manager has received all the acknowledgment’s, the ‘Page Manager’ then sends a request to the third child process of the owner of the specified page indicating the requesting client who needs the ownership and asking the owner to transfer the ownership of the requested page.

The third child process of the present owner after receiving the reply from the Page Manager contacts the owner or the second child process of the owner and grants the ownership. Once the ownership is successfully transferred, the present owner sends an acknowledgement message to the previous owner of the requested page.

After receiving an acknowledgement, the previous owner deletes a copy of the requested page from its local cache and sends an acknowledgement message to the Page Manager indicating the transfer of ownership. The Page Manager then updates the copyset of the requested page, which only contains an entry of the present or requested client as the owner of the requested page. For simulation purposes a page represents a file, which contains all the shared variables.

In terms of a Distributed Database System, each file represents a table within the shared memory segment. The server machine or workstation creates a shared memory segment and all the files, which represent the tables in a distributed database, are stored in the shared memory segment on the server.

The server machine or workstation uses its disk space as a medium of storage. The client machine of workstation also uses its disk space as a medium of storage to store local copies of each page or file that represent a table in a distributed database system.
12. **Page Replacement Policy:**

In distributed shared memory system, a replicated page that another process owns is always a prime candidate to remove because it is known that another copy exists. Consequently, the page does not have to save anywhere. If a directory scheme is being used to keep track of copies, the owner or page manager must be informed of this decision, however. If pages are located by broadcasting, the page can just be discarded.

The second best choice is a replicated page that the evicting process owns. It is sufficient to pass ownership to one of the other copies by informing that process, the page manager, or both, depending on the implementation. The page itself need not be transferred, which results in a smaller message.

If no replicated pages are suitable candidates, a nonreplicated page must be chosen, for example, the least recently used (LRU) valid page. There are two possibilities involved as to where to store it. The first is to write to a disk, if present. The other is to hand it off to another processor.
13. **Limitations of the Page Based Distributed Shared Memory Approach:**

There are advantages and disadvantages to a larger chunk or page size for distributed shared memory. The biggest advantage is that the startup time for a network transfer is substantial, it does not take much longer to transfer 1024 bytes than it does to transfer 512 bytes. By transferring data in larger units, when a large piece of address space has to be moved, the number of transfers may often be reduced.

This property is especially important because many programs exhibit locality of reference, meaning that if a program has referenced one word on a page, it is likely to reference other words on the same page in the immediate future. The client machines or workstations as a memory unit use disk space to store all the copies of the pages, either in read-only mode or in the write mode.

On the other hand, the network will be tied up longer with a larger transfer, blocking other faults caused by other processors. Also, too large an effective page size introduces a new problem called "False Sharing" illustrated in Figure 13.1 shown below.

![Figure 13.1: False sharing of a page containing two unrelated variables.](image)

Figure 13.1: False sharing of a page containing two unrelated variables.
Here we have a page containing two unrelated shared variables, A and B. Processor 1 makes heavy use of A, reading and writing it. Similarly, process 2 uses B. Under these circumstances, the page containing both variables will constantly be traveling back and forth between the two machines.

The problem here is that although the variables are unrelated, but they appear by accident on the same page. When a process uses one of them, it also gets the other. The larger the effective page size, the more often false sharing will occur, and conversely, the smaller the effective page size, the less often it will occur.

In terms of a distributed database system and in this model ‘Page Based Distributed Shared Memory System’ can be thought of as table level locking. For example, let’s say Client C1 is trying to perform a Update to a specific row in table A. Client C2 on the network wants to perform an Insert to the same table. Since Client C1 has locked the entire table, Client C2 has to wait till Client C1 has released the lock on this table.

One problem that is unique to distributed shared memory systems is the network traffic generated when processes on different machines are actively sharing a writeable page, either through false sharing or true sharing. A rule can be enforced to reduce this traffic is that once a page has arrived at any processor, it must remain there for some time.

If requests for it come in from other machines, these requests are simply queued until the timer expires, thus allowing the local process to make many memory references without interface. A more structured and alternative approach is to share only certain variables and data structures that are needed by more than one process.
In this way, the problem changes from how to do paging over the network to how to maintain a potentially replicated distributed database consisting of the shared variables. Using shared variables that are individually managed also provides considerable opportunity to eliminate false sharing. One of the techniques that can be applied here, which often leads to major performance improvement, is “Object-Based Distributed Shared Memory [5]”. 
14. **Object-Based Distributed Shared Memory:**

14.1 Introduction:

In many programming languages, data are organized into objects, packages, modules, or other data structures, each of which has an existence independent of the others. If a process references part of an object, in many cases the entire object will be needed, so it makes sense to transport data over the network in units of objects, not in units of pages.

In terms of a distributed database system and in this model, 'Page Based Distributed Shared Memory' illustrates the concept of table level locking whereas 'Object Based Distributed Shared Memory' illustrates the concept of record level locking where each record is treated as a separate object. Most of the commercial distributed databases such as Oracle, Sybase, Informix etc work on the same principle of record level locking.

14.2 Objects:

An object is a programmer-defined encapsulated data structure as shown in Figure 14.1. It consists of internal data, the Object State, and procedures, called methods or operations, which operate on the Object State. To access or operate on the internal state, the program must invoke one of its methods. The methods can change the internal state, return (part of) the state, or something else. Direct access to the internal state is not allowed. This property called information hiding.
Forcing all references to an object’s data to go through the methods helps structure the program in a modular way, with the help of accessor (functions that start with the Get keyword), mutator (functions that start with the Set keyword), predicate (functions that start with the Is keyword), initializer (default, initializing and copy constructor), terminator (destructor) and utility or helper (functions that are private to the class) functions and declaring all data members as private.

<table>
<thead>
<tr>
<th>Object State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auxiliary variables</td>
</tr>
<tr>
<td>Method 3</td>
</tr>
<tr>
<td>Method 2</td>
</tr>
<tr>
<td>Method 1</td>
</tr>
</tbody>
</table>

*Figure 14.1: An Object*

In an object-based distributed shared memory, processes on multiple machines share an abstract space filled with shared objects, as shown in Figure 14.2. The location and management of the objects is handled automatically by the runtime system. This model is in contrast to page-based distributed shared memory systems such as IVY	extsuperscript{[3]}, which just provide a raw of linear memory of bytes from 0 to some maximum.
Any process can invoke any object’s methods, regardless of where the process and object are located. It is the job of the operating system and runtime system to make the act of invoking a method work no matter where the process and object are located. Because processes cannot directly access the internal state of the shared objects, various optimizations are possible with this approach that are not possible (or at least are more difficult) with page-based distributed shared memory.

A shared memory is structured as a collection of separate objects instead of as a linear address space. The most important issue is whether objects should be replicated or not. If replication is not used, all accesses to an object go through the one and only copy, which is simple, but may lead to poor performance. By allowing objects to migrate from machine to machine, as needed, it may be possible to reduce the performance loss by moving objects to where they are needed.
If objects are replicated, one approach is to invalidate all the other copies, so that only the up-to-date copy remains. Additional copies can be created later, on demand, as needed. An alternative approach is not to invalidate the copies, but to update them.

Shared-variable distributed-shared memory also has this choice, but for page-based distributed-shared memory, invalidation is the only feasible choice because 'Write Invalidation Protocol' overcomes all the limitations of 'Write Through' and 'Write Update' protocol such as high network load, difficult to implement etc.

Similarly, object-based distributed-shared memory, like shared-variable distributed shared memory, eliminates most false sharing since locking is related to records and not tables in terms of a distributed database system.

14.3 Advantages over Page-Based Distributed Shared Memory:

Object-based distributed-shared memory offers three advantages over page-based distributed-shared memory:

1. It is more modular.

2. The implementation is more flexible because accesses are controlled.

3. Synchronization and access can be integrated together cleanly.

The first point states that "A program is whose structure consists of interrelated segments, arranged in a logical and easy understandable order is said to be a modular program. Such programs are easier to develop, correct and modify than programs constructed otherwise" [5].
The second point states that client of the object can be granted controlled access by categorizing the object's data member and member functions as public, private and protected. Hence accesses are controlled.

The third point states that since client accesses can be controlled as stated in the second point, it becomes easier to synchronize any action on a particular object. Rather than maintaining an overhead of synchronizing locks over pages or tables, it is easier to achieve synchronization and control access over objects or records.

