Fault-Tolerant Token Based Algorithm for Mutual Exclusion in Distributed Environment

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

Master of Engineering
in
Software Engineering

Under the supervision of
Mr. R. S. Salaria
Assistant Professor
CSED, TIET, Patiala

Submitted By

Vaibhav Saini
8043124

MAY 2006

COMPUTER SCIENCE & ENGINEERING DEPARTMENT
THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
(DEEMED UNIVERSITY)
PATIALA – 147004

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Fault-Tolerant Token based Algorithm for Mutual Exclusion in Distributed Environment"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted in Computer Science and Engineering Department of Thapar Institute of Engineering and Technology (Deemed University), Patiala, is an authentic record of my own work carried out under the supervision of Mr. R. S. Salaria. The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

Vaibhav Saini

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

Mr. R. S. Salaria
Assistant Professor
Computer Science & Engineering Department
Thapar Institute of Engineering & Technology
Patiala-147004

Countersigned by

Dr. (Mrs.) Seema Bawa

Head
Computer Science & Engineering Department
Thapar Institute of Engg & Technology
Patiala.

Dr. T. P. Singh

Dean
Academic Affairs
Thapar Institute of Engg & Tech
Patiala

The M.E. (Thesis) Viva Voca examination of Vaibhav Saini Roll No. 8043124, M.E (Software Engineering), Thapar Institute of Engineering & Technology, Patiala has been held on ……

Supervisor External Examiner

Acknowledgement

No volume of words is enough to express my gratitude towards my guide, Sh. R. S. Salaria, Assistant Professor, CSED, TIET, who has been very concerned and has aided for all the material essential for the preparation of this seminar report. He has helped me explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research-oriented venture.

I am also thankful to Dr. (Mrs.) Seema Bawa, Head, CSED and Sh. Rajesh Bhatia, P.G. Coordinator, for the motivation and inspiration that triggered me for the seminar work.

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

Most importantly, I would like to thank my parents and the almighty for showing me the right direction out of the blue, to help me stay calm in the oddest of the times and keep moving even at times when there was no hope.

Vaibhav Saini

8043124

Abstract

In distributed computing, many problems require a shared object to be allocated to a number of requesting processing in mutually exclusive manner. Hence the mutual exclusion problem plays a vital role in the design of distributed systems. A considerable amount of literature is available on this problem. The performance metrics of the mutual algorithm are the number of messages required per CS invocation, response time. Moreover, it should be starvation-free and deadlock-free.

The algorithms available in literature vary in different ways viz. the performance of the algorithm under low and high load, resilience behavior the algorithm. Most of the token-based algorithms take *O (n)* messages per CS invocation. Some of the algorithms are not resilient to site and communication failure.

The algorithm implemented here in this thesis for mutual exclusion in distributed environment is token-based. It takes 2-3 messages per CS invocation at low load and a few more at higher load. This algorithm is starvation-free and deadlock-free, but not resilient to site and communication failure. Using broadcast, this algorithm has been modified to be immune to site and communication failure.

To implement this, I have used PARSEC simulator that is a C-based discrete-event simulation language. It adopts the process interaction approach to discrete-event simulation. The simulation results prove that this algorithm is more efficient as compared to algorithms in literature in terms of number of messages passed per CS invocation.

TABLE OF CONTENTS

LIST OF FIGURES & TABLES

CONTENTS **PAGE NO.**

List of Figures

List of Tables

Chapter 1
Introduction

Mutual Exclusion problem arises whenever there is a sharing of resources. A resource can be a part of a single processor system or multi-processor system. In single processor system, mutual exclusion can be efficiently handled by using semaphores, monitors, etc. But the same cannot be used in multiprocessor systems; reason being the same critical section is present at different nodes. To handle critical section in multi-processor systems, we have to pass messages among all the nodes, to intimate about access of the critical section [1]. So it is handled using message passing.

In literature, many algorithms [2,3,4,5,6,7,8,9,10] are present that provide mutual exclusion in distributed environment. The performance of these algorithms depends on two things: number of messages passed per entry/ exit critical section and average delay time.

During my thesis work, I have studied a number of algorithms that implement mutual exclusion in distributed environment. Some algorithms implement it with a high number of message-passing [5,6,8,9], some algorithms cause starvation [7,10] and some are not deadlock-free.

Here, I studied An Efficient Token-Based Algorithm for Distributed Mutual Exclusion [11] proposed by P. K. Dash and R. C. Hansdah. The algorithm uses a distributed queue that is not necessarily FIFO, to enqueue the request messages of the node for entry into the critical sections. These request messages are time-stamped using roughly synchronized clocks. On the average, the algorithm requires 2-3 number of messages per critical section over wide range of workload and a few additional number of messages at lower loads for large networks. This is a contrast to the existing algorithms on the literatures that require around $O(n)$ number of nodes in the system. The algorithm is deadlock-free and starvation-free. But the algorithm is not resilient to site and

communication failures. I have modified this algorithm to resilient to site and communication failures using a broadcast which is called when the timeout period for receiving token expires. I have implemented this algorithm using PARSEC (PARallel Simulation Environment for Complex systems) [12,13].

In fault free environment, the algorithm I implemented requires 2-3 messages per critical section. But when a fault occurs, it takes *O(n)* more messages to handle the fault.

Finally, we conclude with the future scope.

Chapter 2

Background Information

2.1 Introduction

A distributed system is a collection of processes that do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication networks such as high-speed buses or telephone lines.

A distributed system is a collection of loosely coupled processors interconnected by a communication network [1]. From the point of view of specific processors in distributed systems, the rest of the processors and their respective resources are remote, whereas its own resources are local. The processors in distributed system may vary in size and function. The purpose of the distributed system is to provide an efficient and convenient environment for sharing of resources.

2.2 Advantages of Distributed Systems

The four major reasons for building distributed systems are:

- Resource Sharing
- Computation Speed-up
- Reliability
- Communication

Resource Sharing: If a number of different sites are connected to one another, then a user at one site may be able to use the resources available at another. For example, a user at site A may be using a laser printer located at site B. Meanwhile, a user at B may access a file that resides at A. In general, resource sharing in a distributed system provides mechanism for sharing files at remote sites, processing information in a distributed database and performing other operations.

Computation Speed-Up: If a particular computation into sub-computations that can run concurrently, then a distributed system allows us to distribute the sub-computations among the various sites; the sub-computations can be run concurrently and thus provide computation speed up. In addition, if a particular site is currently overloaded, some of them may be moved to other lightly loaded sites. This movement of jobs is load sharing. Automated load sharing, in which the distributed systems automatically moves jobs is not yet common in commercial systems.

Reliability: If one site fails in a distributed system, the remaining sites can continue operating, giving the system better reliability. If the system is composed of multiple large autonomous installations, the failure of one of them should not affect the rest. If on the other hand, the system is composed of small machines, each of which is responsible for some crucial system functions, then a single failure may halt the operation of the whole system. In general, if the enough redundancy, the system can continue operation, even if some of its sites have failed.

The failure of the site must be detected by the system; an appropriate action may be needed to recover from the failure. The system must no longer should use the service of that site. In addition, if the function of the failed site can be taken over by another site, the system must ensure that the transfer of function occurs correctly. Finally, when the failed site recovers or is repaired, mechanisms must be available to integrate it back into the system smoothly.

Communication: When several sites are connected to one another by communication network, the users at different sites have the opportunity to exchange information at a low level message are passed between systems in a manner similar to the single computer message system. Given message passing, all the higher level functionality found in stand alone systems can be expanded to encompass the distributed systems. The advantage of distributed system is that these functions can be carried out over great distances. Two people at geographically disparate sites can collaborate on a project, for example, by transferring the files of the project, logging into each other's remote systems to run

program, and exchanging mail to coordinate the work, users minimize the limitations inherent in long distance work. The advantages of distributed system have resulted in an industry-wide trend toward downsizing. Many companies are replacing the mainframes with networks of workstations or personal computers.

