Theory of Clock Synchronization and Mutual Exclusion in Networked Control Systems

Technical Report of the ISIS Group
at the University of Notre Dame
ISIS-99-007
August, 1999

Joydeep Ganguly
Department of
Electrical Engineering
University of Notre Dame
Notre Dame, IN 46556
jganguly@nd.edu

Michael D.Lemmon
Department of
Electrical Engineering
University of Notre Dame
Notre Dame, IN 46556
lemmon@maddog.ee.nd.edu

Interdisciplinary Studies of Intelligent Systems

Theory of Clock Synchronization and Mutual Exclusion in Networked Control Systems

Joydeep Ganguly and Michael D. Lemmon *
Department of Electrical Engineering
University of Notre Dame

August 31, 1999

Abstract

Process Synchronization, Mutual Exclusion and Deadlock Freedom are basic issues in Distributed Systems. Several idiosyncrasies of distributed systems, which include uncertain communication delays, make these issues more complicated than they appear. Mutual Exclusion algorithms have formed the basis of a lot of research work in the recent past, and has generated a great deal of interest from both the computer science and the controls community. Much of the groundwork on this subject was done in the late seventies, sparked off by path breaking work by Lamport, Dijkstra et al. Interest was further renewed in this topic in the nineties, wherein great emphasis was laid on the development of optimal mutual exclusion algorithms which exhibited resilience to faults and failures. This article provides an introduction to concepts associated with process synchronization and distributed mutual exclusion, and explores ideas related to various algorithms in the open literature. An extension to mutual exclusion to real-time settings is also considered.

1 Introduction

Synchronization between processes is critical in a distributed environment. The notion of a process is fundamental in the study of synchronization mechanisms. A *process* can essentially be thought of as an instance of a program. Two processes are said to be concurrent if their execution overlaps in time [4] [13] [12]. Concurrent processes in general interact through one of the following mechanisms:

- *Shared Memory paradigm:* Shared memory methods allocate a region in memory that is shared by several processes. The processes access this common shared area by using standard pointer operations.
- *Message Passing Paradigm:* In this method, processes communicate by sending and receiving messages.

Message Passing involves the setting up of a one-way communication channel between the processes, over which the processes send and receive messages. It provides a reliable and simple way for specifying the communication between processes, but at the cost of speed. The message passing paradigm is not a very fast way of inter-process communication, which can have adverse effects in a real-time environment, in which the scheduling of jobs in such a way so as to meet their deadlines is of utmost import.

*We gratefully acknowledge the partial financial support of the Army Research Office (DAAH04-96-10285 and DAAG5-98-1-0199) and the National Science Foundation (NSF-ECS95-31485)

The shared memory paradigm provides a solution to this inherent inadequacy of the message-passing paradigm, by providing a two-way communication channel between the processes. However in this form of communication one has to ensure the integrity of the shared memory, which may be violated if access to the shared memory is not coordinated. Instances of violations include (1) a process reading inconsistent values, (2) the shared memory not recording all changes, (3) the final value in the shared memory not being consistent. This problem in which two or more processes try to read or write into a shared variable at the same time captures the spirit of the *mutual exclusion* problem. This problem is referred to as the mutual exclusion problem, since the objective here is to ensure that processes access shared variables/resources in a mutually exclusive manner.

The remainder of this report is organized thus: In the next section some theoretical aspects of the mutual exclusion problem are stated. The third section talks about some early mechanisms for mutual exclusion. The remaining sections of the report then concentrate entirely on mutual exclusion in a distributed environment, methods to achieve them, current algorithms in existence, and a comparson of their effectiveness.

2 Theoretical Preliminaries

Before delving any further into the existing solutions to the mutual exclusion problem, it is essential to understand the concept of a *critical section*. The critical section (region) is a code segment in which a shared resource is accessed. Codes containing critical sections can be viewed as possessing a structure similar to the one given below [4]:

• **Entry Section:** This is the section of code that requests permission to enter the critical section. Once permission to enter the critical section has been acquired, it is assumed that at most one process can enter the critical section.
• **Critical Section:** This is the code segment that accesses the shared variable, and is the section that has to be executed in a mutually exclusive manner.
• **Exit Section:** This the code segment which releases the critical section for use by another process which may request it.
• **Remainder Section:** This constitutes the remainder of the program.

The Critical Section problem can then be defined as that of executing a set of program's critical section such that we satisfy the following requirements:

• **Mutual Exclusion**: The execution ensures that at any one time only one process can execute its critical section.
• **Deadlock-Freedom**: Deadlock-Freedom is achieved if the following are said to hold true in any given execution:
1. If in some state, a process is executing its *entry section*, and no process is executing its *critical section*, then subsequently the process which is executing its entry section enters the critical section.
2. If in any execution some process is in its *exit section*, then subsequently some process enters the *entry section.*

The problem of deadlock is very common in systems where resources are shared between processors. It occurs when a set of processes in a system is blocked waiting on certain requirements that cannot be satisfied. A simple illustration of a deadlock consists of the scenario where two processes exclusively hold different resources, and each is requesting access to the resource held by the other. In this case the processes are said to be involved in a *circular wait.*

The problem of mutual exclusion also embraces the issue of *starvation.* Starvation occurs when a process waits for a resource that continually becomes available, but due to some reason, is never assigned to that process. There are two main differences between deadlock and starvation [12] :

1. In starvation the resource under contention is constantly in use, by some process or the other, this is not true in the case of a deadlock.

2. In starvation there is no certainty as to whether the process will ever get the requested resource, whereas in deadlock, the requested resource will never become available.

There are a few conditions necessary for deadlocks to occur in a computer system:

- *Exclusive access*: Processes request exclusive access to resources.
- *Wait while hold*: Processes hold previously acquired resources, while waiting for additional/new resources.
- *No Preemption*: A resource cannot be preempted from a process without aborting the process.
- *Circular Wait*: There exists a set of processes which are involved in a circular wait.

For a formal proof that the above necessary conditions hold, refer to the work by Coffman et al. in [3]. Their paper states that a system that satisfies the above conditions is prone to deadlock. There are several deadlock prevention strategies. In them resources are granted to requesting process in a way such that a request for a resource never leads to a deadlock i.e. at least one of the above conditions for deadlock are violated.

One very easy way of achieving this is necessitating a process to possess all the needed resources prior to it beginning, thus violating the *wait while hold* requirement for deadlock. However this would not be a very efficient method of preventing deadlock.

In another method, a blocked process releases all its resources to an active process thus violating the *no preemption* condition for deadlock. An optimal solution of this method was proposed by Rosenkratz et al, wherein processes were given a certain unique priority. A process preempts a process, which holds a needed resource, only if it has a higher priority than the process holding the needed resource. Another solution was proposed by Havender, in what is commonly called as the *resource ordering method*. In this method, all the resources are uniquely ordered and all of the processes request the resources in ascending order. That is, if a process holds some resources, it can only request resources which are ranked higher in the ordering than the resources it already possesses. Thus acquiring resources in this approach precludes the formation of a circular wait.

3 Early Mechanisms for Mutual Exclusion

Some of the earliest solutions presented to the mutual exclusion problem involved what is commonly referred to as the *busy waiting* scheme. In this scheme, a process that is unable to enter the critical section repeatedly tests the value of a status variable to determine whether the shared resource is available. The status variable is nothing but a variable that records the status of the shared resource at every instant of time. Various versions of this scheme appeared in the computer science literature during the seventies. This scheme, though very simple to implement, results in the wastage of CPU cycles and the memory access bandwidth. It is also susceptible to violating the deadlock-freedom objective [4] [12] [13] [6].

