Reducing the Inter-Process Communication Time on Local Host by Implementing Seamless Socket like, “low latency” Interface over Shared Memory

(A Messaging System)

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science and Engineering

By:
Mauli Gulati
(80732028)

Under the supervision of:
Dr. Deepak Garg
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

JUNE 2009

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, **"Reducing the Inter-Process Communication Time on Local Host by Implementing Seamless Socket like, "low latency" Interface over Shared Memory"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Dr. Deepak Garg* and refers other researcher's works which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

Mauli Gulati

(Mauli Gulati)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Deepak Garg

(Dr. Deepak Garg)
Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by

Poonam Bawa 01/07/2009

(HEAD)
Computer Science & Engineering Department
Thapar University
Patiala.

R K Sharma 9/7/09

(DR. R.K.SHARMA)
Dean (Academic Affairs)
Thapar University
Patiala.

ACKNOWLEDGMENT

First and foremost, I acknowledge my heartiest thanks to my guide, Dr. Deepak Garg, Assistant Professor, Computer Science and Engineering Department, Thapar University, Patiala, for encouraging me and giving me tremendous guidance during my thesis work. Without his mentoring, this thesis would never have been realized.

I am thankful to Dr. Rajesh Bhatia, Assistant Professor and Head, Computer Science and Engineering Department, Thapar University, Patiala, for providing excellent infrastructural facilities and motivation that helped me in progressing towards the completion of my thesis work. I also thank Dr. Seema Bawa, Professor, Computer Science and Engineering Department, Thapar University, Patiala, for the same.

I am also thankful to Dr. Inderveer Chana, P.G. Coordinator, Computer Science and Engineering Department, for the motivation and inspiration that triggered me for my work.

I also thank all the faculty members and my friends who were always there at the need of hour and provided with all the help and facilities, which I required for the completion of my thesis work.

I am also thankful to the authors and researchers whose works I have consulted and quoted in this work.

Last but not the least I thank God, my husband and my parents for not letting me down at the time of crisis and showing me the silver lining in the dark clouds.

Mauli Gulati

Mauli Gulati

ABSTRACT

The work done here is inspired by businesses today which are increasingly dependent on a connected world and the real-time flow of information across systems. Since there is a need for proper integration of business processes we require an efficient messaging system. There are few customized messaging solutions available which are using shared memory for communication and some already existing messaging solutions for fast and reliable message communication but they use socket underneath both for communication between processes on local host or on remote hosts which results in somewhat overkill of time and resources used in communication for those on same host. To overcome such problem and to achieve maximum throughput on local host we aimed at using shared memory which is fastest known inter-process communication mechanism that would help in reducing the inter-process communication time on local host by implementing fast socket over shared memory.

The notion here is to implement an abstraction layer that encapsulates Shared Memory based Communication Interface and Socket based Communication Interface into one Seamless Interface. The Seamless Interface would help in selecting an appropriate transport based on the locality of the processes. Shared memory would be automatically used by an application if processes on the same host need to communicate. If communication between processes on different hosts is required then socket would be used automatically. This all would be beneath our abstraction layer. For a programmer it would be equivalent to using any other communication library instead of socket interfaces.

At this time, we do not have an equivalent solution in the market. This innovative solution is all set to change the way industry communicates in between processes. The solution tries to give communication on local host its deemed advantage. The resultant system shall result in an extreme low latency, and would be used by the commercial organizations.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER – 1
INTRODUCTION

1.1 General Introduction

Businesses today are increasing dependent on real-time flow of information and connected world. Transferring of information has no relevance if it doesn't reach destination at desired speed. Therefore, transferring of information at faster speed is the necessity. Especially in the financial market where a delay of even a single micro-second could lead to loss of millions of money and could even compel the investors to withdraw money from the market. Hence, timely information is a growing demand of market. Therefore, in order to have faster communication we need to have an effective messaging system.

A communication can takes place between processes on local host or on remote host through number of ways. In present market there are number of messaging solutions available. But all of them use sockets underneath for communication between processes on local host or on remote host. Shared memory is the fastest known inter-process mechanism and there are certain solutions available in market but all those solutions are customized which means they are designed and used by few programmers to meet their own needs. There is no integrated solution available in the market that could encapsulate the best features of both sockets and shared memory into one and use them for their designated tasks. Here we aimed at proposing and designing a Shared Memory Transport Interface in addition to an interface that would actually encapsulate Shared Memory based Communication Interface and Socket based Communication Interface into one Seamless Interface. It would also let the user communicate with an ease without knowing the underlying complexities.

So the main question here is do we really need an interface that would encapsulate the best feature of both shared memory and sockets. And the answer is definitely 'yes', because this would lead to an interface that would allow programmer to use the shared memory for local communication and sockets for remote communication without knowing the complexities of underlying complex design. They would be able to use it

like any other library they are presently using for communication. Programmers can focus on their designated task without being worried about the synchronization problems between processes using shared memory for local communication.

1.2 Background

Shared memory has been used for the inter-process communication in many applications since it is the fastest known inter-process mechanism known so far but all these applications are customized, as we already stated. Similarly sockets are also used for communication between the processes on local host as well as for the communication between the processes on remote host. But there is no integrated solution available in the market that could encapsulate the best features of both sockets and shared memory based inter-process communication. There is a requirement to design a Seamless Interface that would encapsulate Socket based Interface and Shared Memory based Interface into one Seamless Interface to achieve maximum performance. The Seamless Interface would allow programmer to use the best features of both the worlds. Furthermore, it would automatically use Socket Interface if the communication between remote processes is required and if communication between local processes is required then in that case Shared Memory Interface would be used automatically. Moreover, there is also no Shared Memory Transport exists in market that could provide extremely well abstraction that could hide all complexities from user and provide a simple and easy to use interface.

1.3 Shared Memory

A shared memory is a piece of memory that is attached to the address spaces of the processes participating in communication. As a result, all of these processes share the same memory segment and have access to it. Each task or process, executes in its own private memory address space without knowledge of the address spaces of other tasks that execute concurrently [2]. Figure 1 shows two processes and their address spaces. The shared memory is attached to both address spaces and both Process 1 and Process 2, can have access to this shared memory as if it's the part of their own address spaces. It looks like as if, the original address spaces are extended by attaching the shared memory. One process must explicitly ask for an area, using a *key*, to be shared by other processes. This process will be called a Writer Process. All other processes,

the Reader Processes, which know the shared area, can access it. However, there is no protection to a shared memory and any process that knows about its existence can access it freely. To protect a shared memory from being simultaneously accessed by several processes, a synchronization procedure is used. In short, once the memory is being shared, there are no checks on how the processes are using it and processes must synchronize access to the memory by using any of the synchronization primitive for example, System V semaphores etc.

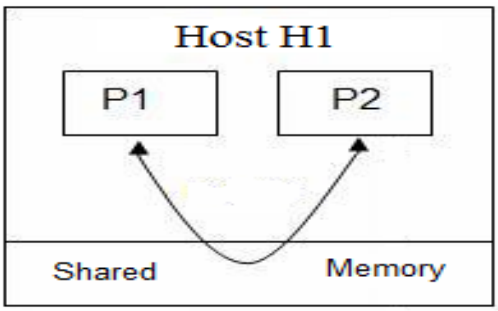

Figure 1: Shared Memory

Each newly created shared memory area is represented by a *shmid_ds* data structure. It describes the size of shared memory region, number of processes using it and other related information. The *shmid_ds* data structure would be discussed later in detail.

1.3.1 Universal Method of Using Shared Memory

Shared Memory is used to facilitate effective communication between multiple processes on local host. In following section we focused on demonstrating the communication between two processes and for this we have named two processes as server process and client process. In context with our proposed solution it resembles Writer Process and Reader Process respectively.

1.3.1.1 Server Process Accountability

The Server Process should be started before any client and performs following tasks:-

1. Request for a shared memory with a memory *key* and store the returned shared memory ID. This is performed by calling *shmget()*.
2. By calling *shmat()* shared memory is attached to the server's address space.
3. If required, initialize the shared memory.
4. Perform required task and wait for all client’s completion.

5. By calling *shmdt(), d*etach a process's address space from the shared memory.
6. By calling *shmctl(),* remove the shared memory segment with appropriate command.

1.3.1.2 Client Process Accountability

The Client Process follows the following steps:-

1. Request for a shared memory segment with the same memory key and remember the returned shared memory ID.
2. Attach this shared memory segment to the client's address space.
3. Utilize the memory.
4. If required, detach all shared memory segments.
5. Exit.

1.4 Increasing Importance of Shared Memory Based Inter-Process Communication

Shared memory technology allows arbitrary processes to exchange data and synchronize execution. It is the fastest form of inter-process communication mechanism known so far because processes do not execute any system calls into the kernel for sharing data between processes that are sharing the memory area. Shared memory allows two or more processes to share a region of memory but they must coordinate and synchronize their use of the shared memory between themselves to prevent any data loss.

Shared Memory is a memory that can be concurrently accessed by multiple processes to facilitate communication among them and to avoid redundant copy operations. Since processes can access the shared memory area like regular working memory, this is a very fast way of communication since it eliminates unnecessary copy operations and context switches (as opposed to other mechanisms of IPC such as named pipes, sockets etc.). Therefore, for the communication between processes on local host shared memory is the best inter-process mechanism. And it definitely leads to faster communication and would certainly help those organization where real-time processing and fast communication is a necessity.

1.5 Significance of Our Proposed Shared Memory Transport

Our proposed Shared Memory Transport is designed as a simplified transport which would hide the complexities involved in using shared memory by providing appreciable level of abstraction. It works on a concept of "Fire and Forget" because of the ease that it offers to user in the form of simple interfaces. This design supports single Writer Process and multiple Reader Processes. However, there could be multiple Writer threads within Writer Process. Moreover, it is a scalable design since there is no constraint on number of readers that can connect to Writer shared memory segment and size of the shared memory is also configurable according to the application needs.

It demonstrates a true example of flexible design since it supports varying size multiple queues as per receiver's application requirement. These queues are designed in a circular lock-free fashion. Moreover, there is no lock between Writer and Reader Processes. The Writer Process can write on head_index of queue and Reader Process can read from tail_index of queue any time without locking queue, provided space is there or data are there respectively.

It would result in an efficient system which would be easy to use and would be used by many commercial organizations with an ease and programmers would be able to use it as any other library without effecting their performance and delivery time. Moreover, use of atomic operations would substantially provide extremely low-latency system that would perform extremely well. The details of our Shared Memory Transport would be discussed in Chapter-4. The Shared Memory Transport along with our Seamless Interface would surely result in a low-latency system and would provide sufficient justification to switch from socket based communication on local host to shared memory based communication.

1.6 Importance of our Proposed Seamless Interface

The Seamless Interface proposed and designed by us is basically used to encapsulate the Shared Memory based Communication Interface and Socket based Communication Interface. In present market there is no integrated solution available which is intelligent enough to identify the locality of process and initiates the

communication between the processes based on their locality. Our Seamless Interface is designed keeping in mind the growing demand for local host and remote host communication. But at present all existing solutions are using sockets both for communication between processes on local host or on remote host because of the flexibility and transparency provided by sockets. But sockets are not good choice for communication across processes on local host because communication via sockets involves kernel which results in more copy operations and more context switches which finally affect the performance of the system.

1.7 Inter-Process Communication

Inter-Process Communication (IPC) is a combination of various methods for the exchange of data between processes on local host or on remote host. IPC has traditionally been the responsibility of the kernel, but kernel-based IPC suffers from a problem that is its performance is architecturally limited by the cost of invoking the kernel [3].

1.7.1 IPC Mechanisms

IPC mechanisms illustrate different ways of sharing information between different processes that are running on some operating system. A particular IPC mechanism can be selected based on the bandwidth and latency of communication and the kind of data being communicated between the processes.

There are various inter-process communication mechanisms which could be used for communication between processes on local host or on remote host depending on the required scenario and features provided by different inter-process communication methods.

1.7.1.1 Unnamed Pipes

This IPC mechanism can be used only for related processes and it allows the flow of data only in one direction. In this case data is buffered from the output process until the input process receives it. Though it is reasonably fast, however since the kernel manages the inter-process synchronization hence, it performs relatively slow.

1.7.1.2 Named Pipes

Named Pipe which is also known as FIFO has a specific name or a pathname name associated with it. It can be used for communication between related or unrelated processes and between the processes that are on different computers. However, even in this case also kernel manages inter-process synchronization hence, it performs relatively slow. FIFO offers only a unidirectional data channel [5].

1.7.1.3 Message Queues

Message queue is an asynchronous communication mechanism which means that the sender and receiver of the message need not interact with the single or multiple message queues, managed by kernel, simultaneously. This facilitates storing of messages in the queue until the receiver retrieves them. Even in this case also, kernel manages inter-process synchronization so the speed is limited by kernel resource contention.

1.7.1.4 POSIX Shared Memory

POSIX shared memory allows the exchange of data between related and unrelated processes through a defined area of memory, technically called shared memory. This doesn't rely on kernel for synchronization between communicating processes rather it is the responsibility of the application program to synchronize access on their own by using any of the synchronization primitives. The unrelated processes can communicate and share memory using any of the following ways provided by POSIX.1 standard:-

1. **Memory-mapped Files:** A file is opened by *open()*. This system call returns a descriptor which is then mapped into the address space of the process by using *mmap()* system call.
2. **Shared Memory Objects:** It uses *shm_open()* to either create a new shared memory object or to open an existing one and returns a descriptor that is then mapped into the address space of the process by using *mmap()* system call.

1.7.1.5 System V Shared Memory

System V shared memory allows the sharing of data between processes through a common region in memory. Processes requiring communication to exchange information can attach to the memory segment and gain access to the data contained

in the segment. However, in this case a shared memory specific data structure called *shmid_ds* is maintained and populate by kernel. Hence, it facilitates finer control as compare to POSIX shared memory. Moreover, it also doesn't rely on kernel for synchronization between communicating processes rather it is the responsibility of the application program to synchronize access on their own by using any of the synchronization primitives.

1.7.1.6 Doors

Doors allow a process to call a procedure in another process on the *same host*. A server process creates a door for a particular procedure available within so that other client processes can call that procedure.

1.7.1.7 RPC

Remote Procedure Call facilitates sharing of information between the processes on *different hosts* connected by some form of network. It allows a client process on one host to call a server process procedure on another host.

1.7.1.8 Sockets

The Socket interface was originally developed in BSD UNIX to provide an interface to the TCP/IP protocol suite [6]. Internet socket or network socket or socket is used for inter-process communication. A socket is one end of a two-way communication link between two programs running on the network.

A socket address is the combination of an IP address and a port number. When the sockets are used for exchanging information and *socket()* system call is used, it returns a unique integer number called socket identifier or socket number.

The socket identifier ensures delivery of incoming data packets to the appropriate application process or thread. This is the most popular interface used for the communication between the processes on same host or on remote host. The reason for this popularity is ease of usage and seamless connectivity, irrespective of location of target process.

1.8 Synchronization Primitives

The synchronization between processes is normally needed to allow the sharing of data between processes or threads on same host or on remote host with an ease and without the lost of information. Now since our solution is meant to improve communication time in-between processes on same host, it makes sense to discuss the synchronization primitives. Following are the different ways for synchronization between processes:-

1.8.1 Mutexes

Mutual Exclusion is the most basic synchronization primitive. It ensures that if one process is executing the code in critical region then no other process should be allowed to access that critical region. Critical region basically contains the data that is being shared between multiple processes and actually it's the data that is being protected. If any process wishes to manipulate or access the data inside the critical region then it must acquire the mutex lock.

1.8.2 Condition Variable

Condition Variable which is associated with mutex is a building block of synchronization. Basically, it is used to synchronize processes depending on the outcome of some conditional test. A process waits on a condition variable if after acquiring a mutex lock it realizes that it needs to wait for some condition to be true. It can release an acquired lock and go into a sleep state in a single atomic operation.

1.8.3 Read-Write Locks

A mutex lock allows only one thread to enter a critical region. But we can allow multiple threads to access critical region based on following read-write locks:-

1. Any number of threads can hold a read-write lock for reading if no other thread is holding the read-write lock for writing.
2. A thread can hold a read-write lock for writing if no other thread holds the read-write lock for reading and writing.

1.8.4 Record Locking

Record locking which is maintained by kernel can be used by related or unrelated processes to share the reading or writing of a locked file which is referenced through a

descriptor. The owner of a lock is identified by its process ID, therefore, this type of locking cannot be used for threads.

1.8.5 System V Semaphores

A semaphore is an IPC mechanism which provides synchronization between various processes or threads. System V semaphores provides a set of counting semaphores which means one or multiple counting semaphores per set. A set has a limitation of having minimum one and maximum 25 semaphores only.

1.8.6 POSIX Semaphores

Like System V, POSIX also provide a counting semaphore but POSIX semaphores mean single counting semaphore which need not be maintained in the kernel. It is used to synchronize processes or threads and can be of two types:-

1. **POSIX named semaphores** can be used for related or unrelated processes and are identified by POSIX IPC names or pathnames in the filesystem.
2. **POSIX memory-based semaphores** are stored in shared memory and are used to synchronize processes which are communicating through shared memory.

1.9 Difference between a System V and a POSIX semaphore

S.No.	System V Semaphore	POSIX Semaphore
1.	In System V, we can control how much the semaphore count can be increased or decreased.	In POSIX, the semaphore count can be increased or decreased by 1.
2.	It allows changing the permissions of semaphores.	It does not allow manipulation of semaphore permissions.
3.	Though it is complex from usage perspective but it offers finer control.	It is straight-forward and simple.
4.	After creating System V semaphore user has to explicitly initialize it.	It allows initialization and creation of semaphores in a single step which means it's atomic.

S.No.	System V Semaphore	POSIX Semaphore
5.	Semaphore creation is expensive in System V semaphores because it creates an array of semaphores when creating a semaphore object.	It is not expensive since it creates only one semaphore.
6.	It provides a mechanism for system-wide semaphore.	It provides a mechanism for process-wide semaphores. Semaphore is automatically cleaned up when process exits.

Table 1: System V Semaphore vs. POSIX Semaphore [7]

1.10 Atomic Operations

An atomic operation refers to a group of operations that are combined and appear as a single operation. The output of the atomic operation is either success or failure. In short, atomic operations are those that cannot be interrupted while accessing any resource like memory location. Atomic operations operate on two conditions:-

1. If one process is executing atomic operations, no other process can execute the same atomic operations and cannot see the changes being made.
2. If atomic operation fails then system's state is restored to original state i.e. the state it was in, prior to executing any atomic operation.

Let us understand the concept of atomic operation with the help of trivial example. Consider two processes Process 1 and Process 2 are running and they both want to increment a value at same shared memory location:-

Step 1: Process 1 reads the value in memory location.
Step 2: Process 1 then increment the value.

Process 1 suspended before writing back the incremented value in the memory location and Process 2 starts:-

Step 1: Process 2 reads the original value in memory location.
Step 2: Process 2 increments the value.
Step 3: The Process 2 writes the new value into the memory location.

The Process 2 is suspended and the Process 1 starts:-

Step 1: Now the Process 1 is unaware that Process 2 has already updated the value in the memory location and writes a wrong value into the memory location.

From the above example it's clear that if we would have used atomic operation then reading, incrementing and writing would have been done in single step and would have not allowed other process to access that value if other process is already using it [9].

1.10.1 Common Atomic Operations

Following are the common atomic operations which are used to maintain consistency in the system:-

1.10.1.1 Atomic read and write

It is an atomic operation which allows reading a memory location and writing a new value into it simultaneously. This operation is used to prevent race conditions in multi-threaded applications.

1.10.1.2 Test and Set

Test-and-Set instruction is used to set the value in a memory location but before setting the value it performs some test. However, the value is set irrespective of the result of the test. If there are multiple processes and if a process is performing test-and-set then no other process is allowed to perform another test-and-set until the first process is completed.