2.3 Architecture

Various hardware and software architectures exist that are usually used for distributed computing [14]. At a lower level, it is necessary to interconnect multiple CPUs with some sort of network, regardless of that network being printed onto a circuit board or made up of several loosely-coupled devices and cables. At a higher level, it is necessary to interconnect processes running on those CPUs with some sort of communication system.

- **Client-server:** Smart client code contacts the server for data, then formats and displays it to the user. Input at the client is committed back to the server when it represents a permanent change.
- **Three-tier architecture:**Three tier systems move the client intelligence to a middle tier so that stateless clients can be used. This simplifies application deployment. Most web applications are 3-Tier.
- **N-tier architecture:** N-Tier refers typically to web applications which further forward their requests to other enterprise services. This type of application is the one most responsible for the success of application servers.
- **Tightly coupled:** refers typically to a set of highly integrated machines that run the same process in parallel, subdividing the task in parts that are made individually by each one, and then put back together to make the final result.
- **Peer-to-peer:** an architecture where there is no special machine or machines that provide a service or manage the network resources. Instead all responsibilities are uniformly divided among all machines, known as peers.
- **Service oriented:** Where system is organized as a set of highly reusable services that could be offered through a standardized interfaces.
- **Mobile code:** Based on the architecture principle of moving processing closest to source of data

2.4 Topology

The sites in the system can be connected physically in a variety of ways. Each configuration has advantages and disadvantages. We can compare the configurations by using the following criteria:

Installation Cost: The cost physically linking the sites in the system.

Communication Cost: The cost, time and money to send a message from site A to site B.

Availability: The extent to which the data can be accessed despite the failure of some sites and links.

Fully Connected Network: Each site is connected to other site. The number of links grows as the number of square of number of sites resulting in a huge installation cost. Therefore, fully connected networks are impractical in any large system.

Partially Connected Network: Direct links exist between some - but not all - pair of sites. Hence the installation cost of such a configuration is lower than that a fully connected network. However, if two sites A and B are not directly connected, messages from one to the other must be routed through a sequence of communication links. This requirement results in a higher communication cost. If a communication link fails, messages that would have been transmitted across the link must be re-routed. In some cases, another route through the network may be found so that messages are able to reach their destination. In other cases, a failure may result in no connection between some pairs of sites. A system is partitioned if it has been split into two more subsystems, called partitions, which lack any connection between them. The different partially connected network types include:

- Tree-Structured Networks
- Ring Networks
- Star Networks

They have different characteristics and installation and communication costs. Installation and communication cost are relatively low for a tree-structured network. However, the failure of a single link in a tree-structured network can result in becoming partitioned. In

a ring network, at least two links must fail for partition to occur. Thus, the ring network has a higher degree of availability than a tree-structured network. However, communication cost is high, since a message may have to cross a large number of links. In a star network, the failure of single link in a network partition, but one of the common partitions has a single site. Such a site can be treated a single site failure. The star network has a low communication cost since each site is at most two links away from every other site. However, the failure of the central site results in every site in the system becoming disconnected.

2.5 Robustness

A distributed system can suffer from various types of hardware failures. The failure of a link, the failure of a site, and loss of message are the most common failures. To ensure the system is robust, we must detect any of these failures, re-configure the system such that the computation may continue and recover when a link or site is repaired.

Failure Detection: In an environment with no shared memory, we are generally unable to differentiate among link failure, site failure and message loss. We can usually detect only that one of these failures has occurred, not what kind of failure it is. Once a failure has been detected, appropriate actions must be taken, depending on the particular application to detect link and site failure, we use a hand-shaking procedure.

Reconfiguration: Suppose that a site A has discovered a failure, it must then initiate the procedure that will allow the system to reconfigure and re-continue its normal mode of operation. If a direct link from A to B has failed, this information must be broadcast to every site in the system, so that the previous routing tables can be updated accordingly. If the system believes that a site has failed then every site in the system must be notified so that they no longer will attempt to use the services of the failed sites.

Recovery from Failure: When a failed site or link is repaired, it must be integrated into the system gracefully and smoothly. Suppose a link between A and B when it is repaired,

both A and B must be notified. We can accomplish this notification by continuously repeating the hand-shaking procedure.

2.6 Design Issues

Making the multiplicity of processors and storage devices transparent to the users has been the challenge to many users. Ideally, a distributed system should look to its users like a conventional, centralized system. The user interface of a transparent distributed system should not distinguish between local and remote resources, that is, user should be able to access the remote distributed system as though the later were local.

A distributed system should be fault-tolerant. A fault tolerant system continues to function, perhaps in a degraded form, when faced with failure. The degradation can be in performance, in functionality or in both. It should be proportional, however, to the failures that cause it. The system that grinds to a halt when only a few of its components fail is certainly not fault-tolerant. Unfortunately, fault-tolerance is difficult to implement. Most commercial systems provide only limited fault tolerance.

The capability of a system to adapt to increased service load is its scalability. Systems have bounded resources and can become completely saturated under increased load. For example, regarding a file system, saturation occurs either when a server CPU runs at a high utilization rate, or when disks are almost full. Scalability is a relative property, but it can be measured accurately. A scalable system reacts more gracefully to increased load than a non-scalable system. Even a perfect design cannot accommodate an ever-growing load. Adding new resources might solve the problem, but it might generate additional indirect load on other resources. Even worse, expanding the system can incur expensive design modifications. A scalable system must have the potential to grow without these problems. In a distributed system, the ability to scale gracefully is of special importance since expanding the network by adding new machines or interconnecting two networks is commonplace. Very large distributed systems, to a great extent, are still only theoretical. No magic guidelines ensure the scalability of the system.

One principle for designing very large-scale systems is that the service demand from any component of the system should be bounded by constant that is independent of the number of nodes in the system. Any service mechanism whose load demand is proportional to the size of the systems is destined to become clogged once the system grows beyond a certain size. Adding resources will not alleviate such a problem. The capacity of this mechanism simply limits the growth of the system.

The practical approximation to symmetric autonomous configuration is clustering, in which the system is partitioned into a collection of semi-autonomous clusters. A cluster consists of set of machines and a dedicated cluster server. So that cross-cluster resource references are relatively infrequent, each cluster server should satisfy request of its own machine most of the time. If the cluster is well balanced, that is, if the server in-charge suffices to satisfy all the cluster demands – it can be used as a modular building block to scale up the system.

Chapter 3

Distributed Mutual Exclusion

3.1 Introduction

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. We will now look at a few examples of how critical regions and mutual exclusion can be implemented in distributed systems [1].

3.2 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, as shown in figure 3-1(a) below. When the reply arrives, the requesting process enters the critical region.

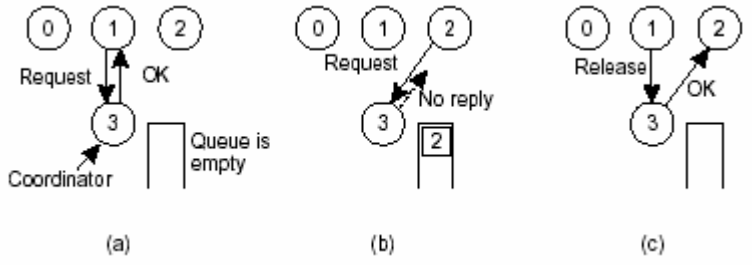

Figure 3-1 (a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted. (b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply. (c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2.

Now suppose that another process, 2 in Figure 3-1(b), 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. In Figure 3-1(b), 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 and waits for more messages.