Another mechanism that was used in practise concerned itself with disabling and enabling interrupts. In this method a process disables interrupts before entering the critical section and enables the interrupt as soon as it exits the critical section. Mutual Exclusion is achieved in this case, as a process is not interrupted when it is in the critical section. This method, however, is only valid for uniprocessor systems.

To achieve mutual exclusion in multi-processor systems, a special instruction called *test and set instruction* is employed. This instruction bascially can be used as a building block for the busy-waiting scheme. The instruction performs a single indivisible operation on a designated memory location/shared variable. When this instruction is executed, a specified memory location is checked for a particular value. If the value is found to match, then the contents of the shared variable are altered, otherwise they are left unaltered. The testing and setting of this designated shared variable is done in the entry and exit sections of the program. The entry section

first tests to see whether the shared variable is set; if it is not set, then the process sets it, and consequently proceeds into its critical section. If it is set, then the process simply waits until the shared variable is unset. After a process has executed its critical section, it enters its exit section and unsets the shared variable, which can now be tested by another process.

The possible implementation of the above would look something like this [4]:

```
while(lock !=0);
lock=1;
    Access Critical Section
lock =0;
```

Algorithm 3.1

In this segment, the program tests the shared variable **lock**. The while loop checks to see whether the lock is set or unset. If it finds it is unset, it first sets it, before entering its critical secion. The setting of the lock variable lets other processes know that it is currently accessing the critical section. After executing its critical section, it resets the lock, which can now be set by some process attempting to execute its critical section. In [17] an illutstration of the application of this segment to a free-floating robotic vehicle with two articulated arms is presented.

In practice, the *lock* variable introduced above could be implemented as a **semaphore**. A semaphore is nothing but a high-level construct, which carries out the locking operation in many preemptive operating systems, for heavyweight processes. For lightweight processes a similar structure known as **mutex** is used. A semaphore S can be looked upon as an integer variable on which processes can perform two indivisible operations, *X(S)* and *Y(S)*. Each semaphore has a queue associated with it, where processes that are blocked on that semaphore, wait. The two operations X and Y are defined as follows:

X(S): if $S \geq 1$ then S=S-1
  else block the process on the semaphore queue;
Y(S): if some processes are blocked on the
  semaphore S, then unblock a process.
  else S=S+1

Whenever a *Y(S)* operation is performed, a blocked process is picked up for execution. Depending on the values a semaphore can take, they are classified as **binary** (the initial value is 1), or **resource counting** (the initial value is generally more than 1). The use of binary semaphores in an attempt to solve the mutual exclusion problem is presented below:

```
mutex: Semaphore (=1);
Process i (i=1,n);        begin
        .
        .
        X(mutex);
        execute critical section;
        Y(mutex);
        .
        .
        end
```

Algorithm 3.2

In the above, if any process has performed a X(mutex) operation without performing a Y(mutex) operation, it gives the indication to the other processes that it is still in its critical section. Then all the other processes that want to enter the critical section, will wait on the X(mutex) operation till the process performs a Y(mutex) operation. This will signal to the other processes that it has exited the critical section. Hence mutual exclusion is achieved.

4 Mutual Exclusion in Distributed Systems

The notion of mutual exclusion is critical in the design of distributed systems. Certain limitations of distributed systems make the problem of mutual exclusion more complex. This section describes some of the issues that arise while solving the mutual exclusion problem for distributed systems, introduces the concept of Lamport's logical clock's, and lays down performance measures to judge the efficacy of the algorithms currently in the open literature.

4.1 Issues in Distributed Systems

To recap, a distributed system, can be looked upon as a collection of computers separated in space, and not sharing a common memory. Thus, approaches that are based on the premise of a shared memory are not applicable to distributed systems. Absence of shared memory, necessitates the use of message-passing techniques to be used in these systems. Another issue in distributed systems, regards the absence of a *global clock*- there exists no common clock, negating the notion of a global time. Though at first sight the obvious solution to this dilemna would seem to be to provide a clock that is common to all the computers in the system, or to provide each computer with its own clock and then synchronize them. But these solutions are not practically feasible, due to the following reasons.

Take the case where a common clock is provided to the system, from which each computer tries to read time. However, two different processes may read the clock value at different instants due to inherent **message delays**, due to which each process will have its own version of the "correct" time. Therefore, two different processes may perceive one instant in physical time to be two different instants, or vice versa.

The other solution of providing a clock to each computer in the system also does not solve the problem, due to the possibility that each clock may **drift** from its real value. This limitation is further aggravated by the fact that each clock may drift at a different rate, giving rise to further complications.

The absence of a global clock has certain ramifications, in that, it dismisses the idea of having events ordered in time. This makes the mutual excluison problem in distributed systems very difficult. Lamport introduced the idea of a logical clock, in order to achieve ordering of events. It is interesting to note here, that this notion champions the cause of executing events in an orderly fashion, but is not immediately concerned with *when* the event is executed. This is of prime importance in control systems, where controllers of physical plants must react in a timely fashion to external events. This introduces the notion of real-time systems in which when the event is executed is of prime concern.

4.2 Lamport's Logical Clocks and Vector Clocks

Lamport proposed a scheme whereby events in a distributed system are ordered using logical clocks. The order in which two events occur are ascertained by computations, which are based upon the *happened before* relation.

Happened Before Relation ($\rightarrow$). This relation emcompasses the causal dependecies between events. The relation $\rightarrow$ is defined as:
• $a \rightarrow b$, if:
1. a and b are events in the same process, and a occured before b.
2. a is the event of sending a message in a process,P_i and b is the event of receiving the same message in another process P_j.
• This relation is transitive, i.e if $a \rightarrow b$, and $b \rightarrow c$, then this implies that $a \rightarrow c$.

A event a is said to causally effect a event b if $a \rightarrow b$. If neither of the events causally affect each other, then the events are said to be concurrent. Lamport's logical clocks follows straight from the above *happened before relation*. There is a clock C_i, associated with each process P_i in the system, which assigns a number $C_i(a)$ to any event a, called the *timestamp* of any event a, at P_i. These numbers that are assigned have no relation whatsoever with the actual physical time, hence the name logical clocks. Some of the properties of these clocks are:
• The numbers assigned by the clocks, C_i are monotonically increasing.
• Generally, the timestamp of the event is the value of the clock when it occurs.
• The clocks can be implemented by counters.
• They satisfy the following conditions:
1. For any two events a and b in any process P_i, if a occurs before b, then,

$$C_i(a) < C_i(b) \tag{1}$$

2. If a is a event of sending a message in process P_i, and b is the event of receiving a message in process P_j, then

$$C_i(a) < C_j(b) \tag{2}$$

The last two conditions are satisfied by the following implementations:
• If a and b are two successive events in the same process, then

$$C_i(b) := C_i(a) + d \tag{3}$$

• If event a is the sending of a message q by process P_i, then it is assigned a time-stamp $t_q = C_i(a)$. On receiving q by Process P_j, C_j is set to a value that is greater than the time-stamp t_q, and a value greater than or equal to its present value.

$$C_j := max(C_j, t_m + d) \qquad d > 0 \tag{4}$$

Usually d, has the value of 1, in most of the existing algorithms, though we cannot apply this same logic to real time systems, where d would be a representation of the delay in the message transmission.