14.4 Disadvantage of Object Based Distributed Shared Memory:

The one disadvantage of an object-based distributed-shared memory is that invoking the object methods must do all accesses to shared objects. In 'Object Based Distributed Shared Memory' approach, extra overhead's such as maintaining scope resolution of objects and variables, memory management issues with objects, pointers, data structures etc is incurred that is not present with shared pages that can be accesses directly.

14.5 Implementation of Object Based Distributed Shared Memory:

The 'Page Based Distributed Shared Memory System' was designed and implemented using C and Inter Process Communication Library. Inter Process Communication Library has the drawback or limitation of not being able to support object's and marshalling and unmarshalling of complex data structures such as linked lists, binary tree et cetera. In order to simulate and implement the 'Object Based Distributed Shared Memory', libraries having the capability to support distributed objects such as Common Object Request Broker Architecture (CORBA), Remote Method Invocation (RMI), and Distributed Component Object Model (DCOM) should be used.
15. **Summary:**

The limitations or bottleneck of this simulation with approximately 100 clients and 100 files or pages is in terms of scalability and performance. The present implementation suffers from two major bottlenecks:

1. **Large Number of Child Processes running on the application server:**

   In the present scenario, there exists a unique child process that is responsible for handling the entire client request for any operation such as read/write on the page for which the process is created. Since every page has a unique process, if there are 100 pages, there exist 100 heavy weight child processes running on the server. This can drastically bring down the performance.

   An ideal solution to overcome this potential problem is by incorporating Load Balancing and Load Sharing Algorithms to the existing software. Another alternative approach is by using Resource or Thread Pooling. A low priority garbage collector thread constantly runs and monitors the state of each process. If a child process is idle for quite a long time, then it can be removed from memory, which lowers the number of processes running on the application server.

2. **Large Number of Open Connections running on the application server:**

   In the present scenario, every client has a connection opened with the Page Manager. When it needs service from another client, it opens another connection with the client satisfying its request. Hence there are multiple connections open at any given point of time. With approximately 100 clients, this approach will drastically bring down the performance. Again, Connection Pooling would be an ideal approach to solve this problem.
Appendices
Appendix A: Inter Process Communication (ipC) Library

ipC is a C library of carefully packaged Unix system calls designed to provide a friendly concurrent-programming environment. The ipC library contains the following procedures and functions.

1.1. Parallel Execution (local)

void ParallelExec(char *Proc[ ], int NProcs)
-- ParallelExec initiates the concurrent execution of the executable files listed in the string array Proc. These processes are numbered 0 . . . NProcs-1. Each process receives its number, as it's only 'command line' argument. Control does not return to the caller until after all the processes have terminated (completed or aborted).

1.2. Shared Memory (local)

char *SharedMem(int Size)
-- SharedMem allocates a memory segment of the specified size and attaches it to the caller's address space. Only one shared segment can be in existence at any one time.

char *AttachMem(void)
-- AttachMem attaches the segment created by SharedMem to the caller's address space.

void RemoveMem(void)
-- RemoveMem deallocates the segment created by SharedMem.
1.3. Semaphores (local)

void Semaphores(int InitVals[ ], int NSems)
-- Semaphores creates a family of semaphores, numbered 0 . . . NSems-1, and initializes them with the values in InitVals. Only one family of semaphores can be in existence at any one time.

void RemoveSems(void)
-- RemoveSems removes the family of semaphores created by Semaphores.

void Down(int SemNo)
-- Down performs a down (or P or wait) operation on a given semaphore in the family created by Semaphores.

void Up(int SemNo)
-- Up performs an up (or V or signal) operation on a given semaphore in the family created by Semaphores.

1.4. Message Queues (local)

void MailBox(char MBoxId)
-- MailBox creates a mail box with the given id.

void RemoveBox(char MBoxId)
-- RemoveBox removes the specified mail box from the system.

void Send(char MBoxId,int MType,int MSize,char *Message)
-- Send sends a message to the given mail box. The length of the message is specified by MSize and a user defined message type is given by MType (MType must be non-zero).
void Receive(char MBoxld, int MType, int MSize, char *Message)
-- Receive receives the next message in the mail box queue of the specified
message type. Note that the message size is specified by Receive.

int RecvAny(char MBoxld, int MSize, char *Message)
-- RecvAny receives the next message in the mail box queue. The message type
is returned as the value of RecvAny.

1.5. Sockets (inter-host)

void InternetAddr((struct sockaddr_in *AddPtr, char *Hostname, int Port))
-- Makes an internet socket address for a destination whose machine and port
are given as arguments. To refer to the local host, use a null char pointer. To let
the system select a port, use 0 as the port number. Otherwise, Port must be an
integer greater than or equal to 1024. These comments also apply to all
instances of Hostname and Port that appear in the other socket calls below.

int PortNm(struct sockaddr_in Addr)
-- Return the port number from a socket internet address

c char *HostName(struct sockaddr_in Addr)
-- Return the host name from a socket internet address

void CloseSocket(int SockDescr)
-- Close a socket.

void PrintInAddr(struct sockaddr_in Addr)
-- Print an internet address in a user readable form.
1.5.1. Datagram Sockets (UDP - inter-host)

int DatagramSocket(int Port)
-- Makes an internet address for a port on this machine and binds it to a socket.
It returns the socket descriptor, or -1 if the port is currently in use.

int SendDatagram(int SockDesc, char *Mess, int Len, struct sockaddr_in *AddPtr)
-- Sends a datagram message via a socket to a given internet address. Returns
the number of bytes sent.

int RecvDatagram(int SockDesc, char *Mess, int Len, struct sockaddr_in *AddPtr)
-- Receives a datagram (and the sender's internet address) via a socket. Returns
the number of bytes received.

int BroadSocket(void)
-- Creates a broadcast socket and returns the socket descriptor.

void Broadcast(int SockDesc, char *Mess, int Len, int Port)
-- Broadcast a message to the given port on all hosts in the local subnet.

1.5.2. Stream Sockets (TCP inter-host)

int ListenSocket(int Port)
-- Makes an internet address for a port on this machine and binds it to a stream
socket. Returns the socket descriptor, or -1 if the port is currently in use.

int AcceptConnection(int SockDesc)
-- Accepts a new stream socket connection. Used to offload service requests, so
that the server can continue to listen for new requests on the original socket.
Returns a socket descriptor.
int ConnectSocket(char *Hostname, int Port)
-- Makes an internet address for a port on a remote machine and connects to it via a stream socket. Returns the socket descriptor.

1.6. Other
void WriteStr(char *S)
-- WriteStr writes a string to standard output. Output is not buffered.

int Random(int R)
-- Random returns the next in a sequence of pseudo-random integers chosen from the range 0 . . . R-1.

void Seed(int S)
-- Seed selects the pseudo-random sequence to be returned by Random.

int gethostname (char *name, int namelen) [UNIX]
-- Gethostname returns the standard hostname for the current processor. The parameter namelen specifies the size of the name array. The returned name is null-terminated unless insufficient space is provided. A return value of 0 indicates success, -1 failure.

waitpid(-1, (int *)0, WNOHANG)
-- This is a quick and dirty way of cleaning up child processes during the execution of a multi-threaded server. Requires
#include <sys/types.h>
#include <sys/wait.h>
Appendix B: Source Code

/************************************************************************/
/* Master's Project : Information Technology */
/* Author : Padmanabhan Raman */
/* File : client.c */
/* Description : This client handles parallel execution of */
/* c_client,c_pgmngr,c_server. */
 /************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "/usr/local/pub/atk/ipC/include/ipC.h"
#include "client.h"

#define NPROCS 3

int main(int ac, char *av[])
{
  /*Local Declarations*/
  char* Proc[NPROCS];
  int Init[1];
  parent_struct*_PtrToParent;

  /*Create a shared memory*/
  _PtrToParent = (parent_struct *)SharedMem(sizeof(parent_struct));
/ * initializing no. of nodes and port number to zero */
PtrToParent->no_of_nodes = 0;
PtrToParent->pm_portno = 0;

/* Copy the hostname and the port number */
strcpy(PtrToParent->pm_hostname, av[1]);
PtrToParent->pm_portno = atoi(av[2]);
PtrToParent->granted = 'N'; /* Initially request not granted */

/* Only one process can enter the critical region */
Init[0] = 1;

