

**A FAULT TOLERANT SEMAPHORE MECHANISM FOR  
MUTUAL EXCLUSION IN DISTRIBUTED SYSTEMS**

*Dissertation Submitted to*  
**JAWAHARLAL NEHRU UNIVERSITY**  
*in partial fulfilment of requirements*  
*for the award of the degree of*  
**Master of Technology**  
*in*  
**Computer Science**

*by*  
**S. Mahadev**



**SCHOOL OF COMPUTER & SYSTEMS SCIENCES**  
**JAWAHARLAL NEHRU UNIVERSITY**  
**NEW DELHI - 110 067**

*January 1996*

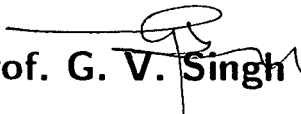
# CERTIFICATE


*This is to certify that the dissertation entitled*

**A FAULT TOLERANT SEMAPHORE MECHANISM FOR  
MUTUAL EXCLUSION IN DISTRIBUTED SYSTEMS**

*which is being submitted by Mr. S. Mahadev to the School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi for the award of Master of Technology in Computer Science, is a record of bonafide work carried out by him under the supervision and guidance of Prof. K. K. Nambiar.*

*This work is original and has not been submitted in part or full to any University or Institution for the award of any degree.*

  
Prof. G. V. Singh  
(Dean SC&SS)

  
Prof. K. K. Nambiar  
(Supervisor)

Library Copy

to my parents ...

## ACKNOWLEDGEMENT

*I take this opportunity to thank various persons who greatly influenced me throughout my stay in JNU.*

*My heartfelt thanks to my guide Prof. K.K. Nambiar for his valuable suggestions throughout this work. My profound thanks for his motivation and guidance during this period.*

*I am indebted to Prof. G.V. Singh, Dean, SC&SS, for providing necessary facilities to carry out my work. My sincere thanks are due to all faculty, SC&SS.*

*I extend my thanks to my classmates and friends.*

*S. Mahadev*

# CONTENTS

<b>1. Introduction</b>	1
1.1. Distributed Systems	1
1.1.1. Goals of Distributed Systems	2
1.2 Mutual Exclusion in Distributed Systems	3
1.2.1. A Centralized Algorithm	3
1.2.2. A Distributed Algorithm	5
1.2.3. A Token Ring Algorithm	7
1.3. Fault Tolerance in Distributed Systems	8
1.3.1. Primary Backup Algorithm	9
1.4. Need for a new Approach	11
<b>2. Detailed Problem Analysis</b>	12
2.1. Problem Statement	12
2.2. Related work & Background	12
2.3. Problem Analysis	15
2.3.1. Data Flow Diagram (DFD) of the System	16
2.3.2. External Interface Requirements	18
<b>3. Design &amp; Implementation Issues</b>	20
3.1. Design Strategies	20
3.1.1. Server Design - <i>D_SemSer</i>	20
3.1.2. Need for the <i>Key_Arbiter</i>	21
3.1.3. Fault Tolerant features	22
3.1.4. Failure Recovery	25
3.2. Design Specification	25
3.2.1. Structure Chart	25

3.2.2. Data Definitions	26
3.3. Implementation Issues	31
3.3.1. Module Algorithms	31
3.3.2. <i>D_SemSer</i> Architecture	37
3.3.3. Development Environment	38
<b>4. Mutual Exclusion using <i>D_SemSer</i></b>	<b>40</b>
4.1. Algorithm	40
4.2. Description	41
4.3. Analysis	44
<b>5. Conclusion</b>	<b>45</b>
<b>Bibliography</b>	<b>46</b>

# 1. INTRODUCTION

## 1.1 Distributed Systems

Two major advancements in technology led computer designers to radically switch from old fashioned centralized systems consisting single CPU, its memory, peripherals & some terminals to a much reliable Distributed Systems. The first advancement was the development of powerful microporcessors. Initially, these were 8-bit machines and later on manufacturers came out with 16 bit, 32 bit and now 64 bit CPUs. Many of these have the computing power of a decent sized mainframe computer, but for a fraction of the price.

The second development was the invention of high-speed computer networks. The Local Area Networks or LANs allow dozens, or even hundreds, of machines within a building to be connected in such a way that small amount of information can be transferred between machines in a millisecond or so. The Wide Area Networks or WANs allow millions of machines all over the earth to be connected at speeds varying from 64 Kbps to Gigabits per second.

The result of these technologies is that, it is now not only feasible, but easy to put together computing systems composed of large numbers of CPUs connected by a high speed network. They are called *Distributed Systems*

Since this is a dissertation involving distributed system, it may be appropriate to begin with an attempt at a definition; as Andrew Tanenbaum [9] says, various definitions of distributed systems have been given in the literature, none of them satisfactory and none of them in agreement with any of the others. For our purpose it is sufficient to give a loose characterization :

*A distributed system is a collection of independent computers that appear to the users of the system as a single computer.*

This definition has two aspects. The first one deals with hardware : the machines are autonomous. The second one deals with software : the users think of the system as a single computer.

### 1.1.1 Goals of Distributed Systems

Not just because it is possible to build distributed systems. due to technological advancements, it came into existence. There are many motivations behind its development.

The real driving force behind the trend toward decentralization is economics. As Grosch's Law states :

*The Computing power of a CPU is proportional to the square of its price.*

With microprocessor technology, Grosch's Law no longer holds. For a few hundred dollars you get a CPU chip that can execute more instructions per second than one of the largest 1980s mainframe. If you are willing to pay twice as much, you get the same CPU, but running at a somewhat higher clock speed. As a result, the most cost effective solution is frequently to harness a large number of cheap CPUs together in a system. Thus the leading reason for the trend toward distributed systems is that these systems potentially have a much better price/performance ratio than a single large centralized system would have.

A next reason for building a distributed system is that some applications are inherently distributed. You can think about a large bank with hundreds of branch offices all over the world. Each office has a master computer to store local accounts and handle local transactions. In addition, each computer has the ability to talk to all other branch computers and thus transactions can be done without regard to where a customer or account is.



Another potential advantage of a distributed system over centralized system is higher reliability. By distributing the workload over many machines, a single chip failure will bring down at most one machine.

Finally, incremental growth is also potentially a big plus. Often, a company will buy a mainframe with the intentions of doing all its work on it. If the company prospers and the workload grows, at a certain point the mainframe will no longer be adequate. The only solutions are either to replace the mainframe with a larger one or to add a second mainframe. Both of these can wreak major havoc on the company's operations. In contrast, with a distributed system, it may be possible to simply to add more processors to the system, thus allowing it to expand gradually as the need arises.

## **1.2 Mutual exclusion in Distributed Systems**

Systems involving multiple processes are often most easily programmed using critical regions. When a process has to read or update certain shared data structures, it first enters a critical region to achieve mutual exclusion and ensure that no other process will use the shared data structures at the same time. In single-processor systems, critical regions are protected using semaphores, monitors, and similar constructs. In the following section we will look at few algorithms which allow us to achieve mutual exclusion in a distributed system.

### **1.2.1 A Centralized Algorithm**

The most straightforward way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system. One process is elected as the coordinator (e.g., the one running on the machine with the highest network address). Whenever a process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission. If no other process is currently in that critical region, the coordinator sends back a

reply granting permission. When the reply arrives, the requesting process enters the critical region.

Now suppose that another process 2, asks for permission to enter the same critical region. The coordinator knows that a different process is already in the critical region, so it cannot grant permission. The exact method used to deny permission is system dependent. For example, the coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply. Alternatively, it could send a reply saying "permission denied". Either way, it queues the request from 2 for the time being.

When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access. The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked (i.e., this is the first message to it), it unblocks and enters the critical region. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic, or block later. Either way, when it sees the grant, it can enter the critical region.

It is easy to see that the algorithm guarantees mutual exclusion : the coordinator lets only one process at a time into each critical region. It is also fair, since requests are granted in the order in which they are received. No process ever waits forever (starvation). The scheme is easy to implement, and requires only three messages per use of a critical region (request, grant, release). It can also be used for more general resource allocation rather than just managing critical regions.

The centralized approach also has shortcomings. The coordinator is a single point of failure, so if it crashes, the entire system may go down. If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since in both cases no message comes back. In addition, in a large system, single coordinator can become a performance bottleneck.

### 1.2.2 A Distributed Algorithm

Having a single point of failure is frequently unacceptable, so researchers have looked for distributed mutual exclusion algorithms. In this section we discuss Ricart and Agrawala's algorithm.