1.10.1.3 Compare and Swap

The CPU instruction compare-and-swap is used to compare the contents of a memory location to a given value and, if they are same, it modifies the contents of that memory location to a given new value. It either returns a simple boolean response or

the value initially read from the memory location, to indicate the result of the operation.

1.10.1.4 Fetch and Add

The CPU instruction fetch-and-add is used to modify the contents of a memory location. It is significantly used in multi-processor systems where it's difficult and undesirable to disable interrupts on all processors at the same time. It prevents multi-processor collision by permitting any processor to atomically increment a value in memory location.

1.10.1.5 Load-Link / Store-Conditional

Load-link also known as "load and reserve" and store-conditional are a pair of instructions that work jointly to ensure a lock-free atomic read-modify-write operation. Load-link returns the recent value of a memory location. A subsequent store-conditional will store a new value in that memory location only if no modifications have taken place to that location since the load-link otherwise, it will fail.

1.11 Locking

Critical sections are protected by locks but locks are accompanied with extra overhead in terms of low performance, processes have to wait until lock is released. An atomic operation is functionally equivalent to a lock and many computer architectures offer dedicated support, moreover, atomic operations are faster than locks. Program performance is improved, if simple critical sections are replaced with atomic operations for non-blocking synchronization which ensures that execution of a process competing for a shared resource is not postponed indefinitely by mutual exclusion.

For example, consider two processes, P1 and P2, use a lock to access a counter count:-

Step 1: lock (count)

Step 2: count ← count + 1

Step 3: unlock (count)

1.12 Organization of Thesis

The Chapter 1 INTRODUCTION provides the general introduction to what the thesis is all about. It also briefly summarizes the question and the reasons why it is a worthwhile question. It also includes a brief section giving background information related to this thesis work for all the intended readers.

The Chapter 2 LITERATURE SURVEY describes the research or work done in inter-process communication field. It provides details on various existing messaging solutions, various existing models in addition to few existing patents in this field.

The Chapter 3 PROBLEM STATEMENT describes the current picture and the missing part. It also includes the solution to existing problem of local host inter-process communication. The goal of this thesis work is also described in this chapter.

The Chapter 4 PROPOSED SYSTEM DESIGN describes in detail the complete design of Seamless Interface and Shared Memory Transport along with few designed algorithms.

The Chapter 5 CONCLUSION AND FUTURE SCOPE concludes the thesis followed by the future scope highlights.

CHAPTER – 2
LITERATURE SURVEY

This chapter describes the methods that were used to attain the final goal of the thesis or rather to explain the procedure. The methodology that we followed here was to study the existing system and find out pros and cons of prevailing techniques used for communication between the processes on local host and for the communication between processes on remote host.

I discussed the present market trend with the people from industry and based on their comments and suggestions, I realized that there is a need to have a new messaging solution that can fit in this vacuum and which could fulfill the growing demand of fast communication between processes on local host. It was also realized that there is no integrated solution in market which could encapsulate the best features of shared memory based communication and socket based communication into one Seamless Interface which could intelligently select appropriate transport based on the locality of processes.

But to design the shared memory based interface it was important to understand the concept of shared memory in detail especially provided by System V and POSIX, how it works, various important data structures, analyzing existing market trends, prevailing systems and other parameters related with easy designing of our proposed Seamless Interface and Shared Memory Transport.

2.1 System V Shared Memory

On a Solaris system, shared memory is an extremely efficient means of sharing data among multiple processes since the data need not actually be moved from one process address space to another. Shared memory leads to the sharing of the same physical memory (RAM) pages by multiple processes, such that each process has mappings to the same physical pages and can access the memory through pointer dereferencing in code.

Shared memory implementation in Solaris would be discussed in detail in subsequent section. In contrast with POSIX Shared Memory, System V Shared Memory supports the data structure *shmid_ds* which is maintained and populated by kernel. For each created shared memory segment there is an associated *shmid_ds* data structure. This data structure contains the complete information about the shared memory it belongs to. [10]

2.1.1 shmid_ds Data Structure

The kernel maintains a unique data structure for every shared memory segment which exists within its addressing space.

```
struct shmid_ds
{
        struct    ipc_perm    shm_perm;
        size_t    shm_segsz;
        size_t    shm_lkcnt;
        pid_t     shm_lpid;
        pid_t     shm_cpid;
        shmat_t   shm_nattch;
        shmat_t   shm_cattch;
        time_t    shm_atime;
        time_t    shm_dtime;
        time_t    shm_ctime;
};
```

Following table, describes each component of *shmid_ds* Data Structure:-

Member Name	Data type	Description
shm_perm	structure	ipc_perm structure maintains permission information.
shm_segsz	unsigned int	Size of the shared segment in bytes.

Member Name	Data type	Description
shm_lkcnt	unsigned short	Number of locks on the shared segment.
shm_lpid	long	Last process PID, which performed a shared memory operation.
shm_cpid	long	Shared memory creator Process PID.
shm_nattch	unsigned long	Number of attaches to the shared segment.
shm_cnattch	unsigned long	Number of ISM attaches to shared memory.
shm_atime	long	Time of last attach to shared segment
shm_dtime	long	Time of last detach from shared segment
shm_ctime	long	Time of last change to shmid_ds structure

Table 2: shmid_ds Data Structure [10]

2.1.2 ipc_perm Data Structure

The kernel also maintains *ipc_perm* data structure per shared memory segment in the system. It maintains information for each IPC object, similar to the information it maintains for files.

```
struct ipc_perm
{
        uid_t       uid;    /* owner's user id */
        gid_t       gid;    /* owner's group id */
        uid_t       cuid;   /* creator's user id */
        gid_t       cgid;   /* creator's group id */
        mode_t      mode;   /* read-write permissions */
        ulong_t     seq;    /* slot usage sequence number */
        key_t       key;    /* IPC key */
}; [11]
```

2.1.3 Interfaces Provided by System V Shared Memory

System V shared memory is similar to POSIX shared memory. But instead of calling *shm_open()* followed by *mmap()*, we use *shmget()* followed by *shmat()*. Here we aimed at discussing all the important predefined functions in detail supported by System V Shared Memory.

Following are the important API's used for System V shared memory:-

System Call	Arguments Accept	Return Values	Explanation
shmget()	key, size, oflag	Shared Memory Identifier	Either creates a new shared segment if one with a corresponding key does not exist, or access an existing one based on the key.
shmat()	Shared Memory Identifier, address, flag	Starting address of shared memory segment	Attaches shared segment to process address space.
shmdt()	Address of shared memory segment	0 on success or -1 on error	Detaches a shared segment from a process address space
shmctl()	Shared Memory Identifier, command, status structure	0 or -1 (success or failure)	Use to change permission and other characteristics of shared memory segment.

Table 3: System V Shared Memory APIs [10]

Let us now discuss all the above mentioned APIs with their complete prototype, parameters and return value in detail:

2.1.3.1 *shmget* Function

The *shmget()* system call is used to create a new shared memory segment if the one corresponding to key doesn't exist, or access an existing one based on the value of key.

Prototype:-

int shmget (key_t key, size_t size, int oflag);

Parameters:-

- The *key* argument is an access value associated with the semaphore ID.
- The *size* argument is the size of requested shared memory segment in bytes.
- The *oflag* argument specifies the initial access permissions and creation control flags.

Return Value:-

It returns the Shared Memory Segment Identifier on success and -1 to indicate error condition. It also returns the ID of an existing shared segment.

2.1.3.2 *shmat* Function

It is used to attach the newly created shared memory segment or an opened existing segment to process address space.

Prototype:-

void * shmat (int shmid, const void *shmaddr, int flag);

Parameters:-

- The *shmid* is a shared memory segment identifier returned by *shmget* function. It is used to recognize the shared memory segment to which the process wants to connect.
- The *shmaddr* is a NULL pointer which allows the system to select the address of the shared memory segment of its own.
- The *flag* indicates the access permissions.

Return Value:-

The return value from *shmat* is the starting address of the shared memory segment within the calling process and returns -1 on error.

2.1.3.3 *shmdt* Function

It is used to detach the shared memory segment when a process is done with it. But if a process terminates, all shared memory segments presently attached by the process are detached. But this call doesn't delete shared memory segment.

Prototype:-

int shmdt (const void *shmaddr);

Parameters:-

- The *shmaddr* is a const pointer returned by *shmat()* which represents the starting address of shared memory segment.

Return Value:-

This function returns 0 for successful execution and -1 on error.

2.1.3.4 *shmctl* Function

shmctl() is used to alter the permissions and other characteristics of a shared memory segment. But the process should have effective *shmid* of owner, creator or superuser to perform this task.

Prototype:-

int shmctl (int shmid, int cmd, struct shmid_ds *buf);

Parameters:-

- The *shmid* represents the effective *shmid* of owner, creator or superuser.
- The *buf* is a structure of type struct shmid_ds which is defined in <sys/shm.h>
- And following are the cmd options:-

Commands	Description	Permission
IPC_STAT	Return the status information contained in the control structure and place it in the buffer pointed to by buf.	The process must have read permission on the segment to perform this command.
IPC_SET	Set the effective user and group identification and access permissions.	The process must have an effective ID of owner, creator or superuser to perform this command.
IPC_RMID	Remove the shared memory segment.	The process must have an effective ID of owner, creator or superuser to perform this command.

Table 4: The *'cmd'* argument in *shmctl* Function [12]

Return Value:-

This function returns 0 for successful execution and -1 on error. [12]

2.1.4 Sample Program using System V Shared Memory

The following code segment indicates simple use of shared memory between two independent processes. Both the processes communicate via shared memory wherein Process 1 accepts input from the user and Process 2 prints that input.

This code throws light on "ZERO COPY" concept that is in this case because of the use of shared memory no copy operations were required because user enters the input directly on shared memory with the help of the pointer returned by *shmat()* function.

Process 1:-

```
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include<stdio.h>
#include<sys/shm.h>
int main()
{
        int shmid;
        key_t key;
        void *ptr;
        system ("touch help");
        key = ftok("help",0);
        shmid = shmget(key,20,IPC_CREAT);
        ptr = shmat(shmid,NULL,0);
        if((void*)-1 == ptr)
        {
                printf("\n Eror: %s",strerror(errno));
        }
        else
        {
                printf("Enter name\n");
                scanf("%s",&ptr);
                sleep(20);
        }
        return 0;
}
```

Process 2:-

```
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include<stdio.h>
#include<sys/shm.h>
int main()
{
        int shmid;
```

```
        key_t key;
        void *ptr;
        system("touch help");
        key = ftok("help",0);
        shmid = shmget(key,20,IPC_CREAT);
        ptr = shmat(shmid,NULL,0);
        if ((void*)-1 == ptr)
        {
              printf("\n Eror: %s",strerror(errno));
        }
        else
        {
              sleep(20);
              printf("%s",ptr);
        }
        return 0;
}
```

2.2 POSIX Shared Memory

Like UNIX System V, POSIX.1 also provides a standardized API for creating and using shared memory.

Though we have already talked about POSIX shared memory in Chapter-1, now let us see various interfaces provided by POSIX to use the shared memory for communication between processes on local host.

2.2.1 Interfaces Provided by POSIX Shared Memory

POSIX shared memory is similar to System V shared memory. But instead of calling *shmget()* followed by *shmat()*, we call *shm_open()* followed by *mmap()*. In the subsequent section we aimed at discussing all the important POSIX shared memory interfaces in detail.

Following are the important API's used for System V shared memory:-

System Call	Arguments Accept	Return Values	Explanation
shm_open()	name, oflag, mode	Non-negative descriptor on success and -1 on error	Either creates a new shared memory object or to open an existing one.
shm_unlink()	name	0 on success or -1 on error	Remove the name of a shared memory object.
ftruncate()	fd, length	0 on success or -1 on error	Change the size of either a regular file or a shared memory object.
fstat()	Fd, buff	0 on success or -1 on error	Use to get the information about existing shared memory object
mmap()	addr, len, prot, flags, fd, offset	On success, starting address of mapped region and MAP_FAILED to indicate error condition.	Maps either a file or a POSIX shared memory object into the address space of a process.
munmap()	addr, len	0 on success or -1 on error	Remove the mapping from the address space of the process.

Table 5: POSIX Shared Memory APIs [12]

Let us now discuss all the above mentioned APIs with their complete prototype, parameters and return value in detail:

2.2.1.1 *shm_open* Function

shm_open is used either to create a new shared memory object or to open an existing shared memory object.

Prototype:-

int shm_open (const char *name, int oflag, mode_t mode);

Parameters:-

- The *name* argument is used by any other processes that want to share this memory.
- The *oflag* argument specifies the initial access permissions and creation control flags.
- *Mode* specifies the permission bits and is used when O_CREAT flag is specified otherwise this argument would be 0.

Return Value:-

When the call succeeds, it returns the non-negative descriptor and -1 to indicate error condition.

2.2.1.2 *shm_unlink* Function

shm_unlink is used to remove the name of a shared memory object. It has no effect on other existing references to the shared memory object, until all references to that object are closed.

Prototype:-

int shm_unlink (const char *name);

Parameters:-

- The *name* argument indicates the shared memory object that needs to be unlinked.

Return Value:-

When the call succeeds, it returns the 0 and -1 to indicate error condition.

2.2.1.3 *ftruncate* Function

It is used to change the size of either a regular file or a shared memory object while dealing with *mmap.*

Prototype:-

int ftruncate (int fd, off_t length);

Parameters:-

- The *fd* argument indicates the descriptor of a file or a shared memory object whose size needs to be changed.
- The size of a regular file or shared memory object is set to *length* bytes.

Return Value:-

When the call succeeds, it returns the 0 and -1 to indicate error condition.

2.2.1.4 *fstat* Function

The *fstat* is used to obtain the information about existing shared memory object when we open it.

Prototype:-

int fstat (int fd, struct stat *buf);

Parameters:-

- The *fd* argument indicates the descriptor of an existing shared memory object whose size needs to be changed.
- The **buf* is the pointer to structure *stat* defined in <sys/stat.h> header file which contains information about shared memory object.

Return Value:-

When the call succeeds, it returns the 0 and -1 to indicate error condition.

2.2.1.5 *mmap* Function

The *mmap* function maps either a file or a POSIX shared memory object into the address space of a process. Once the memory is mapped into the address space of the processes that are sharing the memory region they need not execute any system calls into the kernel for exchanging information which would otherwise be required. Shared memory allows two or more processes to share a region of memory. However, the processes must coordinate and synchronize their use of the shared memory to avoid data loss.

We use *mmap()* function for three purposes:-

- With a regular file to provide memory-mapped I/O.
- With special files to provide anonymous memory mappings.
- With *shm_open* to provide POSIX Shared Memory between unrelated processes.

Prototype:-

void *mmap (void *addr, size_t len, int prot, int flags, int fd, off_t offset);

Parameters:-

- The *addr* specifies the starting address within the process of where the descriptor *fd* should be mapped. Usually it is specified as NULL pointer indicating kernel to select the starting address.
- The *len* represents the number of bytes to be mapped into the address space of the process, starting at *offset* (usually 0) bytes from the beginning of the file.
- The *prot* argument specifies the protection of memory mapped region by using following constants:-

Prot	Descriptor
PROT_READ	Process can read the data.
PROT_WRITE	Process can write the data.
PROT_EXEC	Process can execute the data.
PROT_NONE	Process can not access the data.

Table 6: The '*prot*' argument in *mmap* Function [12]

- The *flag* argument can be specified using following constants:-

Flags	Description
MAP_SHARED	Modifications done by a process to the mapped data are visible to all the processes.
MAP_PRIVATE	Modifications done by a process to the mapped data are visible to only that process.
MAP_FIXED	Interpret the *addr* i.e. the location of memory mapped region. But for portability issues it should not be specified and *addr* should be 0.

Table 7: Flags in *mmap* function specified by the constants [12]

Return Value:-

It returns the starting address of mapped region on success and MAP_FAILED to indicate error condition.

2.2.1.6 *munmap* Function

The *munmap* function is used to remove the mapping from the address space of the process.

Prototype:-

int munmap (void *addr, size_t len);

Parameters:-

- The *addr* argument is the address that was returned by *mmap.*
- The *len* is the size of that mapped region.

Return Value:-

It returns the 0 on success and -1 to indicate error condition. [12]

2.3 Messaging Solutions

A communication can takes place between processes on local host or on remote host through number of ways. In present market, there are number of messaging solutions available. But these solutions use sockets underneath for communication between processes on local host or on remote host.

There are various messaging solutions available in market, below two are two dominating vendors of market in messaging solutions:-

a) TIBCO Messaging Solutions.
b) 29WEST Messaging Solutions.

We use these two market leading vendor solutions as benchmarks against our solution, so let's first discuss them briefly:-

2.3.1 TIBCO Messaging Solutions

For many years TIBCO have been known for providing the most efficient, reliable, and scalable messaging solutions. TIBCO provides businesses the facility to select the messaging solution as per their unique set of systems, business requirements, and IT resources, by providing such a complete set of established and verified solutions. Following are the two messaging solutions provided by TIBCO:-

2.3.1.1 TIBCO SmartSockets

TIBCO SmartSockets provides outstanding performance, scalability, bandwidth efficiency, and fault tolerance and reliable real-time messaging using industry-standard protocols like TCP/IP. With the use of TIBCO's SmartSockets APIs and class libraries, organization can make sure that applications distribute and exchange information quickly, reliably and securely across any platform and any network.

Key Features

a) **Publish-subscribe** for intelligent, streamlined one-to-many communications.
b) **Adaptive multicast** for most efficient network utilization.
c) **Multithreaded, multiprocessor architecture** for full system exploitation.
d) **Online security** safeguards vital communications.

e) **Real-time monitoring** of network applications.

f) **Performance optimization** for maximum throughput.

g) **Robust, enterprise-quality fault-tolerant GMD** for reliable message delivery. [14]

2.3.1.2 TIBCO Enterprise Message Service

TIBCO Enterprise Message Service is used to manage the real-time flow of information by bringing together different IT assets and communications technologies on a common enterprise backbone. By using this solution companies have been able to reliably support over 50,000 messages per second and achieve 99.999% uptime.

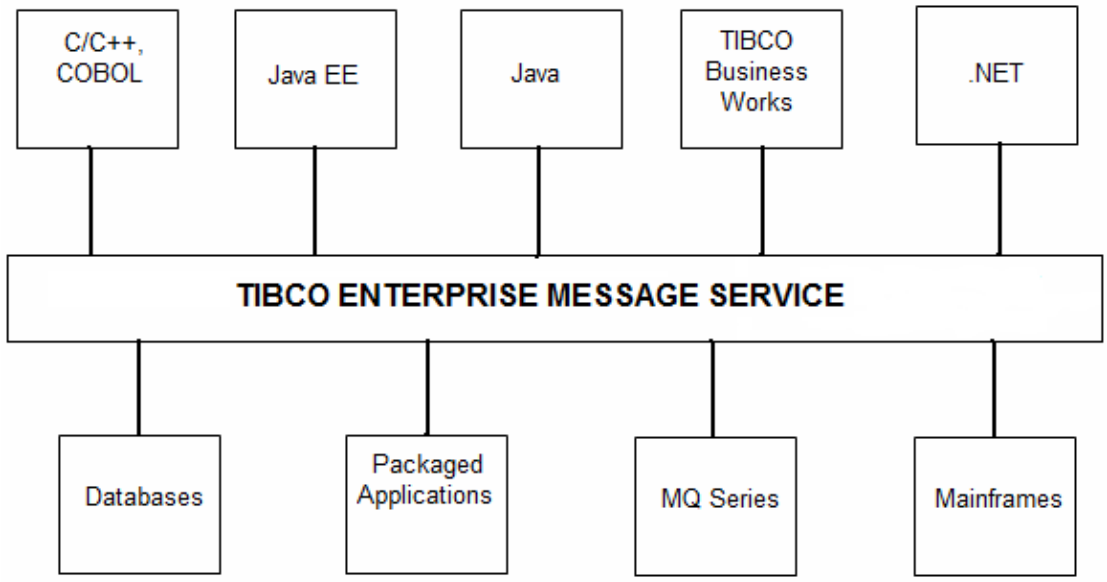

Figure 2: Common Backbone for Services and Real-Time Information Flow [15]

Key Features:-

1. It enables developers and administrators to support different types of service protocols on the same platform and adjust qualities of service for most demanding applications by supporting request/reply and publish/subscribe interactions, synchronous and asynchronous messaging, multicast deployments and different levels of reliable messaging capabilities.
2. It delivers high performance and provides secure messaging solution by supporting security standards with the administrative control.