/* Creating semaphores */
Semaphores(Init, 1);

/* Name of the executable's of the 3 processes executing in parallel */
Proc[0] = "c_client";
Proc[1] = "c_pgmngr";
Proc[2] = "c_server";

/* Executing all of them in parallel */
ParallelExec(Proc, NPROCS);

return 0;
} /* end main */
/* Master's Project : Information Technology */
/* Author : Padmanabhan Raman */
/* File : c_client.c */
/* Description : This client handles only requests to the page manager. Requests such as Information or operation */
/* During Info it registers with the page manager */
/* and during operation it requests the page manager */
/* for a read or write operation */

/*Include Required Libraries*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "usr/local/pub/atk/ipC/include/ipC.h"
#include "common.h"
#include "client.h"

/*Pre-Processor Directives*/
#define TRUE 1
#define FALSE 0
#define SEMAPHORE 0
#define PGMNGR_HOSTNAME "marvin" /* Hostname of the page manager */
#define PGMNGR_PORTNO 3000 /* Port number of the page manager*/

/*Global Declarations*/
parent_struct* _PTR_TO_PARENT;
/* checks file exist or not */
/* if exists -> returns 1 */
/* else -> returns 0 */
int is_file_exist(char f_name[])
{
    /* Local Declarations */
    int node_iterator = 0, file_found = 0;

    /*Lock the shared memory segment*/
    Down(SEMAPHORE);

    /*Loop through each entry of the child copy set to*/
    /* find if the file already exists with the child*/
    for(node_iterator = 0; node_iterator < PtrToParent->no_of_nodes; node_iterator++)
    {
        if (strcmp(f_name, PtrToParent->children[node_iterator].filename) == 0)
        {
            file_found = 1; /*File is found*/
            break;
        }/* end if*/
    }/* end for*/

    /*Release the lock on the shared memory segment*/
    Up(SEMAPHORE);

    /*return the status back*/
    return file_found;
}/*end is_file_exist*/
/ adds new file node in the list */
void add_file_node(char f_name[], char new_mode, char is_owner)
{
    /*Lock the shared memory segment*/
    Down(SEMAPHORE);

    /*Add the new file node in the list*/
    strcpy(PtrToParent->children[PtrToParent->no_of_nodes].filename, f_name);
    PtrToParent->children[PtrToParent->no_of_nodes].mode = new_mode;
    PtrToParent->children[PtrToParent->no_of_nodes].owner = is_owner;

    /*Increment the number of nodes*/
    PtrToParent->no_of_nodes++;

    /*Release the lock on the shared memory*/
    Up(SEMAPHORE);
} /*end add_file_node*/

/* Read from file */
void read_from_file(char filename[])
{
    /*Local Declarations*/
    FILE* fp = null;
    int  ch = 0; /* Int, not char */

    /* Open file for reading */
    fp = fopen(filename, "r");

    /* Error opening file */
    if (fp == NULL)
    {
        perror(filename);
        exit(1);
    } /*end if*/

    /* Next char from file */
    while ((ch = getc(fp)) != EOF)
DATA READ FROM FILE:

/* Output to screen */

EOF occurred due to error */

if (ferror(fp))
{
    perror(filename);
    exit(1);
} /* end if */

/* Close the file */
fclose(fp);
/* end read_from_file */

/* Write into file */
void write_into_file(char filename[])
{
    /* Local declarations */
    FILE* fp = NULL;
    int ch = 0; /* Int, not char */
    int value = 0;

    /* Open file for writing */
    fp = fopen(filename, "w");

    /* Error opening file */
    if (fp == NULL)
    {
        perror(filename);
        exit(1);
    }

    /* Read the value from the user */
    printf("Enter an integer you want to write: ");
    scanf("%d", &value);
/* Store the value in the file */
if (putc(value, fp) == EOF)
    printf("\Error occured in writing.");

/* EOF occured due to error? */
if (ferror(fp))
{
    perror(filename);
    exit(1);
} /* end if */

/* Close the file */
fclose(fp);
} /* end write_into_file */

/* main program begins */
int main(int ac, char *av[])
{
    /* Local Declarations */
    int SockDesc = 0;
    int start_again = 0;
    int frk_portno = 0;
    char hostname[TEXTLEN];
    char i_or_op, mode, is_owner;
    char f_name[TEXTLEN];
    char finished = 'N'; /* Y = Yes, N = No */
    struct sockaddr_in PmAddr;
    common_struct client_struct;
    struct sockaddr_in frkAddr;

    /* Attaching parent structure to sharem memory */
    PtrToParent = (parent_struct *)AttachMem();

    /* Lock the shared memory segment */
    Down(SEMAPHORE);
/* create page manager's internet address */
InternetAddr(&PmAddr, PtrToParent->pm_hostname, PtrToParent->pm_portno);

/*Release the lock on the shared memory segment*/
Up(SEMAPHORE);

/* creating sockdesc for itself */
SockDesc = DatagramSocket(CL_CL_PORTNO);

/* get clients hostname */
if(gethostname(hostname,HOSTLENGTH) == -1)
    printf("\n gethostname failed.");

/* get input from user */
while(TRUE)
{
    /*Indicates that the operation has to started over again*/
    start_again = 0;

    /*Check if the user wants to perform an information or an operation*/
    printf("\n Which operation you want to perform ? (I/O) : ");
    scanf("%c",&i_or_op);
    fflush(stdin);

    /*Read the name of the file*/
    printf("\n Enter filename : ");
    scanf("%s",f_name);
    fflush(stdin);

    /*Enter the mode of ownership - READ or WRITE*/
    printf("\n Enter mode (R/W) : ");
    scanf("%c",&mode);
    fflush(stdin);

    /*Copy all the information required for registration with page manager*/
    strcpy(client_struct.filename,f_name);
    client_struct.mode = mode;
client_struct.req_type = i_or_op;
strcpy(client_struct.client_hostname, hostname);
client_struct.cl_pm_portno = CL_PM_PORTNO;
client_struct.cl_serv_portno = CL_SERV_PORTNO;
client_struct.message_type = 'Z';
client_struct.message_from = 'C';
client_struct.file_present = 'N';

/* Send request to page manager */
SendDatagram(SockDesc,(char *)&client_struct, sizeof(common_struct), &PmAddr);

/*Receive reply from the page manager*/
RecvDatagram(SockDesc,(char *)&client_struct, sizeof(common_struct), &PmAddr);

/*If file doesn't exists*/
if(!is_file_exist(f_name))
{
    /*Check if the operation requested is an information*/
    if(i_or_op == 'I')
    {
        /*Add the file information in its list because file doesn't exist*/
        add_file_node(f_name,mode,client_struct.owner_granted);

        /*Reset the flag. Take another input from the user*/
        start_again = 1;
    }/*end if*/

    /*Check if the operation requested is an operation*/
    else if(i_or_op == 'O')
    {
        /* check if file is registered with page manager */
        if(client_struct.file_present == 'N') /* file not with pgmng */
        {
            printf("File doesn't exist. Service denied.");
            start_again = 1; /* Go back to take i/p from the user. */
        }/* end of if */
    }/*end else*/
}/*end if*/

/* if file exists */
else if(is_file_exist(f_name))
{
    /* if information, go back and take another input from user*/
    if(i_or_op == 'I')
        start_again = 1;
}/*end else*/

/*If the user has requested an operation*/
if(start_again == 0)
{
    /* Waits till the requesting process is granted permission 
        to perform an operation */
    for(;;)
    {
        /*Lock the shared memory segment*/
        Down(SEMAPHORE);

        /*Check if the requesting process has been granted permission*/
        if (PtrToParent->granted == 'Y')
        {
            frk_portno = PtrToParent->fork_process_portno;

            /*Unlock the shared memory segment*/
            Up(SEMAPHORE);

            break;
        } /*end if*/

        /*Unlock the shared memory segment*/
        Up(SEMAPHORE);

        /* timer function */
        while(timer(SockDesc, 1) != 0){}
/* end of for loop */

/* If the requested operation is a write */
if(mode == 'W')
    write_into_file(f_name);

/* If the requested operation is a read */
else if(mode == 'R')
    read_from_file(f_name);