When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access, as shown in Figure 3-1(c). 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 only lets 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 (no starvation). The scheme is easy to implement, too, 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, a single coordinator can become a performance bottleneck.

3.3 A Distributed Algorithm

Having a single point of failure is frequently unacceptable, so researchers have looked for distributed mutual exclusion algorithms. Lamport’s 1978 paper on clock synchronization presented the first one. Ricart and Agrawala (1981) made it more efficient. In this section

we will describe their method. Ricart and Agrawala's 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 actually happened first. Lamport's algorithm is one way to achieve this ordering and can be used to provide timestamps for distributed mutual exclusion. 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 processes, 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:

1. 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.
2. If the receiver is already in the critical region, it does not reply. Instead, it queues the request.
3. 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 requests 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, as shown in
Figure 3-2(a) below.

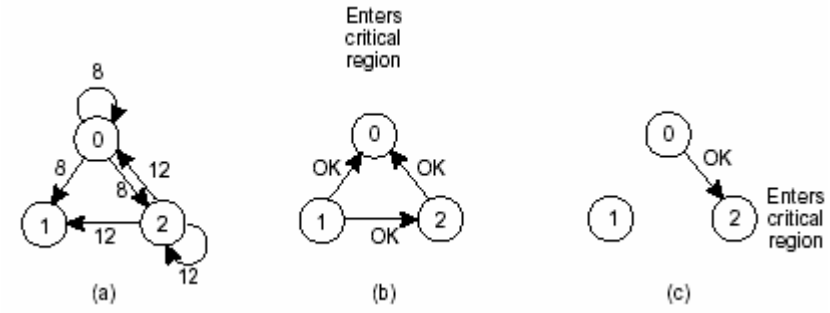

Figure 3-2 (a) Two processes want to enter the same critical region at the same moment. (b) Process 0 has the lowest timestamp, so it wins. (c) When process 0 is done, it sends an *OK* also, so 2 can now enter the critical region.

Process 0 sends everyone a request with timestamp 8, while at the same time, 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, as shown in Figure 3-2(b). When it is finished, it removes the request from 2 from its queue and sends an *OK* message to process 2, allowing the latter to enter its critical region, as shown in Figure 3-2(c). The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps. Note that the situation in figure above would have been essentially different if process 2 had sent its message earlier in time so that process 0 had gotten it and granted permission before making its own request. In this case, 2 would have noticed that it itself was in a critical region at the time of the request, and queued it instead of sending a reply. 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 processes failing is at least n times as large as a single coordinator failing, we have

managed to replace a poor algorithm with one that is more than *n* times worse and requires much more network traffic to boot. The algorithm can be patched up by the same trick that we proposed earlier. When a request comes in, the receiver always sends a reply, either granting or denying permission. Whenever either a request or a reply is lost, the sender times out and keeps trying until either a reply comes back or the sender concludes that the destination is dead. After a request is denied, the sender should block waiting for a subsequent *OK* message. Another 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 that 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. Various minor improvements are possible to this algorithm. For example, getting permission from everyone to enter a critical region is really overkill. All that is needed is a method to prevent two processes from entering the critical region at the same time. The algorithm can be modified to allow a process to enter a critical region when it has collected permission from a simple majority of the other processes, rather than from all of them. Of course, in this variation, after a process has granted permission to one process to enter a critical region, it cannot grant the same permission to another process until the first one has released that permission.

Other improvements are also possible, such as proposed by Maekawa (1985), but these easily become more intricate. Nevertheless, this algorithm is slower, more complicated, more expensive, and less robust that the original centralized one. Why bother studying it under these conditions? For one thing, it shows that a distributed algorithm is at least possible, something that was not obvious when we started. Also, by pointing out the shortcomings, we may stimulate future theoreticians to try to produce algorithms that are

actually useful. Finally, like eating spinach and learning Latin in high school, some things are said to be good for you in some abstract way.

3.4 A Token Ring Algorithm

A completely different approach to achieving mutual exclusion in a distributed system is illustrated in figure below. Here we have a bus network, as shown in Figure3-3(a), (e.g., Ethernet), with no inherent ordering of the processes. In software, a logical ring is constructed in which each process is assigned a position in the ring, as shown in Figure3-3(b) 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.

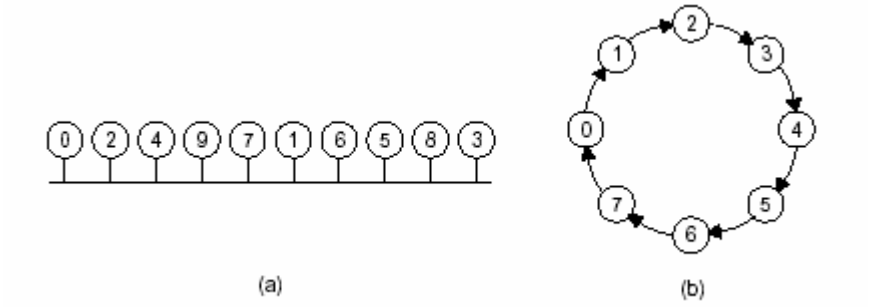

Figure 3-3 (a) An unordered group of processes on a network. (b) A logical ring constructed in software.

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 easy to see. Only one process has the token at any instant, so only one process can actually 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. As usual, this algorithm has problems too. 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, or the one after that, if necessary. Of course, doing so requires that everyone maintain the current ring configuration.

3.5 A Comparison of Algorithms

A brief comparison of the three mutual exclusion algorithms we have looked at is instructive. In the table (figure 3-4), we have listed the algorithms and three key properties: the number of messages required for a process to enter and exit a critical region, the delay before entry can occur (assuming messages are passed sequentially over a network), and some problems associated with each algorithm.

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

Figure 3-4 A comparison of algorithms.

The centralized algorithm is simplest and also most efficient. It requires only three messages to enter and leave a critical region: a request, a grant to enter, and a release to exit. The distributed algorithm requires $n - 1$ request messages, one to each of the other processes, and an additional $n - 1$ grant messages, for a total of $2(n - 1)$. (We assume that only point-to-point communication channels are used.) With the token ring algorithm, the

number is variable. If every process constantly wants to enter a critical region, then each token pass will result in one entry and exit, for an average of one message per critical region entered. At the other extreme, the token may sometimes circulate for hours without anyone being interested in it. In this case, the number of messages per entry into a critical region is unbounded. The delay from the moment a process needs to enter a critical region until its actual entry also varies for the three algorithms. When critical regions are short and rarely used, the dominant factor in the delay is the actual mechanism for entering a critical region. When they are long and frequently used, the dominant factor is waiting for everyone else to take their turn .It takes only two message times to enter a critical region in the centralized case, but $2(n - 1)$ message times in the distributed case, assuming that messages are sent one after the other. For the token ring, the time varies from 0 (token just arrived) to $n - 1$ (token just departed). Finally, all three algorithms suffer badly in the event of crashes. Special measures and additional complexity must be introduced to avoid having a crash bring down the entire system. It is ironic that the distributed algorithms are even more sensitive to crashes than the centralized one. In a fault-tolerant system, none of these would be suitable, but if crashes are very infrequent, they might do.

3.6 PARSEC – An Overview

PARSEC (for PARallel Simulation Environment for Complex systems) is a C-based discrete-event simulation language [12]. It adopts the process interaction approach to discrete-event simulation. An object (also referred to as a physical process) or set of objects in the physical system is represented by a logical process. Interactions among physical processes (events) are modeled by time-stamped message exchanges among the corresponding logical processes.