3. It also provides operational flexibility since it integrates with third-party relational databases.
4. It provides built-in monitoring and management capabilities which help in detailed administrative functions and statistics and support automation through an administrative API or command-line shell. [15]

2.3.2 29WEST Messaging Solutions

It is being used worldwide for ensuring high-performance messaging for financial markets. Many financial institutions worldwide have replaced their legacy messaging systems with 29WEST messaging solutions which have resulted, in latency reductions of 10 times and more remarkable throughput gains.

Following is one of the well-known and widely used messaging solutions provided by 29WEST:-

2.3.2.1 Latency Busters Messaging (LBM)

It is a fast, efficient, and lightweight messaging system aimed to serve as the enterprise messaging solution for the next generation of high-performance applications having very high message rates.

Its exceptional design allows users to gain a competitive edge with the industry's fastest messaging.

LBM design eliminates the need for messaging servers, routers and messaging daemons hence, reduces latency, increases throughput and allows data to flow directly from sender to receiver.

The following two illustrations show messaging before and after LBM.

Before LBM:-

Prior to LBM, daemons, routers and servers were used which created messaging chokepoints for any type of transport used.

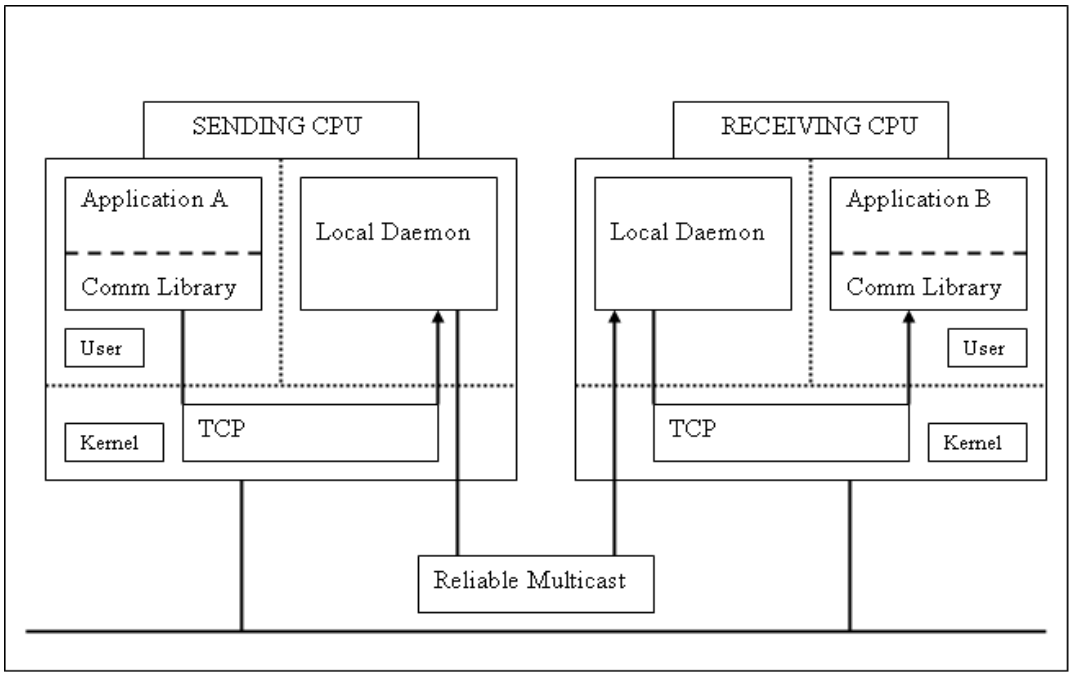

Figure 3(a): Messaging Chokepoints before LBM [17]

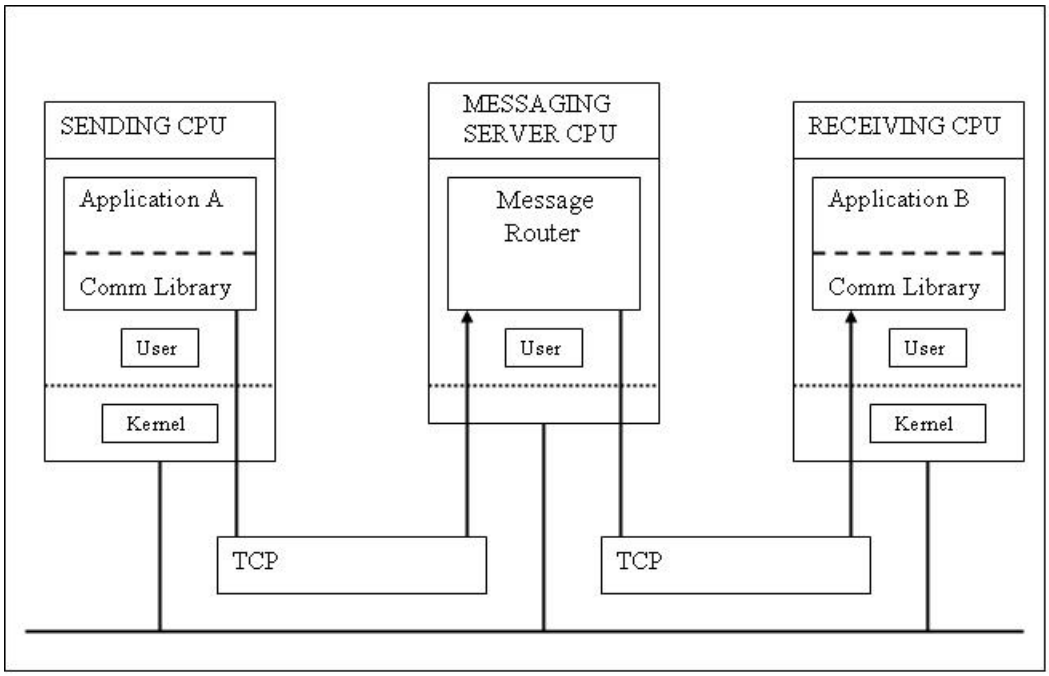

Figure 3(b): Messaging Chokepoints before LBM [17]

With LBM:-

LBM creates a higher throughput, application-to-application model suitable for any transport by utilizing the network infrastructure for message routing,

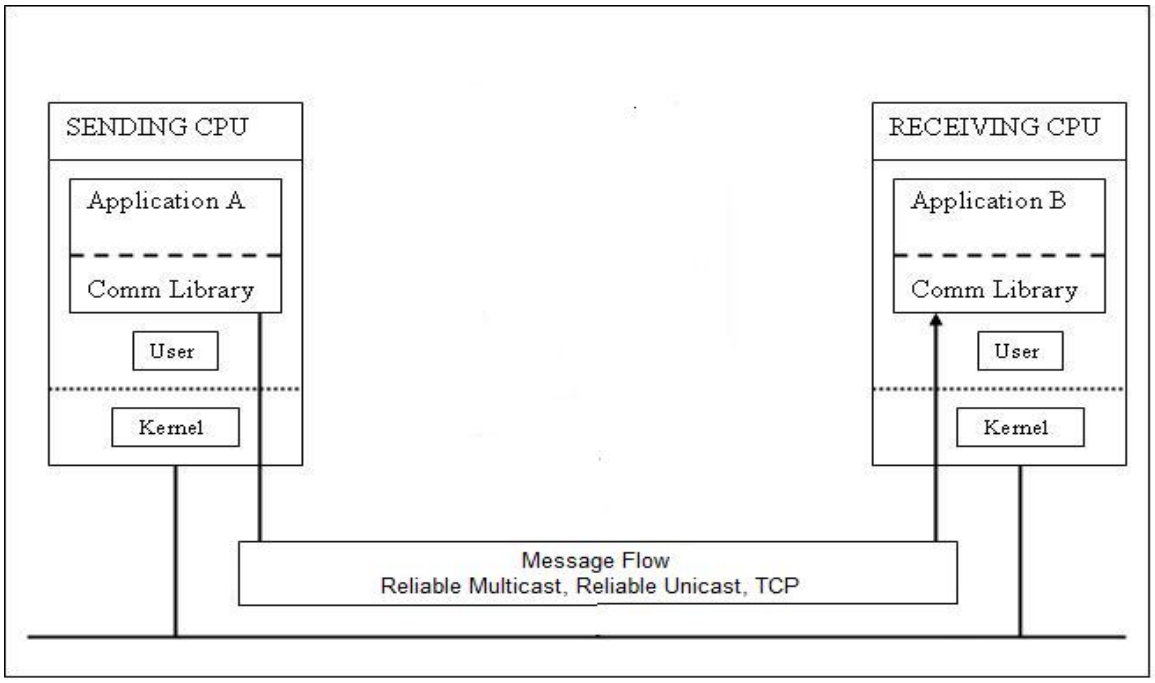

Figure 4: LBM Eliminates Messaging Chokepoints [17]

Key Features

LBM offers various advantages since it provides the ability to link directly with your application:-

1. Reduced data copies.
2. Reduced context switches
3. Reduced number of processes involved in handling each message.
4. Fewer maintenance and upgrade headaches since no new entities to manage in the network. [17]

From above discussion it's very clear that to have fast communication between the processes on remote host these solutions are doing incredibly good. But in order to have equivalent faster communication between the processes on local host we need a better inter-process communication mechanism which is undoubtedly shared memory. But shared memory usage is accompanied with few limitations like extra burden on application program since it has to take care of synchronization between processes. But if we could have such library that could provide us with best of both that is sockets for remote communication and shared memory for local communication then that would help us to achieve a low latency system which is the main demand of

growing IT industry where time means money. This library would provide a Seamless Interface on top that would encapsulate Shared Memory based Communication Interface and Socket based Communication Interface. There are few customized systems that are using shared memory for local communication but no such Seamless Interface exists which could allow user to communicate with other systems without being bothered about underlying complexities and required communication transport.

2.4 Shared Memory Implementation in Solaris

Shared memory is an inter-process communication facility that exists in every major version of UNIX available today. It is omnipresent in its use by applications developed for UNIX systems. On a Solaris system shared memory provides an extremely efficient means of sharing data between multiple processes because the data need not be moved from one process's address space to another. Each process maps to the same physical pages and can access the memory through pointer dereferencing in code.

The use of shared memory in an application requires implementing just a few interfaces bundled into the standard C library, /usr/lib/libc. These interfaces are listed in Table 3 above. These interfaces perform many useful tasks from a kernel implementation standpoint.

The kernel implementation of shared memory requires two dynamically loadable kernel modules:-

- The *shmsys* module, which is located in /kernel/sys directory, contains the kernel support routines for the shared memory library calls (Table 3).
- The *ipc* module, which is located in /kernel/misc directory, contains two kernel routines, *ipcget()* and *ipcaccess()*, that apply to all the inter-process communication (IPC) facilities.

These modules are not loaded automatically by SunOS at boot time. The kernel will dynamically load a required module when a call is made that requires the module. Thus, first time an application makes a shared memory system call (e.g. *shmget()*), the kernel will load the module and execute the system call. The module will remain

loaded until it is explicitly unloaded, via the '*modunload*' command, or the system reboots.

On executing *ipcs* command, it sometimes comes back with a message "facility not in system" which means the module is not loaded.

```
# ipcs
IPC status from as of Mon Aug 11 18:32:30 1997
Message Queue facility not in system.
Shared Memory facility not in system.
Semaphores:
#
```

You can tell the operating system to load the module during bootup by using the '*forceload*' operation in the /etc/system file: forceload: sys/shmsys. We can also use the *'modload'* command, which allows a root user to load any loadable kernel module from the command line. The *'modinfo'* command can be used to see which loadable modules are currently loaded in the kernel. The SunOS is smart enough not to allow the unloading *(modunload)* of a loadable module that is in use. Moreover, the code is written to be aware of dependencies, such that loading the shmsys module will also cause the ipc module to be loaded.

2.4.1 Shared Memory Tuneable Parameters

Various resources are being maintained by kernel for the implementation of shared memory. For example, on successful execution of *shmget()* system call operating system initializes and maintains a shared memory identifier *(shmid)* which identifies a shared segment. It basically has two components:-

- The actual shared RAM pages
- A data structure *shmid_ds* that maintains information about the shared memory segment.

2.4.1.1 *shmmni* Tuneable Parameter

At the boot time, on the basis of *shmmni*, a shared memory tuneable parameter, the system allocates kernel memory for some number of *shmid_ds* structures. The

‘*shmmni’* tuneable parameter defines the requested number of unique shared memory identifiers the system maintains. The size of each *shmid_ds* structure is 112 bytes and has a corresponding kernel mutex lock, whose size is 8 bytes.

Therefore, the amount of kernel memory required by a system to maintain shared memory can be computed by ((shmmni * 112) + (shmmni * 8)). For example, for the default value of *shmmni,* a system allocates (100*112) + (100*8) = 13 kilobytes kernel memory for shared memory support.

But *shmmni* should not be set to a randomly large value merely to ensure sufficient resources since there is a limit to kernel memory that a system can support. For example, on Solaris 2.5, 2.5.1, and 2.6, the limit = 256 MB and on UltraSPARC [sun4u]-based systems, the kernel has its own 4GB address space, so it's not much constrained. Required kernel memory remains in RAM since the kernel is not pageable; this reduces the available memory for user processes. Today this may not be an issue since Sun ships systems with very large RAM capacities, however it should be considered.

In order to protect itself from allocating extra kernel memory for shared memory support, the system, check for the maximum available kernel memory, divide that value by four, and use the result as a maximum value for allocating resources for shared memory. In simply words, the system will not allow more than 25 percent of available kernel memory to be allocated. But this applies to Solaris 2.5, 2.5.1, and 2.6. Prior releases including Solaris 2.4, has no such restriction. Moreover, newer releases don’t require the extra eight bytes per *shmid_ds* for a kernel mutex lock because finer-grained locking was implemented, allowing for greater potential parallelism of applications using shared memory. Whereas in the earlier releases shared memory used very coarse-grain locking and only implemented one kernel mutex in the shared memory code.

In order to determine system’s kernel architecture ‘uname’ command with ‘-m’ option can be used as follows:

```
% uname -m
sun4u
```

2.4.1.2 *shmmax* Tuneable Parameter

It defines the maximum size of a shared segment. The second argument in *shmget()* system call determines the size of a shared memory segment. When the *shmget()* call is executed, the kernel checks to ensure that the size argument is not greater than *shmmax*. If it is, an error is returned. Kernel resources are not allocated based on *shmmax.* Hence, even if we set *shmmax* to its maximum value, it does not affect the kernel size. This parameter can be tuned in /etc/system file entry as:-

set shmsys:shminfo_shmmax=0xffffffff /* hexidecimal (4GB for Solaris 2.5.1, 2.6)*/

set shmsys:shminfo_shmmax=4294967295 /* decimal */

2.4.1.3 *shmmin* Tuneable Parameter

The *shmmin* tuneable defines the smallest possible size a shared segment can be, as per the size argument passed in the *shmget()* call. There's no real compelling reason to set this from the default value of 1.

2.4.1.4 *shmseg* Tuneable Parameter

It defines the number of shared segments a process can attach (map pages) to. Processes may attach to multiple shared memory segments for application purposes, and this tuneable determines how many mapped shared segments a process can have attached at any one time.

Now let us look at two tuneable parameters associated with shared memory in Solaris 10:-

Name	Description
max-shm-memory	Maximum size in bytes of a shared memory segment. When *shmget()* allocates a shared memory segment, the segment's size is allocated and checked against this limit. The *shmget()* fails and set errno equal to EINVAL if the size argument is less than the system-imposed minimum or greater than the system-imposed maximum.
max-shm-ids	Maximum number of *shmid_ds* structures system-wide. When *shmget()* allocates a shared memory segment, one ID is allocated. The *shmget()* fails and sets errno equal to ENOSPC if the system-imposed limit on the maximum number of allowed shared memory identifiers system-wide would be exceeded.

Table 8: Tuneable Parameter Associated with Solaris 10 Shared Memory [10]

2.5 Facilitating Communication within Shared Memory Environment using Lock-Free Queues

One of the other inventions in this field is by Rajeev Sivaram who introduced the yet another concept of using lock-free queues to communicate within shared memory environment. To improve the efficiency in communication within shared memory the lock-free queues are structured to reduce the use of atomic operations and the number of enqueue or dequeue operations.

Each process has an associated lock-free *data queue* and *free queue*. The lock-free queue is concurrently accessible at one end for writing by multiple processes and non-concurrently accessible at another end for reading. The concurrent operations on the queues are managed through atomic operations. Data queue is used to retrieve data from other processes. Each data queue may have zero or more entities containing a pointer of an element in the shared memory. The element consists of data that is to be communicated between processes. Similarly, the free queue includes zero or more entries containing a pointer of an available element. Elements that are available for storing data are tracked by this queue.