/* asks user if the requested operation is completed or still in progress */
while(TRUE)
{
    if(mode == 'W')
        printf("\n Have you finished writing ? ");
    else if(mode == 'R')
        printf("\n Have you finished reading ? ");

    fflush(stdin);
    scanf("%c", &finished);
    fflush(stdin);

    if (finished == 'Y') break;
}/* end of while */

/* Lock the shared memory segment */
Down(SEMAPHORE);
/*Don't grant access to any process for any operation on this file till I'm done*/
PtrToParent->granted = 'N';

/*Unlock the shared memory segment*/
Up(SEMAPHORE);

/* inform page manager that it has finished operation*/
client_struct.mode = 'F';

/* creating internet address for the forked process*/
InternetAddr(&frkAddr, PtrToParent->pm_hostname, frk_portno);

/* Sends a message to the page manager*/
SendDatagram(SockDesc,(char *)&client_struct, sizeof(common_struct), &frkAddr);
}/* end of if */
}/* end of while */
return 0;
}/*end of main*/
/**********************************************************/
/*  Master's Project : Information Technology
/*  Author        : Padmanabhan Raman
/*  File          : c_pgmngr.c
/*  Description   : This client handles only requests from the  
/*                      page manager. Requests such as Delete or  
/*                      requesting another client if this client does not  
/*                      have a copy of the file on which it wants to  
/*                      perform an operation.  */
/***********************************************************************************/

/*Include Required Libraries*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <usr/local/pub/atk/ipC/include/ipC.h
#include "common.h"
#include "client.h"

/*Pre-Processor Directives*/
#define SEMAPHORE 0

/*Global Declarations*/
parent_struct*    PtrToParent;

/* checks file exist or not */
/* if exists -> returns 1 */
/* else -> returns 0 */
int is_file_exist(char f_name[])
{
    /*Local Declarations*/
    int node_iterator = 0, file_found = 0;

    /*Lock the shared memory segment*/
    Down(SEMAPHORE);

    /*Loop through each entry of the child copy set to
find if the file already exists with the child*/

for (node_iterator = 0; node_iterator < PtrToParent->no_of_nodes; node_iterator++)
{
    if (strcmp(f_name, PtrToParent->children[i].filename) == 0)
    {
        file_found = 1; /*File is found*/
        break;
    } /*end if*/
} /*end for*/

;/*Release the lock on the shared memory segment*/
Up(SEMAPHORE);

;/*return the status back*/
return file_found;
} /*end is_file_exist*/

/* delete file from the list */
void delete_file_from_list(char f_name[])
{
    /*Local declarations*/
    int inner_node_iterator = 0, outer_node_iterator = 0, file_found = 0;

    /*Lock the shared memory segment*/
    Down(SEMAPHORE);

    /*Loop through each entry of the child copy set to
     find if the file already exists with the child*/
    for (inner_node_iterator = 0; inner_node_iterator < PtrToParent->no_of_nodes; inner_node_iterator++)
    {
        if (strcmp(f_name, PtrToParent->children[inner_node_iterator].filename) == 0)
        {
            file_found = 1; /*File is found*/
            break;
        } /*end if*/
    } /*end for*/
/*Move the list one position above to delete an entry from the list*/
for(outer_node_iterator = inner_node_iterator; outer_node_iterator < PtrToParent->no_of_nodes - 1; outer_node_iterator++)
{
    strcpy(PtrToParent->children[outer_node_iterator].filename, PtrToParent-
>children[outer_node_iterator+1].filename);
    PtrToParent->children[outer_node_iterator].mode = PtrToParent-
>children[outer_node_iterator+1].mode;
    PtrToParent->children[outer_node_iterator].owner = PtrToParent-
>children[outer_node_iterator+1].owner;
} /*end for*/

/*Decrement the number of nodes by 1*/
PtrToParent->no_of_nodes--;

/*Unlock the shared memory segment*/
Up(SEMAPHORE);
} /*end delete_file_from_list*/

/* main program begins */
int main(int ac, char *av[])
{
    /* Local declarations */
    int iterator = 0;
    int SockDesc = 0;
    int is_owner = 0;
    int frk_portno = 0;
    struct sockaddr_in other_client_addr;
    struct sockaddr_in PmAddr;
    common_struct c_pm_struct;
    char hostname[HOSTLENGTH];

    /* Attaching parent structure to sharem memory */
    PtrToParent = (parent_struct *)AttachMem();

    /* Lock the shared memory segment */

Down(SEMAPHORE);

/* creating page manager's internet address */
InternetAddr(&PmAddr, PtrToParent->pm_hostname, PtrToParent->pm_portno);

/* UnLock the shared memory segment */
Up(SEMAPHORE);

/* create socket descriptor for itself */
SockDesc = DatagramSocket(CL_PM_PORTNO);

/* Receive input from Client's */
while(TRUE)
{
    /* Receive from page manager */
    RecvDatagram(SockDesc, (char *)&c_pm_struct, sizeof(common_struct), &PmAddr);

    /* get the portno of forked process */
    frkj_portno = PortNm(PmAddr);

    /* check request is granted or denied */
    if (c_pm_struct.message_type == 'G')
    {
        /* Lock the shared memory segment */
        Down(SEMAPHORE);

        /* Get the port number of the fork process */
        PtrToParent->fork_process_portno = frkj_portno;

        /* Unlock the shared memory segment */
        Up(SEMAPHORE);

        /* Check if file exists */
        if(is_file_exist(c_pm_struct.filename))
        {
            /* Lock the shared memory segment */
        }
    }
}
Down(SEMAPHORE);

/* Now client can perform operation on requested file */
PtrToParent->granted = 'Y';

/* UnLock the shared memory segment */
Up(SEMAPHORE);
}/*end if*/
else
{
    /* create receiving(other) client's internet address */
    InternetAddr(&other_client_addr,c_pm_struct.owner_hostname,CL_SERV_PORTNO);

    /* Request file from the owner */
c_pm_struct.message_type = 'R'; /* R = Request */

    /* get clients hostname */
    if(gethostname(hostname,HOSTLENGTH) == -1)
        {
            printf("\n GetHostName Failed.");
            exit(1);
        }/*end if*/

    /* copy the clients hostname */
    strcpy(c_pm_struct.client_hostname,hostname);

    /* Send the information to the client */
    SendDatagram(SockDesc, (char *)&c_pm_struct, sizeof(common_struct),
    &other_client_addr);
}/*end else*/
}/*end if*/

/* check message is for deleting file */
if (c_pm_struct.message_type == 'D')
{

    /* Check if file exists */
if (is_file_exist(c_pm_struct.filename))
{
    printf(" Deleting %s from list", c_pm_struct.filename);

    /* Lock the shared memory segment */
    Down(SEMAPHORE);

    /* Search through the list for the owner of the file */
    for(iterator = 0; iterator < PTrToParent->no_of_nodes; iterator++)
    {
        if(strcmp(PTrToParent->children[iterator].filename, c_pm_struct.filename) == 0)
        {
            if(PTrToParent->children[iterator].owner == 'Y')
            {
                is_owner = 1;
                break;
            }
        }
    }

    /* UnLock the shared memory segment */
    Up(SEMAPHORE);

    /* Delete from list */
    if (is_owner == 0)
        delete_file_from_list(c_pm_struct.filename);

    /* Ack. msg. that it has deleted the file */
    c_pm_struct.message_type = 'A';

    /* Send the message to the page manager */
    SendDatagram(SockDesc, (char *)&c_pm_struct, sizeof(common_struct), &PmAddr);
}
}
return 0;
}/*end if*/
}/*end while*/
return 0;
} /*end of main*/
/* Include Required Libraries */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "/usr/local/pub/atkC/include/ipC.h"
#include "common.h"
#include "client.h"

/* Pre-Processor Directives */
define SEMAPHORE 0

/* Global Declarations */
parent struct* PtrToParent;

delete file from the list */
void delete_file_from_list(char f_name[])
{
    /* Local declarations */
    int i = 0, k = 0, file_found = 0;

    /* Lock the shared memory segment */
    Down(SEMAPHERE);

    /* Search for the position of the file in the list */
    for(i = 0; i < PtrToParent->no_of_nodes; i++)
{ 
    if (strcmp(f_name, PtrToParent->children[i].filename) == 0) 
    {
        file_found = 1;
        break;
    } /*end if*/
} /*end for*/