Equations (1),(2),(3),(4) describe a partial ordering of events as suggested by Lamport. A total order of events (the ordering relation denoted by $\Rightarrow$), using the above system of clocks, is also introduced:

Given that a is an event at process P_i, and b is an event at process P_j, then $a \Rightarrow b$ if and only if,

$$C_i(a) < C_j(b) \qquad or \tag{5}$$

$$C_i(a) = C_j(b) \quad and \quad P_i \prec P_j \tag{6}$$

Here $\prec$ is an arbitrary relation that totally orders the processes to break ties. There are a few points which one must note here. Lamport's logical clocks give us an approximation to physical time, known as *virtual time*.

Virtual time progresses with the progression of events. Therefore, it is discrete. If no events occur, virtual time stops. This is not the case with physical time and hence this concept can be difficult to support in the context of real-time operating systems. Lamport's logical clock's suffer from one more drawback. We can't get information about the causal relationship between events in different processes, by merely observing the timestamps. This is because each clock can advance independently due to some local occurence of events, and Lamport's clock system cannot distinguish between advancements due to local events and that due to message passing between processes.

In an endeavor to determine whether two events are causally related by just observing their timestamps, the notion of **vector clocks** was introduced. They are another approximation of virtual clocks, where each process keeps its own local clocks, these being based on the knowledge of clocks of all the other processes in the system. When a message is sent from process P_i, it carries with it $C_i(i)$. The implementation rules for vector clocks are given as follows:

• As in the case with Lamport's logical clocks, for two events in the same process:

$$C_i(i) := C_i(i) + d \qquad (7)$$

• If event a is the sending of a message q by process P_i, then it is assigned a time-stamp $t_q = C_i(a)$. On receiving q by Process P_j, C_j is set to a value that is greater than the time-stamp t_q, and a value greater than or equal to its present value.

$$C_j[k] := max(C_j[k], t_m[k]) \qquad d > 0 \qquad (8)$$

Thus, upon receiving a message, all the other clocks are updated to be the highest value known to the sender and receiver. In this manner, vector clocks provide us with a way to order events and determine the causal relationship between two events, by merely observing their timestamps.

4.3 Causal Ordering of Messages

The concept of vector clocks aids us greatly in the causal ordering of events. But what would really be useful in achieving mutual exclusion would be to obtain a causal ordering of *messages*. This is something that is not automatically guaranteed in a distributed system. Figure [1] shows us how the causal ordering of messages can be violated in a distributed system. Even though it is evident *send(*msg_1*)* $\rightarrow$ *send(*msg_2*)*, msg_2 is delivered before msg_1. The numbers in the circle represent the correct causal order which should have been obeyed in the delivery of messages.

The causal ordering of messages aims at maintaining the same causal relationship that holds among the "send message" and "receive message" events. That is to say if *send(*msg_1*)* $\rightarrow$ *send(*msg_2*)*, (where *send(*msg_1*)* and *send(*msg_2*)* are the events of sending messages msg_1 and msg_2), then every process that receives both msg_1 and msg_2, receives msg_1 before msg_2. Schemes that achieve the causal ordering of messages are very useful, and simplify the algorithms for distributed mutual exclusion to a great extent. Two protocols which achieve this objective, exist in the open literature- the *Birman-Schiper-Stephenson* protocol, and the *Schiper-Eggli-Sandoz* protocol. The essence of both protocols is the same, with the main difference between the two being that the Birman-Schiper-Stephenson protocol relies on the ability of processes to communicate using broadcast messages, while the second protocol does not require that the processes only communicate through broadcast messages. The main idea in both protocols is to deliver a message only if the message preceding it has been delivered to the process. In the case that the message preceding it has not been delivered, the message is not delivered immediately, but it is buffered until the message immediately preceding it has been delivered. Both protocols necessitate the existence of a vector accompanying each message that carries all the necessary information for a process to decide whether there is a message preceding it, or it is free to deliver its message. The first protocol is described in detail below:

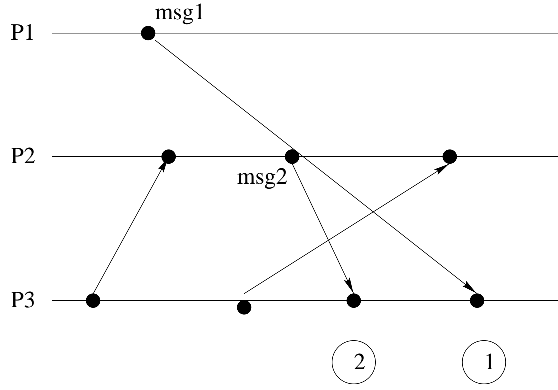

Figure 1: Violation of Causal Ordering of messages

4.3.1 Birman-Schiper-Stephenson Protocol

Prior to broadcasting a message, msg_1, a process P_i increments a vector time VT_{P_i}[i], and timestamps the message msg_1. This vector is critical to the functioning of the algorithm since it carries all the relevant information regarding the number of messages preceding a particular message. This means that (VT_{P_i}[i]-1) indicates how many messages from P_i precede msg_1. Any process P_j which receives the timestamped message msg_1, will delay its delivery until the following conditions are satisfied:

- VT_{P_j}[i]=VT_{msg_1}[i]-1, where VT_{msg_1} represents the timestamp of msg_1.
- VT_{P_j}[k] $\geq$ VT_{msg_1}[k] $\forall$ $k \in$ 1,2,..., n - i. Here n represents the number of processes in the system.

The delayed messages are queued *at each process* in a queue that is sorted according to the vector time of the messages. When a message arrives at process P_j, VT_{P_j} is updated according to the rules governing vector clocks stated in equation [8].

The critical step here are the two bulleted items. The first bullet ensures that process P_j has received all the messages from P_i that precede msg_1. The second bulleted item ensures that process P_j has received all the messages received by P_i prior to the time when it sent message msg_1. Using these two conditions, causal ordering of messages is achieved.

4.4 Performance Specifications of Distributed Mutual Exclusion Algorithms

Before studying the current distributed mutual exclusion algorithms in existence, we need to specify certain measures of performance, by which we gauge the efficacy of each algorithm. There are four universally accepted metrics, by which we evaluate the performance of each algorithm:

- **Number of messages**: Sent, received or both. This metric is with respect to the number of times the critical section is accessed.
- **Synchronization Delay**: This refers to the time it takes for a process to enter its critical section after a process has exited its critical section.
- **Response Time**: This is the time interval a request waits for its critical section to be over, after its request messages have been sent out.
- **System Throughput**: This is the rate at which system executes requests for the critical section. System throughput is given by the following equation:

$$SystemThroughput = \frac{1}{2T + E} \tag{9}$$

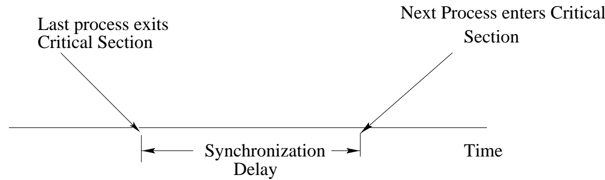

Figure 2: Synchronization Delay

where, *2T* is representative of the synchronization delay, and E is the average critical section time.

These metrics give us a good index to judge the performance of a distributed mutual exclusion algorithm. Each algorithm generally has a best case performance, and a worst case performance. The best performance of an algorithm generally is achieved under low load conditions, and the worst case achieved under high load conditions. The average of the two case performances gives us the average behavior of the algorithm.