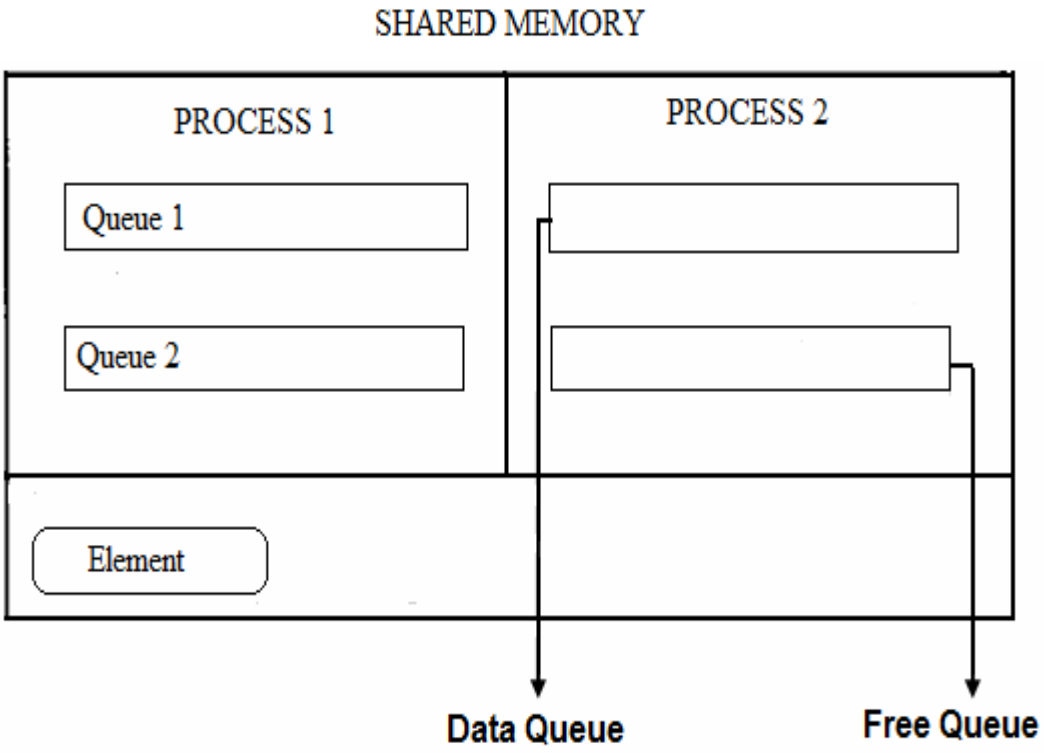

Figure 5: Processing Entities in the Shared Memory

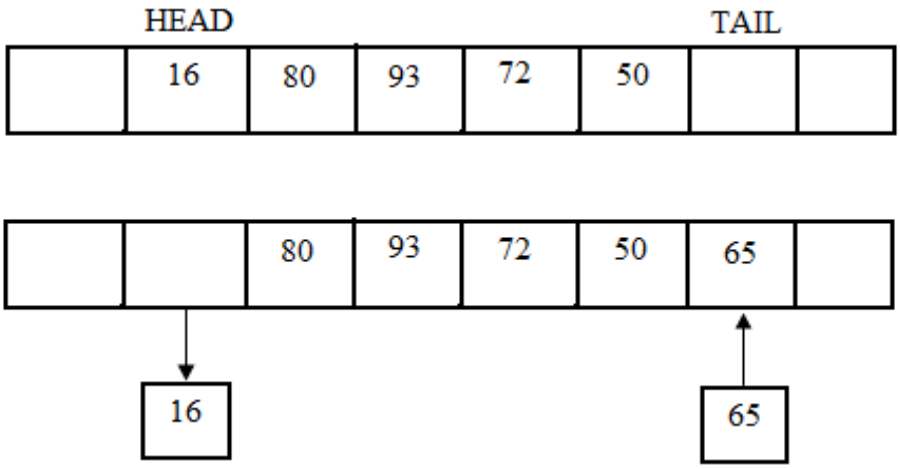

Figure 6: Lock-Free Queues in Shared Memory

2.5.1 Queue size

The queues are designed and sized to not reach a full condition. The queue is initialized before performing any enqueue or dequeue operations. For instance, int queue[size] is used to initialize the queue. The queues are designed not to reach full condition by making the size of the queue as a power of 2 which is sufficiently large such that the queue does not become full. This is achieved by making a queue size equal to the total number of elements available in the processes. For example, queue size for 64 processes where each can submit up to 128 on-the-fly elements, 64X128 = 8192 is sufficient.

2.5.2 Enqueue Operation

He further used the term ‘slot’ for the elements in memory. Sender process sends the data to receiver process by following these steps:-

1. Sender process obtains the pointer of a slot from a sender’s free queue data structure to place data. (Dequeue operation)
2. Store data in the slot specified by the pointer.
3. The pointer of the slot is placed on receiver’s data queue. (Enqueue operation)

A sending process enqueue an element on a receiving process data queue by placing a pointer of the element on the queue. Atomic operations are required for enqueue operations because concurrency is provided for enqueue operation that enables multiple processes to concurrently write to the queue. Moreover, enqueue is

performed using single atomic operation only once. The enqueue operation does not check for a full queue, since the queue is designed not to be full.

To enqueue the slot on the receiver's message queue following steps are performed:-

1. Atomically determining the current tail of message queue and increment the tail to obtain new tail by employing a single fetch_and_add operation.
2. Place the slot index into old tail index.

2.5.3 Dequeue Operation

The receiver periodically checks its queue to determine if there are any messages. Following steps are performed by receiver process:-

1. Initially, the receiver attempts to dequeue a slot from the receiver's message queue.
2. Receiver checks whether a slot was dequeued or not. If a pointer of the slot was not returned by the dequeue process, then processing is complete. However, if the pointer was returned, then the data in the slot specified by the pointer is processed.
3. Return the slot to the free queue.

An element is dequeued from the data queue of receiver process when it wishes to access the data. An element is dequeued by retrieving a pointer from the queue. Since a queue is owned by single process and only that process can process the data of the queue therefore, non-concurrency for dequeue operation is provided. Dequeuing an element of the lock-free queue absent an atomic operation. It basically includes following steps:-

1. A determination is made as to whether the queue is empty. If the queue is empty, then slot_index is set to empty indicating that there are no slots on the message queue. However, if the queue is not empty, then the value of the head of the queue is obtained.
2. The queue head is checked next that whether it's empty or not. If its not empty then slot_index is set to queue[old_head] containing the pointer of the slot that has the data to be retrieved.
3. Moreover, the queue at queue[old_head] is set to empty.
4. The value of head is incremented to indicate the new head.

2.5.4 Reusing the Slots to Increase Efficiency and Performance

Each process holds up to one extra slot that is not in its message queue or free queue. There will be no starvation of free slot as long as the number of free slots owned by a process is greater then a total number of processes. Since the total number of extra slots for one process can be at most the total number of processing entities communicating in the shared memory, and the remaining slots are returned to the free slot queue of the process sooner or later.

2.5.4.1 Sender Reuse

To reduce the number of enqueue and dequeue operations, the slot used by the last incoming message is not returned to the free slot queue immediately rather it is saved for the next outgoing message. In short, a pointer of the slot is saved. For example, in communications protocols, in general the receiver send a reply to the sender for the message received, so slot reuse saves one enqueue and one dequeue operation, hence, performance and efficiency is improved since the sender reuses the slot by placing data in it and enqueues the slot on the receiver's queue.

2.5.4.2 Receiver Reuse

The receiver dequeues a slot from its queue and processes the data in the slot. The slot is then saved for the next outgoing message. Then a determination is made as to whether there is a previous saved slot. If so, then the previous saved slot is enqueued on the free queue, otherwise, the processing is complete.

Hence, we can conclude that to facilitate communication, lock-free queues are provided that minimize atomic operations, as well as dequeue/enqueue operations. The queues have the characteristics of not becoming full and only having concurrency at one end e.g., the tails for enqueue operations. By using these queues, performance is enhanced. Performance is further enhanced by reusing slots, thus minimizing the number of enqueue/dequeue operations. [20]

After analyzing everything we concluded that businesses today look forward for an integrated solution. Today industry is dependent on already existing socket based solutions for message passing which is slower in case we require inter-process communication on local host. So we aimed at developing an integrated solution that

would encapsulate shared memory interface and sockets interface into one seamless interface. The resultant interface would automatically use shared memory for faster communication if processes are on same host else would use sockets for communication between processes on remote system.

CHAPTER - 3
PROBLEM STATEMENT

3.1 The Current Picture

In current scenario communication plays a vital role since scalable solutions generally are not designed within single process. Such systems have large number of cooperative processes generally serving a set of services, where each service may rely on data from other co-operative process. This results in large amount of data communication need. These days data have become an integral part of any organizations IT Infrastructure and an organization should have access to all kind of data at all time.

Taking into account today's scenario we understand that data is required to cross process boundaries and at times be available to applications simultaneously, and this in turn requires efficient inter-process communication. There are various middleware applications available in market to server specific need of inter-process communication and many vendors provide various efficient messaging solutions. However, to our analysis till date all the existing messaging solutions are using sockets underneath for communication between the process on local host as well as for the communication between the processes on different hosts. This is a very obvious choice considering the flexibility sockets provide in terms of communication and almost no synchronization need. However, there are some solutions that use shared memory underneath for communication but these are a very limited set of highly customized solutions. These solutions as such are not really available as a generic communication middleware till date, though they serve specific needs very well. We observed that there is a desperate need in this vacuum for an integrated solution to be available in the market for diverse range of applications to take advantage from.

Now that we understand there is a need for such solution, we might want to find, why such solution doesn't exist? Well the answer has to lie within these two facts:

1. Shared memory is the fastest known inter-process communication mechanism along with the fact that it is one of the most complex forms of IPC to deal with. The complexity involved is simply too huge, like synchronization between processes is of atmost important and should be dealt with care while using shared memory for inter-process communication.
2. Sockets on the other hand shifts all synchronization needs from programmers shoulder to kernel. This in itself is good enough to choose sockets over shared memory when dealing with complex multithreaded applications.

Application program or the programmer should not be overburdened with the shared memory usage. They should be able to use shared memory as an efficient mechanism for communication on local host as they are using sockets.

Sockets which are being used today for inter-process communication on local host as well as on remote host is though a reliable method but lack in efficiency when it comes to inter-process communication on local host since number of copy operations and context switches between processes and kernel increases resulting in low latency system. Though internet sockets are easy to use because of the available libraries but if we could provide users with an equivalent library that allow user to work with same ease and which is more efficient in terms of time would be far better option in prevailing market.

3.2 The Missing Part

As discussed earlier, there is no integrated solution available in market that could encapsulate the best features of shared memory and sockets so that the best possible efficiency can be derived from them. Moreover, to use them for their best possible ways to achieve maximum throughput that is to use shared memory for communication between the processes on local host and sockets for communication between the processes on different hosts.

Another view worth paying attention is ‘market’ these days, which is highly sensitive towards ‘time to market’. In present market condition one needs to best use his time and resources. All the available efficiency whether of human resource or other resources should be used efficiently and in time efficient manner. Similarly, available

resource in terms of computing power and memory utilization should be put to its maximum efficiency so that it can be used in an effective way. And the best way to use this memory for inter-process communication is to use shared memory concept for local communication and sockets for remote communication. Though both these concepts already exist in market and are being used too but there is no integrated solution that could help an application program or a programmer to use both these concepts efficiently.

This is the solution that best fits this vacuum. This would fill the gap of having a proper and efficient mechanism for IPC, that is a Seamless Interface covering both Shared Memory based Communication Interface and Socket based Communication Interface into one and help a programmer to concentrate on his designated task without being worried about the underlying complexities of IPC and synchronization in-between co-operative processes.

3.3 The Solution

The solution to above problem is to have an integrated Seamless Interface encapsulating Shared Memory based Communication Interface and Socket based Communication Interface into one seamless interface. We call it seamless because user would be free to use it the way he likes without being worried about underlying complexities and without paying attention on required synchronization between processes because all this would be taken care by our efficient and intelligent interface.

Secondly, the proposed system would be inter-operable i.e. C++ Application would be able to communicate with a C or Java Application and vise-versa. The solution would be portable across various operating Systems (currently we focused on UNIX and LINUX variants). It would provide an efficient, consistent and maintainable solution in the sense that it could be used by many organizations and firms without making modifications.

The solution would be generic and would allow all kind of organizations that deal with real-time flow of information to use it in an efficient manner like any other

library without being worried about cost involved and time required to implement. Once they will start using this solution, it would result in low latency system resulting in more accurate and timely information. This solution would be used by any financial institutions, Scientific Research Organizations and any other organizations without modifications thus ensuring consistency. We aimed at using existing APIs to design an efficient and error-free system.

3.4 The Goal of Thesis Work

The goal of this thesis work is to study and understand the existing inter-process communication mechanisms which are being used in existing market for real time flow of information. These includes a detailed study of various architectures designed and implemented by some of the well known messaging solution providers such as TIBCO (EMS and SmartSockets) and 29WEST (LBM) for inter-process communication and then design our own Shared Memory Transport and Seamless Interface that would encapsulate the Shared Memory based Communication Interface for local communication and Socket based Communication Interface for remote communication.

This thesis would also throw light on various pros and cons of existing system and other related work done in this field. And finally to come up with a portable and scalable design that would allow programmers to use our Seamless Interface with an ease and with minimal complexities.

The following considerations are to be taken into account:-

1. The seamless interface or the library that we aimed at designing should be easy to use.
2. All kind of applications that require any kind of inter-process communication should be able to use it and hence should be portable.
3. It should free the programmer from any extra overhead required when dealing with shared memory like synchronization. Hence, no extra overhead for a programmer that would shorten the delivery time.
4. The solution should not impede communication across processes written in different programming languages. It should provide similar flexibility in this area as provided by sockets based communication systems.

3.5 The Platform

This thesis work concentrates primarily on analyzing the existing well-known messaging solution providers in market and designing a new Seamless Interface that would encapsulate the Shared Memory based Communication Interface and Socket based Communication Interface so as such no platform was required to carry out this thesis work. This work mainly focuses on analysis part so that correct and efficient solution can be designed. Besides designing a Seamless Interface, we also focused on designing a Shared Memory Transport with other supporting modules. All the designed modules and algorithms are discussed in detail in Chapter-4 for proper understanding of all readers.

Our design in not limited to few Writer or Reader Processes but in its fully blown form, it'll accommodate processes across several servers and multiple processes with complex connection requirements on same server. All processes operate by using the system which in turn seamlessly would choose in-between Socket and Shared Memory Transport depending on the locality of receiver process.

CHAPTER - 4

PROPOSED SYSTEM DESIGN

4.1 Conceptual Design of a Proposed System

The proposed Seamless Interface API is designed keeping in mind the growing demand for fast communication of information and data between the processes. The desired output of this proposed system is the low latency system. The conceptual design of the system presents the overall architecture of the proposed system and would help to understand and analyze the various modules and aspects of the proposed system. The Seamless Interface that encapsulate both Shared Memory based Communication Interface and Socket based Communication Interface, takes the advantages of both worlds and gives us a solution that is more flexible and would be used by many organization which require real time processing of data.

Our generic "Shared Memory Transport" which is designed taking into considerations the goals of this thesis work. We would analyse this transport in detail because without knowing about it completely we won't be able to get the flavour of our proposed system. Our proposed system would take care of synchronization between communicating processes and would help to share data between multiple processes using writer shared memory segment queues which we would be discussing in detail in subsequent section.

Conceptual model will help us to see at a glance the complete architecture of our proposed system. It throws light on various modules and components and their proper arrangement which help in easy and fast communication of data between processes on local host. This conceptual diagram doesn't express in detail the procedure used by existing messaging system since that is out of the scope of this thesis work. The architecture of the complete proposed model gives accurate information about our Seamless Interface encapsulating Shared Memory based Communication Interface and Socket based Communication Interface.

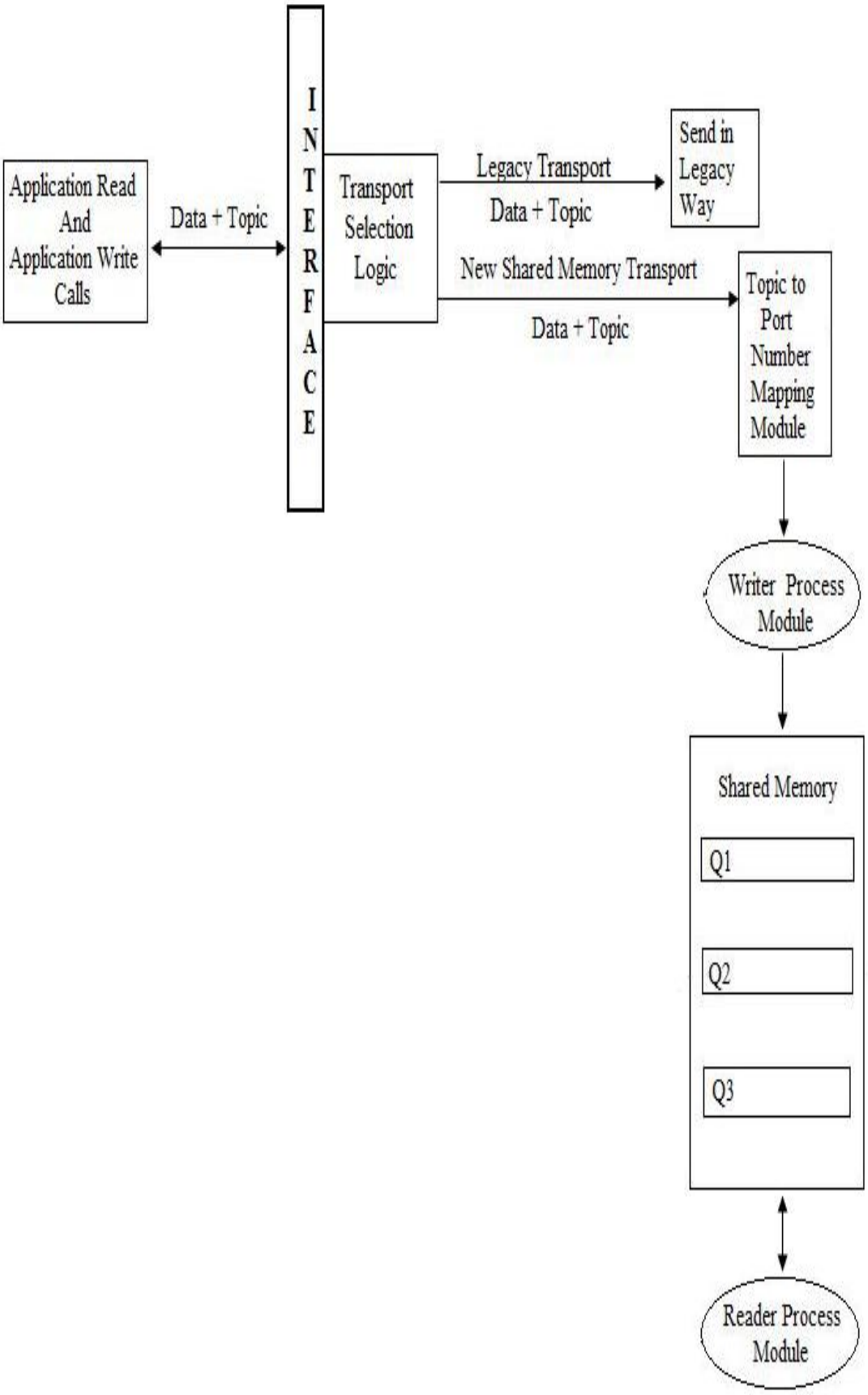

Figure 7: Architecture of the Proposed System

In order to have a complete and accurate picture of our proposed system it's important to first understand the systems with and without shared memory usage and the present communication scenario.

4.2 Systems With and Without Shared Memory Usage

In present market all the current messaging solutions as we discussed in Chapter – 2 are using sockets for communication between processes on local host as well as on different hosts. But this affects the performance and slows down the communication speed on local host. To overcome such problem, we designed a solution that would take the benefits of both shared memory and sockets and would result in a low latency system.

Before we consider about the various modules and their application in proposed system in detail let us also see the benefits of using shared memory for communication on local host. Let us take an example and show what exactly happens when User Process 1 communicate with User Process 2 with shared memory and without shared memory on local host.

Without using Shared Memory on local host:-

At application layer, we create message and write it to a buffer (1 copy operation) and then send it to socket which writes data from user space (area where program executes on RAM) to kernel space (2 copy operations). Now, Kernel thread would again write this data from kernel space to second user process space (3 copy operations). Therefore, it requires following number of copy operations and context switches:-

1. 3 copy operations.
2. 2 context switches.

Using Shared memory on local host:-

At application layer, we create message and write it to shared memory (1 copy operation) from where other process can access the data Therefore, it requires only following number of copy operations and context switches:-

1. 1 copy operation.
2. 1 context switch.