One of the important distinguishing features of PARSEC is its ability to execute a discrete-event simulation model using several different asynchronous parallel simulation protocols on a variety of parallel architectures. PARSEC is designed to cleanly separate the description of a simulation model from the underlying simulation protocol, sequential or parallel, used to execute it. Thus, with few modifications, a PARSEC program may be

executed using the traditional sequential (Global Event List) simulation protocol or one of many parallel optimistic or conservative protocols.

In addition, PARSEC provides powerful message receiving constructs that result in shorter and more natural simulation programs. Useful debugging facilities are available.

3.6.1 Simulation

A simulation model may be used to predict the behavior of a physical system under a variety of operating conditions. In the process-interaction approach to simulation, a physical system is assumed to consist of a set of physical processes that interact with each other at discrete points in time; these interactions are referred to as events. In its simulation model, a logical process (LP) is used to model one or more physical processes (PP); the events in the physical system are modeled by message exchanges among the corresponding logical processes in the model.

Consider a bank teller as a simple example of a physical system. We assume that the bank has *n* teller windows that are open. A teller is said to be idle if she is not servicing a customer and is otherwise said to be busy. Customers arrive at the bank and wait for service in a common queue. The customer at the head of the queue goes to any teller that is idle. After receiving the desired service the customer leaves the bank. We are interested in measuring the average and maximum time that is spent by a customer in the bank. We may be interested in using the model to examine how these metrics change as the number of tellers, the service rate of the teller, or the arrival rate of customers is modified. The processes in this system are the tellers and the queue of waiting customers; interesting events are the arrival of a customer at the queue (henceforth called the arrival event), departure of a customer from the queue, arrival of a customer at a teller (the begin-service event), and departure of a customer from a teller (the end-service event). We will ignore the time spent by a customer in walking from the head of the queue to a teller window; thus the departure of a customer from the queue and his/her arrival at the teller are really the same event.

In its simulation model, we can define logical processes to model the teller and queue processes in the physical system. The logical processes are referred to as teller and queue respectively. Although the model will contain *n* tellers, for simplicity we will use teller to refer to a specific teller process. We also define a source process that creates customers. The customers in the physical system are abstracted via the events and need not be modeled as explicit logical processes. The model defines a source process that generates customers and a sink process that collects statistics on the time spent by a customer in the bank. The arrival event can now be modeled by sending a message from the source process to the queue process; the message contains the arrival time of the customer and perhaps a unique id that is used to track the customer. Similarly the begin-service event is simulated by sending a message from the queue to a teller and the end-service event by sending a message from the teller to the sink.

A primary activity in a simulation model is the scheduling of future events. For instance, the source must schedule the arrival of future customers and a teller must schedule the end-service event. Some of these events simulate the passage of time in the physical system, whereas others are assumed to occur instantaneously. Messages used to simulate physical events are associated with a send time and a receive time. By default, the send and receive time of a message are set to the simulation time of the sender LP. In general, the receive time of a message can be explicitly set to any future value greater than or equal to its send time. An LP may also suspend itself for a specific interval of simulation time to simulate passage of physical time in performing an activity. For instance, assume that the service time for a client at a teller is *t* units. On receiving a begin-service message at say time *T*, a teller may either immediately send an end-service message with a receive time of $T+t$ to the sink LP or it may suspend itself for *t* time units and then send an end-service message with receive and send times of $T+t$ to the sink.

3.6.2 Parallel Simulation

A PARSEC model can be executed using sequential or parallel simulation algorithms. The sequential algorithm typically uses an ordered data structure called the global event list that stores all events that are generated in the system in their timestamp order. Details

on the commands to compile and execute a model on a sequential platform are contained in the manual.

For parallel execution, each LP or entity in the model is mapped to a specific processor. Each processor, say P, has its own event list which stores the events for the entities that are mapped to P. Parallel synchronization algorithms have been defined to ensure that events in the event lists stored on different processors are executed in their global timestamp order. Enforcing this requirement, referred to as the causality constraint, is the central problem in efficient execution of parallel simulations. Two primary approaches have been suggested to solve the synchronization problem: conservative and optimistic.

Conservative algorithms do not permit any causality error: each object in the simulation processes an incoming message only when the underlying synchronization algorithm can guarantee that it will not subsequently receive a message with a smaller timestamp. This constraint may introduce deadlocks, which are typically avoided by using *null messages*. A null message is a timestamped signal sent by an LP to indicate to other LPs a lower bound on the timestamp of its future messages. If an LP sends a null message with timestamp T+la at simulation time T, we say that the LP has a lookahead of *la*, (the LP is looking *la* units ahead in its future). In general, the larger the lookahead of an LP, the better its performance with conservative protocols. Efficient implementation of null messages is also facilitated if each LP maintains the set of its source and/or destination LPs. PARSEC provides a set of constructs to specify the lookahead and topology of a model and thus improve its performance with conservative implementations.

In optimistic protocols, an LP is allowed to process events in any order; however, the underlying synchronization protocol must detect and correct violations of the causality constraint. The simplest mechanism for this is to have each LP periodically save (or checkpoint) its state. Subsequently, if it is discovered that the LP processed messages in an incorrect order, it can be rolled back to an appropriate checkpointed state, following which the events are processed in their correct order. The rollback may also require that the LP unsend or cancel the messages that it had itself sent to other LPs in the system. An optimistic algorithm is also required to periodically compute a lower bound on the

timestamp of the earliest global event, also called the Global Virtual Time or GVT. As the model is guaranteed to not contain any events with a timestamp smaller than GVT, all checkpoints timestamped earlier than GVT can be reclaimed. PARSEC provides a set of facilities that can be used to control the various parameters that affect the performance of an optimistic implementation. These include setting the frequency of checkpointing, and GVT computation among others.

3.7 PARSEC LANGUAGE

3.7.1 Introduction

PARSEC is a C-based language that was designed to cleanly separate the simulation model from the underlying algorithm (sequential or parallel) that may be used to execute the model [13]. A program written in PARSEC is independent of any synchronization algorithm. When it is compiled, the analyst can indicate the specific simulation algorithm that is to be used to synchronize execution of the model: sequential, parallel conservative, or parallel optimistic. The compiler generates the appropriate code to interface the model with the corresponding run-time system: a splay-tree based implementation of the global event-list algorithm for the sequential implementation, a null-message or conditional event implementation of the parallel conservative synchronization algorithm, or a space-time implementation of the optimistic synchronization algorithm.

PARSEC has been implemented on a variety of sequential workstations and laptop machines, on networks of workstations and on scalable MPP platforms like the distributed memory IBM SP2 and on symmetric multiprocessor (SMP) platforms like the shared memory SparcStation 1000[15]. It has been used for the parallel simulation of a number of applications in diverse areas including queuing networks, VLSI designs, parallel programs, mobile wireless multimedia networks, ATM networks, and high-speed communication switches and protocols.

PARSEC adopts the process interaction approach to discrete-event simulation. An object (also referred to as a PP for physical process) or set of objects in the physical system is represented by a logical process or LP. Interactions among PPs (events) are modeled by

message exchanges among the corresponding LPs. We first describe the PARSEC primitives to define processes and the inter-process communication primitives and subsequently indicate how they are used to describe events.

3.7.2 Entities

A PARSEC program is a collection of entity definitions and C functions. An entity definition (or an entity type) describes a class of objects. An entity instance, called simply an entity, represents a specific object in the physical system. Every PARSEC program must have a **driver** entity. This entity initiates execution of the simulation program and serves essentially the same purpose as the main function in C.