5 Mutual Exclusion Algorithms

Distributed mutual exclusion algorithms have to guarantee that at one time only one process accesses the critical section. To formally state the problem, a distributed mutual exclusion problem needs to satisfy the following requirements:

• **Deadlock Freedom**: A process should not endlessly wait for a message that will never arrive.
• **Starvation Freedom**: A process should not wait indefinitely to execute its critical section, while other processes are executing their critical sections.
• **Fairness**: This requirement ensures that requests for the critical section must be executed, in the order they were received. While achieving this objective, the concept of *logical clocks* are used to define an *order* of execution. If an algorithm is *fair*, it is also *starvation free*, but the reverse is not true.
• **Fault Tolerance**: The algorithm has to be fault-tolerant, in that in the event of a failure, it recognizes that a fault has occurred and it continues to function in an orderly fashion.

The distributed mutual exclusion algorithms in the open literature are broadly classified into two categories: [1] Token-based algorithms [2] Non-token based algorithms. Examples of both these types of algorithms are given in the following sections. In all the algorithms the tacit assumption is that clocks run at a perfect rate, they do not drift relative to each other, and once clocks are synchronized, they remain synchronized.

5.1 Token-based Algorithms

The concept in this class of algorithms is very simple. A unique token circulates amongst the processes. A process can only enter the critical section if it has the token. If it wishes to enter the critical section, a process makes a request for the token. The token can be thought of as a *key* which opens the lock, which in this case is the critical section. This is the basic essence of all the token-based algorithms. The differences between them lie in the way the processes search for the token. A point worth noting here is that these algorithms do not use

timestamps, but a sequence of numbers, which advances independently every time a process makes a request for the critical section.

5.1.1 Suzuki-Kasami's Broadcast Algorithm

In this algorithm, any process that does not possess the token, but wishes to enter its critical section, sends a request out to all the other processes in the system. After broadcasting the request it waits for action by the process currently in possession of the token. If the process in possession of the token is *not* executing its critical section, then it gives the token up to the process requesting for it. If it is in the process of executing its critical section, then it gives the token up after it exits the critical section.

There are a few issues that come straight to mind as regards the functionality of this algorithm. Firstly, consider the case where a process P_1 is executing its critical section, and another process P_2 makes a request for the token. In the interim that it makes a request, and P_1 exits its critical section, say another process P_3 also makes a request for the token. How does P_1 recognize which process, P_2 or P_3 does it give the token to? That is to say, that there must be a way to keep track of all the requests for the critical section made so far, so as to give the token to the process which made the request first. Also there must be a method to keep track of *outdated* messages from *current* ones. Secondly, what if there is a conflict, when two or more processes request the use of the token simultaneously?

The solution to the first part is relatively simple. To distinguish between outdated and current messages, and to determine which process requested use of the token the earliest, the following is done: A REQUEST message from process P_j has the form REQUEST(j,n), where n(n=1,2,3......) is a sequence number that indicates that process P_j is making its $n^(th)$ request for the critical section. So process P_i keeps an array of integers RN_i[1...N], where RN_i[j] is the largest sequence number received so far in a REQUEST message from process P_j. A message is considered to be outdated if the following hold:

$$RN_i[j] > n \tag{10}$$

This follows form the fact that as soon as process P_i receives a REQUEST message from process P_j, it sets RN_i[j]:=max(RN_i[j],n). Thus the problem of determining whether or not the request is outdated or current is resolved. Now to determine which process should the token go to, after the process with the token has exited the critical section. The solution to this involves keeping a queue, with all the requests for the token. After executing its critical section, the process checks the queue to see if any other process has requested to enter the critical section, and gives the token to the process at the head of the queue. The contents of the queue are then decremented by 1.

This algorithm however does not take the possibility that two processes may request the token at exactly the same instant into account, due to its low probability.

5.1.2 Raymond's Tree Based Algorithm

In this algorithm, processes are logically arranged as a directed tree, such that edges of the tree are assigned directions towards the process that has the token. Each process in this algorithm has a local variable, *holder* and a FIFO queue, called *request-q*. The *holder* variables at processes define a logical tree structure among the processes, by pointing to an immediate neighboring node on a directed path to the process that has the token. An example of such a tree configuration is shown in [3]. The *request-q* stores the requests of the neighboring processes that have made requests for the token, but have not as yet acquired it. The working of this algorithm is best illustrated with the aid of an example.

If a process, say P_6 wants to enter its critical section, it sends a REQUEST message to the node along the directed path to that node that has the token (the *root* node), provided that the *request-q* of the neighboring

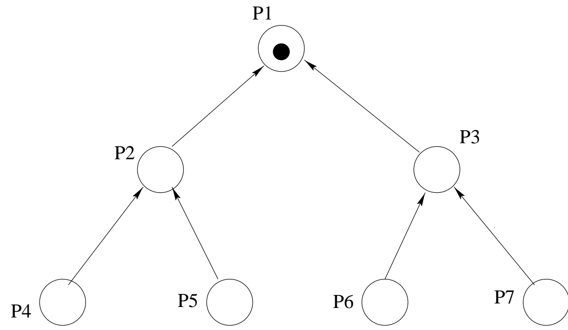

Figure 3: Tree Configuration

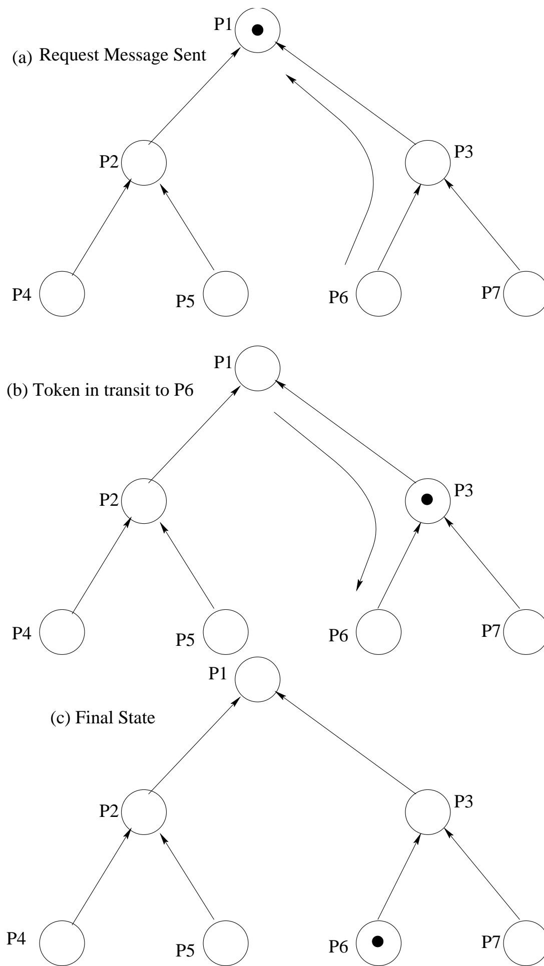

Figure 4: Illustrative Example

node, which in this case happens to be P_3, is empty. This follows from the fact that a non-empty *request-q* indicates that the process has sent a REQUEST to the root node, for the token, and hence wishes to be placed at the head of the root nodes' $request - q$. When the process P_6 sends the REQUEST, it adds its own request to its *request-q*.

When a node on the path receives the message, it places this REQUEST in its *request-q*, and sends a message along the directed path to the root node. If some other process , say P_7had sent a REQUEST earlier, then P_3 would have had P_7*'s* REQUEST in its *request-q*, and would not send P_6*'s* REQUEST along the directed path. When the root process receives a REQUEST message , provided it has finished executing its critical section, it sends the token to the process from which it received the token,i.e P_3, and sets its *holder* variable to point to that process. When a process receives the token, it deletes the top entry from its *request-q*, and sends the token to that entry which it just deleted. In our example, this would mean that P_3 would delete P_6*'s* entry and send the token to P_6 When then token finally reaches the process which had initially requested it, it deletes its own entry from its *request-q* and keeps the token, and enters the critical section.