/*Remove the entry from the list*/
for(k = i; k < PtrToParent->no_of_nodes - 1; k++)
{
    strcpy(PtrToParent->children[k].filename, PtrToParent->children[k+1].filename);
    PtrToParent->children[k].mode = PtrToParent->children[k+1].mode;
    PtrToParent->children[k].owner = PtrToParent->children[k+1].owner;
} /*end for*/

/*Decrement the number of nodes*/
PtrToParent->no_of_nodes--;

/*Unlock the shared memory segment*/
Up(SEMAPHORE);
} /* end of delete_file_from_list function */
/* Add new file node in the list */
void add_file_node(char f_name[], char new_mode, char is_owner)
{
    /*Local declarations*/
    int k = 0;

    /*Lock the shared memory segment*/
    Down(SEMAPHORE);

    /*Add the node in the list*/
    strcpy(PtrToParent->children[PtrToParent->no_of_nodes].filename, f_name);
    PtrToParent->children[PtrToParent->no_of_nodes].mode = new_mode;
    PtrToParent->children[PtrToParent->no_of_nodes].owner = is_owner;

    /*Increment the number of nodes*/
    PtrToParent->no_of_nodes++;

    /*UnLock the shared memory segment*/
    Up(SEMAPHORE);
} /*end of add_file_node*/

/* main program begins */
int main(int ac, char *av[])
{
    /*Local declarations*/
    int SockDesc = 0;
    int is_owner = 0;
    char hostname[TEXTLEN];
    struct sockaddr_in other_client_addr;
    struct sockaddr_in request_client_addr;
    common_struct c_serv_struct;

    /* Attaching parent structure to sharem memory */
    PtrToParent = (parent_struct *)AttachMem();

    /* create socket descriptor for itself */
    SockDesc = DatagramSocket(CL_SERV_PORTNO);
/* Receive from other client */
while(1)
{
    RecvDatagram(SockDesc, (char *)&c_serv_struct, sizeof(common_struct), &other_client_addr);
}

/* creating the internet address for requesting client ( C:2 ) */
InternetAddr(&request_client_addr, c_serv_struct.client_hostname, CL_SERV_PORTNO);

/*Check the request is for a write*/
if((c_serv_struct.message_type == 'R') && (c_serv_struct.mode == 'W'))
{
    delete_file_from_list(c_serv_struct.filename); /* delete from list */
    c_serv_struct.message_type = 'G'; /* Grant request */
    SendDatagram(SockDesc, (char *)&c_serv_struct, sizeof(common_struct), &request_client_addr);
    c_serv_struct.message_type = 'Z';
}

/*Check the request is for a read*/
if((c_serv_struct.message_type == 'R') && (c_serv_struct.mode == 'R'))
{
    c_serv_struct.message_type = 'G'; /* Grant request */
    SendDatagram(SockDesc, (char *)&c_serv_struct, sizeof(common_struct), &request_client_addr);
    c_serv_struct.message_type = 'Z';
}

/*Check the request is granted*/
if(c_serv_struct.message_type == 'G')
{
    /*if request is for write, then ownership is granted*/
    if(c_serv_struct.mode == 'W')
    {
        add_file_node(c_serv_struct.filename, c_serv_struct.mode, 'Y');
    }
    else
{  
    add_file_node(c_serv_struct.filename,c_serv_struct.mode,'N');  
}

/*Lock the shared memory segment*/  
Down(SEMAPHORE);

PtrToParent->granted = 'Y';

/*Unlock the shared memory segment*/  
Up(SEMAPHORE);

*/ end if */

*/ end of while */  
return 0;

*/ end of main */
# Include Required Libraries

#include "page_manager.h"

// Pre-Processor Directives

#define SEMAPHORE 0
#define MUTEX 1

// Global Declarations

parent_pm*  PtrToPM;
int SockDesc;

int is_file_exists(char f_name[])
{
    /* Local declarations */
    int iterator = 0, file_found = 0;

    /* Lock the shared memory segment */
    Down(SEMAPHORE);

    /* Search for the file in the list */
    for (iterator = 0; iterator < PtrToPM->no_of_list_nodes; iterator++)
    {
if (strcmp(f_name, PtrToPM->Cl_list[iterator].filename) == 0)
{
    file_found = 1; /* File is found */
    break;
}/*end if*/
}/*end for*/

/*UnLock the shared memory segment*/
Up(SEMAPHORE);
fflush(stdout);

/*Return the status*/
return file_found;
}/*end is_file_exists*/

/* adds new file node in the list */
void add_newfile_in_list(char f_name[], char new_hostname[], int new_portno)
{
    /*Lock the shared memory segment*/
    Down(SEMAPHORE);

    /* Add the file to the list */
    strcpy(PtrToPM->Cl_list[PtrToPM->no_of_list_nodes].filename, f_name);
    strcpy(PtrToPM->Cl_list[PtrToPM->no_of_list_nodes].owner_hostname, new_hostname);

    /* Increment the number of list nodes */
    PtrToPM->no_of_list_nodes++;

    /*UnLock the shared memory segment*/
    Up(SEMAPHORE);
}/* end add_newfile_in_list */

/* adds new file node in the queue */
void add_newfile_in_queue(char f_name[], char new_hostname[], int new_portno, char new_mode)
{
    /*Lock the shared memory segment*/
Down(SEMAPHORE);

/*Add the requested file for operation in the queue */
strcpy(PtrToPM->Queue[PtrToPM->no_of_queue_nodes].filename,f_name);
strcpy(PtrToPM->Queue[PtrToPM->no_of_queue_nodes].request_hostname[PtrToPM->Queue[PtrToPM->no_of_queue_nodes].no_of_request_for_file],new_hostname);
PtrToPM->Queue[PtrToPM->no_of_queue_nodes].request_mode[PtrToPM->Queue[PtrToPM->no_of_queue_nodes].no_of_request_for_file] = new_mode;

PtrToPM->Queue[PtrToPM->no_of_queue_nodes].no_of_request_for_file++;
PtrToPM->no_of_queue_nodes++;

/*UnLock the shared memory segment*/
Up(SEMAPHORE);
}*/ end add_newfile_in_queue */
/* Checks queue exists for a file or not */
int is_queue_exists(char f_name[])
{
    /* Local Declarations */
    int iterator = 0;
    int found = 0;

    /*Lock the shared memory segment*/
    Down(SEMAPHORE);

    /*Search for the file name in the queue*/
    for(iterator = 0; iterator < PtrToPM->no_of_queue_nodes ; iterator++)
    {
        if(strcmp(PtrToPM->Queue[iterator].filename, f_name) == 0)
        {
            found = 1; /* File is found */
            break;
        }/*end if*/
    }/*end for*/

    /* UnLock the shared memory segment */
    Up(SEMAPHORE);

    /* Return the status */
    return found;
}/*end is_queue_exists*/
/* add request in the existing queue */
void add_request_in_queue(char f_name[], char new_hostname[], char new_mode)
{
    /* Local declarations */
    int iterator = 0;

    /* Lock the shared memory segment */
    Down(SEMAPHORE);

    /* Add the request in the queue */
    for(iterator = 0; iterator < PtrToPM->no_of_queue_nodes ; iterator++)
    {
        if(strcmp(PtrToPM->Queue[iterator].filename,f_name) == 0)
        {
            strcpy(PtrToPM->Queue[iterator].request_hostname[PtrToPM->Queue[iterator].no_of_requestsorfile],new_hostname);
            PtrToPM->Queue[iterator].request_mode[PtrToPM->Queue[iterator].no_of_requestsorfile] = new_mode;
            PtrToPM->Queue[iterator].no_of_requestsorfile++;
            break;
        } /* end if */
    } /* end for */

    /* Unlock the shared memory segment */
    Up(SEMAPHORE);
} /* end add_request_in_queue */
/* Checks if the queue is empty */
int is_queue_empty(char f_name[]) {
    /* Local declarations */
    int iterator = 0;
    int found = 0;

    /* Lock the shared memory segment */
    Down(SEMAPHORE);

    /* Checks if the list is empty */
    for(iterator = 0; iterator < PTrToPM->no_of_queue_nodes ; iterator++) {
        if(strcmp(PTrToPM->Queue[iterator].filename,f_name) == 0) {
            if (PTrToPM->Queue[iterator].no_of_request_for_file == 0) {
                found = 1;
                break;
            } /* endif */
        } /* end if */
    } /* end for */

    /* UnLock the shared memory segment */
    Up(SEMAPHORE);