This algorithm requires that there be a total ordering of all events in the system. That is, for any pair of events, such as messages, it must be unambiguous which one happened first. The algorithm works as follows. When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number, and the current time. It then sends the message to all other process, conceptually including itself. The sending of messages is assumed to be reliable; that is, every message is acknowledged. Reliable group communication if available, can be used instead of individual messages.

When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message. Three cases have to be distinguished :

- o If the receiver is not in the critical region and does not want to enter it, it sends back an OK message to the sender.
- o If the receiver is already in the critical region, it does not reply. Instead, it queues the request.
- o If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message is lower, the receiver sends back an OK message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

After sending out request asking permission to enter a critical region, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may enter the critical region. When it exits the critical region, it sends OK messages to all processes on its queue and deletes them all from the queue.

Let us try to understand why the algorithm works. If there is no conflict, it clearly works. However, suppose that two processes try to enter the same critical region simultaneously. Process 0 sends everyone a request with timestamp 8, while at the sametime, process 2 sends everyone a request with timestamp 12. Process 1 is not interested in entering the critical region, so it sends OK to both senders. Processes 0 and 2 both see the conflict and compare timestamps. Process 2 sees that it has lost, so it grants permission to 0 by sending OK. Process 0 now queues the request from 2 for later processing and enters the critical region. When finished, it removes the request from 2 from its queue and sends an OK message to process 2 allowing the later to enter its critical region. The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps.

As with the centralized algorithm discussed above, mutual exclusion is guaranteed without deadlock or starvation. The number of messages required per entry is now  $2(N-1)$ , where the total number of processes in the system is  $N$ . Best of all, no single point of failure exists.

Unfortunately, the single point of failure has been replaced by  $n$  points of failure. If any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions. Since the probability of one of the  $N$  times as large as a single coordinator failing, we have managed to replace a poor algorithm with one that is  $N$  times worse and requires much more network traffic to boot.

The problem with this algorithm is that either a group communication primitive must be used, or each process must maintain the group membership list itself, including

processes entering the group, leaving the group, and crashing. The method works best with small groups of processes never change their group memberships.

Finally, recall that one of the problems with the centralized algorithm is that making it handle all requests can lead to a bottleneck. In the distributed algorithm, 'all' processes are involved in 'all' decisions concerning entry into critical regions. If one process is unable to handle the load, it is unlikely that forcing everyone to do exactly the same thing in parallel is going to help much.

### 1.2.3 A Token Ring Algorithm

A completely different approach to achieving mutual exclusion in a distributed system is discussed in this section. Here we have a bus network, with no inherent ordering of the processes. In software, a logical ring is constructed in which each process is assigned a position in a ring. The ring positions may be allocated in numerical order of network addresses or some other means. It does not matter what the ordering is. All that matters is that each process knows who is next in line after itself.

When the ring is initialized, process 0 is given a TOKEN. The token circulates around the ring. It is passed from process  $K$  to process  $K+1$  (modulo the ring size) in point-to-point messages. When a process acquires the token from its neighbor, it checks to see if it is attempting to enter a critical region. If so, the process enters the region, does all the work it needs to, and leaves the region. After it has exited, it passes the token along the ring. It is not permitted to enter a second critical region using the same token.

If a process is handed the token by its neighbor and is not interested in entering a critical region, it just passes it along. As a consequence, when no processes want to enter any critical regions, the token just circulates at high speed around the ring.

The correctness of this algorithm is evident. Only one process has the token at any instant, so only one process can be in a critical region. Since the token circulates

among the processes in a well-defined order, starvation cannot occur. Once a process decides it wants to enter a critical region, at worst it will have to wait for every other process to enter and leave one critical region.

The problem with this algorithm is, if the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is unbounded. The fact that the token has not been spotted for an hour does not mean that it has been lost; somebody may still be using it. The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbor tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can throw the token over the head of the dead process to the next member down the line.

### 1.3 Fault Tolerance in Distributed Systems

A system is said to fail when it does not meet its specification. In some cases such as a distributed air traffic control system, a failure may be catastrophic. As computers and distributed systems become widely used in safety-critical missions, the need to prevent failures becomes correspondingly greater. A hardware and/or software fault may reduce the availability of a system. A fault tolerant system continues to operate even after some hardware/software components have failed and are not available to the user. The key advantage of distributed systems is that they can be more fault tolerant than the centralized systems.

In a critical distributed system, often we are interested in making the system be able to survive component (in particular, processor) faults, rather than just making these unlikely. System reliability is especially important in a distributed system due to the large number of components present, hence the greater chance of one of them being faulty.

## Use of Redundancy

The general approach to fault tolerance is to use redundancy. Three kinds are possible : information redundancy, time redundancy, and physical redundancy. With information redundancy, extra bits are added to allow recovery from garbled bits. For example, a Hamming code can be added to transmitted data to recover from noise on the transmission line.

With time redundancy, an action is performed, and then, if need be, it is performed again. If a transaction aborts, it can be redone with no harm. Time redundancy is especially helpful when the faults are transient or intermittent.

With physical redundancy, extra equipment is added to make it possible for the system as a whole to tolerate the loss or malfunctioning of some components. For example, extra processors can be added to the system so that if a few of them crash, the system can still function correctly. This type of redundancy is helpful when the faults are permanent.

There are two ways to realize : active replication and primary backup. Consider the case of a server. When active replication is used, all the processors are used all the time as servers (in parallel) in order to hide faults completely. In contrast, the primary backup scheme just uses one processor as a server, replacing it with a backup if it fails.

Here, we give emphasis to Primary Backup scheme, as this method has been incorporated in the developed system for achieving fault tolerance.

### 1.3.1 Primary Backup Algorithm

The essential idea of the primary-backup method is that at any one instant, one server is the primary and does all the work. If the primary fails, the backup takes over. Ideally, the cutover should take place in a clean way and be noticed only by the

client operating system, not by the application programs. Few examples in real world scenario would be : Government (the Vice-President), aviation (co-pilot) etc.,

Primary-backup fault tolerance has two major advantages over active replication. First, it is simpler during normal operation since messages go to just one server (the primary) and to a whole group. The problems associated with ordering these messages also disappear. Second, in practice it requires fewer machines, because at any instant one primary and one backup is needed (although when a backup is put into service as a primary, a new backup is needed instantly). On the downside, recovery from a primary failure can be complex and time consuming.

As an example of the primary-backup solution, consider the simple protocol, in which, the client sends a message (1. request) to the primary, which does the work (2. do work) and then sends an update (3. update) message to the backup. When the backup gets the message, it does the work (4. do work) and then sends an acknowledgement (5. Ack) back to the primary. When the acknowledgement arrives, the primary sends the reply (6. reply) to the client.

Now, let us consider the effect of a primary crash at various moments during a client request. If the primary crashes before doing the work(2), no harm is done. the client will time out and retry. If it tries often enough, it will eventually get the backup and the work will be done exactly once. If the primary crashes after doing the work but before sending the update(3), when the backup takes over and the request comes in again, the work will be done a second time. If the work has side effects, this could be a problem. If the primary crashes after step 4 but before step 6, the work may end up being done three times, once by the primary, once by the backup as result of step 3, and once after the backup becomes the primary. If requests carry identifiers, it may be possible to ensure that the work is done only twice, by getting it done exactly once is difficult to impossible.



One theoretical and practical problem with the primary-backup approach is when to cat over from the primary to the backup. In the protocol above, the backup could send 'Are you alive?' messages periodically to the primary. If the primary fails to respond within a certain time, the backup would take over.

#### **1.4 Need for a new approach**

It is very evident from the previous sections that the problem of providing mutually exclusive access to a critical region (section) in a distributed system is not trivial. Various algorithms were published to provide solution for Mutual Exclusion in a distributed system, each of which fall into any one of the class discussed earlier viz., Centralized, Distributed and Token Ring. But each one has got its own limitations. In this dissertation, we discuss about implementation of Distributed Semaphore mechanism for providing mutual exclusion in distributed systems, which incorporates a variation of both Centralized and Distributed mutual exclusion algorithms, with a provision of Fault Tolerance for any node failure.

## 2. DETAILED PROBLEM ANALYSIS

In this chapter, we define the complete problem statement and review some corresponding work about the solutions of Mutual Exclusion in Distributed systems. Complete problem analysis is given depicting Data Flow Diagram (DFD) for the system along with the Functional specification.

### 2.1 Problem Statement

Providing solution for Mutual Exclusion in a distributed system is a much involved task. Conventionally, in a single CPU system, Semaphores are the classic mechanism of solution for this problem. The intent of this dissertation is to develop a Distributed Semaphore mechanism that meets the following criteria. First, it must always maintain mutual exclusion regardless of any node failure. In addition, the whole system should be distributed as evenly as possible across all the nodes in the network.