The declaration of an entity is *syntactically* similar to that of a C function; the key difference is that unlike a function, an entity does not return a value, and, in fact, generally doesn't return at all! An entity type consists of a *heading* that declares the name and formal parameters of the entity type and a *body* that declares local variables and defines its actions. For instance, the entity type declared in Figure 3-5 describes an object that models a resource manager. This entity type will be used as a running example to illustrate various PARSEC constructs. The heading in line 3 of the figure indicates that the entity type is called *Manager* and has one integer parameter called *num_resources*. The body of the entity is contained within curly braces in lines 4-13. The body is a compound C statement that may also include constructs to send and receive messages as described subsequently.

An entity is created by the execution of a **new** statement and is automatically assigned a unique id on creation. PARSEC defines a type called **ename** that is used to store entity-identifiers. The following statement will create an instance of entity type *Manager* and store its identifier in variable *manager*, which must be declared to be of type **ename**.

manager = **new** *Manager(*10);

An entity can internally refer to its id using the keyword **self**. An entity instance terminates itself by `falling off the end'.

Figure 3-5: A Resource Manager: Single Resource

```
 1 message Request {ename requester;};
 2 message Release {};
 3 entity Manager (int num_resources) {
 4   int resources = num_resources;
 5   for (;;)
 6     receive (Request req) when (resources > 0) {
 7       resources--;
 8       send Resource{} to req.requester;
 9     }
10     or receive (Release rel) {
11       resources++;
12     }
13 }
```

An entity may contain an optional.

3.7.3 Messages

Entities communicate with each other using buffered message passing. Every entity has a unique message buffer; asynchronous send and receive primitives are provided to respectively deposit and remove messages from the buffer.

- **Declaring Message Types**

  PARSEC uses typed messages. An entity type must define the types of messages that may be received by its instances. A message type is similar to a struct in C and consists of a name and a (possibly empty) parameter list. For instance, two message types are defined for the *Manager* entity type(lines 1-2): *Request* which has one parameter of type **ename**, and *Release* which has no parameters.

- **Sending a Message**

  An entity sends a message by executing a **send** statement. Each message is transparently timestamped with the current simulation time and is deposited in the

destination buffer at the same (simulation) time at which it is sent. The send statement has the following form:

send *message_type* **to** *destination*;

where *destination* is a variable of type **ename** and *message_type* is a message whose type has been declared by the destination entity *destination*. For instance, execution of the send statement in line 8 of the *Manager* will deposit a message of type *Resource* in the message buffer of the entity identified by *requester*. Message type declarations are global to a program (even if declared in separate files), so should be declared in a common header file.

- **Receiving a message**

An entity accepts messages from its message-buffer by executing a PARSEC **wait** statement. In its most commonly used version, the receive statement has the following syntactic form:

```
receive (mt1 mv1) [when b1] statement1;
or receive (mt2 mv2) [when b2] statement2;
 .
 .
 .
or receive (mtn mvn) [when bn] statementn;
```

where mt_i is a message-type, mv_i is a read-only message variable, b_i an optional boolean expression referred to as a *guard*, and $statement_i$ is any C or PARSEC statement. The guard is a side-effect free boolean expression that may refer to local variables or message parameters. If omitted, the guard is assumed to be the constant *true*. The message-type and guard are together referred to as a *resume condition* and the resume condition together with the statement is referred to as a resume statement. A resume condition with message-type mt_i and guard b_i is said to be *enabled* if the message buffer contains a message of type mt_i, which if delivered to the entity would

cause b_i to evaluate to *true*; the corresponding message is called an *enabling message*. A resume condition that is not enabled is said to be *disabled*.

For instance, the receive statement in line 6 of the *Manager* entity consists of two resume statements. The resume condition (line 6) in the first statement ensures that the entity accepts a *Request* message only if enough resources are available (*resources > 0*). If so, the resource is allocated to the requesting entity (line 7-8). The second resume statement (line 10) accepts a *Release* message and simply increments the count of available resources. As discussed subsequently, the resume condition in a receive statement may include multiple message-types, each with its own boolean expression. This allows many complex-enabling conditions to be expressed directly, without requiring the programmer to describe the buffering explicitly.

If two or more resume conditions in a receive statement are enabled, the timestamps on the corresponding enabling messages are compared and the message with the earliest timestamp is removed and delivered to the entity. By default, if all resume conditions in the receive statement are disabled, the corresponding entity is suspended until it receives an enabling message. A timeout mechanism is also supported as described in the section after the following.

3.7.4 Events

Each event in a discrete-event simulation model simulates some activity of interest in the physical system and may involve one or more objects. Every event is associated with a time stamp that indicates the time at which the corresponding event occurs in the system. Execution of a discrete-event simulation model requires that all events in the system be executed in their strict time stamp order.

Since an event is modeled by a message in a PARSEC program, each message carries a timestamp. By default, the timestamp placed on a message is the current (simulation) time of the entity that sends the message. A message may be explicitly timestamped with a future time as described later.

The simulation time of an entity can be advanced only when it receives a message or when it executes a **hold** statement. When an entity with a simulation time t_s, receives a message with time stamp t_e, the simulation time of the entity is set to the larger of t_s or t_e. This ensures that the simulation time of an entity increases monotonically.

A **hold** statement is used to suspend the entity for a given duration in simulation time. For instance, the following statement will cause the entity executing the statement to suspend until the simulation time of the system advances by *t* time units. In other words, if the simulation time of the entity was t_s prior to execution of the **hold** statement, it will be t_s + t when the entity executes the statement following the **hold** statement.

hold (*t*);

As explained earlier, advances in simulation time such as this are used to represent the time required by some activity (i.e. event) in the physical system being simulated.

As an example, consider the *Job* entity in Figure 3-6. The entity periodically sends a request for a resource to the manager and waits to receive the *Resource* message indicating availability of the resource. It simulates use of the resource by executing the hold statement (line 8) that causes the simulation time for the entity to advance by the specified time interval. It then returns the resource to the pool by sending a *Release* message to the manager (line 9).

Figure 3-6: A Job Entity

```
 1 message Resource {};
 2 entity Job(int   processing_time,
 3            ename manager}{
 4   for (;;) {
 5     hold(processing_time);
 6     send Request{self} to manager;
 7     receive (Resource res) {
 8       hold(processing_time);
 9       send Release to manager;
10     }   }  }
```

3.7.5 Conditional Events and Timeouts

PARSEC also provides constructs to schedule conditional future events. These are events that may be cancelled after they have been scheduled. A conditional event is scheduled by using a variation of the receive statement introduced in the previous section. A receive statement may optionally specify a delay-time as follows:

```
receive (mt1 mv1) [when b1] statement1;
...
or timeout after (tc) {
   ...
}
```

Assume that an entity executes a receive statement with delay-time t_c at time t. This causes a message of type **timeout**, with timestamp $t + t_c$, to be conditionally scheduled for the entity. This message is cancelled if an enabling message is available in the message buffer of the entity within the inclusive interval $[t, t + t_c]$; otherwise the timeout message is sent to the entity by the runtime system. The message type **timeout** is declared automatically by the compiler for every entity and must not be declared by the user. If t_c is omitted, it is set by default to an arbitrarily large number such that the corresponding entity is blocked (potentially forever) until an enabling message is available in its buffer. In particular, if t_c is specified to be 0, the statement works like a non-blocking receive: if the message buffer does not contain any enabling message, the entity will continue execution at the same simulation time at which it was suspended.

The event corresponding to the receipt of a timeout message is conditional because it is cancelled if an enabling message is received by the entity. Such a statement can be used to model interruptible and asynchronous activities in the physical system.

3.7.6 The Driver Entity

Every PARSEC program must include an entity called **driver**. This entity serves a purpose similar to the main function of a C program. Execution of a PARSEC program is initiated by executing the first statement in the body of entity **driver**. The **driver** entity

takes the same **argc** and **argv** parameters as the C main function (except that argv must be declared *char** argv* because of PARSEC's requirement that array parameters have a constant size). Parameters recognized by the PARSEC runtime system will be removed from argc and argv before the driver is invoked.