    /* Return the status */
    return found;
} /*end is_queue_empty */
Adds the hostname of the client holding a copy of the file in its copyset */
void add_in_copyset(char f_name[], char new_hostname[])
{
    /* Local declarations */
    int iterator = 0;

    /* Lock the shared memory segment */
    Down(SEMAPHORE);

    /* Add the file name in the owner's list */
    for(iterator = 0 ; iterator < PtrToPM->no_of_list_nodes ; iterator++)
    {
        if(strcmp(PtrToPM->CI_list[iterator].filename, f_name) == 0)
        {
            strcpy(PtrToPM->CI_list[iterator].copyset_hostname[PtrToPM->CI_list[iterator].no_of_copies_of_file], new_hostname);
            PtrToPM->CI_list[iterator].no_of_copies_of_file++;
            break;
        } /* end if */
    } /* end for */

    /* UnLock the shared memory segment */
    Up(SEMAPHORE);
} /* end add_in_copyset */

/* Sends a delete message to all clients from its copyset */
void invalidate_copies(char f_name[], char hostname[], int tempSockDesc)
{
    int i = 0;
    int k = 0;
    int temp = 0;
    struct sockaddr_in CI_Addr;

    Down(SEMAPHORE);

    for(i = 0 ; i < PtrToPM->no_of_list_nodes ; i++)
    {

if(strcmp(PtrToPM->Cl_list[i].filename,f_name) == 0)
{
    for(k = 0 ; k < PtrToPM->Cl_list[i].no_of_copies_of_file ; k++)
    {
        InternetAddr(&CI_Addr, PtrToPM->Cl_list[i].copyset_hostname[k], CL_PM_PORTNO);
        pm_struct.message_type = 'D';
        SendDatagram(tempSockDesc,(char*)&pm_struct,sizeof(common_struct),&CI_Addr);
    }
    break;
}
}
temp = PtrToPM->Cl_list[i].no_of_copies_of_file;

Up(SEMAPHORE);

while (temp!= 0)
{
    RecvDatagram(tempSockDesc,(char*)&pm_struct,sizeof(common_struct),&CI_Addr);

    if(pm_struct.message_type == 'A')
    {temp--;
    }
}

Down(SEMAPHORE);

PtrToPM->Cl_list[i].no_of_copies_of_file = 0;
strcpy(pm_struct.owner_hostname,PtrToPM->Cl_list[i].owner_hostname);
strcpy(PtrToPM->Cl_list[i].owner_hostname,hostname);
pm_struct.message_type = 'G';
printf("\n Ownership of file %s is transferred from %s to %s ",f_name,pm_struct.owner_hostname,PtrToPM->Cl_list[i].owner_hostname);

Up(SEMAPHORE);
}

/* Forks a new process for a new file request */
void fork_new_process()
{  
  int NewSockDesc = 0;
  int StreamSockDesc = 0;
  int i = 0;
  int k = 0;
  int no_of_reads_serviced = 0;
  int fr_portno = 0;
  char temp_hostname[HOSTLENGTH];
  char temp_mode;
  char temp_filename[FILE_LENGTH];
  struct sockaddr_in CPMAddr;
  struct sockaddr_in C_Addr;
  struct sockaddr_in f_Addr;

  /*NewSockDesc = AcceptConnection(SockDesc);*/

  if(fork() == 0) /* Forks a child for a new file request */
  {
    CloseSocket(SockDesc);

    for(fr_portno = 1024; fr_portno < 5001; fr_portno++)
    {
      if((NewSockDesc = DatagramSocket(fr_portno)) != -1)
        break;
    }

    /* Attaching parent structure to sharem memory */

    PtrToPM = (parent_pm *)AttachMem();

    strcpy(temp_filename, pm_struct.filename);

    while(1)
    {
      while(fis_queue_empty(temp_filename))
      {
        Down(SEMAPHORE);
      }
for(i = 0; i < PtrToPM->no_of_queue_nodes ; i++)
{
    if(strcmp(PtrToPM->Queue[i].filename,temp_filename) == 0)
    {
        printf("Filename : %s ",PtrToPM->Queue[i].filename);
        strcpy(temp_hostname,PtrToPM->Queue[i].request_hostname[0]);
        printf("Requsting host : %s", temp_hostname);
        temp_mode = PtrToPM->Queue[i].request_mode[0];
        for (k = 0; k < PtrToPM->Queue[i].no_of_request_for_file - 1; k++)
        {
            strcpy(PtrToPM->Queue[i].request_hostname[k],PtrToPM->Queue[i].request_hostname[k+1]);
            PtrToPM->Queue[i].request_mode[k] = PtrToPM->Queue[i].request_mode[k+1];
        }
        PtrToPM->Queue[i].no_of_request_for_file--;
        break;
    }
}

Up(SEMAPHORE);

// Creating Internet Address for Client handling Page Manager's Request */
InternetAddr(&CPMAddr, temp_hostname, CL_PM_PORTNO);

if(temp_mode == 'R')
{
    pm_struct.message_type = 'G';
    add_in_copyset(temp_filename,temp_hostname);

    Down(SEMAPHORE);
}

129
for(i = 0; i < PtrToPM->no_of_queue_nodes ; i++)
{
    if(strcmp(PtrToPM->CI_list[i].filename,temp_filename) == 0)
    {
        strcpy(pm_struct.owner_hostname,PtrToPM->CI_list[i].owner_hostname);
        break;
    }
}

Up(SEMAPHORE);

pm_struct.mode = temp_mode;

SendDatagram(NewSockDesc,(char*)&pm_struct,sizeof(common_struct),&CPMAAddr);

no_of_reads_serviced++;
}
else
if(temp_mode == 'W')
{
    while(no_of_reads_serviced != 0)
    {
        RecvDatagram(NewSockDesc,(char*)&pm_struct,sizeof(common_struct),&C_Addr);
        if(pm_struct.mode == 'F')
            no_of_reads_serviced--;
    }

invalidate_copies(temp_filename,temp_hostname,NewSockDesc);

    pm_struct.mode = temp_mode;
SendDatagram(NewSockDesc,(char*)&pm_struct,sizeof(common_struct),&CPMAAddr);
    pm_struct.mode = 'Z';

    while(pm_struct.mode != 'F')
    {
        RecvDatagram(NewSockDesc,(char*)&pm_struct,sizeof(common_struct),&f_Addr);
    }
no_of_reads_serviced = 0;
}
} /* End of is Queue Empty While */
} /* End of while(1) */
}

/* Checks if a process is forked for that file */
int is_forked(char f_name[])
{
  /* Local declarations */
  int iterator = 0;
  int file_found = 0;

  /* Lock the shared memory segment */
  Down(SEMAPHORE);

  /* Check if the file is found */
  for(i = 0; i <PtrToPM->no_of_list_nodes ; i++)
  {
    if(strcmp(PtrToPM->Cl_list[i].filename,f_name) == 0)
    {
      file_found = 1;
      break;
    }/* endif */
  }/* end for */

  /*if file is found, check if a child process already exists */
  if(file_found == 1)
  {
    if(PtrToPM->Cl_list[i].forked == 'Y')
    {
      Up(SEMAPHORE);
      return 1;
    }/* end if */
  }/* end else */
{ 
    Up(SEMAPHORE);
    return 0;
} /* end else */
} /* end if */
else
{
    Up(SEMAPHORE);
    return 0;
} /* end else */
} /* end is_forked */

/* set file forked */
void set_file_forked(char f_name[])
{
    /* Local declarations */
    int iterator = 0;

    /* Lock the shared memory segment */
    Down(SEMAPHORE);

    /* Fork a child for every new file request */
    for(iterator = 0 ; iterator < PtrToPM->no_of_list_nodes ; iterator++)
    {
        if(strcmp(PtrToPM->Cl_list[iterator].filename,f_name) == 0)
        {
            PtrToPM->Cl_list[iterator].forked = 'Y';
            break;
        } /* end if */
    } /* end for */

    /* UnLock the shared memory segment */
    Up(SEMAPHORE);
} /* end set_file_forked */

/* Begin of main program */
int main(int ac, char *av[])

/* Local Declarations */
int pm_portno = 0;
char hostname[HOSTLENGTH];
struct sockaddr_in ClientAddr;
int portno = 0;
int Init[1];
int i = 0;