### 2.2 Related work & Background

There has been a lot of research activity in the area of mutual exclusion algorithms for distributed systems. A survey of some existing mutual exclusion algorithms appears in *Table 2.1*. The schemes proposed fall into two categories, Consensus and Token-Ring algorithms.

In distributed systems the implementation of mechanisms for synchronization has traditionally taken one of the two approaches : Application-level based and Operating System Kernel-level based. One of the early attempts at realizing a distributed semaphore mechanism was in LOCUS [5]. It is an OS Kernel-level implementation and concerns itself with augmenting the LOCUS distributed operating system.

However, the usefulness of such a system is limited, because they were tailor-made for specific environments. The reality is that most of the popular workstations are

Table 2.1. Survey of Mutual Exclusion Algorithms

Algorithm	Number of Messages	Method of Resolution
Lamport	$3(N-1)$	Consensus
Ricart-Agrawala	$2(N-1)$	Consensus
Suzuki-Kasami	$N$	Token Ring
Carvalho-Roucairol	$0 - 2(N-1)$	Consensus
Maekawa	$O(\sqrt{N})$	Consensus
Mishra-Srimani	$N$	Token Ring
Naimi-Trehel	$O(\log N)$	Token Ring
Helary et al.	$0 - N-1+d$	Token Ring
Raymond	$O(\log N)$	Token Ring
Singhal	$\frac{(N+1)}{2}$	Token Ring

being marketed with one of the predominant UNIX systems (BSD, System V, OSF), and UNIX will continue to be the preferred operating system by most users. Since new facilities for distributed systems are also being developed, requiring changes to the kernel in order to support a specific facility, becomes a costly and self defeating approach. The alternative is to develop these facilities entirely at the application level. The work published by Shyan-Ming Yuan, et al.[11], suggests an architecture at the application level.

Consequently, the design presented here requires no changes to the UNIX kernel. It is entirely implemented at the application level. So it gains insight into the portability for all workstations running a version or a derivative of UNIX operating system which supports BSD Sockets and System V IPC.

## Background

Semaphore is a synchronization primitive that prevents two or more processes from accessing a shared resource simultaneously; our distributed semaphore is also the same. The processes can synchronize their operations on the different nodes by means of the distributed semaphore.

Dijkstra published the Dekkers algorithm that describes implementation of semaphore, integer valued object that has the following operations defined for it :

- o Creation and initialization of a semaphore to a non-negative value.
- o A P operation that decreases the value of the semaphore. If the value of the semaphore is less than 0 after decreasing its value, the process that did the P goes to sleep.
- o A V operation that increases the value of the semaphore. If the value of the semaphore becomes greater than or equal to 0 as a result, one process that had been sleeping as the result of a P operation wakes up.

- o Close and remove an existing semaphore.

## 2.8 Problem Analysis

In this section, we analyse the problem statement defined above completely to gain an insight over the data flow in the system and also specify the functional requirements of the system.

In this system, each of the resources which are to be shared among different processors (processes) are given a unique KEY. Whenever a user wants to access the shared resource, creates a Distributed Semaphore with the corresponding KEY to the resource, and gets an ID inreturn. An important property of any distributed system is the ability to access any resource in the system transparently, i.e., the user need not know where exactly the resource is existing. Thus the system we are going to present should also provide a transparent access to the Semaphores. These user requests are serviced by a SERVER running in each node (Workstation). If a user tries to create a distributed semaphore with a KEY which has already been created and being used, then the system defers from creation, and requests the SERVER which created the semaphore, and gets the ID. Once the user wants to enter the critical region, it DOWNs (P operation) the semaphore using the ID. This request goes to the SERVER where the distributed semaphore was created. The DOWN operation is carried out as discussed in the Semaphore operations in the previous section and thus if any other user is using the resource, which would have DOWNed the distributed semaphore, then this user is blocked until the other releases it by an UP (V operation) operation on the same distributed semaphore.

Another important feature to be discussed about the system is that, the fault tolerance it provides. We discussed various methods of achieving fault tolerance in a distributed system in the introduction. Here we try to provide an overall view of how we could achieve fault tolerance in case of any node failure in the system.

Every distributed semaphore operation (create, P, V, close) needs to be replicated in any node other than the node in which it is created & serviced. Thus before servicing (performing) any user request, the request should propagate to its replication node. The actual service is carried out by the SERVER only after receiving the acknowledgement from the replica node. Once any node failure is identified, the replica node takes care of recovery.

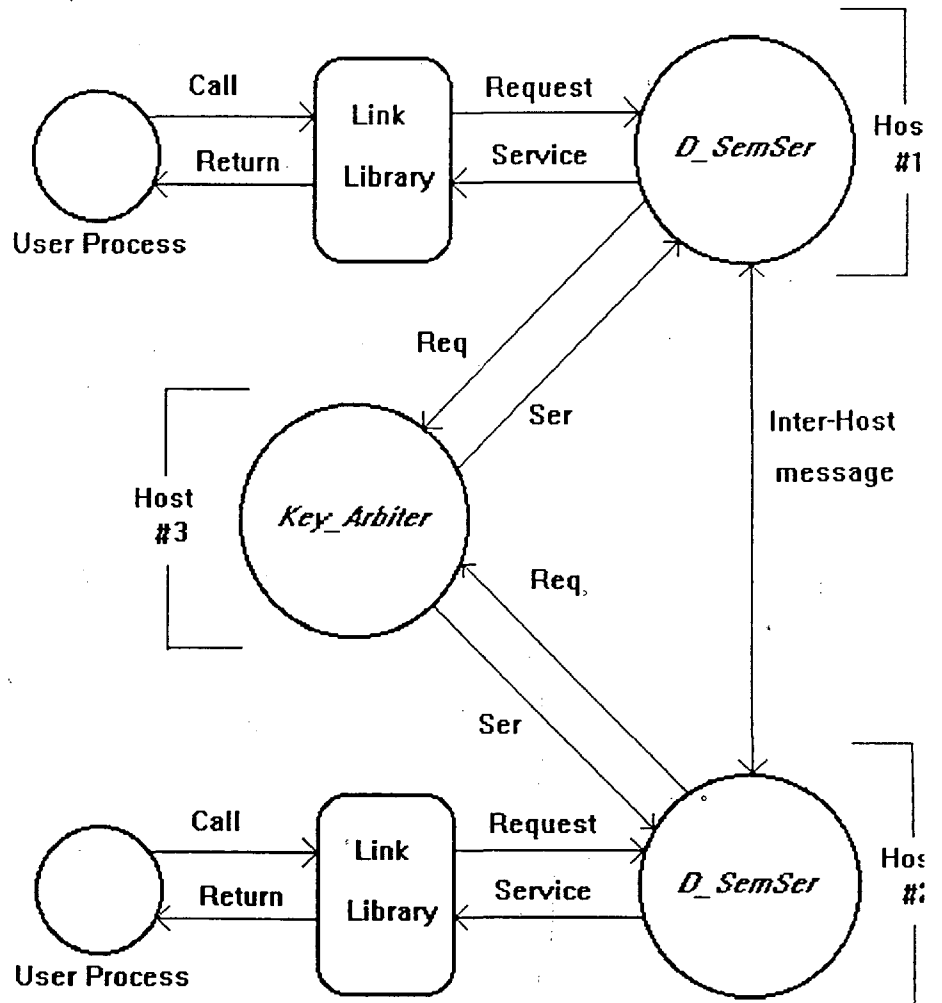
### 2.3.1 Data Flow Diagram (DFD) of the System

With the information we obtained from the Problem Analysis section, we try to define the Data Flow Diagram of the system and also the decisions made about the system. Before that, we understand what a DFD is :

A DFD shows the flow of data through a system. Data Flow Diagram also shows the movement of data through the different transformations or processes in the system. The processes are shown by named circles and data flows are represented by named arrows entering or leaving the circles.

The Data Flow Diagram of Distributed Semaphore mechanism is depicted in *Figure 2.1*. The decisions made regarding the system are :

- o Every node involving Distributed Semaphore mechanism runs, the only one and the same copy of *D\_SemSer* (Distributed Semaphore Server).
- o Every process requests are directed to its local *D\_SemSer*, where decision is taken, which sometimes require consulting other nodes (*D\_SemSer* running in those nodes), and serviced accordingly.
- o Specific KEYS are assigned to shared resources in the distributed system, so that any process requires to use a shared resource can create a distributed semaphore with its (resource's) corresponding KEY.