Figure 8 represents, communication between User Process 1 and User Process 2 with shared memory and without shared memory on local host:-

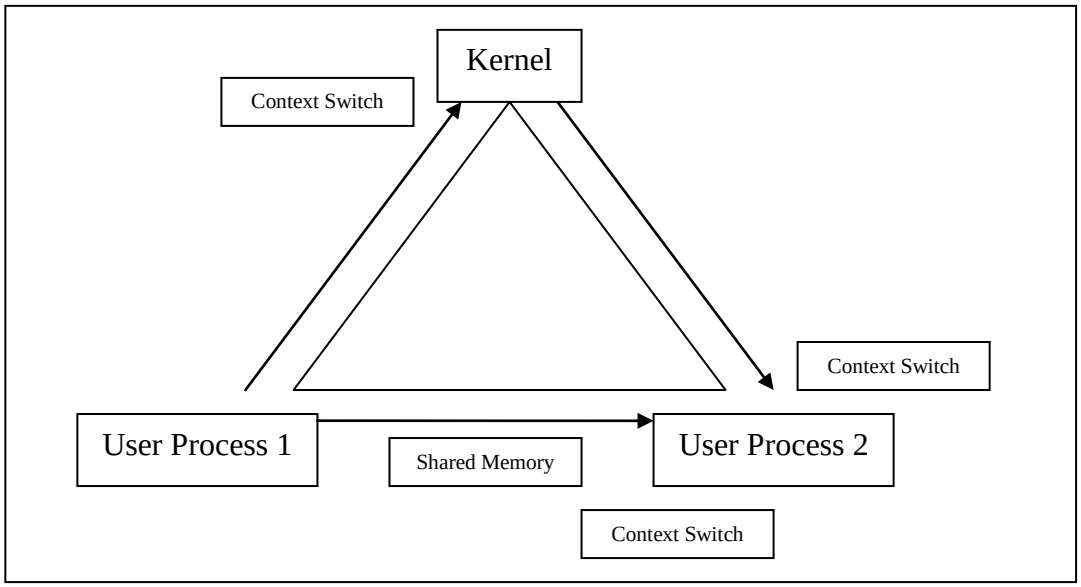

Figure 8: Local Host

In case of socket we just write data that needs to be communicated on socket. But if an application decides to use shared memory directly than:-

1. It has to take care of inter-process synchronization on shared memory.
2. And memory allocation and de-allocation on shared memory.

These two problems are not trivial and there is every possibility to introduce a bug. Therefore, in other to solve above mentioned problems we need to have an algorithm that would reduce the inter-process communication time on local host by implementing fast socket over shared memory. That is our main notion is to provide a Seamless Interface that would allow transparent communication for the message intended for any destination. Shared memory would be automatically used by an application if processes on the same host need to communicate. If communication between processes on different hosts is required than socket would be used automatically. Sockets are the most robust and convenient whereas shared memory is fastest with a limitation that processes should be on the same host. We need to marry these two and come up with a solution which provides Seamless Interface on top providing best features of both worlds.

4.2.1 Communication via Various Methods

If processes P1 and P2 are on same host and if P1 wants to communicate with P2, though shared memory is available however currently, sockets still seems to be a better choice considering the complexity involved when dealing with shared memory and flexibility provided when using sockets. If we remove the complexity of shared memory usage from application programmer than it suddenly becomes a better choice. However, we still have another limitation where we cannot communicate with a process on remote host using shared memory.

4.2.1.1 Communication via Sockets

Communication through sockets suffers from one main disadvantage and one main advantage. Advantage being the flexibility which means that with the help of sockets a process can communicate with a process on another host within a same network or on different network with an ease. And the disadvantage is that processes on same host communicate via kernel resulting in slower communication speed because of extra copy operations and more context switches.

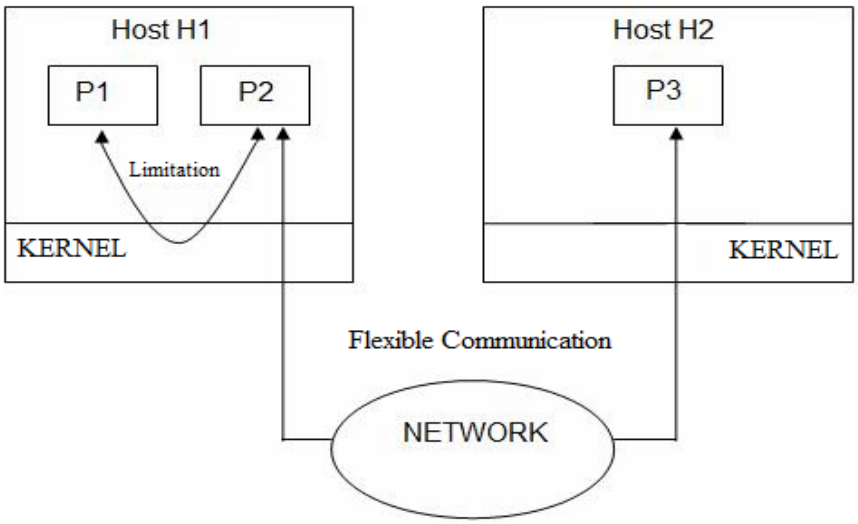

Figure 9: Communication via Sockets

4.2.1.2 Communication via Shared Memory

Shared memory offers relatively low latency, deterministic, high bandwidth inter-process communication [22]. Communication through shared memory suffers from one main disadvantage and one main advantage. Advantage is the fast speed of communication when processes on same host communicate. And the disadvantage is that with the use of shared memory processes on different host cannot communicate.

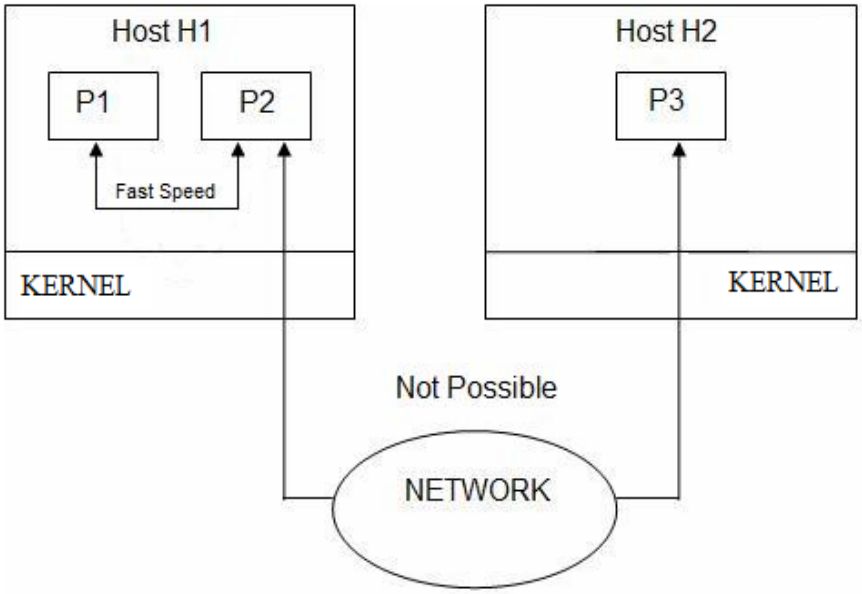

Figure 10: Communication via Shared Memory

4.2.1.3 Communication via Seamless Interface

Communication through seamless interface has two main advantages. First, advantage is the fast speed of communication when processes on same host communicate using Shared Memory Transport. Second, advantage is that it offers high flexibility with the use of sockets for the communication between processes on different hosts. The Seamless Interface selects the Shared Memory Transport or the Socket Interface seamlessly that is it's transparent from the user and the application program.

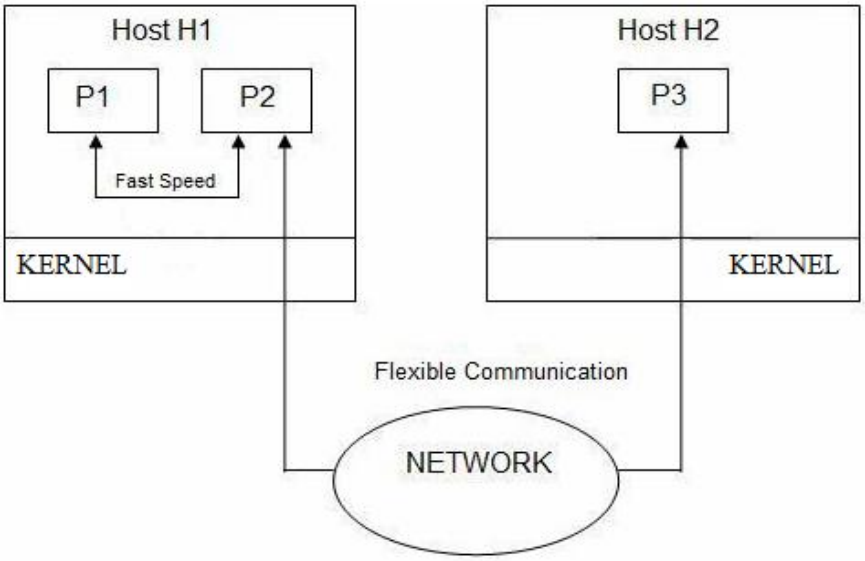

Figure 11: Communication via Seamless Interface

At present there is no integrated solution available. Our solution and design of Seamless Interface would definitely result in a low latency system that would be used by many organizations to reduce the time required to communicate data between processes.

4.3 Design Goals of Seamless Interface

Now, if we were to architect the best possible solution for the desired "Seamless Interface", it should have following features:-

1. It should be as simpler as sockets or even more simpler.
2. It should be as fast as shared memory on local host and as fast as using sockets for remote host communication.
3. It should be seamless. User should not be bothered to know about the underlying transport he is supposed to use.
4. User should be given with simple interface such as:

    transport.send (DataBuffer, TopicName)

Here, the topic name could be associated with one or more processes. Topics are resolved to a group of one or more IP address and port number pairs. All processes subscribe on topics they want to receive messages from and publish to topic they want to send messages to.

4.4 Algorithm for Seamless Interface

All design goals requires us to implement transport class which would have a send routine accepting a 'DataBuffer' and a 'TopicName'.

Here, we assume any of the available topic resolution mechanism would be used which would provide us with one or more IP address and a port number pairs associated with that topic. To simplify the matter for now we assume, it returns just one IP address and port number pair. Other scenarios would be trivial once we explain this use-case. So now here we are with an IP address and port number pair given by topic resolution agent. The first step is to figure out if this IP address is same as our own IP address. Remember we can have multiple IP addresses for a same host. A simple parser which would parse output of ipconfig file and give us a set of IP addresses associated with our local host. So let us keep this set of IP addresses in a

hash table which is initialized at process start-up. So we'll end up doing a lookup in this table. If address is found we would use shared memory as underlying transport otherwise we would use socket as underlying transport.

Step 1: User Application 1 → send (DataBuffer, TopicName);
Step 2: ipAddr = TopicResolver ("TopicName");
Step 3: status = localIPaddrHashTable.lookup (ipAddr);

```
if (status == true)
        sharedMemoryTransport.send (DataBuffer, TopicName);
else
        socketTransport.send (DataBuffer, IP, PortNo);
```

Now the problem boils down to writing shared memory transport and socket transport. Here we assume that any of the available transport would be used for socket transport and below we'll describe in detail only shared memory transport.

4.5 Shared Memory Transport

Shared Memory Transport would take care of inter-process synchronization while using shared memory and also memory allocation and de-allocation on shared memory. It would reduce the inter-process communication time on local host by using shared memory and facilitates implementation of a Seamless Interface that would encapsulate Shared Memory based Communication Interface and Socket based Communication Interface. Shared Memory Transport would be automatically used by an application if processes on the same host need to communicate. If communication between processes on different hosts is required than socket would be used automatically. Hence, we would require Shared Memory Transport and a Seamless Interface built on top of it.

Figure 12, helps visualizing "Shared Memory Transport" and gives insight on data flow within the system. It is simplified to have just two processes. However, Shared Memory Transport should be designed to do much more, logically there is no limit on number of queues a process can write to or read from. Utmost care should be taken to maintain contention free multi process accesses.

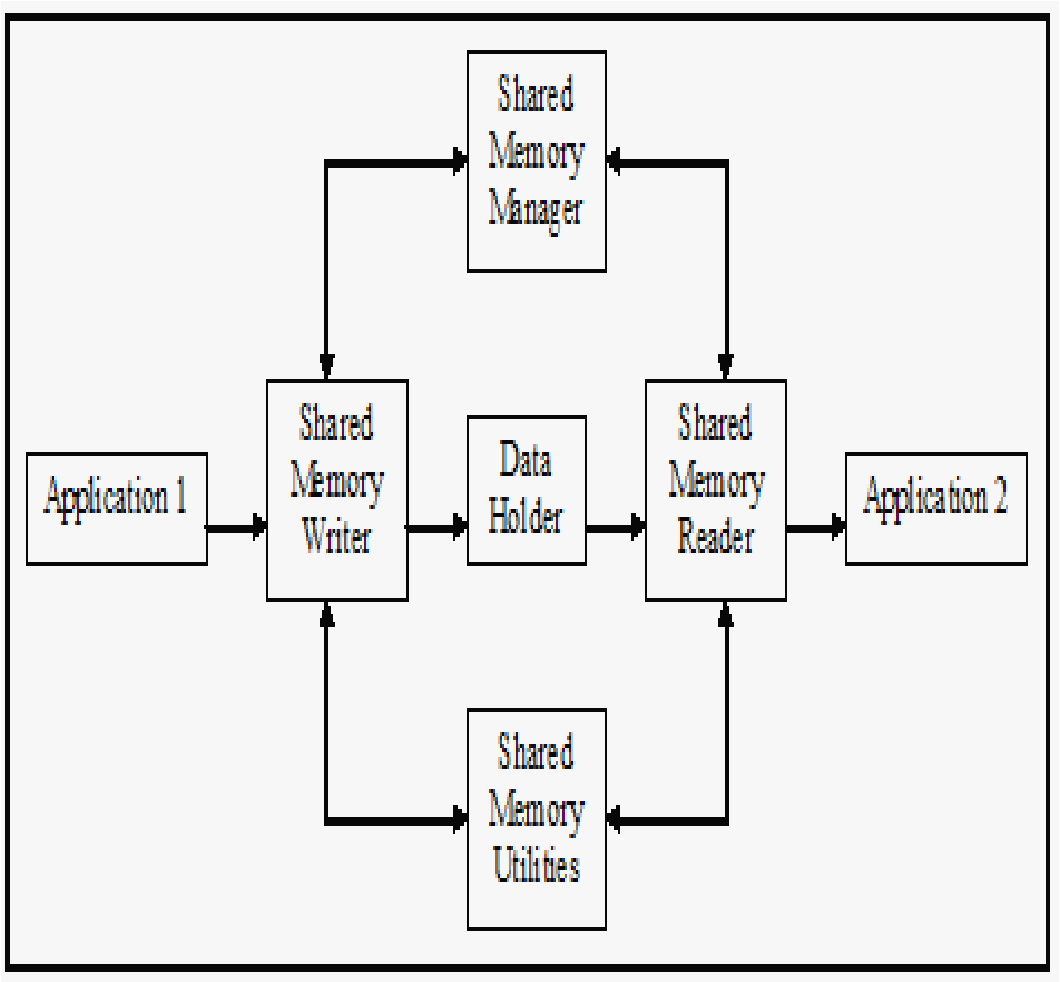

Figure 12: High Level View of Shared Memory Transport

Processes can share queues (similar to "Anycast-IPv6" with additional advantage of destination load balancing). Shared Memory Transport provides lock-free access to processes accessing common queue. This lock-free concept would be discussed in detail in subsequent sections.

Shared Memory Transport should follow a layered architecture. It should be well abstracted, easily extendable. It should be designed to give very low latency and at the same time maintainability should not be compromised. It fulfils all the above mentioned criteria and the out come would be a well-designed low latency system.

Figure 13, helps us to have a high level or broad understanding of the Shared Memory Transport and its components.

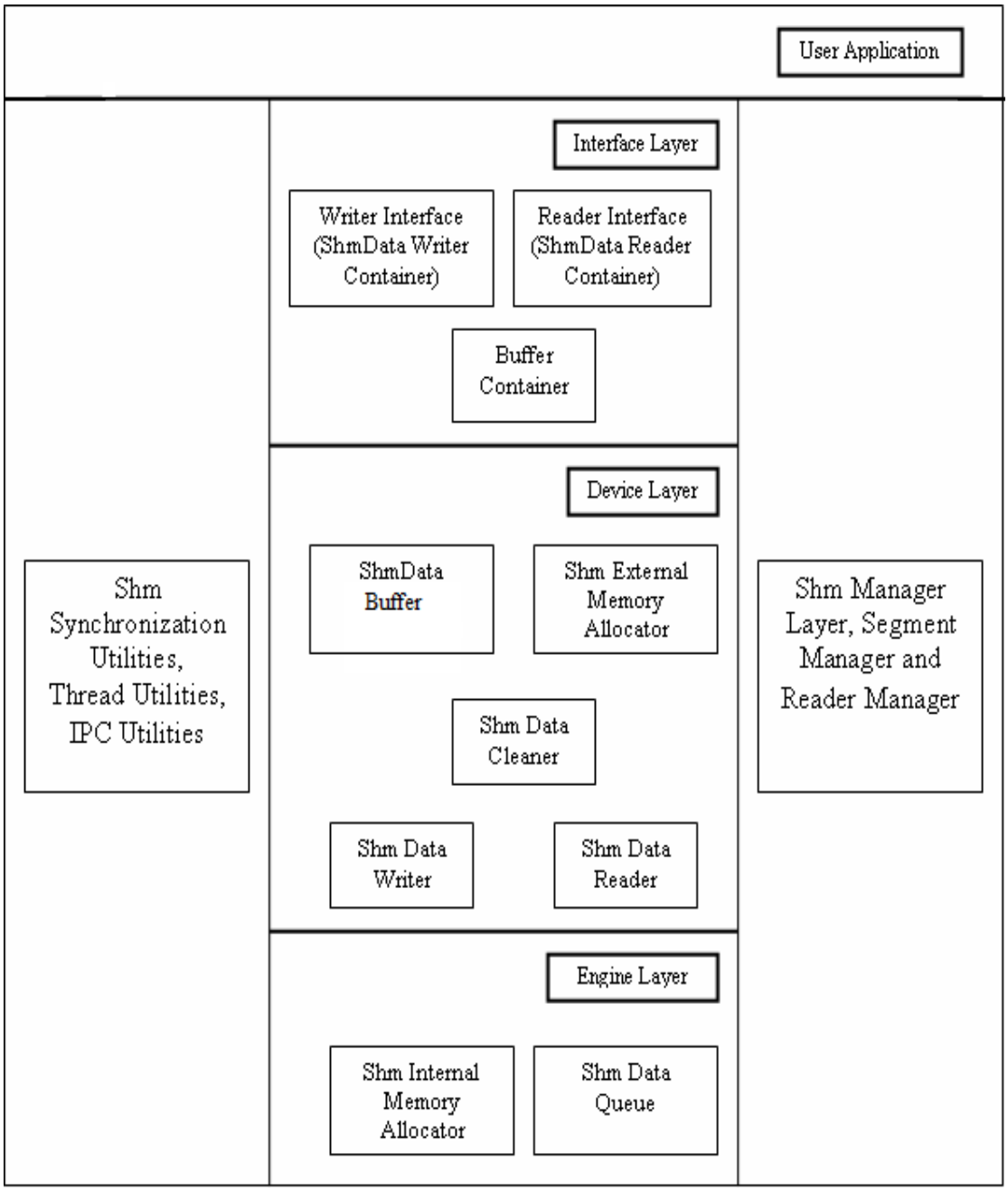

Figure 13: Shared Memory Transport Architecture

4.5.1 Main Components of Shared Memory Transport

The Shared Memory Transport architecture is divided into 3 blocks: Interface Layer, Device layer and Engine layer. Here, user application represents actual running application. Interface Layer includes all the required user interfaces, the device layer shows the implementation and engine layer shows the helping modules that are used by modules at device layer for their working. We separate interface from implementation to keep changes of underlying implementation transparent from the

application. Interfaces are most important asserts of an organization, an interface change can mean reworking an entire application which is not desirable. Hence, we keep interfaces independent of implementation. To draw an analogy you can consider C++ virtual base class as an interface while the derived class as an implementation. You can replace the entire derived class with a new derived class leaving the interface or base class or virtual base class intact. Interface class has all methods defined as pure virtual methods. If you are giving a generic transport to a wide group of applications then you would want to give something which is not prone to changes. However, you know that any solution you give is prone to changes to inculcate future technologies so to separate these two we give interfaces to user application and keep implementation to ourselves so that implementation can change and interface remain intact.