After the process P_6 has finished executing its critical section, it can take the following actions based on the contents of its *request-q*. If its *request-q* is not empty, then it deletes the top entry from its *request-q* and gives the token up to the process whose entry it just deleted, and sets its *holder* variable to point to that process. If its *request-q* is empty, then it sends a REQUEST message to the site which is pointed to by its *holder* variable. This is a very efficient algorithm which is free from deadlocks and starvation.

5.1.3 Singhal's Heuristic Algorithm

This algorithm, attempts to try and optimize performance with respect to the number of messages sent and received in an attempt to achieve mutual exclusion. In this, each process maintains information about the state of other processes in the system, and uses this information to try and select a subset of processes that are most likely to have the token, or at least will possess the token in the near future. Then after selecting the processes, it sends a request message only to these processes. As processes are heuristically chosen for sending request messages, this algorithm is called heuristic.

In this algorithm each process, P_i maintains two distinct arrays, PS_i[1...N] and PN_i[1...N], which stores information about the state and the highest known sequence number for each process. The tokens too maintain two similar arrays, *TS*[1...N] and *TN*[1...N]. According to this algorithm, a process can be in any of the following states:

- *R* - Requesting the critical section.
- *E* - Executing the critical section.
- *H* - Holding the idle token.
- *N* - None of the above.

If a process, P_i, wishes to possess the token, it does the following:

- It first sets PS_i[i] to *R*, and increments PN_i[i] := PN_i[i]+1.
- It then sends a request message, REQUEST(i,*pn*), to all other processes P_j for which PS_i[j]= *R*. Here *pn* is the updated value of the array PN_i[i].

When a process P_j receives the request, with the process id. and the sequence number, it first checks to see if that request is outdated or not i.e it checks to see if PN_j[i] $\geq$ *pn*. It it finds out, that the relation holds, it realizes this is an outdated message and discards it. If not it sets PN_j[i] to *pn*, and then based on its current state, does the following:

- If PS_j[j]=*N*, then set PS_j[i] := *R*.
- if PS_j[j]=*R*, the set PS_j[i] := *R* and send a REQUEST(j, PN_j[j]) message to P_i to let it know its status.

• if PS_j[j]=E, then set PS_j[i] := R.
• if PS_j[j]=H, then set PS_j[i] := R, PS_j[j] := N, TS[i] := R, and TS[i]=pn. It then sends the token to process P_i

As soon as process P_i, acquires the token then it is free to execute its critical section. Prior to entering its critical section, it sets PS_i[i] := E. After it finishes executing its critical section, it sets PS_i[i] := N and TS_i[i] := N. Thus, this algorithm achieves mutual exclusion, and exhibits optimality in terms of the number of messages sent.

5.1.4 Comparison between the Algorithms

We now evaluate the performance of the above token-based algorithms using the metrics proposed in section 4.4. As far as the response time is concerned, both the Suzuki-Kasami algorithm and Singhal's heuristic algorithm have the same performance, in that the response time is equal to $2T+E$, where $2T$ represents the total round-trip time taken for the messages to be passed and E is the time it takes to execute the critical section. This is not the case with Raymond's algorithm, where the response time is $T(\log N)+E$. It must be noted that as the load on the system increases, wherein more processes vie for the critical section, the response time increases. A closed form analytical expression for the response time as a function of load does not exist for the algorithms. But it is evident, that in the case where there is heavy load, or even non-uniform load (this refers to the condition when the rate of critical section requests are not uniformly distributed over the processes), Singhal's heuristic algorithm potentially has the ability to significantly reduce message traffic.

As far as the message traffic goes, the Suzuki-Kasami algorithm requires N messages per critical section execution, for a N process system. Singhal's heuristic algorithm requires just $N/2$ messages. Raymond's algorithm requires $\log N$ messages (this comes about from the fact that the distance between two nodes in a typical tree with N nodes is $(\log N)/2$).

5.2 Non-Token Based Algorithms

The fundamental principle in all non-token based algorithms lie in processes communicating with each other to determine which one of them should execute the critical section next. This class of distributed mutual exclusion algorithms use timestamps to order requests for the critical section. It is also used to resolve conflicts when two processes simultaneously request the critical section. In this section, a few non-token based algorithms are presented which are representative of most of the algorithms of this class. In all the algorithms presented, processes maintain logical clocks in accordance with Lamport's scheme presented earlier in this report. Every request for the critical section gets a timestamp, with the priority lying with processes which have smaller timestamps.

5.2.1 Lamport's Algorithm

Lamport was the first to present a distributed mutual exclusion algorithm, in response to the naive centralized approach that existed till that time. To summarize the centralized solution, it consisted of a *control site*, which received requests for the critical section from all the processes. It queued all the requests, and granted the processes permission one by one. Lamport identified some obvious drawbacks of this centralized solution to the distributed mutual exclusion problem. First, there was the possibility that the *control site* might fail, and that would mean failure of the entire system. Secondly, this put a lot of pressure on one "supervisor" to coordinate the entry of all processes into the critical section and the release of the critical section by the processes. There was also the case when many processes request the critical section at one time, which would lead to congestion

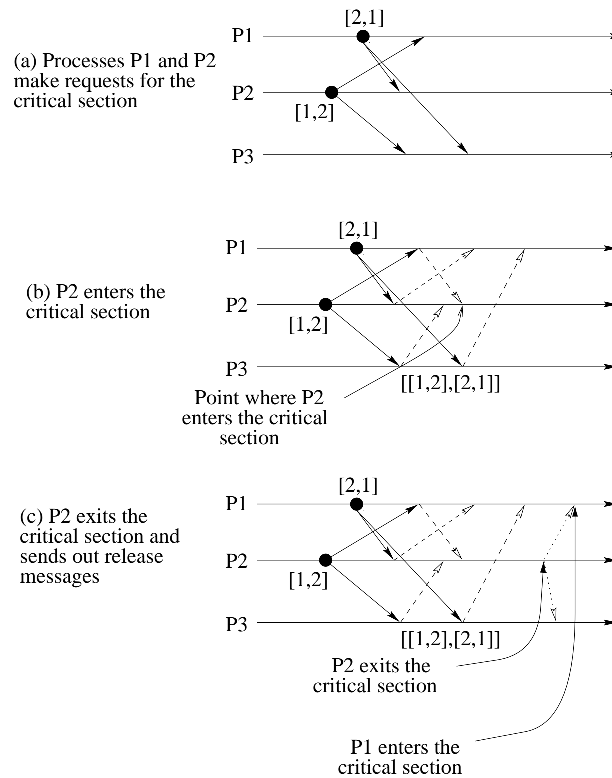

Figure 5: Lamport's Algorithm

at the communication links near the *control site*. Many researchers further came up with more drawbacks of this centralized approach, such as this approach did not support the existence of a *fast* algorithm.

Lamport's algorithm is a very elegant algorithm which has formed the basis of most of the other algorithms in the open literature. Part of its attraction lies in its simplicity. When a process P_i wishes to enter its critical section, it sends a REQUEST(ts,i) message to all the other processes in the system (ts is its timestamp). It then places its own request in its *request-q*. This can be seen in the example illustrated in figure 5. Here two processes P_1 and P_2 make requests for the critical section, and send messages to the other two processes with their timestamps. At the same time, they enter this request in their respective queues (Refer to figure 5(a)).