**Figure 2.1 Data Flow Diagram (DFD) of the System**

- *Key\_Arbiter*, played by any one of the *D\_SemSer* running in any node, makes sure that no other process creates a distributed semaphore with the same KEY.
- Every distributed semaphore operation must be replicated before being acknowledged to the user.
- *D\_SemSers* broadcast ALIVE message periodically to every other *D\_SemSers* such that node failures can be identified.
- Link Library , *L\_Library* , takes care of forwarding all user programs' distributed semaphore calls to the local *D\_SemSer* . Thus the system allows any user program to use the distributed semaphore facility as a normal function call, providing transparent service.

### 2.3.2 External Interface Requirements

The system is used by System programmers and Application developers. Thus the usage of *D\_SemSer* system is made entirely transparent. It enables the users to use much of these library functions as simply across machine boundaries as within a single machine. The system provides access facilities that are invoked in the user program by calling a set of C functions :

A new distributed semaphore with a specified initial value is created by using

```
int semid = dsem_create(key_t key, int init_val);
```

*dsem\_create()* returns a positive value as the semaphore identifier (semid) or -1 when an error occurs.

If the distributed semaphore already exists, we do not initialize it and return -1. If the caller knows that the distributed semaphore already exists, then *dsem\_open()* function should be used.



```
int semid = dsem_open(key_t key);
```

*dsem\_open()* returns a positive integer as the semaphore identifier (semid) or -1 when error occurs. If the distributed semaphore does not exist, we return -1.

Once a distributed semaphore is created (or opened), operations are performed on semaphore value using *dsem\_down()* and *dsem\_up()* functions.

```
int dsem_down(int semid, int amount);
```

```
int dsem_up(int semid, int amount);
```

The *dsem\_down()* function decreases the distributed semaphore value by a user-specified amount. The *dsem\_up()* function increases the distributed semaphore value by a user-specified amount.

The *dsem\_close()* function is for a process to call before it exits, when it is done with the distributed semaphore. If this is the last user of the distributed semaphore, then we remove the distributed semaphore.

```
int dsem_close(int semid);
```

It returns a value to indicate the operation is successful or failed.

### 3. DESIGN & IMPLEMENTATION ISSUES

The design of a system is essentially a blueprint, or a plan for a solution for the system. Here we consider a system to be a set of components with clearly defined behaviour, which interact with each other in a fixed, defined manner, to produce some behaviour or services to its environment. Implementation Issues essentially deals with the translation of design of the system produced during the design phase into code in a given programming language which can be executed by a computer, and which performs the computation specified by the design. In this Chapter, we talk about the strategies followed for designing our model of distributed semaphore, followed with design specifications. In the implementation issues, we provide the *D\_SemSer* architecture, development platform, its implementation details and the algorithms.

#### 3.1 Design Strategies

##### 3.1.1 Server Design - *D\_SemSer*

Semaphores, which are the best suited for mutual exclusion and other synchronization problem, are not incorporated in Distributed operating systems itself, because they invariably rely (imply) on the existence of shared memory. For example, two processes that are to synchronize their activity on sharing a resource using a semaphore, must both be able to access the semaphore. If they are running on the same machine, they can share the semaphore by having it stored in the kernel, and execute system calls to access it, as it is in the case of System V IPC. If, however, they are running on different machines, this method no longer works.

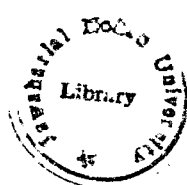
Our approach to realize distributed semaphore mechanism is, in an abstract shared memory on a network. An abstract shared memory simulates a shared physical memory.

In our model, *D\_SemSer* plays two kinds of roles at the sametime : one is the *arbiter* for some distributed semaphores, i.e., the one which creates distributed semaphores and coordinates its operations performed on it, on user requests. Another is the *agent* that is responsible to user for providing the distributed semaphore service. i.e., the one which makes the distributed semaphores created elsewhere in the network to be available to the user.

As from our problem analysis decisions, every node runs the only one and same copy of code for *D\_SemSer*, and thus each of them act as *agent* to the processes (users) in their respective nodes where they are running. The problem here is, how to select a *D\_SemSer* as the *arbiter* of a distributed semaphore ?. We choose the simplest and effective way : we let the *D\_SemSer* of the node which first uses (creates) the distributed semaphore be the *arbiter* of that distributed semaphore. Therefore, performance is optimised because operations on the semaphore will be local operations for the arbiter.

### 3.1.2 Need for the *Key\_Arbiter*

Since we are talking about arbiter, we consider the other arbiter involved in our system, that is *Key\_Arbiter*. The requirement for this could be best explained with an example. Consider an user wants to acquire a shared resource, first issues a create distributed semaphore call (*dsem\_create*) with the KEY corresponding to the resource. The *D\_SemSer* running in the node creates a semaphore and returns an ID to the user, which allows user to do other operations like DOWN (*dsem\_down*), before actually using the resource. Supposing, another user from some other node, with the same requirement, issues a create distributed semaphore call (*dsem\_create*) with the same KEY. The *D\_SemSer* running in that node creates a semaphore and returns an ID. This totally defeats the idea of our model, to provide mutual exclusion, since both users can , after getting the ID, do a DOWN operation (*dsem\_down*), which would be



serviced locally for them, and they try to acquire the shared resource when any one of them is already using it.

To avoid such scenarios, every distributed semaphore create call (`dsem_create`) will be directed to the *Key\_Arbiter* running in any one of the nodes in the network. This maintains a table which contain all the distributed semaphores created over the network at any instant along with the creators Node address. Thus if any user tries to create a distributed semaphore with a KEY which has been already created elsewhere in the network, *Key\_Arbiter* returns an error message. Thus the user can use distributed semaphore open call (`dsem_open`) with the same KEY, to synchronize its activity with the user who created the distributed semaphore elsewhere, as it returns the ID of that distributed semaphore. Subsequent operations like 'dsem\_down' will be forwarded to the *D\_SemSer*, whose address is maintained in the *Key\_Arbiter*, which is the arbiter of the distributed semaphore.

There is a problem of how to select a *D\_SemSer* as the *Key\_Arbiter* ?. The simplest solution would be the first node (*D\_SemSer*) which participates in the network, will *fork* a child, which acts as *Key\_Arbiter*. Since every *D\_SemSer* sends a broadcast message when the node becomes alive, it is easy for a node to identify itself whether it is the first node to become alive.

### 3.1.3 Fault Tolerant Features

The necessity for fault tolerance in any distributed system was discussed in great detail in chapter 1. Evidently, node failure of the arbiter of some distributed semaphores will lead to loss of all informations of some distributed semaphore, which may lead to undesirable results in the system.

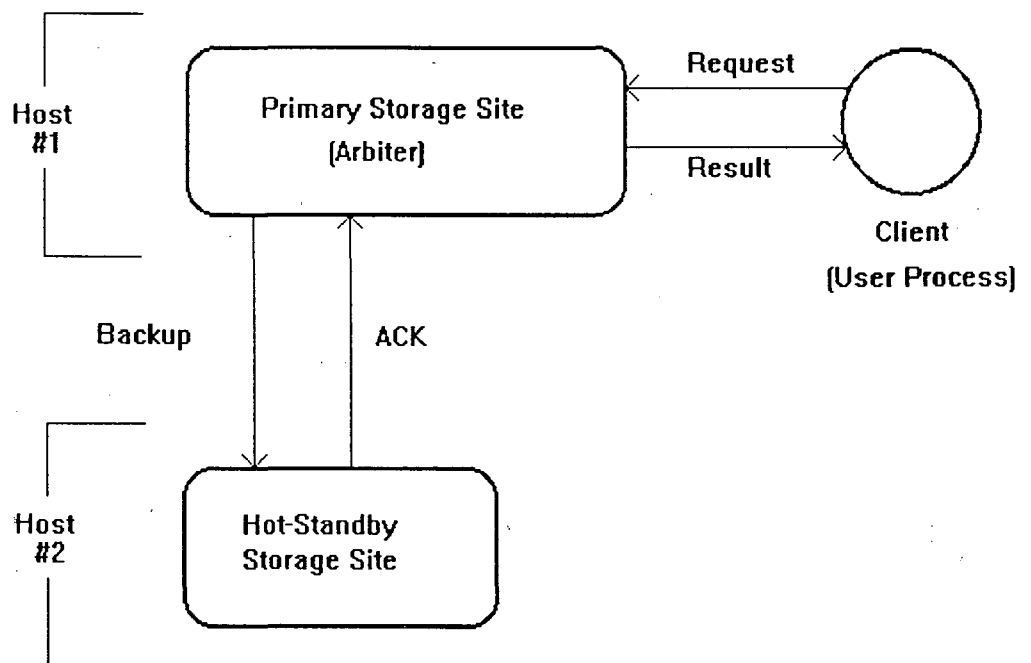
Typically, in our model, the fault tolerance is enforced through replication. Here, each distributed semaphore has its own hot-stand by replica at the other site. Usually, the hot-standby copy of a distributed semaphore is located at the site which is the

second site using the distributed semaphore in the distributed system. Therefore, it is fault tolerant provided that there exists at least one correct copy in the system except that the both sites crash simultaneously.