4.5.1.1 Buffer Container

This shall be the shared memory buffer container class. It would have abstract interfaces to access shared memory buffer. This abstraction would keep the changes to underlying system transparent from user applications.

4.5.1.2 Writer Interface

This abstract interface shall serve as the medium to write data to shared memory transport. This abstraction would keep user application transparent from underlying implementation changes.

4.5.1.3 Reader Interface

This abstract interface shall serve as the medium to read data from shared memory transport. This abstraction would keep user application transparent from underlying implementation changes.

4.5.1.4 Shared Memory Data Buffer

This class would encapsulate shared memory buffer handling specific operations and any of the data units. In particular, we need a buffer while serializing data. Buffer Container internally will use memory allocator to get a buffer from the shared memory for serializing data. This is basically used to increase the efficiency since the other option would have been to serialize in an application buffer and then later copy

it in a shared memory buffer. So the better option was to directly get a buffer from shared memory and directly using it for serializing and storing data.

4.5.1.5 Shared Memory Data Writer

This class would encapsulate methods and data member required in facilitating efficient multi-threaded shared memory data writing capabilities to the user application. This encapsulation layer is an implementation of Shared Memory Data Writer Container. This would contain two methods:-

1. 'IsQueueReady' would be used by implementation to return status of queue whether it is initialized or not, whether it is ready to write or not.
2. 'write' would be the actual interface to be used to write data on to the shared memory. However, write interface would accept a shared buffer container (shmBuffer Container) type object so actual data writing on shared memory would take place when you will create and populate shared memory buffer container object. This write method would take care of writing the object on the shared memory data queue. In short, writer interface will take care of writing pointer of memory location containing buffer, in the queue pointer.

4.5.1.6 Shared Memory Data Reader

This class would encapsulate methods and data member required in facilitating efficient multi-threaded shared memory data reading capabilities to the user application. This encapsulation layer is an implementation of Shared Memory Data Reader Container.

It will expose interfaces to client application for registering their data call back functions. These call back functions would be given shared memory buffer container object. So the reader interface will take care of reading data from the queue, putting it into the buffer container object and passing it to client application call back function.

4.5.1.7 Shared Memory Data Cleaner

This class would encapsulate the functionality of shared memory garbage collector. It is a supportive class to implement memory allocation and de-allocator module. Its main usage can be explained as follows:-

1. If Reader Process fails before setting ‘used’ in ‘shmDataHolder’ Structure (discussed later).

   No effect
2. If Reader Process set ‘used’ and fails or kills before setting ‘finish’ in ‘shmDataHolder’ Structure.

   In this case if Writer Process tries to write at that location, it will find that ‘finish == false’ and ‘used == true’ which means some other process is using that location. In that case writer will skip this location and goes to next location and performs same task. This will result in many unused skipped holes.

Therefore, there would be other cleaner thread in a writer process which traverses through all the queues of Writer Process checking the timestamp (user can configure expire timeout externally) and cleaning up any of the expired data from the queues. Here, expired data is the one, whose ‘used == true’ and ‘finish == false’. Cleaner thread would clean this slot and hence, there would be a loss of only 1 unit of data. In contrast to RT Server, which has queues of size 50MB which drops completely if client disconnects and in case of sockets also, if socket disconnect, data in socket buffer will also be cleaned.

Algorithm for Cleaner Thread in Writer Process:-

if (timestamp_time + expire_time < current_time)

    Data not expired

else

    Data expired

    Perform cleaning operation

4.5.1.8 Memory Allocator

Memory Allocator would be used by application to allocate memory within shared memory segment and send to ‘Shared Memory Data Writer’ to write. The shared transport will implement a two stage memory allocation: Shared Memory External Memory Allocator and Shared Memory Internal Memory Allocator. The concept is in-line with operating system implementation of malloc() and free() in the sense that malloc() maintains application specific heap in form of a data structures, allocates and

de-allocates from within. If need be requests from the operating system's memory allocator for extra heap space by may be using brk() or sbrk() system calls. Similarly this external memory allocator would be used by buffer container module to get memory. This memory in-turn would be requested by external memory allocator from internal memory allocator which would be implemented as a fixed sized block allocator. So in effect every request to internal memory allocator would return a fixed size block. If the application will require more space then external memory allocator would request for another block from internal memory allocator.

4.5.1.9 Shared Memory Data Queue

Shared Memory Data Queue would be the actual storage area, where the Writer Process would write and from where Reader Process would read. The queue is designed to optimize single Writer and multiple Readers. This design is inline with the fact that every Writer would have its own shared memory segment to write to and multiple Reader processes would connect to the queue. Multiple Readers access the queue in a lock-free fashion by using atomic CAS operations. Every element within this queue is of type 'shmDataHolder' where every holder would contain pointer to data with other control information which would be discussed in detail in subsequent section.

4.6 Shared Memory Transport Design Details

Shared memory technology allows processes to exchange data and synchronize execution. It is the fastest form of inter-process communication mechanism known so far because the memory is mapped into the address space of the processes that are sharing the memory region and then processes do not execute any system calls into the kernel in passing data between processes. Therefore, for the communication on local host shared memory is the best inter-process communication mechanism.

Our Shared Memory Transport design would definitely lead to faster communication and would certainly help those organizations where real-time processing and fast communication is a necessity. But efficient implementation of shared memory requires proper coordinate and synchronize between processes that are using shared memory for communication.

Our proposed Shared Memory Transport is a simple transport which would hide the complexities involved in using shared memory by providing a layer of abstraction. It provides a convenient and easy interface to be used by users.

This design supports single Writer Process and multiple Reader Processes. However, there could be multiple Writer threads within Writer Process. Moreover, it is a scalable design since there is no constraint on number of Readers that can connect to Writer shared memory segment and size of the shared memory is also configurable according to the application needs.

It offers a flexible design since it supports varying size multiple queues as per receiver's application requirement. These queues are designed in a circular lock-free fashion since there is no lock between Writer and Reader Processes. Writer Process would write on head_index of queue and Reader process would read from tail_index of queue any time without locking queue, provided space is there or data are there respectively. Moreover, use of atomic operations would substantially provide extremely low-latency system that would perform extremely well.

Inputs to the Shared Memory Transport are the data and the IP address-port number pair i.e. send (DataBuffer, TopicName). So the transport should some how be able to send this data to a queue which is in shared memory and is read by the recipient i.e. the Reader process.

To enable this, following operations are performed by Writer and Reader Processes:-

1. Every Writer Process has a shared memory writer component. This component creates a shared memory segment and initializes data queues for reader groups (one or more readers) within this component.

2. Every Reader Process has a reader component which connects to shared memory segments of all writers that it intends to read from. Within those segments it connects to its queue. Whenever some data arrives it sets 'used == true' in 'shmDataHolder' structure atomically and starts reading. Once reading is finished it sets 'finish == true' in 'shmDataHolder' structure so that the writer can clean it.

3. Reader/Writer synchronization is done with semaphore. Writer after writing data, signals on a semaphore on which reader is waiting. Every reader connects to its queue in writer segment and waits on its queue named semaphore. This semaphore can be signalled by either of the writers. On the receipt of signal reader iterates through all its queues in various writer segments and processes data found there.

In figure 14, Writer after writing to queue Q1 will signal to Q1 semaphore. All writers would signal to this common counting semaphore if the request is for a reader waiting on unique queue id Q1. If multiple writers signal simultaneously, the counting semaphore will increment. This is how synchronization would be done between Writer and Reader Processes.

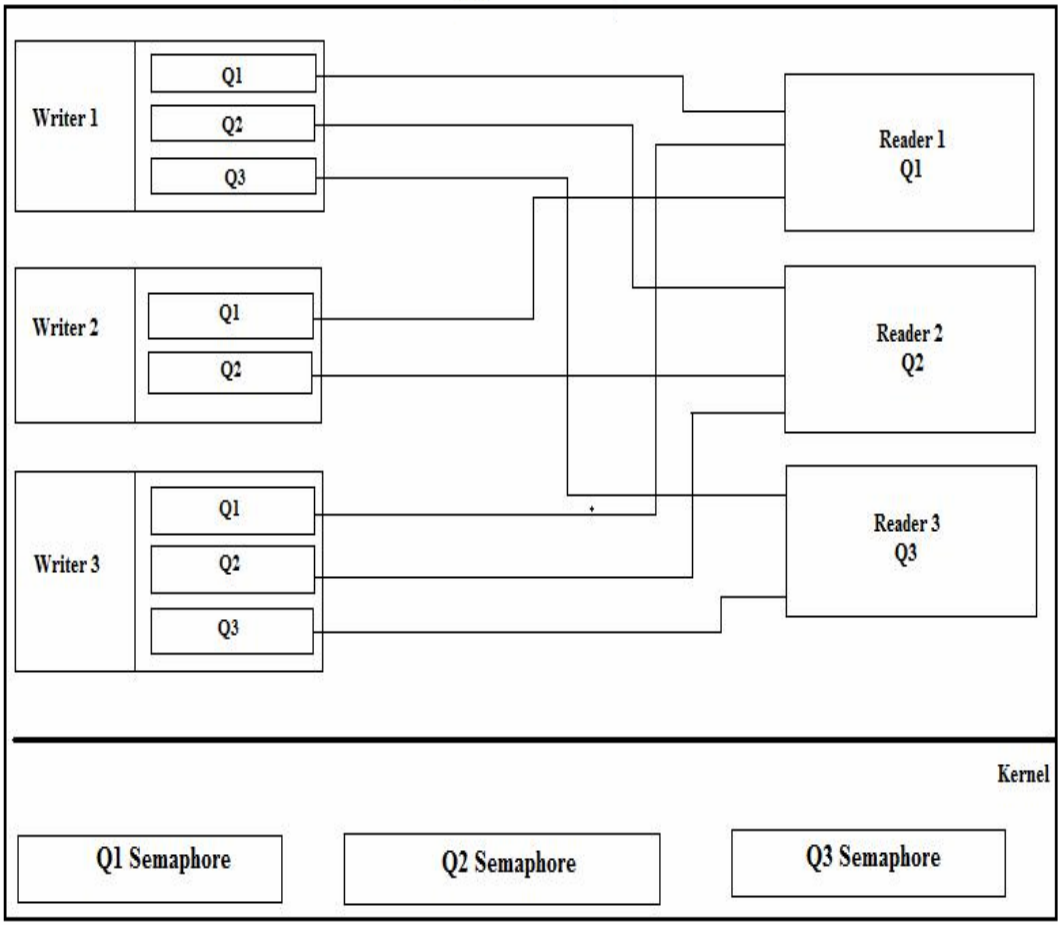

Figure 14: Simple Connection Diagram between Writer Processes and Reader Processes in Shared Memory

4.6.1 Other Design Challenges with Solutions

4.6.1.1 UNICAST

Our Shared Memory Transport design would support Unicast which means one Reader Process would be associated with 1 queue only. MULTICAST is not supported by this design though it can be considered as future scope of this work; multicast basically means multiple readers reading from same location.

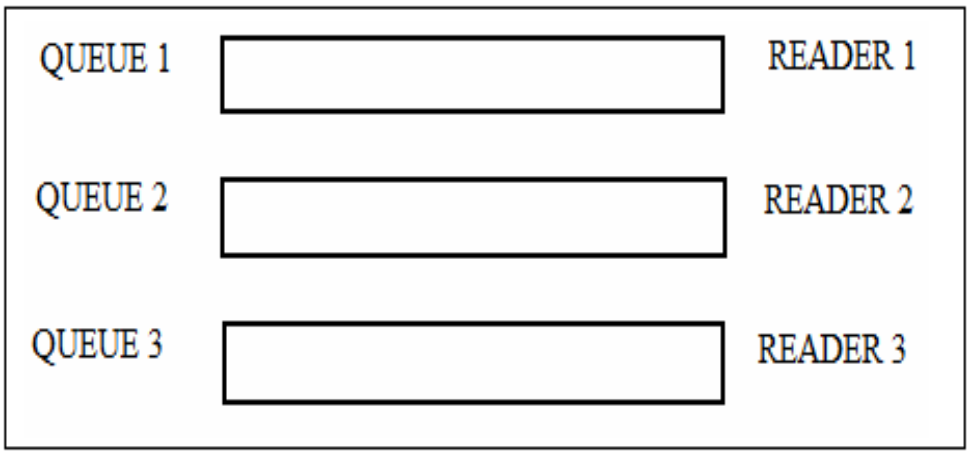

Figure 15: UNICAST Supported Design

4.6.1.2 ANYCAST

Our Shared Memory Transport design would also support Anycast which means multiple readers can access a particular queue but a particular location in queue would be accessed by one reader only.

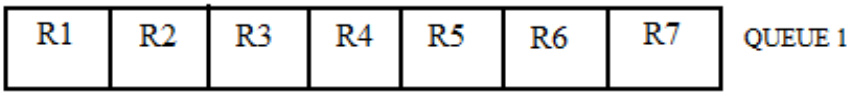

Figure 16: ANYCAST Supported Design

There is another scenario where multiple readers can be connected to the same queue and either of them would receive data (true load balanced). In the diagram below though a queue has a unique id (Q1-Q2) but still Reader1 and Reader2 which are connected to this queue would consider it as two separate queues.

Here Reader1 and Reader2 share the queue as well as semaphore. On receipt of signal either of the readers would wakeup and read the data exactly same as single process connected would have done. True load balancing would be achieved here since whichever process will be ready goes to 'wait' of semaphore and hence is signalled. In event of both processes are ready and waiting, only one of them would receive the signal and hence, process the data. In short, if there are multiple readers waiting on same semaphore, they all will go to 'wait' of semaphore and when the Writer Process will signal, either of the reader waiting would be signalled and will process the data.

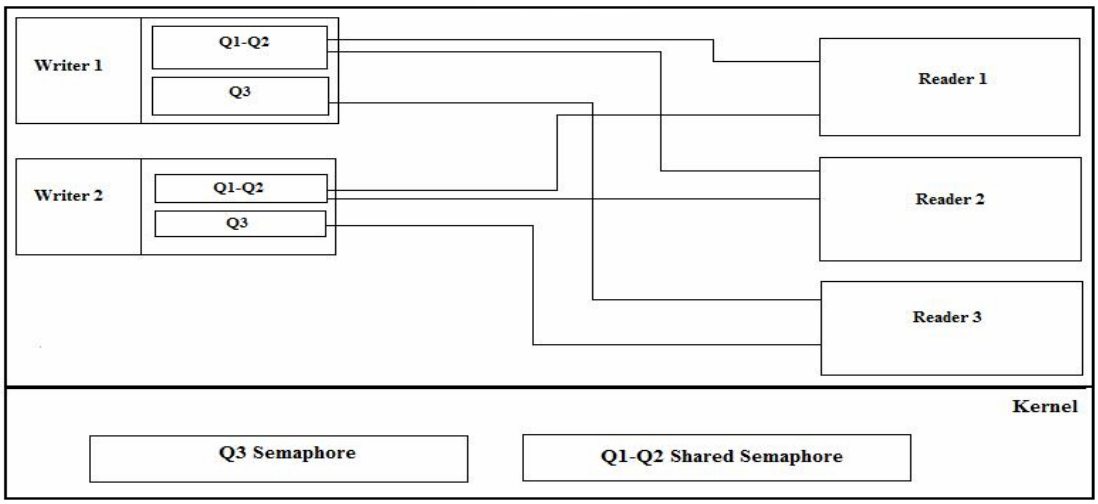

Figure 17: Multiple Readers Connecting Same Queue (ANYCAST)

4.6.1.3 Process as a Writer and Reader Both

It is not necessary that a process can be either Reader or Writer. A process can be Reader and Writer both.

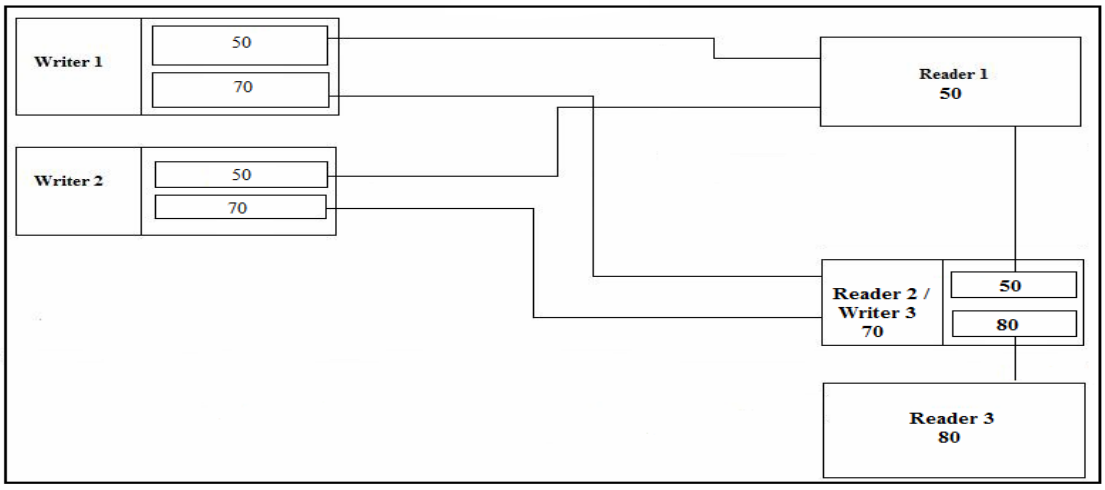

Figure 18: Process as a Writer and Reader Both

4.6.1.4 No lock between Writer and Reader Process

Writer Process or Reader Process never waits. As per this design there is no lock between Reader and Writer process, in the sense, that no common mutexes between these two. Anyone can read or write at any time provided data is there and space is there. This is accomplished by having only one Writer Process and multiple Reader Processes architecture. Moreover, Writer is designed to write on head_index and Readers are designed to read from tail_index atomically.

4.6.1.5 Readers accessing queues in lock-free fashion

When queues are shared between Readers then queues become accessible to multiple Readers in a lock-free fashion. All readers share the same volatile tail accessible through header of the corresponding queue. The tail is incremented automatically using CAS (compare and swap) CPU instruction. No reader process would be blocked and all will access current tail atomically.

4.7 Writer Process Module

Each Writer Process has a shared memory segment associated with it. And this shared memory has one or more queues. Each queue has a corresponding port number associated with it. This port number represents the corresponding Reader which is associated with that particular queue. Moreover, synchronization between processes is also achieved by giving unique name to semaphore, same as the queue id and the Reader Process waits on this semaphore and Writer Process signal on this semaphore. head_index is the one end of queue which always points to current location to write data. This head_index is always an empty location where writer can write.

After writing, ‘Data Writer’ sets ‘used == false’ and ‘finish == false’ in ‘shmDataHolder’ structure. When reader starts reading this unit, it sets the ‘used == true’ and after consuming data sets ‘finish == true’. This queue is implemented in a circular way. When Writer Process comes back to same location and finds ‘finish == true’ then it calls ‘free’ of memory allocator module to free the memory being used by data. Each queue in Writer shared memory segment contains a ‘Queue Header’ associated with it. This Queue Header is basically a data structure containing

information like address of header pointer, tail pointer and queue registration data array.

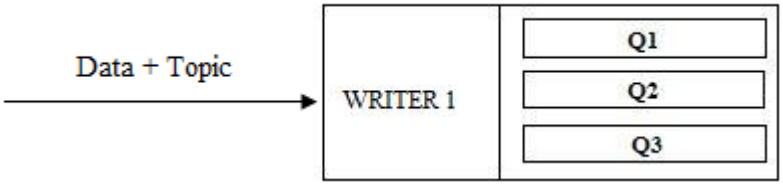

Figure 19: Writer Process in Shared Memory Segment

4.7.1 Algorithm for a Writer Process

Step 1: Verifies if reader is alive, using 'Heart Beat Maintenance' module.