When a process P_j receives such a request, it returns an ANSWER message with its own timestamp, and places the process' request in its own *request-q*. In our example this would translate to processes P_1, P_2 and P_3 sending their timestamps to P_1 and P_2. This can be seen in fig [5(b)]. According to Lamport's algorithm, a process P_i can only enter its critical section if two conditions are satisfied:

[C1] P_i has received an ANSWER message from all the other processes in the system with a larger timestamp than its own.
[C2] P_i's request is at the top of its own *request-q*. The second condition basically is self-evident. It just restates the fact that if P_i received a REQUEST message from some other process, P_j before it sent out its own REQUEST message, it cannot possible have a lower timestamp than process P_j. As seen in figure 5(b), P_2 receives clearance from the other two processes that its timestamp is lesser than both P_1's and P_3's.

Once the process P_i finishes executing its critical section, then it exits the critical section, and sends out timestamped RELEASE messages to the other processes in the system. Along with this, it also removes its own request from the top of its *request-q*. When a process P_j receives the timestamped RELEASE message from P_i, it removes P_i's entry from the top its *request-q*, and if its own request is on the top of its *request-q* and it satisfies condition **[C1]** of Lamport's algorithm, then it can enter its critical section. This is illustrated

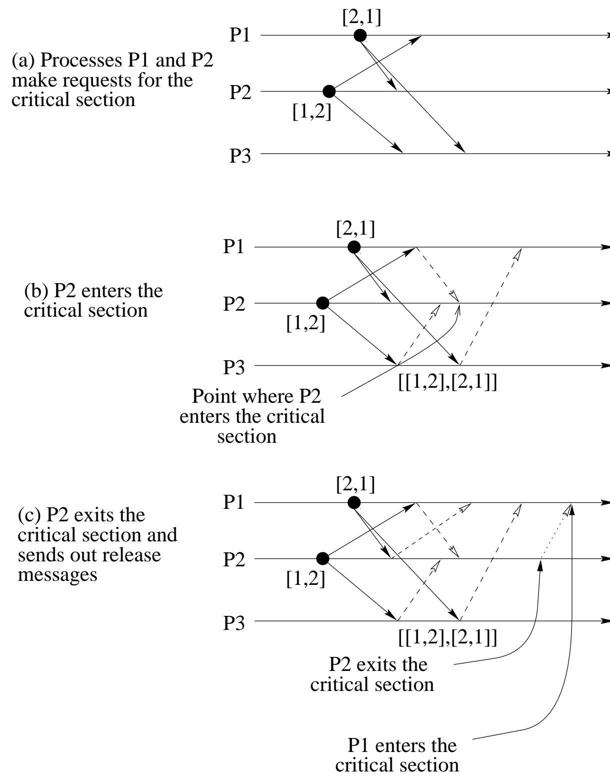

Figure 6: Ricart-Agrawala's Algorithm

in fig[5(c)], where after getting process P_2's timestamped RELEASE message, process P_1 enters the critical section.

As far as performance goes, Lamport's algorithm requires 3(N-1) messages, everytime a critical section is executed. This follows from the fact that (N-1) messages are required for the request of the critical section, (N-1) messages for the answer, and (N-1) messages for the release. Certain methods of optimizing this algorithm are possible. In order to reduce the number of messages involved in this algorithm, it is possible to suppress certain ANSWER messages. For instance if a process P_j receives a request from another process P_i, after it has sent out a REQUEST with a higher timestamp than P_i, it need not send an ANSWER message to P_i confirming that P_i has a lower timestamp than it. This is because, in any account, P_i upon receiving P_j's REQUEST will realize it has a lower timestamp than P_j, and assume that it is free to enter the critical section. This is just one of the optimizations suggested on the Lamport algorithm. Lamport's algorithm pioneered a new approach towards distributed mutual exclusion, and a lot of the current algorithms are modifications/optimizations of Lamport's distributed mutual exclusion algorithm. The next algorithm is an example of one such optimization of Lamport's algorithm.

5.2.2 Ricart-Agrawala Algorithm

The Ricart-Agrawala algorithm optimizes Lamport's algorithm by merging the RELEASE messages with the ANSWER messages. As in the Lamport algorithm when a process P_i wants to enter the critical section, it will send a timestamped request message to all the processes in the system. When a process P_j receives the message, it will send an ANSWER message to process P_i only if process P_j is not requesting or executing the critical section, or if process P_j is requesting the critical section and P_i's timestamp happens to be less than process P_j's timestamp. If not the request message is deferred. Once process P_i has acquired ANSWER messages from all the processes and confirms that it has the lowest timestamp, it then proceeds to enter the critical section.

When process P_i is done executing the critical section, it will only send an ANSWER message to those processes that have deferred requests. Thus the RELEASE and ANSWER messages are merged into one message. Process P_i will send an ANSWER message to the process which has a deferred request with the lowest timestamp. The working of this algorithm is illustrated by an example, shown in fig[6]. The main difference between this and the example illustrated in fig[5] is seen when process P_2 exits the critical section (see fig[6(c)]). Here process P_2 only sends an ANSWER message to process P_1 which has a deferred request in P_2's *request-q*.

This algorithm reduces the number of messages involved in a critical section execution from (3N-1) to (2N-1).

5.2.3 Proof of Correctness

In this section, a formal proof of Lamport's algorithm is given. The proof's of most of the algorithms follow in a similar manner. We prove that Lamport's algorithm achieves mutual exclusion, by contradiction. Assume two processes P_i and P_j are executing the critical section concurrently. This automatically implies that both the processes have satisfied both the conditions imposed by Lamport's algorithm to enter the critical section. that is to say, that both P_i and P_j have their own requests at the top of their *request-q*'s and the first condition, **[C1]** holds for both the processes. Without a loss of generality, we can assume that process P_i has a lower timestamp than process P_j. Due to **[C1]**, we know that the request of process P_i must be present in the *request-q* of process P_j, during the execution of its critical section. This further implies that if process P_j has to execute the critical section with process P_i's request in its *request-q*, it has to have a smaller timestamp than process P_i - which is a contradiction. Hence, we have proved that Lamport's algorithm satisfies the mutual exclusion property.

The Ricart-Agrawala algorithm is proved in a similar way, since it is nothing but an optimization of Lamport's algorithm.The proof is again by contradiction. Assume that two processes P_i and P_j are executing the critical section simultaneously, and process P_i's request has a higher priority than the request of process P_j. That is, it has a smaller timestamp. Thus it is clear that process P_i received P_j's request after it made its own request. Otherwise P_i's request would have had lower priority. Therefore, P_j can concurrently execute the critical section with P_i, only if P_i returns an ANSWER message to P_j, in response to P_j's request, before P_i exits the critical section. However this is impossible since P_j's request has lower priority.

6 Generalized Non-Token Based Algorithm

In [18] Sanders presented a generalized non-token based algorithm for mutual exclusion. All the existing non-token based algorithms are special cases of the this algorithm. To unify all the different non-token based algorithms, Sanders developed the concept of *information structures*, which forms the basis of his proposal.