First, we assume that failures are absent. In normal circumstances, a service request, from the user is sent to the *D\_SemSer* running in that node, which is the primary replica of the distributed semaphore. Then after the request is handled by *D\_SemSer*, the request will propagate to the hot-standby replica of the distributed semaphore. The primary replica will wait for the acknowledgement of its hot-standby replica after its *D\_SemSer* has sent out the request. Upon receiving that signal from its hot-standby replica, the primary replica sends the result to the requesting user. But from the viewpoint of client, the request is made only once, and only one copy of the result is received by the client. The request and the acknowledgement sent by the client and *D\_SemSer* are shown in *Figure 3.1*.

Secondly, the failure may occur in the primary replica. This kind of failure can occur at two times: before or after the the *D\_SemSer* of primary replica has propagated the request. If the failure occurs in the primary replica before the request is propagated to the standby, the client will detect that the primary replica had failed, and same request will be sent to the hot-standby replica. Then everything continues just as if no error has occurred, except that the client resends the request and result is sent directly from the standby to the client.

Now, let us consider the case of failure occurring in the primary replica after the request is propagated to its standby. If the primary replica fails after the request has propagated to the standby replica, the standby replica will receive the same request twice. Since the request carries with it a sequence number, the standby will process the request only once, and the result is sent directly from the standby to the client, rather than from the primary replica.



**Figure 3.1 *D\_SemSer* Replication Operation**

### 3.1.4 Failure Recovery

Recovery from the failure is important for a fault tolerant system. Our recovery mechanism is rather simple when compared with other complex schemes. Where the responsibilities of the failed primary will be taken over by its hot-standby, and then we must recreate a new standby replica to ensure that there are always available hot-standby replica. The using site is in the client machine that issues the request and receives the result; the storage site is the place where a distributed semaphore is stored actually.

Dangerous situation could occur if a process does a semaphore operation, presumably locking some resource, and then exits without resetting the semaphore value. Such situations can occur as the result of receipt of a signal that causes sudden termination of a process or site crash. Hence other processes could find the semaphore locked even though the process that had locked it no longer exists. To avoid such problems, it is necessary to back out the updates so that changes appear invisible. The mechanism which is used to implement this is an *undo log*. Each storage site records an undo log that indicates how to undo the semaphore if an application updates the semaphore and is aborted by a signal or site crash. The undo log entries are removed when the semaphore is removed.

## 3.2 Design Specification

### 3.2.1 Structure Chart

Structured design methodology views every software system as having some inputs which are converted into the desired outputs by the software system. The software is viewed as a transformation function that transforms the given inputs into the desired outputs, and the central problem of designing software systems is considered to be properly designing this transformation function. The goal is to produce a design for the software system that consists of many modules such that the modules have the least

interconnections between them. With such a design, each module can be implemented and changed with minimal consideration of the other modules of the system.

In Structured design methodology, the design is represented by structure charts. The structure of a program is made up of the modules of that program together with the interconnections between modules. The structure chart of a program is a graphic representation of its structure. In the structure chart, we represent a module by a box. Here, the focus is on representing the hierarchy of modules. The structure chart representing the high-level design is shown in *Figure 3.2*.

### 3.2.2 Data definitions

For our model to execute successfully, each node in the network (*D\_SemSer*) must maintain certain information locally. In this section, we define the datastructures which are used represent these informations in detail.

#### 1. MESSAGE

{		
	msg_type	→ Specifies whether internal or external
	req_type	→ Specifies the type of service required
	proc_id	→ Process ID of the user requesting for service
	key	→ KEY of a distributed semaphore
	sem_id	→ Corresponding ID of distributed semaphore
	value	→ Distributed semaphore value
	Address	→ Node address of the user requesting for service

*Description* : The basic entity by which the dataflow within the system and among the nodes are achieved through message passing. The message structure is depicted above. Whenever a user makes a distributed semaphore call, a message structure as defined above is constructed in the library function and passed to the *D\_SemSer*. Again, for any inter-server communication, i.e., between *D\_SemSers*, the message format is as above. Other than *msg\_type*, all the other fields are meaningfully interpreted according to the type of service specified in the *req\_type*. Different Service request references are given in *Table 3.1*.



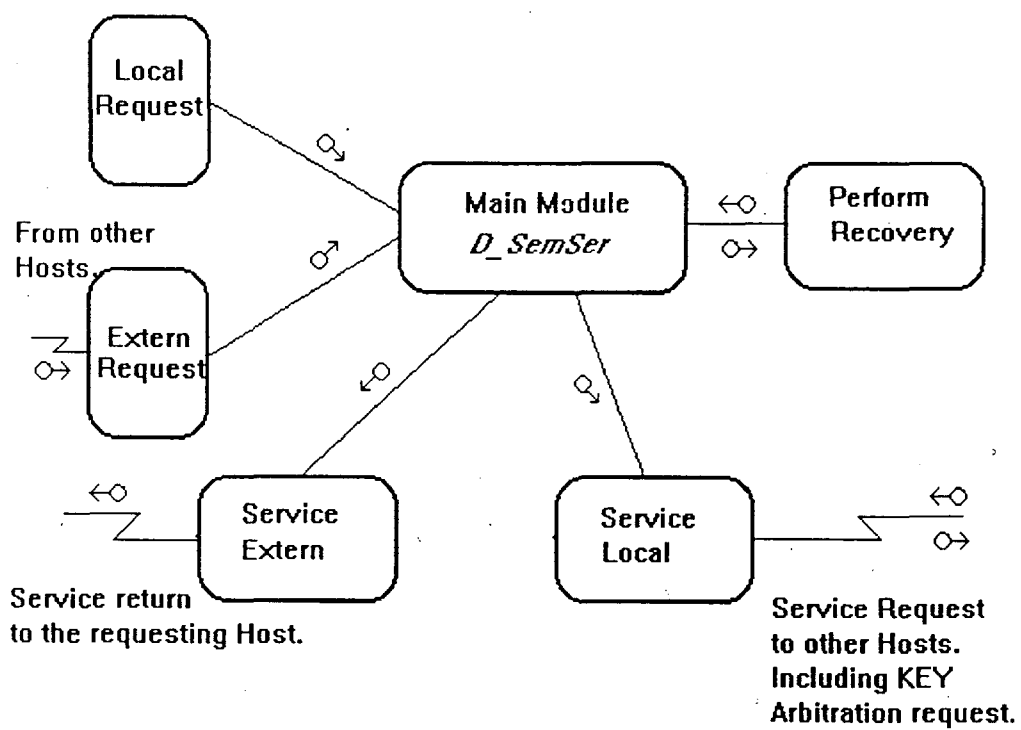


Figure 3.2 Structure Chart for *D\_SemSer*

Table 3.1. Request Message Reference

Request Message	Purpose
CREATE	Create a distributed semaphore
OPEN	Open an existing distributed semaphore
DOWN	Down (P) Operation on specified <i>d_sem</i>
UP	Down (V) Operation on specified <i>d_sem</i>
CLOSE	Delete requested user entry for the specified <i>d_sem</i>
REPLY	Message carrying service to other HOSTS
NACK	Erronious request or request could not be serviced
ALIVE	Broadcast message for other HOSTS
ADD_USR	New user participation to a distributed semaphore
KEY_ARB	For KEY arbitration
KEY_ACK	Creation with given KEY agreed
KEY_NACK	KEY already exists
KARB_ADD	<i>Key_Arbiter</i> address broadcast message
EXT_MSG	Request message from other HOSTS

## 2. SEM\_TABLE

```
{
    key           → KEY of a distributed semaphore
    sem_id        → Corresponding ID of distributed semaphore
    value         → Distributed semaphore value
    SEM_USERS     → List containing users of this semaphore
    replica_add   → Node address of the standby node
}
```

*Description* : Every *D\_SemSers* maintain a list of above structure. Whenever a distributed semaphore is created in that node, a new structure entry with appropriate field contents is appended to the list. The field *replica\_add* stores the address of the node which acts as the hot-standby replica for that distributed semaphore. *SEM\_USERS* field contains the list of users who are using that distributed semaphore. When this list becomes NULL, for any entry, that entry is removed from the list.