Step 2: Writes data at head_index of queue and then increment.

```
if (head_index + 1 == tail_index)
        Queue full
        Wait
else
        if (ptr == 0)
/* pointer in shmDataHolder structure, if its '0' it means no data at this location
and Writer can write */
                Write
                Increment head_index
        else if (finish == true)
                Delete old data      /* call free() of memory allocator */
                Write
                Increment head_index
        else            /* Data is there but not used by any Reader */
                Increment head_index (skip hole)
```

Step 3: Signals on semaphore.

4.7.2 Heart Beat Maintenance Module

'Queue Registration Data' is a structure which contains heart beat or a time stamp of a Reader Process. This is an important module to avoid Writer Process to write in that queue which doesn't have corresponding Reader Process. If a Writer Process is writing in a queue whose corresponding reader does not exists then in that case there

would be memory overflow and unnecessary wastage of CPU time by Writer Process. To avoid that and to inform Writer Process about the existence of Reader Process this module is really helpful.

This module can be explained with an example. Say, every Reader Process needs to time stamp its presence after every 20ms in shared memory. And Reader Process current time = 10ms. Therefore, before writing anything in a queue every Writer Process should check for following condition:-

$$WT - RT > 20$$

Where,

WT = Writer Current Time

RT = Reader Current Time

And constant 20 represents the worst case. After which Writer Process can be sure that Reader Process doesn't exists any more.

4.8 Reader Process Module

It's the responsibility of a Reader Process to connect to the corresponding Writer shared memory segment. Reader performs an atomic increment, that is, it takes two parameters current tail_index and next tail_index and return incremented tail_index. tail_index always points to 'ready to read' location from where a Reader can read.

4.8.1 Algorithm for a Reader Process

Step 1: Waits on a semaphore.

Step 2: Checks if the data is available (head != tail)

then

Atomically increment the tail and read data from original tail_index.

else /* head_index == tail_index) */

Queue empty, iterate to next queue pointer in ready queue until all queue pointers in ready queue are checked and then goto Step 1.

Each Reader Process has two threads associated with it Thread 1 and Thread 2. Thread 1 is a ‘Reader Manager’ and Thread 2 is a ‘Reader Main’.

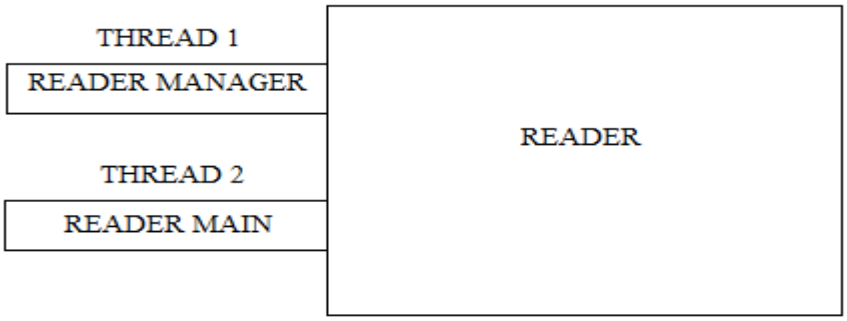

Figure 20: Reader Process in Shared Memory Segment

4.8.2 Reader Manager Thread Module

Reader Manager Thread which we also call a connection manager is responsible for creating a connection with all possible Writers segments on the same host and adds the queue address to a ready queue which in turn would be used by Thread 2 to read data from.

It is also accountable for creating a ready queue with pointer to different shared memory queues. Ready queue is nothing but a circular queue that contains the pointers or address to the connected queues.

Additionally Reader Manager Thread has the responsibility to timestamp its presence by populating ‘Queue Registration Data’ structure and then maintaining heart beat or time stamping through out the connection period to indicate its presence. This would be taken care off by ‘Heart Beat Maintenance’ module discussed before.

4.8.3 Reader Main Thread Module

This thread keeps on iterating through the ready queue which contains the pointer to corresponding writer queues to know when the data arrives in write queue so that it can read it. But this results in ‘Busy Waiting’ condition which can be avoided by using semaphores. So its main responsibilities are:-

1. Iterates through ready queue.
2. Reads data when it arrives.

4.8.3.1 Algorithm for Reader Main Thread Module

While iterating ready queue

if (head == tail)

no data, goto next field in ready queue.

else

read data from tail and increment tail.

Figure 21 shows the connection between Writer and Reader Process along with the Reader Process Threads with the ready queue.

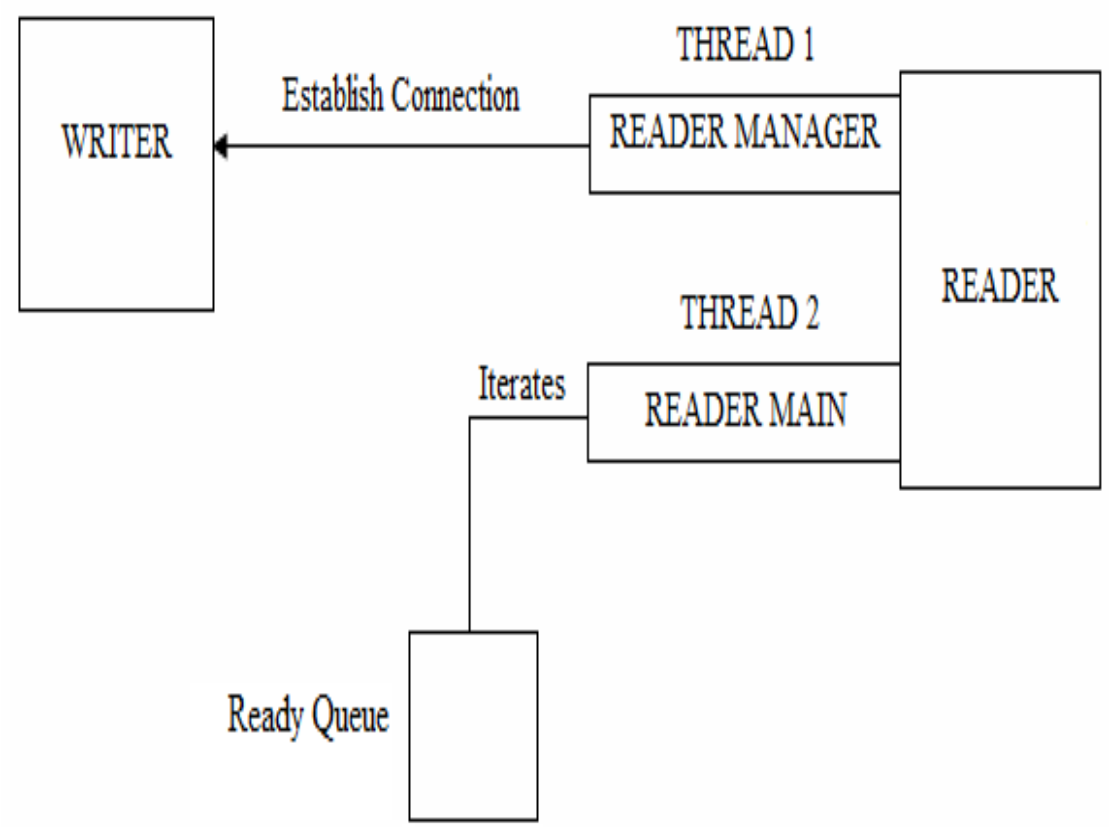

Figure 21: Threads in Reader Process

4.8.4 To avoid Busy Waiting in Ready Queue of Reader

1. All Writer and Reader Manager try to create same semaphore. However, in 'Non-exclusive' mode which means that create semaphore if it doesn't exist otherwise connect to the existing semaphore.
2. All writer signals on same semaphore and Reader wait on same semaphore.
3. As soon as Writer writes data on queue it signals the Reader so that it can read the data from that queue.

4.9 Shared Memory Design

The entire shared memory segment is divided into number of blocks where each block performs an important function and contributes in efficient functionality of an entire shared memory segment.

4.9.1 Writer Process Role

It's the responsibility of the Writer Process to create a shared memory segment. After creating shared memory segment it writes 0x00000000 in first 8 bytes of first 8K size of block of that segment. These 8 bytes represents 'Signature' of the shared memory segment which is used to verify that shared memory segment is correct and can be used for further processing. After writing 0x00000000 in first 8 bytes Writer Process writes 0xAA55AA55 at that very location again to indicate the shared memory segment has created all the required queues and it's ready to be worked upon by the Reader Process. This concept of initializing memory location with 0xAA55AA55 has been picked from operating system boot sector signature.

Following are the snapshots of shared memory segment at different steps:-

Step 1: On creating shared memory segment

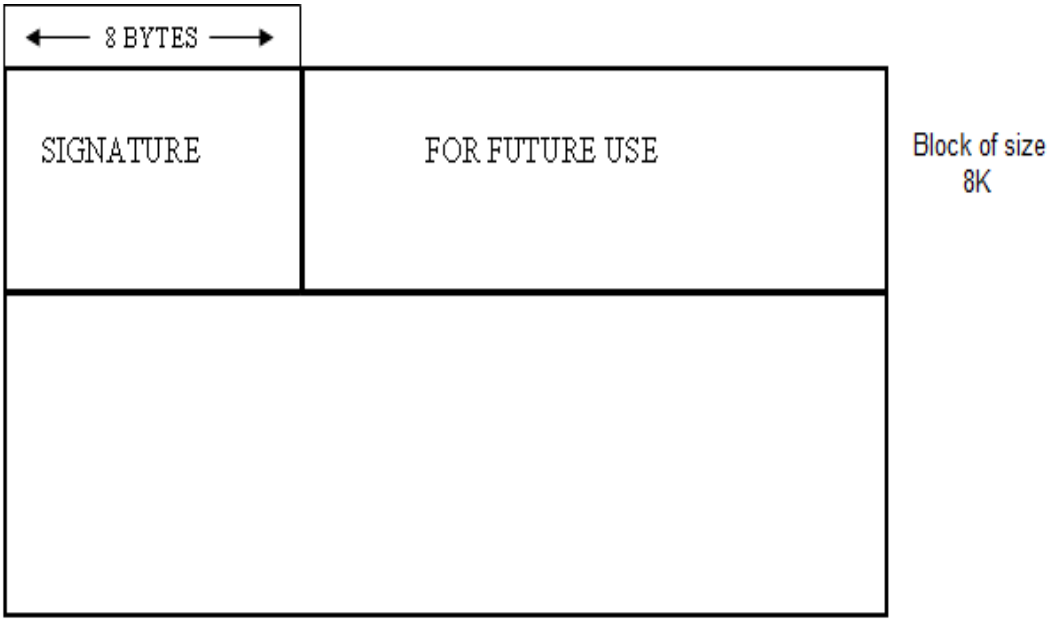

Figure 22(a): Shared Memory Segment – First Block

Step 2: Immediately after creating shared memory segment and writing 0x00000000 to first 8 bytes of the first block of size 8K.

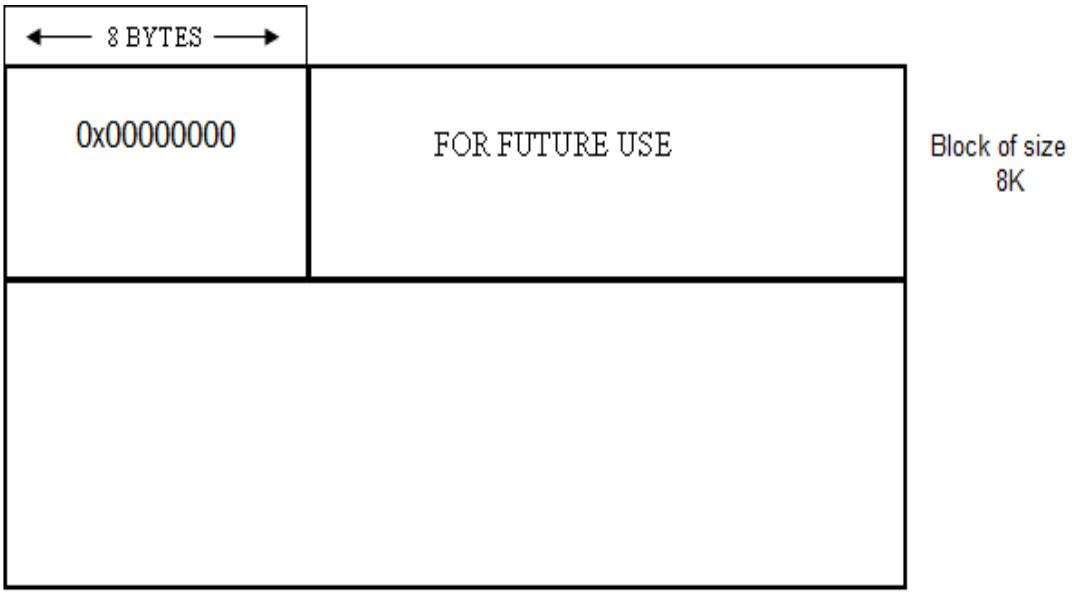

Figure 22(b): Shared Memory Segment – First Block

Step 3: After creating all the required queues in the shared memory segment and when it's ready to be used by Reader Process.

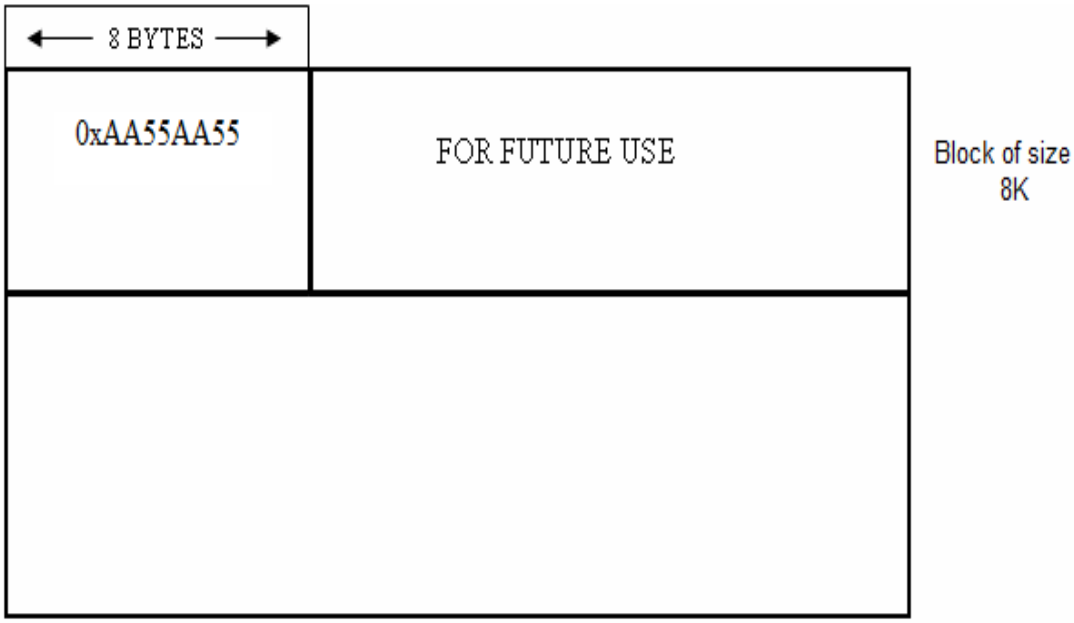

Figure 22(c): Shared Memory Segment – First Block

4.9.2 Reader Process Role

It's the responsibility of the Reader Process to connect to the shared memory segment and after connecting it checks the first 8 bytes of that particular shared memory segment. If its other then 0xAA55AA55, it will wait else will start reading.

The remaining part of first block is reserved for the future use which can be used to implement more control functions to have a better control over shared memory segment or for other future requirements.

Now, in the shared memory segment next 4K size of block is reserved for 1024 blocks of 4 bytes each. Each block contains the starting address of each queue header in the shared memory. This particular design of shared memory support 1024 blocks only, which seems to be sufficient number of queues in the shared memory.

Initially when shared memory segment is created at that time each of these blocks will be initialized with '0'. After creating queues these blocks are initialized with addresses of queue headers.

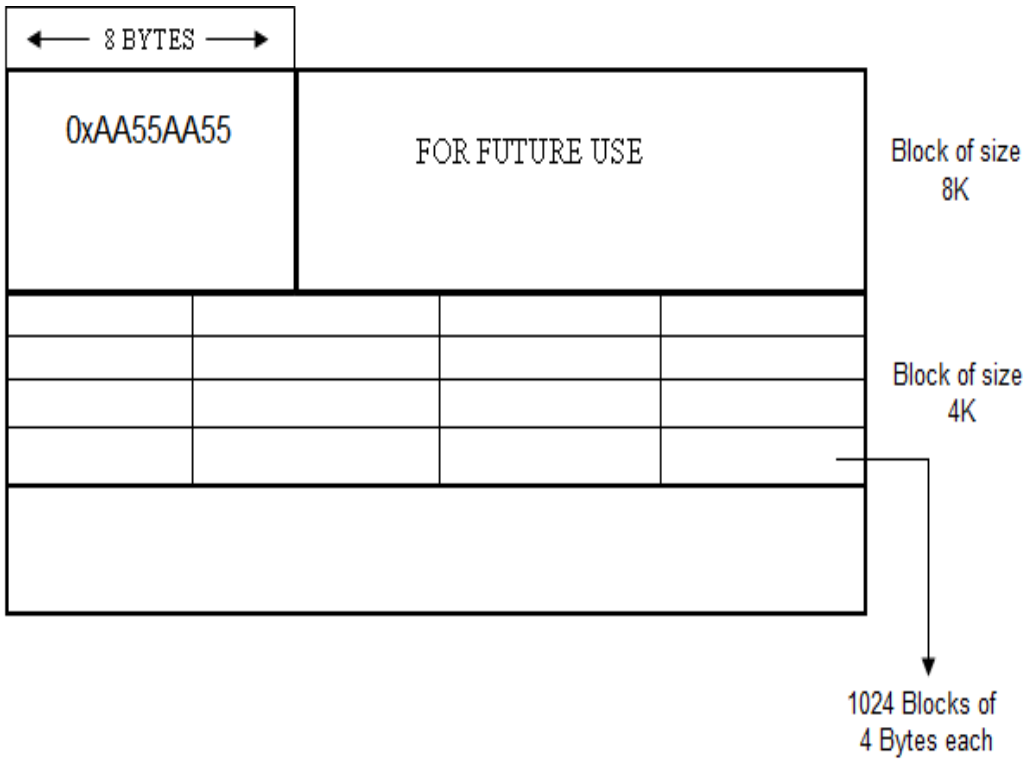

Figure 23: Shared Memory Segment – Second Block

4.10 Space Efficient Lock-Free Virtual Queue Design

The queue design is closed to virtual queue but not really a virtual queue. Every queue is designed as a fixed length virtual array where length of array is equivalent to the amount of data the complete shared memory segment can support. The queues are designed as fixed size circular queue and asserts are used to identify queue empty and queue full conditions.