6.1 Information Structures

The information structure of any mutual exclusion algorithm defines the data structure required at a process to record the status of other processes. The process uses the information in the information structure to make decisions (such as, from which processes to request permission etc.) Basically, the information structure at a process, P_i consists of three sets:

- A *request set* R_i
- An *inform set* I_i
- A *status set* S_i

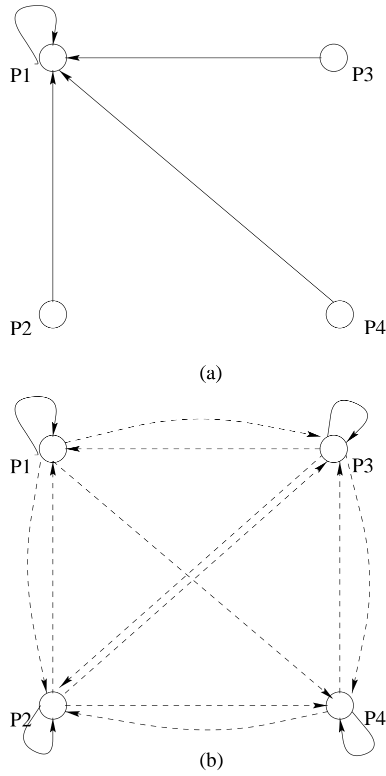

Figure 7: Information Structures

These sets consist of all the id's of the processes of the system. A process, prior to entering its critical section has to acquire permission from each of the processes in the request set. Also, each process must inform all the processes in its inform set about any status change that occurs, such as entering the critical section or exiting the critical section. The status set simply contains the id's of the processes for which a process maintains status information. It is worth noting here that the information set and the status set are dependent on each other. If $P_i \in I_j \Rightarrow P_j \in S_i$.

A process also maintains a queue which contains REQUEST messages in order of their timestamps, for all requested made for the critical section so far, for which no GRANT message has been sent out so far. It also maintains a variable *CSstatus* which represents the process' knowledge of the status of the critical section. Sanders proposed in [19] that to guarantee mutual exclusion, the information structure of processes in any algorithm must satisfy the conditions given by the following theorem:

Theorem: If $\forall\ i:\ 1 \leq i \leq N :: P_i \in I_i$, then the following conditions are necessary and sufficient to guarantee mutual exclusion:

S1: $\forall\ i:\ 1 \leq i \leq N :: I_i \subset R_i$
S2: $\forall\ i,j:\ 1 \leq i,j \leq N :: (I_i \cap I_j \neq \emptyset) \vee (S_i \in R_j \wedge S_j \in R_i)$

Here both $\wedge$ and $\vee$ have their usual meaning of conjunction and disjunction respectively. **S2** simply states that for every pair of processes, either they request permission from each other, or they request permission from a common process, that contains status information of both the processes.

6.2 Generalized Algorithm

In the generalized algorithm it is assumed that each request for the critical section is assigned a timestamp that is maintained according to Lamport's scheme. If a process wants to execute the critical section, it sends timestamped REQUEST messages to all the processes in the request set. When a process receives a REQUEST message, it places the request in its queue in order of the timestamps (from lowest to highest). If *CSstatus* indicates that the critical section is free, then it sends a GRANT message to the site at the top of its queue, and then deletes its entry. A process basically executes the critical section only after it receives a GRANT message from all the processes in its request set. Whenever a process sends out a GRANT message to another process, it sets the status set S_i to indicate which process is entering the critical section. When a process finishes executing the critical section and is about to exit it, it sends a RELEASE message t every process in its inform set.

When a process receives a RELEASE message, it sets the status set to *free.* it then checks to see if its queue is non-empty. If it is, it sends a GRANT message to the entry at the top of its list, and the process is repeated. Most of the algorithms are variations on this generalized algorithm.

Figure 7 shows two mutual exclusion algorithms in terms of their information structures. A solid arrow from a process P_i to a process P_j indicates that process P_j is in process P_i's inform and request set i.e $P_j \in R_i$ and $P_j \in I_i$. A dashed arrow from a process P_i to a process P_j indicates that process P_j is in process P_i's request set, but not in process P_i's inform set i.e $P_j \in R_i$ and $P_j \notin I_i$.

Figure 7(a) shows the information structure of a mutual exclusion algorithm in which there e is a central process, so to speak, that controls the entry of all the processes into the critical section. In this example the process here performing this supervisory task is process P_1. As can be seen in the figure, each process request permission from this process before entering the critical section. This is reflective of most of the centralized algorithms in the open literature. Figure 7(b) shows the information structure of a mutual exclusion algorithm wherein each process requests permission from every other process in the system prior to entering the critical section. In addition to requesting permission from other processes, each process keeps track of its own request made. This is indicated by the arrow starting and terminating at its own node. An example of such an algorithm would be the Ricart-Agrawala algorithm.

7 Recent Work

Much of the recent work in this area has been directed towards finding more "optimal" algorithms. Optimality in the sense of trying to minimize some performance metric, for example the number of messages passed, or trying to make the algorithm fault tolerant [9] [11] [14], or making the algorithm fast [8] [3]. This section is intended to give an overview of some of the recent developments, which have used the underlying principles stated earlier in this paper as their basis.

There is a lot of literature which talks about the functioning of algorithms in the presence of faults. Welch and Dolev describe their algorithms on fault-tolerant clock synchronization, inspired by the architecture and failure patterns of shared-memory multiprocessors [2]. Fault-tolerant algorithms must possess the property of tolerance to any number of process failures, and for the non-faulty processors' clocks to be unaffected by the failures. It is equally important for the processors that have ceased being faulty to be able to rejoin the system and become synchronized. Dolev and Welch describe the requirements of a fault-tolerant algorithm as, an algorithm that guarantees that, for some fixed k, once a processor P has been working correctly for at least k pulses, then as long as P continues to work correctly:

1. P does not adjust its own clock, and,
2. P's clock agrees with the clock of every other processor that has also been working correctly for at least k pulses.

They consider two types of faults -

1. *Napping faults*: A Napping failure causes a processor to stop operation and then resume (with or) without recognizing that a failure has occurred.
2. *Transient faults*: A Transient fault is a fault that causes the state of a process to change arbitrarily. By state of the process we refer to its local state, program counter and shared variables.

The first type of fault captures the spirit of *wait-freedom.* A wait-free algorithm ensures that the working processor must synchronize in a fixed amount of time regardless of the actions of the other processors.

The second type of fault encompasses the concept of *self-stabilization.* An algorithm is self-stabilizing if it is resilient to any number and any type of faults. Basically it refers to the fact that, starting with an arbitrary state of the system, a self-stabilizing algorithm eventually reaches a point after which it correctly performs its task. This is conditioned upon the fact that no further fault occurs. Their paper was motivated by the attempt to prove the existence of an algorithm that can tolerate any number of faulty processors and work correctly when started in an arbitrary system state.

The practical appeal of self-stabilizing protocols is that they are simpler, and they are more robust (they can recover from transient faults such as memory corruption as well as common faults such as link crashes). From a theoretical point of view, a self-stabilizing protocol is considered "cleaner", since the need to specify an initial state is eliminated[14].

Prior to the work of Dolev and Welch, there had been algorithms that were self-stabilizing [20], and there were algorithms that were wait-free, but none of them were self-stabilizing and wait-free. A fault model had been presented by Lamport and Melliar-Smith [21], which was strong, and stated that no algorithm can work unless more than one-third of the processors are non-faulty. A weaker model was suggested called authenticated Byzantine, which allowed an algorithm to tolerate any number of faulty processors. However in that algorithm faulty processors could influence the clocks of non-faulty processors, by speeding them up. Lamport and Melliar-Smith had done extensive work involving Byzantine faults, and provided self-stabilizing algorithms [21].