## 3. EXT\_SEM\_TAB

```
{
    key           → KEY of a distributed semaphore
    sem_id        → Corresponding ID of distributed semaphore
    value         → Distributed semaphore value
    owner_add     → Node address of the distributed semaphore creator
    SEM_USERS     → List containing users of this semaphore within
                  the node
    is_replica    → Flag to check is it a standby replica for
                  the Distributed semaphore
}
```

*Description* : A list of above structure is maintained in every *D\_SemSers*, which stores the information about the distributed semaphores created elsewhere in the network, and used by the processes within the node. When *SEM\_USERS* field becomes NULL, for any entry, that entry is removed from the list. If the field *is\_replica* is TRUE, for any entry in this list in a *D\_SemSer*, it would take part in the recovery, if the creator crashes.

## 4. SEM\_USERS

```
{
    proc_id       → Process ID of the user process
    address       → Node address of the user process
}
```

*Description* : A list of above structure is maintained in every entry of the SEM\_TABLE and EXT\_SEM\_TAB. A new entry to this list is added whenever a new user process issues a *dsem\_open*, pertaining to the corresponding distributed semaphore entry in SEM\_TABLE or EXT\_SEM\_TAB.

5. HOSTS\_TABLE

```
{  
    address      → Node address of a workstation  
    status       → Specifies whether it is ALIVE or DEAD  
}
```

*Description* : A static list of above structure is constructed at the time when every *D\_SemSers* are started from the file HOSTS. Each *D\_SemSer* polls every entry in the list (i.e., every other *D\_SemSers*). The field *status* is marked ALIVE if a positive acknowledgement is received.

6. ARB\_TABLE

```
{  
    key          → KEY of a distributed semaphore  
    owner_add    → Node address of the distributed semaphore creator  
}
```

*Description* : A list of above structure is maintained in the *Key\_Arbiter*. An entry is added to the list for every distributed semaphore created anywhere over the network. It is deleted whenever that distributed semaphore contains no active users.

7. HOSTS → File containing Host addresses

*Description* : This file contains all Host addresses in the network, which is used by the *D\_SemSer* to construct the HOST\_TABLE.

8. UNDO\_LOG → File containing Log entries of semaphore operations performed by various users.

*Description* : This file maintained by every *D\_SemSer*, enters all the operations performed on a distributed semaphore. This is helpful in recovering information due to node crashes. The entries corresponding to a distributed semaphore are removed when the distributed semaphore is removed.

### 3.3 Implementation Issues

In this section, we discuss the algorithms used for implementing our model. Here, we follow the TOP DOWN approach of starting from the main module to the inner most fragments. Every algorithm is presented in the pseudo code fashion for better understanding. We have also discussed about the entire architecture of the system along with the techniques used for providing the programming solution. The development environment of the system is also mentioned.

#### 3.3.1 Module Algorithms

Following module algorithms constitute the *D\_SemSer* system.

(a) *Module Name*  $\Rightarrow$  *D\_SemSer*

*Purpose*  $\Rightarrow$  *Receives messages from user process and from other nodes and does the required service by 'forking' a child. At the initialization, identifies itself whether it is the Key\_Arbiter. Invokes PERFORM\_RECOVERY incase of any node failure.*

BEGIN

Initialize datastructures SEM\_TABLE and EXT\_SEM\_TAB.

Initialize Communication channels.

Construct HOSTS\_TABLE using HOSTS file.

Broadcast ALIVE message to all other nodes.

if (first node to participate in the network)

begin

    Create a child.

    make it *Key\_Arbiter*

$\rightarrow$  Only Child executes

end

$\rightarrow$  Parent continues here

LOOP FOREVER

    if ( message received from user )

    begin

        Create a child.

        SERVICE\_LOCAL

$\rightarrow$  Only Child executes

    end

$\rightarrow$  Parent continues here

    if ( message received from othernode )

    begin

        Create a child.

```

SERVICE_EXTERN          → Only Child executes
end                       → Parent continues here
Poll every node from the HOSTS_TABLE periodically.
if ( node N found DEAD )
    PERFORM_RECOVERY
LOOP END
END.

```

(b) *Module Name* ⇒ *Key\_Arbiter*

*Purpose* ⇒ *This module gets 'forked' as a separate process if any one of the D\_SemSer identifies itself as the Key\_Arbiter. It allows only one distributed semaphore to be created for a given KEY over the network at any instant.*

```

BEGIN
Initialize ARB_TABLE.
Broadcast KEY_ARB_ADDRESS to all D_SemSers.
LOOP FOREVER
    receive_message_from_any_D_SemSer
    if ( CREATE request )
        begin
            if ( Given KEY present in the ARB_TABLE )
                return ERROR.
            else
                append new entry in the ARB_TABLE
                    with creator's node address.
            end
        end
    else if ( OPEN request )
        begin
            Get semaphore ID and creator's node address for the
                given KEY from the ARB_TABLE.
            Send ADD_USR to the creator along with the new
                user process id and Address.
            return semaphore ID.
        end
    else if ( CLOSE request )
        begin
            remove entry from ARB_TABLE for the given KEY.
        end
    end
LOOP END
END.

```

(c) Module Name  $\Rightarrow$  **PERFORM\_RECOVERY**

*Purpose  $\Rightarrow$  Does necessary recovery steps needed incase of any node failure detected in the network.*

```
BEGIN
  for ( all semaphores in EXT_SEM_TAB whose owner_add is node
        N and is_replica is TRUE )
    begin
      Create Semaphore with fields of EXT_SEM_TAB entry.
      Make entries in SEM_TABLE.
      Send request to Key_Arbiter for change of creator.
      Elect any other node as its hot-standby replica.
    end
  for ( all semaphore entries in the SEM_TABLE )
    begin
      if ( any user from node N in SEM_USERS list)
        begin
          Remove the entry from SEM_USERS list.
          Undo the operations it had performed from undo_log
        end
      end
    end
  end
END.
```

(d) Module Name  $\Rightarrow$  **SERVICE\_LOCAL**

*Purpose  $\Rightarrow$  Any service request from the users within the node will be taken care by this module. It also keeps the hot-standby replica of any particular distributed semaphore be informed i.e., records every operation.*

```
BEGIN
  if ( CREATE request )
    begin
      send KEY to Key_Arbiter .
      if ( ERROR returns )
        return ERROR.
      else
        Branch to CREATE_DSEM.
      end
    end
  Branch to OPEN_DSEM or DOWN_DSEM or UP_DSEM or CLOSE_DSEM
  corresponding to the incoming request.
  Send request to the hot-standby replica.
  Record performed operations into UNDO_LOG.
  return MESSAGE.
```

END.

(e) Module Name  $\Rightarrow$  SERVICE\_EXTERN

*Purpose  $\Rightarrow$  Every request from any other nodes are serviced in this module. It returns the result to the requested node through network communication.*

BEGIN

Branch to OPEN\_DSEM or DOWN\_DSEM or UP\_DSEM or CLOSE\_DSEM corresponding to the incoming request.

Construct MESSAGE

Send request to the hot-standby replica.

Establish connection with the requested NODE and send MESSAGE.

END.

(f) Module Name  $\Rightarrow$  CREATE\_DSEM

*Purpose  $\Rightarrow$  Creates a distributed semaphore with the given KEY, if no other distributed semaphore with the same KEY has been created elsewhere in the network.*

BEGIN

Create a semaphore with the given KEY.

Make entry in the SEM\_TABLE.

return semaphore ID.

END.

(g) Module Name  $\Rightarrow$  OPEN\_DSEM

*Purpose  $\Rightarrow$  Allows any user to participate in a distributed semaphore which has been already created. For user requests within the node, if the node is not the creator of the distributed semaphore, then request will be forwarded to the creator. It also caters for the OPEN requests from other nodes.*

BEGIN

if ( semaphore not in SEM\_TABLE for the given KEY )

begin

if ( LOCAL request )

begin

Send KEY to the Key\_Arbiter .

if ( ERROR returns )

return ERROR.

else



```

begin
    Make entry in the EXT_SEM_TAB.
    return semaphore ID.
end
end
else if ( GLOBAL request )
    return ERROR.
end
else
    return semaphore ID.
END.

```

**(h) Module Name  $\Rightarrow$  DOWN\_DSEM**

*Purpose  $\Rightarrow$  Does DOWN operation on a specified semaphore. For the user requests within the node, if the node is not the creator, the request will be forwarded to the creator. It also caters for the DOWN requests from other nodes.*

```

BEGIN
if ( semaphore not in SEM_TABLE for the given sem_ID )
begin
    if ( LOCAL request )
begin
        Get node address of the semaphore creator for the
            given sem_ID from EXT_SEM_TAB.
        Send request to that node.
        Get reply.
        return value.
    end
    else if ( GLOBAL request )
        return ERROR.
end
end
else
begin
    Perform DOWN operation.
    return value.
end
END.