Following are the Main Asserts:-

1. Reader Process Assert or Queue Empty

   if (rear == front)

2. Writer Process Assert or Queue Full

   if (rear == front + 1)

3. The increment of indices happens in following way:-

   if (index == max_index-1)

   index = 0;

   else

   index++;

The next important block of shared memory segment whose size is variable and depends on the arguments that we pass through environment variable can be represented as follows:-

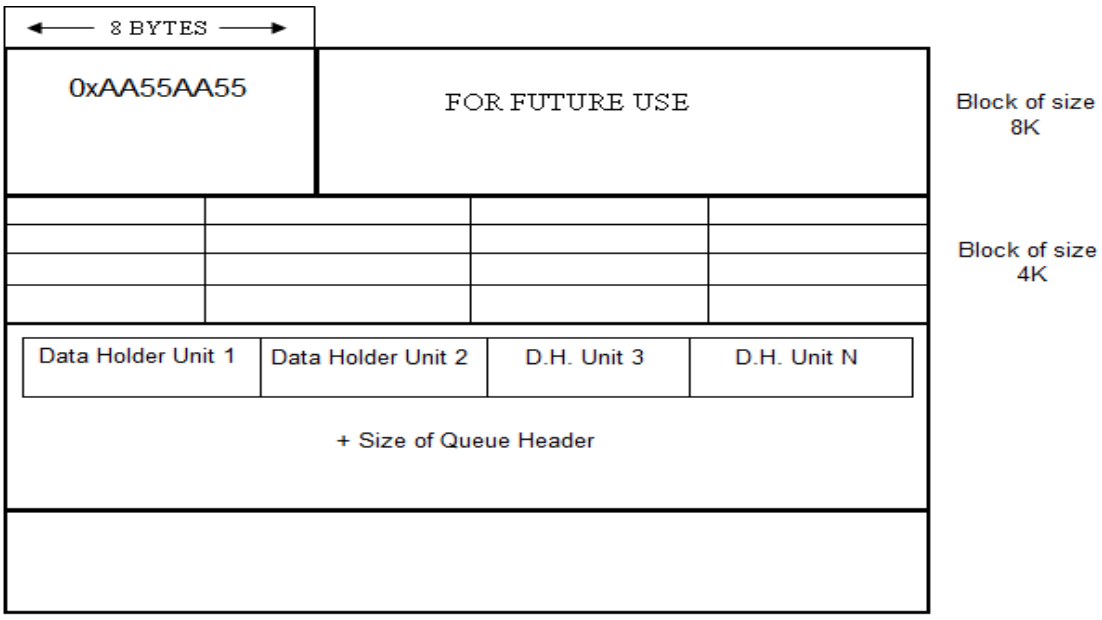

Figure 24: Shared Memory Segment – Third Block

Figure 25, represents pointer to actual data in the shared memory. Each shared memory data unit which is basically a data structure whose data members would be discussed later has a pointer which points to a memory location that contain data which is inserted by Writer Process to be consumed by Reader Process.

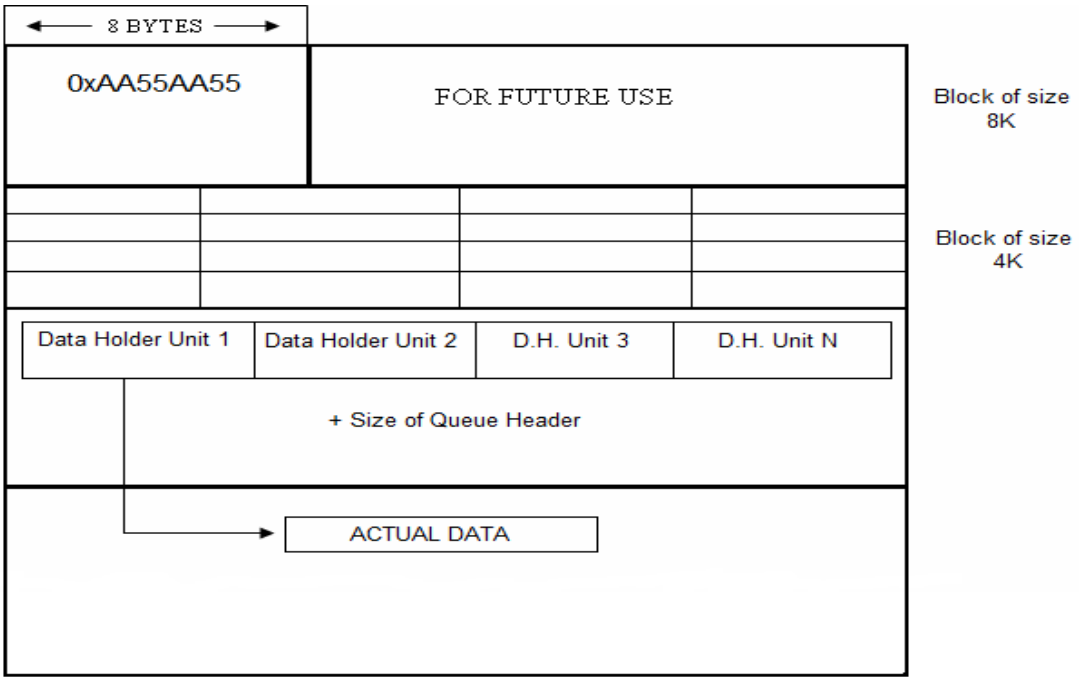

Figure 25: Shared Memory Segment – Fourth Block

Every queue is an array of data holder units. Here, every data holder unit is a structure with following data members:-

```
struct shmDataHolder
{
        bool used;
        bool finished;
        void * offsetPtrData;
        time_t timestamp;
};
```

Size of above structure = 1 + 1 + 1 (alignment padding) + 1 (alignment padding) + 4 + 4 = 12 Bytes.

We are using the concept of data holder unit and using a pointer in it to point to actual data. Since placing the data at that very location would waste lot of space and would not allow Writer to share the data space in between data queues which otherwise makes the sharing possible and hence, space efficiency is increased tremendously.

Now, say user has specified following data through environment variable:-

QUEUE_SIZE_LIST = 25,500:26,600:99,100

This is a ':' separated list of ',' separated pairs which is read by Writer Process. Each pair represents QueueId and number of elements in that queue.

e.g. In above QUEUE_SIZE_LIST, following three queues are specified:-

1. Queue Id = 25 and Number of elements = 500.
2. Queue Id = 26 and Number of elements = 600.
3. Queue Id = 99 and Number of elements = 100.

Size of data holder unit 1 whose QueueId is 25 = 12 * 500 = 6000 Bytes.

Apart from the space reserved for the queue elements i.e. 6000 Bytes, another chunk of memory is allocated to store 'Queue Control Information' or 'Queue Header' structure which is as follows:-

```
struct queueHeader
{
        unsigned int head_index;
        unsigned int tail_index;
        unsigned int max_index;
        shmDataHolder * offsetPtrDataHolderArray;
        mutex_t tail_mutex;
};
```

Note: Any pointer which is stored in shared memory is offset pointer to get actual pointer. You have to add offset address and base address to get actual address.

Reader performs following calculation in constant time to calculate its queue address:-

```
unsigned * arrPtr = baseAddress + 8K;
myQueueAddr = arrPtr[queueId];
```

Now, Reader Process can read and process the data from the address stored in myQueueAddr.

4.11 Generic Serialization and De-serialization Module

4.11.1 Serialization by Writer Process

Generic Serialization Module is used to serialize any application data structure into a character buffer terminated by '\0'. It implements this by storing function pointer variable that stores the address of application serialization function at application start-up (while initializing transport).

Each application implements its data structure serialization function, which accepts a "void pointer", a "pointer to buffer" and "size of buffer". This function is used while registering serializer function with the transport. The process is known as *"Registering Serialization Function Pointer with Transport".*

Step 1:
Send the address of serialization function to the generic serialization module. In turn, it stores the address of that function in its pointer variable. This step is important because function name can be different across applications.

Step 2:
When data arrives, get buffer from shared memory and call the registered serialization function pointer. This step is performed to get data serialized into shared memory buffer.

This can be explained with following example. Consider an Application 1 which has following structure and a serialization function which returns void * and then the application can type-cast it, into required type.

```
struct emp
{
        char name[10];
        int salary;
        char address[20];
};

void * fun1 (void *abc, char *buff, int buff_size);
{
        return (snprintf(buff, buff_size, "%d%f", (money *)abc -> i,  (money *)abc -> j));
}
```

Note: Prototype of serialization functions in both applications should be same. And same goes for de-serialization function.

4.11.2 De-Serialization by Reader Process

Generic De-Serialization Module is used to de-serialize any data buffer received from Shared Memory Transport into application data structure. It implements this by storing application de-serialization function pointer registered with transport by application at application start-up (while initializing transport).

Each application implements its data structure de-serialization function, which accepts a "pointer to buffer" and returns a newly created application data structure pointer as a "void pointer". This function is used while registering de-serializer function with the transport. The process is known as *"Registering De-serialization Function Pointer with Transport".*

Step 1:

Receive buffer from writer queue.

Step 2:

Call the registered de-serialization function pointer with shared memory buffer as argument.

Step 3:

Accept de-serialized application data structure pointer in the form of a void pointer as return value from de-serialization function pointer.

This can be explained with following example. Consider an Application 2 which has following structure and a de-serialization function which returns void * and then the application can type-cast it, into required type.

```
struct emp
{
        char name[10];
        int salary;
        char address[20];
};

void * fun2 (const char *buff)
{
        struct emp temp;
        sscanf(buff,%d%f",&temp -> i, &temp -> j);
        return temp;
}
```

The complete process of serialization and de-serialization can be explained with the help of following diagram which provides the insight of the complete proposed system in addition to various supporting modules. This diagram show in detail that how the DataBuffer and TopicName is provided by the application to Seamless Interface and then how that information is processed to form a buffer and finally how the recipient get that information converted again from buffer to actual data. This diagram clearly represents all the intermediate steps necessary in carrying out desired task.

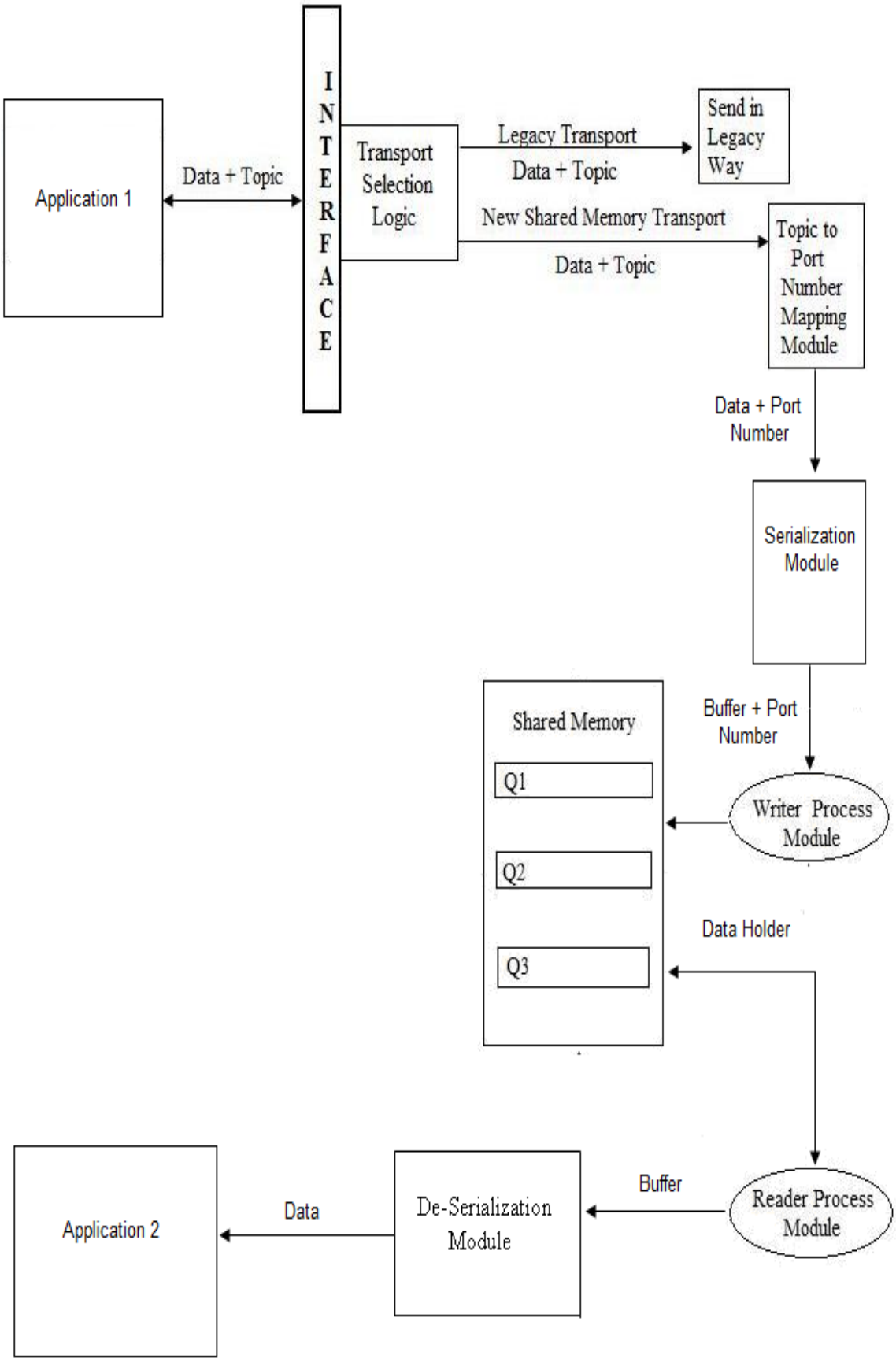

Figure 26: Diagram Elaborating Serialization and De-Serialization Modules

4.12 The Final Picture

Our conceptual model and its solution is designed keeping in mind all the existing messaging solutions and the design goals of our thesis work.

The complete solution consists of two main modules:-

- Seamless Interface
- Shared Memory Transport

Both these modules work together and with the help of other supporting modules like Writer Process Module and Reader Process Module etc. to achieve the final goal of our proposed system.

Each Writer module has its own shared memory with multiple queues and each queue in turn has its own port number associated with it. It is the responsibility of Reader's Process Manager Thread to connect to the Writer Process shared memory segment. Each Reader is associated with one unique port number within a host and has two threads for managing connection and for reading data from Writer queues when it arrives. Synchronization is the important part of shared memory concept and in order to handle it gracefully and without effecting system performance counting semaphores are the best option.

Our Seamless Interface and Shared Memory Transport would together lead to an efficient messaging solution that would be used by many commercial organizations to have low latency system. The above design and algorithms makes it clear that our proposed system would definitely result in an effective low latency system which is the demand of current industry.

CHAPTER - 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

Today all IT organizations or any other business, require communicating and exchanging data. And the focus is on saving time in processing and accessing data. All the major messaging middleware solutions today are using sockets underneath for communication between processes on local host or for communication on different hosts.

Shared memory is the fastest known inter-process communication mechanism for communication between processes on local host. Hence, in order to increase the effectiveness in terms of communication time between processes on local host we aimed at designing an algorithm. This effort resulted in an end to end design solution that would reduce the inter-process communication time on local host by implementing fast socket over shared memory. This design when implemented would not only reduce the communication time between processes on local host but would also provide Seamless Interface on top which would hide the underlying complexities from the user and would provide simple to use and efficient interface that could be used by any organization.

The notion here is to provide a Seamless Interface encapsulating Shared Memory based Communication Interface and Socket based Communication Interface into one API which is finally achieved by designing effective algorithms. The Shared Memory based Interface would be automatically used by API, if processes in question for communication are on the same host otherwise legacy Socket based Interface will be used. We also discussed and designed an efficient Shared Memory Transport Interface.

At present there is no integrated solution available to do similar job. And our solution would result in an API which is time efficient, still generic enough to be used by a large variety of commercial applications be it a web server or a high availability real-time server. We target this solution to commercial organizations as their future messaging solutions for communication.

5.2 Future Work

Businesses today look forward for an integrated solution. Today industry is dependent on already existing socket based solutions for message passing which is slower in case we require inter-process communication on local host. Our solution of providing Seamless socket like “low latency” interface over shared memory would definitely help businesses and IT organizations to have fast communication between processes on local host.

I got a chance to discuss this with people from industry, I asked them why weren’t they using Shared Memory for local communication, and their simple answer was “additional complexity involved in its implementation which reduces maintainability and affects deliverable time”. Then I asked how good it would be to have a solution that could integrate both sockets based messaging solution and shared memory based messaging solution, and provide a seamless API for data transmission. Now the answer was quite as expected, “if it’s simple enough and safe to use, we would be more than happy to use it”.

Looking at above, the future scope of this work is that this design can be implemented and integrated with already existing messaging solutions whether it’s TIBCO’s SmartSockets or TIBCO’s Enterprise Message Service or 29WEST LMB solution or may be any other messaging solution.

In nut shell we can say that it can be married with any industry standard communication library with a minimal effort and will provide great advantages.

REFERENCES

[1] Introduction to Shared Memory
http://www.csl.mtu.edu/cs4411/www/NOTES/process/shm/what-is-shm.html

[2] Douglas E. Corner and Steven B. Munson. "Efficient Interprocess Communication Using Shared Memory", February 1988.

[3] Brain N. Bershad, Thomas E. Anderson, Edward D. Lazowska and Henry M. Levy, "User-Level Interprocess Communication for Shared Memory Multiprocessors", in ACM Transactions on Computer Systems, Vol. 9. No. 2, May 1991.

[4] Inter-Process Mechanisms
http://www.linuxhq.com/guides/TLK/ipc/ipc.html

[5] Douglas C. Schmidt, "IPC SAP C++ Wrappers for Efficient, Portable, and Flexible Network Programming", 1992.

[6] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, "The Design and Implementation of the 4.4BSD Operating System", Addison Wesley, 1996.

[7] Semaphores in Linux
http://www.linuxdevcenter.com/pub/a/linux/2007/05/24/semaphores-in-linux.html?page=4

[8] Synchronization Primitives
http://www.usenix.org/events/bsdcon/full_papers/baldwin/baldwin_html/node5.html

[9] Introduction to Atomic Operations
http://en.wikipedia.org/wiki/Atomic_operation

[10] Jim Mauro and Richard McDougall. Solaris Internals: Solaris 10 and Open Solaris Kernel Architecture, Solaris Series, Second Edition: Prentice Hall, March 2006.

[11] ipc_perm Data Structure
http://tldp.org/LDP/tlk/ds/ds.html

[12] W. Richard Stevens. "UNIX Network Programming: Interprocess Communications", Volume – 2, Second Edition: Prentice Hall, 1995, pp. 321-369.

[13] (2000) The TIBCO homepage [Online].
Available: http://www.tibco.com/software/messaging/default.jsp

[14] (2000) The TIBCO homepage [Online].
Available: http://www.tibco.com/software/messaging/smartsockets/default.jsp

[15] (2000) The TIBCO homepage [Online].
Available:http://www.tibco.com/software/messaging/enterprise_messaging_service/default.jsp

[16] (2002) The 29WEST website [Online].
Available: http://www.29west.com/

[17] (2002) The 29WEST homepage. [Online].
Available: http://www.29west.com/products/lbm/

[18] W. Richard Stevens, Bill Fenner and Andrew M. Rudoff. UNIX Network Programming: The Sockets Networking API, Volume – 1, Third Edition: Prentice Hall, 2004.

[19] Shared memory Implementation in Solaris
http://sunsite.uakom.sk/sunworldonline/swol-09-1997/swol-09-insidesolaris.html

[20] Sivaram; Rajeev (West Orange, NJ), Xue; Hanhong (Poughkeepsie, NY), "Facilitating communication within shared memory environments using lock-free queues," U.S. Patent 7219198, May 15, 2007.

[21] Abraham Silberschatz, Peter Baer Galvin and Greg Gagne. "Operating System Principles", Seventh Edition: WSE, 2006.

[22] Paul J. Christensen, Daniel J. Van Hook, Harry M. Wolfson, "HLA RTI Shared Memory Communication", sponsored by the Defense Modeling and Simulation Office (DMSO) under Air Force Contract No. F19628-95-C-0002.

LIST OF PAPERS PUBLISHED

[1] Mauli Gulati, Dr. Deepak Garg, "Low Latency Seamless Transport Interface", in CiiT International Journals, in May 2009 Issue.
Available: http://www.ciitresearch.org/ncemay2009.html