PtrToPM = (parent_pm *)SharedMem(sizeof(parent_pm));

PtrToPM->no_of_list_nodes = 0;
PtrToPM->no_of_queue_nodes = 0;

for (i = 0; i < SIZE; i++)
{
    PtrToPM->Queue[i].no_of_request_for_file = 0;
    PtrToPM->Cl_list[i].no_of_copies_of_file = 0;
    PtrToPM->Cl_list[i].forked = 'N';
}/* end for */

if(ac == 1)
{
    fprintf(stderr, "Format: %s integer\n", av[0]);
    exit(-1);
}

/* Reads the port number of PM from command line arg */
pm_portno = atoi(av[1]);

/* get the hostname of itself (page manager) */
if(gethostname(hostname, HOSTLENGTH) == -1)
    printf("\nGETHOSTNAME FAILED.");

Init[MUTEX] = 1;
Semaphores(Init, 1);
/* Listen from client */
SockDesc = DatagramSocket(pm_portno);

while(1)
{
    RecvDatagram(SockDesc,(char*)&pmjstruct,sizeof(commonj3truct),&ClientAddr);

    /* Gets the port number of the client */
    portno = PortNm(ClientAddr);

    /* Check for request type Info or Operation */
    if(pm_struct.req_type == 'I')
    {
        if(is_file_exists(pm_struct.filename))
        {
            pm_struct.owner_granted = 'N';
            SendDatagram(SockDesc,(char*)&pmjstruct,sizeof(commonjstruct),&ClientAddr);
        }
        else
        {
            pm_struct.owner_granted = 'Y';
            printf("Ownership granted to client %s for filename %s",
                   pmjstruct.client_hostname,pmjstruct.filename);
            fflush(stdout);
            SendDatagram(SockDesc,(char*)&pmjstruct,sizeof(commonjstruct),&ClientAddr);
            add_newfile_in_list(pm_struct.filename,pmj3truct.client_hostname,portno);
        }
    }
    else
    {
        if(pm_struct.req_type == 'O')
        {
            if(is_file_exists(pm_struct.filename))
            {
                pmjstruct.file_present = 'Y';
                SendDatagram(SockDesc,(char*)&pmjstruct,sizeof(commonjstruct),&ClientAddr);
            }
            else
            {
                pmjstruct.file_present = 'N';
                SendDatagram(SockDesc,(char*)&pmjstruct,sizeof(commonjstruct),&ClientAddr);
            }
            if(is_queue_exists(pm_struct.filename))
            {
                pmjstruct.queue_present = 'Y';
                SendDatagram(SockDesc,(char*)&pmjstruct,sizeof(commonjstruct),&ClientAddr);
            }
        }
    }
}
{ 
    add_request_in_queue(pm_struct.filename, pm_struct.client_hostname, pm_struct.mode);
}

else
{
    add_newfile_in_queue(pm_struct.filename, pm_struct.client_hostname, portno, pm_struct.mode);
}

if(!isJorked(pm_struct.filename))
{
    fork_new_process();
    set_file_forked(pm_struct.filename);
}
else
{
    pm_struct.file_present = 'N';
    SendDatagram(SockDesc,(char*)&pm_struct,sizeof(common_struct), &ClientAddr);
}

/* end of while */
return 0;
/* end of main */
/* Master's Project : Information Technology */
/* Author : Padmanabhan Raman */
/* File : client.h */
/* Description : This file contains all the global declarations required for the client.c file. */

/*Include required Libraries*/
#include "/usr/include/sys/time.h"

/*Pre-Processor Directives*/
#define TEXTLEN 80
#define SIZE 100
#define CL_CL_PORTNO 2300
#define CL_PM_PORTNO 2400
#define CL_SERV_PORTNO 2500
#define HOSTLENGTH 20

/*Global Declarations*/
typedef struct child
{
    char filename[TEXTLEN];
    char mode;
    char owner;
} child_struct;

typedef struct parent
{
    child_struct children[SIZE];
    int no_of_nodes;
    char granted;
    int pm_portno;
    char pm_hostname[TEXTLEN];
    int fork_process_portno;
} parent_struct;
/*Global Functions*/

int timer(int sock, int sec)
{
    /*Local declarations*/
    int n = 0;
    struct timeval timeout;
    fd_set rdset;

    timeout.tv_sec = sec;
    timeout.tv_usec = 0;
    FD_ZERO(&rdset);
    FD_SET(sock, &rdset);

    if(( n = select(sock+1, &rdset, 0, 0, &timeout )) < 0)
        printf("Error in timer.");

    if(n == 0)
    {
        printf("Request not granted yet.");
        return 0;
    }/*end if*/
    else
    return 1;
}/*end timer*/
/* Global Declarations */

/* This structure is used both by the client and the server */
typedef struct common {
    char filename[80];
    char mode;          /* R = Read, W = Write, F = Free */
    char req_type;      /* O = Operation, I = Information */
    char owner_hostname[80];
    int owner_portno;
    char client_hostname[80];
    int cl_pm_portno;
    int cl_serv_portno;
    char message_type;  /* G = Granted, D = Delete from list, */
    /* A = Acknowledge Delete */
    /* R = Request */
    char message_from;  /* C = Client, P = Page manager */
    char file_present;  /* Y = Yes, N = No */
    char owner_granted; /* Y = Yes, N = No */
} common_struct;
/* Master's Project : Information Technology */
/* Author : Padmanabhan Raman */
/* File : page_manager.h */
/* Description : This client contains the declaration of the */
/* structure used by the page manager. */

#include <stdio.h>
#include "/usr/local/pub/atk/ipC/include/ipC.h"
#include "common.h"

/*Pre-Processor Directives*/
#define TEXTLEN 80
#define SIZE 100
#define CL_CL_PORTNO 2300
#define CL_PM_PORTNO 2400
#define CL_SERV_PORTNO 2500
#define HOSTLENGTH 20
#define MAX_COPIES 20
#define FILE_LENGTH 20

/*Global declarations*/
typedef struct queue
{
    char filename[TEXTLEN];
    char request_hostname[MAX_COPIES][HOSTLENGTH];
    int request_portno[MAX_COPIES];
    char request_mode[MAX_COPIES];
    int no_of_request_for_file;
} queue_struct;

typedef struct client_list
{
    char filename[TEXTLEN];
    char copyset_hostname[MAX_COPIES][HOSTLENGTH];
int copyset_portno[MAX_COPIES];
char owner_hostname[HOSTLENGTH];
int owner_portno;
char file_status; /* Not required */
char owner_mode; /* Not required */
int no_of_copies_of_file;
char forked; /* Y = Yes  N = No */
} client_list_struct;

typedef struct pm_parent
{
  queue_struct    Queue[SIZE];
  client_list_struct  Cl_list[SIZE];
  int no_of_queue_nodes;
  int no_of_list_nodes;
} parent_pm;
IPC_LIB = /usr/local/pub/atk/ipC/lib/sgi_libipc.a
INC = /usr/local/pub/atk/ipC/include/ipC.h

all : client c_client c_pgmng c_server page_manager

client: client.c $(INC) client.h
    cc client.c $(IPC_LIB) -o client

c_client: c_client.c $(INC) common.h client.h
    cc c_client.c $(IPC_LIB) -o c_client

c_pgmngr: c_pgmngr.c $(INC) common.h client.h
    cc c_pgmngr.c $(IPC_LIB) -o c_pgmngr

c_server: c_server.c $(INC) common.h client.h
    cc c_server.c $(IPC_LIB) -o c_server

page_manager: page_manager.c $(INC) page_manager.h
    cc page_manager.c $(IPC_LIB) -o page_manager
Appendix C: Simulation Results

The goal behind development of the ‘Page Based Distributed Shared Memory’ was to test and determine how the system would help prevent consistency problems in a distributed database system. The entire simulation test was performed in the Computer Science Labs on the SUN workstations.

The simulation test was performed with 5 different client workstations namely kansas, newhampshire, rhodeisland, illinois and arizona. A Centralized Page Manager running on a dedicated server machine or workstation namely Georgia. Files were used instead of pages to represent tables in terms of a distributed database system. The simulation test was done using 5 different files namely faculty.txt, raman.txt, vikas.txt, test.txt and student.txt.