```

(i) *Module Name* ⇒ *UP\_DSEM*

*Purpose* ⇒ *Does UP operation on a specified semaphore. For the user requests within the node, if the node is not the creator, the request will be forwarded to the creator. It also caters for the UP requests from other nodes.*

```
BEGIN
  if ( semaphore not in SEM_TABLE for the given sem.ID )
  begin
    if ( LOCAL request )
    begin
      Get node address of the semaphore creator for the
        given sem.ID from EXT_SEM_TAB.
      Send request to that node.
      Get reply.
      return value.
    end
    else if ( GLOBAL request )
      return ERROR.
  end
  else
  begin
    Perform UP operation.
    return value.
  end
END.
```

(j) *Module Name* ⇒ *CLOSE\_DSEM*

*Purpose* ⇒ *The CLOSE requests are serviced within this module. The specified distributed semaphore will be removed from the system, if this request comes from the only user of it. In other cases, only the user entry is removed.*

```
BEGIN
  if ( semaphore not in SEM_TABLE for the given sem.ID )
  begin
    if ( LOCAL request )
    begin
      Get node address of the semaphore creator for the
        given sem.ID from EXT_SEM_TAB.
      Send request to that node.
      Get reply.
    end
  end
END.
```

```

        Remove user entry from SEM_USERS in EXT_SEM_TAB
        entry corresponding to the given sem_ID.
    end
    else if ( GLOBAL request )
        return ERROR.
    end
    else
    begin
        Remove user entry from SEM_USERS in SEM_TABLE entry
        corresponding to the given sem_ID.
    end
END.

```

All user processes' (programs) requests are forwarded to the local *D\_SemSer* only through the library module, which should be linked with the user programs. Algorithm for the Library module is depicted below.

**(k) Module Name  $\Rightarrow$  L\_LIBRARY**

*Purpose  $\Rightarrow$  This module contains the function definitions of the distributed semaphore calls used in the user programs and should be used as a link library. This acts as the interface between the user programs and the D\_SemSer.*

```

BEGIN
    Initialize communication channel.
    Construct MESSAGE structure according to the function call made.
    Send to D_SemSer.                                 $\rightarrow$  Local D_SemSer
    receive from D_SemSer.
    return necessary result.
END.

```

**3.3.2 *D\_SemSer* Architecture**

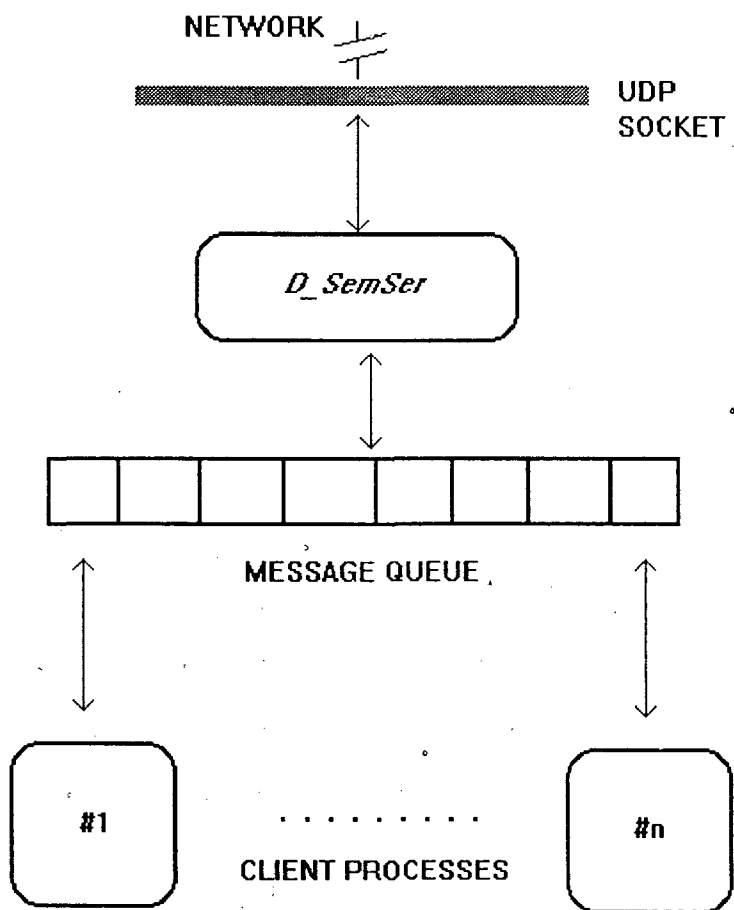
The architecture of *D\_SemSer* is shown in *Figure 3.3*. Every user process function call (distributed semaphore's) will be passed through the L\_LIBRARY module, where a MESSAGE structure is constructed according to the call made. This MESSAGE has to be forwarded to the *D\_SemSer* running in that machine. This intrasite communications are based on *UNIX System V IPC - MESSAGE QUEUES*. The communication

between any two *D\_SemSers* and with the *Key\_Arbiter* is based on *DATAGRAM PROTOCOL (UDP/IP) SOCKETS*. An utility *d\_sem*, like *ipcs -s* in UNIX System V, is provided for the user to view the network wide distributed semaphores, i.e., which lists all the distributed semaphores over the network at that instant, with its KEY, ID and ADDRESS of the creator.

### 3.3.3 Development Environment

We have implemented an initial version of the *D\_SemSer*, running on a cluster of DEC ALPHA Workstations connected by a 10Mbps Ethernet. The details regarding the development environment are ,

Hardware	: DEC ALPHA Workstations.
Operating System	: OSF/1.
Programming Tool	: C.
Networking Protocol	: UDP/IP .
Other utilities	: UNIX System V IPC - Message Queues.



**Figure 3.3 *D\_SemSer* Architecture.**

## 4. MUTUAL EXCLUSION USING *D\_SemSer*

The design of a mutual exclusion algorithm consists of defining the protocols to synchronize entry into the critical section. Here, in this chapter we describe, how the model discussed in earlier chapters, is helpful in achieving mutual exclusion in distributed systems.

The following algorithm gives code to be executed for entry into the critical section, exit from the critical section. Before giving the algorithm, we assume the following, for our example :

- o The shared resource is associated with a unique KEY
- o It is a single resource

### 4.1 Algorithm

```
begin
    id = dsem_create(KEY, 1);
    if (ERROR returns)                                i.e., already created
    {
        id = dsem_open(KEY);
    }
    repeat
        Execute Non-Critical Section.
        dsem_down(id, 1);                            Enter Critical Section
        Execute Critical Section.
        dsem_up(id, 1);                               Exit Critical Section
    until done
    dsem_close(id);
end.
```

## 4.2 Description

Consider user1 executing in node1 (workstation 1) and user2 executing in node2 (workstation 2), both want to synchronize their activity over acquiring a shared resource which is associated with a unique KEY. Thus both users execute the above algorithm.

Initially, both users (through *D\_SemSer*) try to create a distributed semaphore with the KEY. The *Key\_Arbiter* allows any one of the user to create the semaphore (Module *Key\_Arbiter* algorithm), and thus the *D\_SemSer* which creates it becomes the 'arbiter' of that distributed semaphore (Refer *Figure 4.1a*). The other user simply gets the ID of the distributed semaphore. For our convenience, say user1 is the creator.

Now, because of unequal non-critical section code, say user2 issues the *dsem\_down* call first. This call directed to the creator (*D\_SemSer1*), downs the semaphore, finds the value equal to 0 and returns to user2. Now user2 can execute critical section. Subsequently, now user1 issues *dsem\_down()* call, for it to enter the critical section. Now since it the creator, the request is a local request for *D\_SemSer1* and it downs the semaphore, finds the value less than 0 and BLOCKS, i.e., user1 is waiting on *dsem\_down()* call. (Refer *Figure 4.1b*).

Whenever, in user2, the critical section code is over, it issues a *dsem\_up()* call, which is directed to *D\_SemSer1* through *D\_SemSer2*, increments the distributed semaphore's value by 1, finds value equal to 0, UNBLOCKS for user1 (i.e., makes a return to the call *dsem\_down()* made by user1) and returns to user2. Now user1 is executing its critical section code, and user2 is in its non-critical section code.

Thus this algorithm allows, at any instant of time, only one user to be in its critical section.

Now, we consider how our model behaves in case of Faults i.e., node crashes. Suppose, we assume that the system is as in *Figure 4.1b*, and suddenly node2 crashes.