In [2] the self-stabilizing protocol copes with both transient faults and napping faults. Algorithms are presented which use both bounded and unbounded clock values. A non-faulty processor P synchronizes its clock even in the case in which all *n-1* other processors are faulty. In other words, P does not wait for the other processors to participate in the protocol (thus capturing the essence of wait-freedom). P chooses the maximal clock value in the system, designated by clk, and assigns $clk + 1$ to its own clock. The other processors then search for the maximal value of clock in the system, which happens to be the clock value of P, and then set their clock to the value of P's clock- hence achieving synchronization.

Boaz Patt-Shamir extensively researched into the issues of self-stabilization. In the self-stabilization model presented by Patt-Shamir, he defines faults ot be two-fold in nature: catastrophic faults and anticipated faults. Catastrophic faults encompass those faults that cause arbitrary corruption of global state, and anticipated faults are those which are known to occur. [14] and [15] discusses the design of general algorithmic transformers that take a protocol as input and produce as their output, a self-stabilizing protocol. The transformers exhibit trade-offs between their generality (i.e. the range of input protocols that they can transform) and the efficiency of the resulting protocols. The basic idea in these transformers is to periodically check the local network state and reset the system if there is some inconsistency. Work also has been carried out in checking the global network state and then applying a correction if an inconsistency is detected.

Another method of dealing with fault tolerance that has formed the focus of much research in the recent past has been concerning *reset-based self-stabilization* [14] [15]. The principle behind this is very simple- In reset-based self-stabilization, the state is constantly monitored. If an error is detected, a special reset protocol is invoked. The effect of this reset protocol is to consistently establish a pre-specified global state from which the system can resume normal operation. However one of the main drawbacks of this approach is that the detection mechanism triggers a system-wide reset in the face of the slightest inconsistency.

Patt-Shamir also researched into the field of time-adaptive self-stabilization. In [22], a protocol is defined as being time-adaptive if it recovers from faults in time which depends only on the number of faults. Ideally we would like a system to recover in the shortest possible time. Recovery clearly depends on the severity of the transient fault. In particular, it is influenced by the number of processors whose state was corrupted by the fault. In the same paper an interesting question is addressed concerning the notion of a "faulty processor". How does one define or more importantly identify a faulty processor? In other words, what constitutes a faulty processor. The need for a clear distinction between the concept of a faulty processor and a non-faulty processor is explained by this example: Consider a system with n processors. In this system, one of the processors is not consistent with the other *(n-1)* processors. How do we know which of the two- the one processor, or the *(n-1)* processors is faulty. Did a transient fault hit the outcast processor, or did the fault corrupt the state of all other *(n-1)* processors in a consistent way.

In [22], a state-based measure is used to come up with the answer to this question. The number of faults is defined to be the minimum number of processors, whose state needs to be changed to attain a stable global state. From this we come to the definition of a time-adaptive self-stabilization protocol as being on whose recovery time depends only on the number of faults. There are many useful references on self stabilization, which go into much greater detail on this issue [14] [15] [21] [22].

8 Possible Future Directions

All the work done so far in the mutual exclusion problem relies heavily on the ideas of Lamport's logical clocks and trying to obtain a logical scheduling of events. One area however, which has been neglected has been the extension of the problem to *real-time* systems. Most of the algorithms that appear in the open literature are concerned primarily with the order in which processes access the critical section. In achieving a complete ordering of the way in which different processes access the critical section, they achieve mutual exclusion. This does not provide a satisfactory solution in a *real time environment*, wherein *when* the process does what, is of primary concern. We would like to be able to get processes to perform certain actions at certain times, in this way facilitating real-time control.

Therefore in a real-time system, we would want the processes to be able to react in a timely fashion to any requests made. In this way we could get processes to start performing certain actions together, *at a particular instant in time*. The concept of logical clocks can help us determine if a certain process can access the critical section before another process, but not *when* it can access it. Thus we need the processes in the system to be synchronized as closely as possible. Extending the concepts covered in this paper to real-time systems is a problem which would be of great interest to both the controls and the computer science community.

References

[1] Jennifer Lundelius and Nancy Lynch, "An Upper and Lower Bound for Clock Synchronization", In*Information and Control*,62, pp. 190-204, 1984

[2] Shlomi Dolev and J.L.Welch, "Wait-Free Clock Synchronization", In *Proc. of the 12th Annual ACM Symp. on Principles of Distributed Computing*,pp. 97-108, 1993

[3] R.Alur and G.Taubenfield, "Fast Timing-based Algorithms", In *Distributed Computing* ,10,pp. 1-10, 1996

[4] Michael D.Lemmon, "Computer Controlled Systems- An Example", In*Lecture Notes*, University of Notre Dame, 1998.

[5] N.Lynch and N.Shavit, "Timing-based Mutual Exclusion", In *Proc. 13th IEEE Real-Time systems Symposium, IEEE Computer Society Press* ,pp. 2-11, 1992

[6] Boaz Patt-Shamir and Sergio Rajsbaum, “A Theory of Clock Synchronization”, In *Proc. 26th Symp. on Theory of Computing* , 1994

[7] Rajeev Alur, Hagit Attiya, and Gadi Taubenfield, “Time-Adaptive Algorithms for Synchronization”, In *Siam J. Comput* , 1997

[8] Rajeev Alur and Gadi Taubenfield, “How to Share an Object: A fast timing-based solution”, In *Proc. of the 5th IEEE Symp. on Parallel and Distributed Processing* , pp. 470-477, 1993

[9] Fred B. Seconder, “Understanding Protocols for Byzantine Clock Synchronization ”, 1987

[10] Yail Amir, “Clock Synchronization ”, In *Lecture Notes, JHU* , 1996

[11] Shlomi Dolev, “Possible and Impossible Self-Stabilizing Digital Clock Synchronization in General Graphs”, 1996

[12] Mukesh Singhal and Niranjan Shivaratri, *Advanced Concepts in Operating Systems*, McGraw Hill, 1994

[13] Jean Bacon, *Concurrent Systems - Operating Systems, Database and Distributed Systems: An Integrated Approach*, Addison-Wesley, 1997

[14] Baruch Awerbuch, Boaz-Patt Shamir and George Varghese, “Self-Stabilizing By Local Checking and Correction”, In *Proc. 32nd IEEE conference on Foundations of Computer Science*, 1991

[15] Baruch Awerbuch, Boaz-Patt Shamir and George Varghese, “Self-Stabilizing End to End Communication”, In *25th Symposium on Theory of Computing* , 1993

[16] L.Lamport, “Time, Clocks and Ordering of Events in a Distributed System”, In *Communications of the ACM*, vol. no. 21, no. 7, July 1978, pp. 558-565

[17] Michael D. Lemmon, Kevin X. He, and Ivan Markovsky, “Supervisory Hybrid Systems”, To appear in Control Systems, 1999

[18] E. Coffamn, M.J. Elphick and A. Shoshani, “System Deadlocks”, In *ACM Computing Surveys* , pp. 66-78, June 1971

[19] Sanders.B, “The Information Structure of Distributed Mutual Exclusion Algorithms”, In *ACM transactions on Computer Systems*, 1987

[20] Anish Arora, Shlomi Dolev and Mohamed Gouda, “Maintaining Digital Clocks in Step”, In *Parallel Processing Letters*, Vol.1, No.1, pp. 11-18, 1991

[21] L.Lamport and P.M. Mellair-Smith, “Synchronizing Clocks in the Presence of Faults”, In *Journal of the ACM*, Vol. 32, No. 1, pp. 1-36, 1985

[22] Shay Kutten and Boaz Patt-Shamir, “Time Adaptive Self-Stabiliztion”, In *Proceedings ACM Symposium on Principles of Distributed Computing* , 1997