3.7.7 Program Termination

A PARSEC simulation terminates in one of the following ways:

- *The simulation clock exceeds the maximum simulation time specified by* ***setmaxclock()****.*
- *All entities are suspended and no messages (including timeouts) are in transit.*
- *An entity executes an* ***exit()*** *or* ***pc_exit()*** *statement.*

When a termination condition is detected, each entity's (optional) finalize statement will be called. The entity may take appropriate actions before termination, including printing accumulated statistical data.

3.7.8 Clock Operations

In order to allow simulations to be executed over longer durations with fine grained clock values, the PARSEC system clock is implemented as a large integral type called **clocktype**. All clock operations make use of **clocktype** variables. The following functions are provided to manipulate the simulation clock:

- ***simclock(*****void*****)****: This function returns the value of the current simulation clock as a clocktype value.*
- ***setmaxclock(clocktype)****: This function sets the maximum simulation time to the value specified in the numerical-string. The simulation is terminated when the simulation clock exceeds this value.*
- ***atoc(*****char*, clocktype*****):*** *Places the clocktype value represented by the string in the clocktype parameter.*
- ***ctoa(*****clocktype, char*****):*** *Like sprintf, it prints the value of the clocktype parameter into the string parameter.*

Example:

```
clocktype time = (clocktype) 100000000;

setmaxclock (time);    /* set maximum simulation time
                                    to 100,000,000 */
```

The value CLOCKTYPE_MAX is predefined to contain the maximum value of **clocktype**. The default C type for **clocktype** is unsigned long, a 32 bit integer.

3.8 Advanced PARSEC Facilities

3.8.1 Explicit Message Timestamps

By default, the timestamp placed on a message is the (simulation) time of the entity that sends the message. A message may be explicitly timestamped with a future time using the following variation of the send statement:

send *message_type* **to** *destination* **after** *t*

The preceding statement will cause the message *message_type* to carry a timestamp of *clock*+*t*, where *clock* refers to the current simulation time of entity *destination*.

For instance, the body of the job entity in Figure 3-6 could be written as shown in Figure 3-7 where both send statements use the after clauses to store explicit future timestamps on the messages. However, there is one significant difference between using a hold statement followed by a send and directly specifying the timestamp in the send statement: in the former case, the **hold** statement causes the simulation time of the entity to advance; not so in the latter. In particular, the **hold** statement is used to simulate processing time within the entity, while the **send ... after** construct is used to simulate a transmission delay.

Figure 3-7: Job Entity with Transmission delays

```
entity Job(int processing_time, int transmission_delay, ename manager)
{
  for (;;) {
    send Request{self} to manager after transmission_delay;
    receive(Resource res) {
      hold(processing_time);
      send Release to manager after transmission_delay;
    }
  }
}
```

3.8.2 Receive Statement

Several PARSEC facilities are available for specifying more complex resume conditions in receive statements, such as using message parameters in guards, compound resume statements, etc. Most of these facilities are more useful to the parallel programmer rather than the parallel simulator. In a simulation, it is usually appropriate for the message with the lowest time-stamp to be chosen. Nevertheless, some of these features will be described briefly.

3.8.3 Message Parameters

The guard in a resume statement can reference both local variables of the entity and message parameters. For instance, assume that the manager in our running example receives requests for one or more printer units and that incoming requests are serviced using the *first-fit* discipline. The following fragment shows how the resume condition is modified to ensure that a *Request* message is accepted only if the requested number of units is available.

```
message Request {ename requester; int count; };
 .
 .
 .
receive (Request req) when (req.count <= resources)
```

.
.
.

PARSEC also provides a number of pre-defined functions that may be used by an entity to *inspect* its message buffer. For instance, the function **qsize** (*message_type*) returns the number of messages of type *message_type* in the buffer. A special form of this function called **qempty** (*message_type*) is defined, which returns *true* if the buffer does not contain any messages of type *message_type*, and returns *false* otherwise. This function may be used to impose priorities on incoming messages. Assume that the manager is to be modified such that a request message is accepted only if no *Release* messages are present in the buffer. A simple way to do this is to strengthen the guard for the *Request* message as follows:

```
receive (Request req) when (qempty(Release) && (req.count <= resources));
```

Compound Resume

At times, we would like to not process an event until two or more enabling conditions are met. For example, in the Dining Philosophers problem, the Philosopher cannot eat until he acquires two forks. This can be specified in PARSEC using a *compound resume*. The following is a generic example of its use.

```
receive (M_a m_a, M_b m_b, ... , M_n m_n) when (b_a && b_b && b_n) {
  statement;
}
```

However, as of this writing, compound resume conditions have not been implemented in PARSEC.

3.9 Execution of PARSEC Programs

PARSEC programs are executed just as normal C programs are executed. A PARSEC program is a collection of entity and function definitions. Each program must include an entity called ***driver***. This entity plays a role similar to the main function of a C program -- execution of a PARSEC model begins by executing the first statement in the body of the driver entity.

A PARSEC simulation terminates when the simulation clock exceeds the specified simulation horizon (the time period over which the model is to be simulated). The simulation horizon is specified using the predefined *setmaxclock* function which takes a value of type clocktype (or any numeric value, typically).

The PARSEC compiler, called *pcc*, accepts all the options supported by the C compiler, and also supports separate compilation. C programs (files with .c suffix) and object files (files with .o suffix) can also be compiled and linked with PARSEC programs. PARSEC programs are usually given a .pc extension.

Chapter 4

Problem Statement

Here an overview of Efficient Token-Based Algorithm for Distributed Mutual Exclusion [11] proposed by P.K. Dash and R. C. Hansdah.

4.1 Problem Statement

4.1.1 Computation Model

The system model for this algorithm is as follows:

The distributed system consists of n distinct failures free processors (or nodes or sites) Pr_i, $1 \leq i \leq n$, which communicate with each other via message passing over a complete network. The message takes finite but arbitrary time to reach at the receiving sites in FIFO order through the reliable communication links. All the processor clocks are roughly synchronized internally with known upper bound on error as e.

That is $\forall i,\ j\ \varepsilon 1..n$ and $\forall t \geq 0, |C_i(t) - C_j(t)| \leq e$ where $C_i(t)$ is a real value monotonically non decreasing function indicating Pr_i's local clock value at real time t. At each site only one process competes with the processes at other sites to enter CS. Inside CS the process access a shared object having a state that can be changed process by executing inside CS. A Process at a site will be able to enter CS only when it owns the token or privilege to do so. We consider the token having a state associated with it so that not only the token but also its current state has to be transferred to a requesting site as the token state may be changed by a node while executing inside CS. The following messages are used in the algorithm:

- **Request Message** *Req (i, TS)*: This message is initiated by the node Pr_i with timestamp $TS = C_i(t)$ at real time t to enter CS.

- **Token Message** *Tok (token, Q_i)*: When the node Pr_i comes out of the CS, it sends this message with the current state of the token to node corresponding to the first

entry in the request queue Q_i. Also when an ideal node having token receives a request message from another node, it sends this message to the requesting node.

The state of processor Pr_i in the context of the algorithm are as follows:

- **IDLE:** At present, Pr_i is not trying to enter CS.
- **WAITING:** Pr_i has sent its request message to enter CS and is waiting to receive the token.
- **INCS:** Pr_i, after receiving the token, changes its state to INCS and enter CS.

It is assumed that any node has at most one request for CS entry at any time. We assume that the requesting process at each node ahs two sub-processes: one to carry out job inside CS and the other to handle the incoming messages.

4.1.2 ALGORITHM

The algorithm for Pr_i

```
Procedure Initialization;
begin
        State_i := IDLE; next_i := last_i := 1;
        If (i == 1) then
                TokenState_i := Init_Token_State;
end;

Procedure Enter-CS;
begin
        if (next_i == i) then
                begin
                        state_i := INCS;
                end
        else
                begin
                        state_i := WAITING;
                        send Req ( i , clock_i() ) to Pr_last_i ;
```

```
                    receive Tok (token, Q_j) from Pr_j;
                    TokenState_i := token;  Q_i := Merge (Q_i, Q_j);
                    next_i := i; state_i := INCS;
            end;
end;

Procedure Release-CS;
begin
        state_i :=IDLE;
        if (Q_i != ϕ) then
                begin
                    next_i := id of first entry in Q_i;
                    last_i := id of last entry in Q_i;
                    Q_i := Q_i - <first entry in Q_i>
                    Send Tok (TokeState_i, Q_i) to Pr_next_i ;
                    Q_i = ϕ;
                end
        else next_i  := last_i := i;
end;

procedure Process-Request-Msg (Req (j, TS));
begin
        case state_i of
                INCS, WAITING:
                    insert Req (j, TS) in Q_i in sorted order;
                IDLE:
                    if (next_i == i) then
                            begin
                                next_i := last_i := j;
                                send Tok (TokenState_i, Q_i) to Pr_j;
                            end
```

```
                else
                        begin
                                send Req (j, TS) to Pr_last_i ; last_i :=j;
                        end;
        end;
end;
```

4.1.3 Description Of The Algorithm

The following variables are used in the algorithm for Pr_i

- $state_i$: (IDLE, WAITING, INCS);
- Q_i : < Req (i, TS_j)>. The incoming requests to Pr_i are kept in this queue in increasing timestamp order.
- $next_i$:It is the id of the node to which Pr_i had given the token earlier.
- $last_i$: It is the id of the node which had requested earlier for the token and may be having it currently.
- $TokenState_i$: State of the token known to Pr_i.

Informally, the algorithm works as follows: Initially, only Pr_i has the token and next and last variables of all other nodes are set to 1. When Pr_i wants to enter CS, it does so if $next_i=i$ (i.e. it holds the token); otherwise it sends Req(i, $clock_i$()) message to the node Pr_{lasti} and remains in the WAITING state. Upon receiving Tok (token, Q_j) from Pr_j, it updates its $TokenState_i$ to the received token. It also merges the received Q_j with Q_i. After changing $state_i$ to INCS, it enters CS. When Pr_i comes out of CS, it keeps the token with it if Q_i is empty; other, it sends the token to node corresponding to the first entry in Q_i. It also updates the $next_i$ and $last_i$ variables. Then it makes Q_i empty and becomes IDLE. When Pr_i receives Req (j, TS_j) in INCS or WAITING state, it puts the message in its local queue in sorted order as per timestamps (the contention between timestamp is resolved by id comparison). If Pr_i receives request Req (j, TS_j) in the IDLE state, it does the following:

If it has the token, it send the same to Pr_j; otherwise it forwards Req (j, TS_j) to node Pr_{last_i} .The variables $next_i$ and $last_i$ are updated properly. When the token is transferred or the request message from any other node is forwarded to Pr_{last_i} .

The above-described algorithm is not resilient to site and communication failure. Here, this algorithm is enhanced by introducing fault tolerance feature into it. The existing features have not been disturbed.

The algorithm described stop working whenever a node goes down thereby paralyzing the ability of the network to provide mutual exclusion for the critical section. So it becomes very important to introduce the resilient feature to avoid such situation.

4.2 Objective and Sub-tasks

The primary objective of this thesis is to

To analyze, implement and evaluate fault-tolerant algorithm for mutual exclusion in distributed environment to make the network immune of site and communication failures.

The above problem can be broken into following sub-tasks:

1. Studied the various distributed mutual exclusion algorithms for their performance and impact of site and communication failure on their working.
2. Investigate and learn how to use the simulation tool PARSEC exploring its various features and facilities provided by it.

Analyze and implement the efficient fault-tolerant algorithm for distributed mutual exclusion and study the performance and network throughput.

Chapter 5

Implementation of the Fault-Tolerant Token-Based Mutual Exclusion Algorithm

5.1 Modified Algorithm

To implement fault tolerance given in the algorithm given in section 4.1.2, we have done following modifications:

```
Procedure Enter-CS;
begin
    if (next_i == i) then
        begin
            state_i := INCS;
        end;
    else
        begin
            state_i := WAITING;
            send Req ( i , clock_i () ) to Pr_last_i ;
            set timer_i = clock_i () + MAXCLOCK;
            //clock_i () tells the current time node i
            if ( clock_i () == timer_i)
                begin
                    send FaultAlert () to every other Pr;
                    //assumption: If one is raising FaultAlert, none else
                    //can raise FaultAlert till next MAXCLOCK
                    receive FaultReply () from every other Pr_jkl.... except node faulted
                    if (state_jkl..... == WAITING)
                        begin
                            insert element of q_jkl... in q_i in sorted order
                        end;
```

```
                    TokenState_i := Init_Token_State;
                    State_i := INCS;
                    next_i = i;
                end;
            else
                begin
                    receive Tok (token, Q_j) from Pr_j;
                    TokenState_i := token;  Q_i := Merge (Q_i, Q_j);
                    next_i := i; state_i := INCS;
                end;
        end;
    end;

Procedure FaultAlert;
begin
    receive FaultAlert from Pr_j;
    if (State_i == IDLE OR State_i == WAITING)
        begin
            send FaultReply ( State_i, i,  next_i, last_i, Q) to Pr_j;
            next_i = j; last_i =j;
        end;
end;
```

5.2 Correctness Proof

The following lemmas and theorems prove the correctness of the algorithm:

Lemma 1: At any time, if we start from any node Pr_j and traverse along the chain of pointers maintained by the last variables, then we will arrive at a node Pr_k which either has the token or is in the WAITING state.

Proof: The lemma can be proved by considering the initial state and instances when the *last* variable of each node is changed.

Lemma 2: At any time, one of the following two conditions holds in the system:
All nodes are in the idle state except the one which has the token and which is in IDLE or INCS state. Also from any node, if we traverse along the chain of the pointers maintained by the last variable, we will arrive at the node that has the token. If the request of the node Pr_j is en-queued at Pr_k which is in INCS state or in waiting state, there is a sequence of node Pr_{i1}, Pr_{i2}, ……., Pr_{im} where $i_m=j$ and $i_{m-1}=k$ such that the request of Pr_{i_j} , is en-queued at $\mathrm{Pr}_{i_{j-1}}$, $1<i<=m$ and Pr_{i1} holds the token.

Proof: Part (b) of the lemma describes the distributed queue. This lemma can be proved using lemma 1, and by considering the initial state and the instances when the distributed queue is modified.

Theorem 1: The algorithm ensures mutual exclusion.

Proof: At any time during execution of the algorithm, only a single token is either located a node or in transit between two nodes in the network. Therefore, mutual exclusion is always ensured by the protocol.

Theorem 2: The algorithm ensures freedom from deadlocks in the system.
Proof: A set of sites is said to be in deadlock when no site is executing the CS and no requesting site can ever execute the CS. From Lemma 1, it is clear that at any point of time the request of a site will either served by the node in the IDLE state having token or en-queued at a node in INCS or WAITING state. Now assume that there is a queue of request and no more requests are coming. When more nodes enter and leave CS, every node in the WAITING state will enter the queue moving with token as there is a sequence of nodes for every waiting node as stated in part (b) of lemma 2. There, every node will eventually get the token. Hence the algorithm is deadlock-free.

Theorem 3: The algorithm is starvation-free.

Proof: In order to show that the algorithm is starvation-free, we need to show that if a node (say Pr_j) makes a request for entering CS, it will do so within a finite time even if more request arrive repeatedly from other nodes. Assume that there is already a queue of request pending when Pr_j makes a request for entering CS. Then Pr_j*'s* request will get en-queued at a node that is in INCS or WAITING state as per lemma 1. According to lemma 2, there is a sequence of nodes $Pr_i=Pr_{i1,}$ $Pr_{i2,}$ ……$Pr_{im}= Pr_j$, satisfying part (B) of lemma 2. Now, Pr_i has the token. Let $t_2, t_3,$ ....., t_m be the timestamps of the request of $Pr_{i2}, Pr_{i3},$ ……, Pr_{im} respectively. Now only a finite number of requests whose timestamps are less than t_2 can be made as each node takes a finite time to enter and leave CS, and roughly synchronized clocks increase monotonically. Hence, token will definitely come to site Pr_{i2} at least after all requests whose timestamps are less than t_2 are fulfilled. This is because the request of Pr_{i2} will be the first in the queue at the node having token after all request whose timestamps are less than t_2 are served. Similarly, it can be shown that token will definitely come to Pr_{i3}, and so on. Therefore token will definitely come to Pr_{im}, i.e., Pr_j, within a finite time after it makes the request.

Theorem 4: The algorithm is fault-tolerant.

Proof: In order to show the algorithm is fault-tolerant, the clock of the concerned node (say X) will become equal to the value of the timer at that node. Then the node X will broadcast the FaultAlert () message to all the nodes and will wait for FaultReply message. All the nodes, except the faulted one, receiving the FaultAlert () message will send back the FaultReply () message. The node X receiving the FaultReply () messages will enqueue these messages. Now the network will be re-initialized by the node X.

5.3 Simulation Results

The table given below shows the number of messages being passed per entry/ exit critical section:

Processes	Number of Messages when no fault has occurred	Number of Messages when fault has occurred
2	19	43
3	24	48
4	29	53
5	34	58
6	39	63
7	44	68
8	49	73
9	54	78
10	59	83

Table 5-1 Simulation Results

Taking example, for Process 6 (in the table above):

Messages when no fault occurs:

During initialization, Number of messages passed: 10

Number of messages required for driver request: 6

Number of messages to release from CS: 6

Number of messages required by process requesting first: 2

Number of messages required by remaining processes: 3 * 5 = 15

Total Number of messages for Process 6: 39

Messages when fault occurs:

Number of messages required by FaultAlert: 9

Number of messages required by FaultReply: 8

Number of messages required by RESUME: 8

Number of messages required by 6 Processes: 38

Total Number of Messages: 63

Chapter 6
Conclusions and Future Scope

6.1 Conclusions

In Chapter 4, we have stated that the objective of the thesis is to analyze, implement and evaluate fault-tolerant algorithm for mutual exclusion in distributed environment to make the network immune of site and communication failures. To achieve this objective we designed several sub-tasks. We sincerely hope that our work will contribute in providing further research directions in the area of trust-based security. The contribution of the thesis is as follows:

- A detailed study has been conducted about the algorithms for mutual exclusion in distributed environment. The study focused on various performance features of these algorithms like number of messages per entry/ exit critical section, average delay time, deadlock-free and starvation-free capabilities, etc. The algorithm available in literature vary in different ways viz. assumptions made about the network configuration, resilience behavior of the algorithm, performance algorithm under low and high load conditions, design approaches to achiever mutual exclusion etc.
- For implementing the Efficient Token-Based Mutual Exclusion Algorithm, a detailed study of the PARSEC (PARallel Simulation Environment for Complex systems) simulator was conducted. All the features and facilites provided by this simulator were understood in detail. PARSEC is a C-based discrete-event simulation language. It adopts the process interaction approach to discrete-event simulation. In addition, PARSEC provides powerful message receiving constructs that result in shorter and more natural simulation programs.
- A fair amount of simulation of the Efficient Token-Based Mutual Exclusion Algorithm has been carried out using PARSEC simulator. Simulations prove the algorithm to starvation-free

6.2 Future Scope

We have implemented and evaluated the performance of the Fault-Tolerant Efficient Token-Based Algorithm for Mutual Exclusion in Distributed Environment. We also encountered some problems that we feel worth to investigate but were not able to complete to limited time. In this section we illustrate some of the topics that could be explored in future:

- While implementing this algorithm, we have not considered the routing algorithm for routing the messages being passed among various nodes. If we combine the routing algorithm with this proposed algorithm, the throughput of the can be increased.
- Ad hoc networks are the advancement of the distributed computing. The algorithm proposed here could be modified to become compatible with ad hoc environment.

References

[1] Andrew S Tanenbaum and Maarten Van Steen, “Distributed System: Principles and Paradigms”,Pearson Education(sept 2001).

[2] L. Lamport. “The Mutual Exclusion Problem: Part-I –A Theory of Interprocess Comunication”, Journal of the ACM, 33(2): 313–326, 1986.

[3] L. Lamport. “The Mutual Exclusion Problem: Part-II - Statementand Solutions”, Journal of the ACM, 33(2): 327–348, 1986.

[4] Mukesh Singhal, “A taxonomy of Distributed Mutual Exclusion”, Journal of Parallel and Distributed Computing, vol. 18, pp. 94 – 101, 1993.

[5] D.Agarwal and A. El. Abbadi, “ A Token-Based Fault-Tolerant Distributed Mutual Exclusion Algorithm”, Journal of Parallel and Distributed Computing vol. 24, pp. 164-176, 1995.

[6] P. K. Dash, “Design of Resilient Distributed algorithms Using Synchronized Block”, M. E. Project, Dept. of Computer Science and Automation,Indian Institute of Science,Bangalore,India,January 1996.

[7] F.Mueller,“Priority inheritance and ceilings for distributed mutual exclusion”, In IEEE Real-Time Systems Symposium, pages 340–349, Dec. 1999.

[8] Frank Mueller,“Fault tolerance for token-based synchronization protocols”, Workshop on Fault-Tolerant Parallel and Distributed Systems, IEEE, april 2001.

[9] M. Bertier, L. Arantes, and P. Sens, “Hierarchical token based mutual exclusion Algorithms”,In 4th IEEE/ACM CCGrid04, 10 April 2004.

[10] S. Lin, Q. Lian, M. Chen, and Z. Zhang, A Practical Distributed Mutual Exclusion Protocol in Dynamic Peer-to-Peer Systems, Microsoft Research, Technical Report MSR-TR-2004-13.

[11] P. K. Dash and R. C. Hansdah, “An Efficient Token-Based Algorithm for Distributed Mutual Exclusion” ,Indian Institute of Science , Bangalore , India, **http://citeseer.ist.psu.edu/98612.html**

[12] R. Bagrodia, "Parallel Languages for Discrete-Event Simulation Models," IEEE Computational Science and Engineering”, Apr.-June 1998, pp. 27-38.

[13] Rajive Bagrodia, Richard Meyer, Mineo Takai, Yu-an Chen, Xiang Zeng, Jay Martin, Ha Yoon Song, "Parsec: A Parallel Simulation Environment for Complex Systems," Computer, vol. 31, no. 10, pp. 77-85, Oct., 1998.

[14] Vijay K. Garg. Elements of Distributed Computing. John Wiley and Sons Inc., 2002.

[15] PARSEC: **http://pcl.cs.ucla.edu/projects/parsec**

List of Publications

ACCEPTED

1. Vaibhav Saini and R. S. Salaria, "Implementing Mutual Exclusion in Distributed Environment", in Proceedings of National Conference on Computing Sciences and Information Technology, B.B.S.B.E.C, Fatehgarh Sahib, 18-19 March 2006.