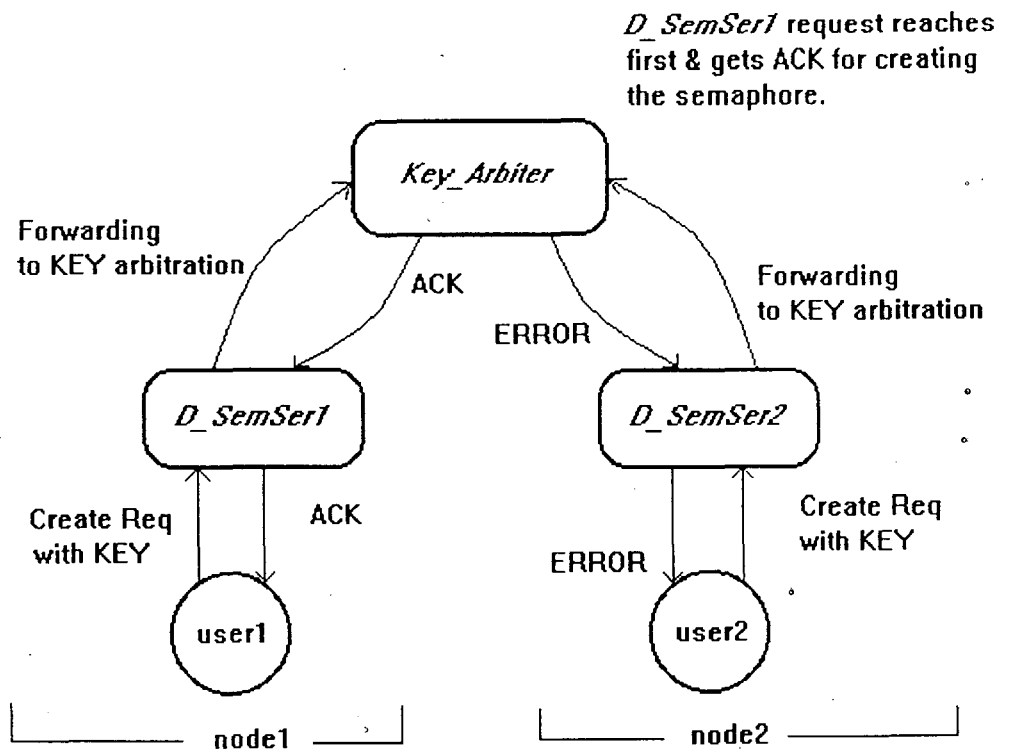
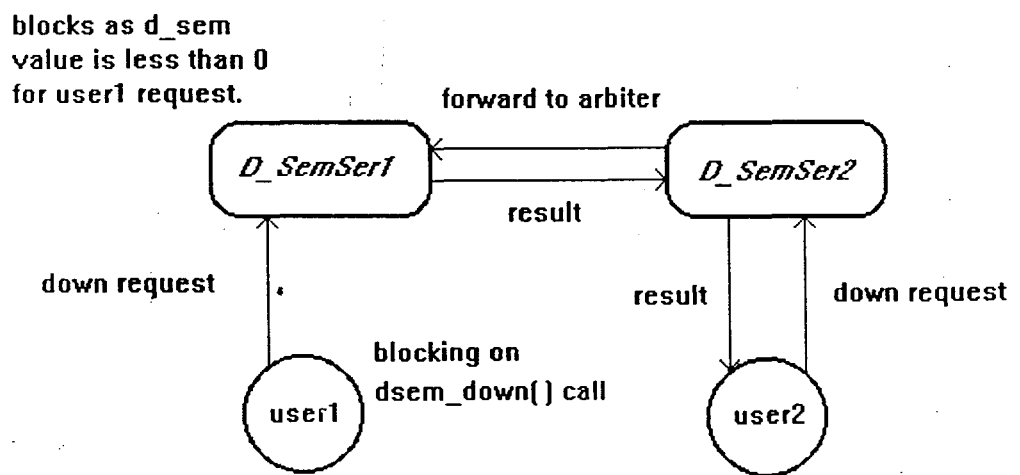


Figure 4.1(a)



42  
Figure 4.1 (b)



Here, user1 is waiting for user2 to release the resource and user2 is no longer using the resource, moreover is no longer available. Now, our Fault tolerant feature incorporated in our *D\_SemSer* comes handy in this situation. In our model, each *D\_SemSer* polls every node periodically, thus *D\_SemSer1* in node1 polls and finds out that node2 is no longer 'alive'. Hence, it initiates the PERFORM\_RECOVERY module. This works as follows :

First it checks, whether it is hot-standby replica for any distributed semaphore and since no other semaphores are there in the system other than the one that is created here in node1, this part does nothing.

Next thing it does is that, checks any distributed semaphore has users from the failed node (node2), and finds that user2 is there for the distributed semaphore we are talking about. *D\_SemSer1* removes the user2 entry from its SEM\_TABLE's user list and undoes whatever operations user2 has done on the semaphore, which are stored in the UNDO\_LOG file. This makes the value of the distributed semaphore to be 0 and thus user1 is released from its BLOCKING state as *D\_SemSer1* sends a message to user1 to UNBLOCK from its *dsem\_down()* call. Hence, user1 enters its critical section. Thus, the algorithm works fine even in the case of any node failure and hence it is Fault Tolerant.

Thus, here also , the model makes sure that at most only one user enters its critical section, even in the case of the node failures, at any point of time. That too, all this fault tolerant recovery algorithm is executed transparently to the user that, any distributed semaphore user hardly notices that there was a node failure, except those who were using the failed node.

### 4.3 Analysis

A brief discussion over the performance of the above algorithm is given here. The number messages required for a process to enter and exit a critical region contribute mainly to the analysis.

Every process willing to enter the critical section, first issues a `dsem_create()` call. In case if distributed semaphore already exists, the process needs to issue `dsem_open()` call.

Before entering the critical section, the process issues a `dsem_down()` call, and while exiting issues `dsem_up()` call. The process disengages its participation with the distributed semaphore with a `dsem_close()` call.

Thus for any process, under the proposed model, the number of messages per entry/exit to a critical section is 4 (incase if it is the creator, then the number messages become 3).

## 5. CONCLUSION

A fault tolerant distributed semaphore mechanism for mutual exclusion in distributed system has been presented. The main advantages claimed for the new architecture are the reduction of the network traffic overhead due to fault tolerance and the relative simplicity of the strategy. The fault tolerant distributed control strategy described in this dissertation is useful in realizing fault-tolerant global control of resources in distributed computing systems. The mechanism discussed here has been implemented without any Operating System Kernel modification (i.e., at the application layer), so that it can be ported to any system which supports UNIX System V IPC and BSD Sockets. User Interface has been provided as set of C library function calls, which resembles any standard System call in UNIX.

The mechanism described has been implemented on OSF/1 platform, and is successfully running on a cluster of DEC Alpha Workstations connected by a 10 Mbps Ethernet.

Future works include improving the protection access, management of replicas, examining the performance of a number of typical asynchronous concurrent programs using *D\_SemSer*.

## BIBLIOGRAPHY

1. **Bach, M.J.** : The Design of the UNIX Operating System, Englewood cliffs, N.J : Prentice Hall, 1986.
2. **Brown, C.** : UNIX Distributed Programming, Prentice Hall International (UK), 1994
3. **Comer, D.E.** : Internetworking with TCP/IP vol I : Principles, Protocols and Architecture, 2nd ed, Englewood cliffs, N.J : Prentice Hall, 1991.
4. **Comer, D.E.** : Internetworking with TCP/IP vol III : Client Server Programming and Application, Englewood cliffs, N.J : Prentice Hall, 1993.
5. **Fleisch, B.D.** : "*Distributed System V IPC in Locus: A Design and Implementation Retrospective,*" Proc. SIGCOMM' 86 Symp. on Communications Architectures and Protocols, ACM, pp. 386-396, 1986.
6. **Jalote, P.** : An Integrated Approach to Software Engineering, Narosa, 1991.
7. **Stevens, R.W.** : Advanced Programming in the UNIX Environment, Addison-Wesley, 1992.
8. **Stevens, R.W.** : UNIX Network Programming, Englewood cliffs, N.J : Prentice Hall, 1990.
9. **Tanenbaum, A.S.** : Distributed Operating Systems, Englewood cliffs, N. J : Prentice Hall, 1995.
10. **Wong, J., Thambu, P., Stoen, R.** : "*A fault tolerant Algorithm for Mutual exclusion in a distributed system,*" Journal of Systems Software, vol. 29, pp. 121-134, 1995.
11. **Yuan, S.M., Wu, C.J., Lien, H.M., Chen, I.N.** : "*Design and Implementation of a Distributed Semaphore facility,*" Proc. 12th Conf. on Distributed Computing Systems, IEEE, pp. 180-184, April 1992.