The output scripts from each of the client machine or workstation and the server or ‘Page Manager’ is as shown below:

I stands for Information and O stands for Operation. R stands for Read mode and W stands for Write mode. To run the client, 3 parameters that are required are as follows: <Client Executable> <Page Manager Hostname> <Page Manager Port Number>

- Output Script File for Client Kansas:

Script started on Mon Aug 02 21:17:52 1999
kansas% hostname
kansas
kansas% client georgia 3000
Which operation you want to perform? (I/O): I
Enter filename: faculty.txt
Enter mode (R/W): R
Adding file faculty.txt in the copyset.
Which operation you want to perform? (I/O): O
Enter filename: raman.txt
Enter mode (R/W): R
Request not granted yet.
DATA READ FROM FILE: N
Have you finished reading? Y
Which operation you want to perform? (I/O): O
Enter filename: raman.txt
Enter mode (R/W): R
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
DATA READ FROM FILE: O
Have you finished reading? Y
Which operation you want to perform? (I/O):
kansas% exit
Script done on Mon Aug 02 21:35:24 1999
Output Script File for Client Newhampshire:

newhampshire% script client_newhampshire
Script started, file is client_newhampshire
newhampshire% hostname
newhampshire
newhampshire% client georgia 3000
Which operation you want to perform? (I/O): I
Enter filename: test.txt
Enter mode (R/W): W
Adding file test.txt in the copyset.
Which operation you want to perform? (I/O): O
Enter filename: faculty.txt
Enter mode (R/W): R
Request not granted yet.
DATA READ FROM FILE: N
Have you finished reading? Y
Which operation you want to perform? (I/O): O
Enter filename: vikas.txt
Enter mode (R/W): R
Request not granted yet.
DATA READ FROM FILE: N
Have you finished reading? Y
Which operation you want to perform? (I/O): O
Enter filename: raman.txt
Enter mode (R/W): W
Request not granted yet.
ENTER AN INTEGER YOU WANT TO WRITE: 79

Have you finished writing? Y

Which operation you want to perform? (I/O): 0

Enter filename: raman.txt

Enter mode (R/W): W

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

Request not granted yet.

ENTER AN INTEGER YOU WANT TO WRITE: 78

Have you finished writing? Y

Which operation you want to perform? (I/O):

newhampshire% exitScript done, file is client_newhampshire
Script done on Mon Aug 02 21:35:50 1999

- **Output Script File for Client RhodeIsland:**

  Script started on Mon Aug 02 21:19:17 1999
  rhodeisland% hostname
  rhodeisland
  rhodeisland% client georgia 3000
  Which operation you want to perform? (I/O): O
  Enter filename: faculty.txt
  Enter mode (R/W): R
  Request not granted yet.
  DATA READ FROM FILE: N
  Have you finished reading? Y
  Which operation you want to perform? (I/O): I
  Enter filename: raman.txt
  Enter mode (R/W): W
  Adding file raman.txt in the copyset.
  Which operation you want to perform? (I/O): O
  Enter filename: vikas.txt
  Enter mode (R/W): R
  Request not granted yet.
  DATA READ FROM FILE: N
  Have you finished reading? Y
  Which operation you want to perform? (I/O): O
  Enter filename: raman.txt
  Enter mode (R/W): R
Request not granted yet.

DATA READ FROM FILE: 0

Have you finished reading? Y

Which operation you want to perform? (I/O):

rhodeisland% exit

Script done on Mon Aug 02 21:36:11 1999

- **Output Script File for Client Arizona:**

  Script started on Mon Aug 02 21:21:30 1999

  arizona% hostname

  arizona

  arizona% client georgia 3000

  Which operation you want to perform? (I/O): I

  Enter filename: student.txt

  Enter mode (R/W): W

  Adding file student.txt in the copyset.

  Which operation you want to perform? (I/O): I

  Enter filename: vikas.txt

  Enter mode (R/W): R

  Adding file vikas.txt in the copyset.

  Which operation you want to perform? (I/O): O

  Enter filename: faculty.txt

  Enter mode (R/W): R

  Request not granted yet.

  DATA READ FROM FILE: N

  Have you finished reading? Y

  Which operation you want to perform? (I/O): 0
Enter filename: raman.txt
Enter mode (R/W): R

Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
DATA READ FROM FILE: N
Have you finished reading?
arizona% exit
Script done on Mon Aug 02 21:34:44 1999

- **Output Script File for Client Illinois:**

Script started on Mon Aug 02 21:17:52 1999
illinois% hostname
illinois
illinois % client georgia 3000
Which operation you want to perform? (I/O): 0
Enter filename: raman.txt
Enter mode (R/W): R
Request not granted yet.
DATA READ FROM FILE: N
Have you finished reading? Y
Which operation you want to perform? (I/O): 0
Enter filename: raman.txt
Enter mode (R/W): R
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
Request not granted yet.
DATA READ FROM FILE: N
Have you finished reading? Y
Which operation you want to perform? (I/O):
illinois% exit
Script done on Mon Aug 02 21:35:24 1999

- **Output Script File for Page Manager Georgia:**

Script started on Mon Aug 02 21:17:26 1999
georgia% hostname
georgia
georgia% page_manager 3000
Ownership granted to client arizona for filename student.txt
Ownership granted to client kansas for filename faculty.txt
Ownership granted to client newhampshire for filename test.txt
FILENAME: faculty.txt
REQUESTING HOST: rhodeisland
REQUESTING MODE: R
Adding file faculty.txt in copyset for host rhodeisland.
Ownership of file faculty.txt is retained with the previous owner kansas.

Ownership granted to client rhodeisland for filename raman.txt

Ownership granted to client arizona for filename vikas.txt

FILENAME: faculty.txt
REQUESTING HOST: newhampshire
REQUESTING MODE: R

Adding file faculty.txt in copyset for host newhampshire.

Ownership of file faculty.txt is retained with the previous owner kansas.

FILENAME: faculty.txt
REQUESTING HOST: arizona
REQUESTING MODE: R

Adding file faculty.txt in copyset for host arizona.

Ownership of file faculty.txt is retained with the previous owner kansas.

FILENAME: raman.txt
REQUESTING HOST: kansas
REQUESTING MODE: R

Adding file raman.txt in copyset for host kansas.

Ownership of file raman.txt is retained with the previous owner rhodeisland.

FILENAME: raman.txt
REQUESTING HOST: illinois
REQUESTING MODE: R

Adding file raman.txt in copyset for host illinois.

Ownership of file raman.txt is retained with the previous owner rhodeisland.
FILENAME: vikas.txt

REQUESTING HOST: newhampshire
REQUESTING MODE: R
Adding file vikas.txt in copyset for host newhampshire.
Ownership of file vikas.txt is retained with the previous owner arizona.

FILENAME: vikas.txt
REQUESTING HOST: rhodeisland
REQUESTING MODE: R
Adding file vikas.txt in copyset for host rhodeisland.
Ownership of file vikas.txt is retained with the previous owner arizona.

FILENAME: raman.txt
REQUESTING HOST: newhampshire
REQUESTING MODE: W
File raman.txt deleted by host kansas
File raman.txt deleted by host illinois
Ownership of file raman.txt is transferred from rhodeisland to newhampshire in write mode.
File raman.txt deleted by host rhodeisland

FILENAME: raman.txt
REQUESTING HOST: kansas
REQUESTING MODE: R
Adding file raman.txt in copyset for host kansas.
Ownership of file raman.txt is retained with the previous owner newhampshire.

FILENAME: raman.txt
REQUESTING HOST: rhodeisland
REQUESTING MODE: R

Adding file raman.txt in copyset for host rhodeisland.

Ownership of file raman.txt is retained with the previous owner newhampshire.

FILENAME: raman.txt

REQUESTING HOST: newhampshire

REQUESTING MODE: W

File raman.txt deleted by host kansas

File raman.txt deleted by host rhodeisland

Ownership of file raman.txt is retained with the previous owner newhampshire.

FILENAME: raman.txt

REQUESTING HOST: arizona

REQUESTING MODE: R

Adding file raman.txt in copyset for host arizona.

Ownership of file raman.txt is retained with the previous owner newhampshire.

FILENAME: raman.txt

REQUESTING HOST: illinois

REQUESTING MODE: R

Adding file raman.txt in copyset for host illinois.

Ownership of file raman.txt is retained with the previous owner newhampshire.

gorgia% exit

Script done on Mon Aug 02 21:36:33 1999
References:


