

1-1-1993

Distributed mutual exclusion algorithms

Paul Banta

University of Nevada, Las Vegas

Follow this and additional works at: <https://digitalscholarship.unlv.edu/rtds>

Repository Citation

Banta, Paul, "Distributed mutual exclusion algorithms" (1993). *UNLV Retrospective Theses & Dissertations*. 316.

<http://dx.doi.org/10.25669/t1yh-ub7e>

This Thesis is protected by copyright and/or related rights. It has been brought to you by Digital Scholarship@UNLV with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 1356699

Distributed mutual exclusion algorithms

Banta, Paul, M.S.

University of Nevada, Las Vegas, 1993

Copyright ©1993 by Banta, Paul. All rights reserved.

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS

by

Paul Banta

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

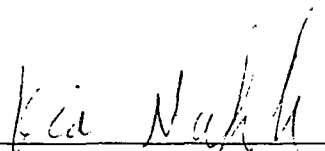
in

Computer Science

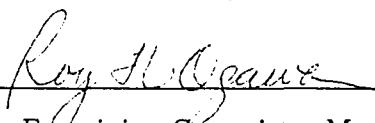
Department of Computer Science
University of Nevada, Las Vegas
December, 1993

©1993 Paul Banta
All Rights Reserved

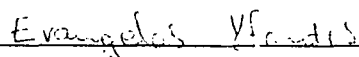
The thesis of Paul Banta for the degree of Master of Science in Computer Science is approved.



Chairperson, Kia Makki Ph. D.



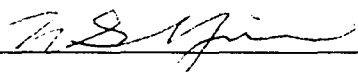
Examining Committee Member, Roy Ogawa Ph. D.




Examining Committee Member, Evangelos A. Yfantis Ph. D.



Examining Committee Member, Niki Pissinou Ph. D.



Graduate Representative, Woosoon Yim Ph. D.



Graduate Dean, R. W. Smith Ph. D.

University of Nevada, Las Vegas

January, 1993

ABSTRACT

In this thesis we present three original algorithms which solve the distributed mutual exclusion problem. Two of the three solve the problem of allowing only one site at a time into the critical section. The third solves the more difficult problem of allowing a specific number of sites (k sites) into the critical section at a time.

All three algorithms are "Token Based". That is, they make use of a token and token queue in order to guarantee mutual exclusion. Only the site that currently has the token is allowed to enter its critical section in the 1 mutual exclusion algorithms. Only the sites that have seen the token since they requested it are allowed to enter their critical sections in the k mutual exclusion algorithm.

The primary goal of our algorithms is efficiency. Both of our 1 mutual exclusion algorithms require between 2 and n messages per critical section (n being the number of sites) depending on the number of requests for the critical section. Our k mutual exclusion has similar requires between 3 and n messages per critical section depending on the number of requests for the critical section. In all three algorithms, the number of messages per critical section improve as the number of requests for the critical section increase.

TABLE OF CONTENTS

Abstract	iii
List Of Figures	vii
Chapter 1 - Introduction	1
1.1 Definitions	2
1.2 Problem Statement	5
1.3 Constraints And Assumptions	5
1.4 Goals	5
1.5 Limitations	6
1.6 Organization	6
Chapter 2 - Related Research	7
2.1 Token Based	8
2.1.1 Suzuki And Kasami	8
2.1.2 Ricart And Agrawala	9
2.1.3 Singhal	11
2.1.4 Raymond	12
2.1.5 Trehel And Naimi	14
2.2 Response Based	15
2.2.1 Lamport	16
2.2.2 Ricart And Agrawala	17
2.2.3 Carvalho And Roucairol	18
2.2.4 Singhal	19
2.2.5 Maekawa	20
Chapter 3 - Preliminaries	22
3.1 The Distributed System	22
3.2 Messages	23
3.3 Token	24
3.4 Token Requests	25
Chapter 4 - A Simple Distributed Mutual Exclusion Algorithm	31
4.1 Description Of The Simple Algorithm	31
4.2 Initialization	33
4.3 Detailed Description Of The Simple Algorithm	34
4.4 Discussion	38
4.5 Correctness	39
4.5.1 Mutual Exclusion	39
4.5.2 Starvation	39
4.6 Elimination Of Starvation	40

4.7 Performance Analysis	41
4.7.1 Light Token Request Analysis	42
4.7.2 Heavy Token Request Analysis	42
4.7.3 Performance In General	43
Chapter 5 - An Efficient Distributed Mutual Exclusion Algorithm	44
5.1 Outline	44
5.2 Token And Token Queue	45
5.3 Local Queue	45
5.4 Good Site	46
5.6 Message Types	48
5.7 Initialization	49
5.8 Description Of The Efficient Algorithm	49
5.9 Discussion	55
5.10 Correctness Of The Algorithm	55
5.10.1 Mutual Exclusion	56
5.10.2 Starvation	56
5.10.3 Fairness	56
5.11 Performance Analysis	57
Chapter 6 - An Efficient Distributed Algorithm For The k Mutual Exclusion Problem	61
6.1 Introduction	61
6.2 Description Of The Algorithm	63
6.2.1 Token And Token Queue	63
6.2.2 Token Semaphore	64
6.2.3 Local Queue	65
6.2.4 Good Site	66
6.2.5 Message Types	68
6.2.6 Initialization	69
6.2.7 Description Of The Distributed k Mutual Exclusion Algorithm	70
6.2.8 Discussion	74
6.3 Correctness Of The Algorithm	75
6.3.1 Mutual Exclusion	75
6.3.2 Starvation	76
6.3.3 Fairness	76
6.4 Performance Analysis	76
Chapter 7 - Conclusion	80
7.1 Summary	80
7.2 Future Research	82
Bibliography	84

LIST OF FIGURES

Figure 3.1: Distributed System	23
Figure 3.2: Message Passing	24
Figure 3.3: Passing The Token	25
Figure 3.4: One Cycle	29
Figure 4.1: Token Queue	32
Figure 5.1: Token Queue	45
Figure 5.2: Messages Per Critical Section	59
Figure 6.1: Token Queue	64
Figure 6.2: Token Queue	65

CHAPTER 1

INTRODUCTION

The Mutual Exclusion Problem is the problem of allowing a system resource to be shared in such a way that only one process or a small number of processes are allowed access to the resource at a time. For example, we would want all processes to be able to use a line printer, however the line printer should be restricted to one process at a time.

This thesis deals with the problem of Mutual Exclusion in a Distributed System, or Distributed Mutual Exclusion. In a computer system, centralized or distributed, there exist resources that are shared between multiple processes. Some of these resources must be restricted so as to allow only one or a small number of processes access at any given time. For example, it would not be appropriate to allow more than one process to print to the line printer at the same time. Therefore, use of the line printer resource must be restricted to one process at a time.

Three algorithms are presented which solve the distributed mutual exclusion problem. The first two allow only one process to enter the critical section at a time and require between 2 and n messages to be sent between processes for each entry into the critical section (n being the number of nodes in the distributed system). The third algorithm

allows up to k processes at a time to enter the critical section and requires between 3 and n messages to be sent between processes for each entry into the critical section.

The distinguishing feature of the first two algorithms is that as the number of requests for the critical section increases, the number of messages sent between processes remains the same. This has the effect of reducing the number of messages per critical section entry as the number of requests goes up.

The third algorithm is similar to the first two in the sense that as the number of critical section requests increases, the average number of messages required per critical section entry decreases.

1.1 DEFINITIONS

Shared resources are typically accessed only through a limited number of code segments. Therefore, by limiting access to these code segments, one could effectively limit access to the resource itself. These code segments are referred to as **Critical Sections**. The term critical section is used interchangeably with mutual exclusion and in this thesis should be taken to mean the same thing.

A **Distributed System**, by definition, is a system in which processes are allowed to

communicate with each other only by passing messages back and forth. Processes do not share any common memory and do not share a common clock.

There are three main requirements for a mutual exclusion algorithm. The algorithm should be free from **Starvation**, free from **Deadlock**, and provide a large degree of **Fairness**.

Starvation occurs when a process makes a request to enter the critical section but the request is never granted. There are a number of causes of starvation. In the final analysis, however, it is a shortcoming of the particular algorithm.

Deadlock can occur when multiple processes are requesting multiple resources. For example, process P_1 makes requests for resources R_1 and R_2 . At the same time, process P_2 makes requests for resources R_2 and R_1 . It may happen that P_1 is granted R_1 and P_2 is granted R_2 . When P_1 requests R_2 , it is unavailable due to the fact that it has been granted to P_2 . Therefore, P_1 waits for R_2 to become available. Meanwhile, P_2 makes a request for R_1 . Since R_1 is being held by P_1 , P_2 waits until it becomes available. This is the classic deadlock scenario. Both P_1 and P_2 will wait forever for their second resource to be granted while not releasing the first resource granted to them.

The last requirement of a mutual exclusion algorithm is that it provides some degree of **Fairness**. An unfair algorithm gives some sort of preference to certain processes. The

other processes are not given the same preference and are said to be treated unfairly. For example, we would say that an algorithm is unfair if it is more likely to grant process P_1 's request than to grant process P_2 's request.

The three algorithms presented in this thesis are free from starvation and are fair. They do not address the issue of deadlock due to the fact that they are designed to handle the mutual exclusion of a single resource while deadlock can only occur in an environment with multiple resources.

All three algorithms are **Token Based** and use a **Token Queue**. One simple method of achieving mutual exclusion is to have a single message called the **Token**. A site can only enter the critical section when it has the token. Once the site is done with the critical section, it sends the token to some other site which needs to use the token in order to enter its critical section. The question arises of "How does the site with the token know where to send the token next?". A queue is appended to the token with a list of all sites that are to receive the token. This is called the **Token Queue**. The site with the token has received the token because it reached the front of the queue. It removes itself from the queue and sends the token and token queue to the new site at the front of the queue. It is important to note that the queue is part of the token and not a separate message. When sending the token, the queue automatically is sent with the token as a single message.

1.2 PROBLEM STATEMENT

We are now ready to state the problem that this thesis attempts to solve: Given a distributed system, access to a shared resource must be limited to either one or a small number of processes at a time while guaranteeing that no process will starve and that all processes will be treated fairly.

1.3 CONSTRAINTS AND ASSUMPTIONS

There is one assumption that all three algorithms rely upon. This assumption is that all messages will take a finite and measurable amount of time to reach their destinations. This is to say that there is some maximum amount of time, T_{max} , in which each message is guaranteed to reach its destination. Furthermore, T_{max} is both measurable and known.

As stated earlier, our algorithms do not solve the problem of deadlock. Our algorithms deal with only one resource. Deadlock is a situation arising in a multiple resource environment.

1.4 GOALS

The primary goal of these algorithms is to minimize message traffic during periods when the critical section is being heavily requested. The more processes that request the

critical section at one time, the fewer messages per critical section entry are required. Our secondary goal is to present our algorithms in a way that highlights their simplicity and elegance.

1.5 LIMITATIONS

The algorithms in this thesis are somewhat limited in scope. As mentioned earlier, the issue of deadlock is not dealt with due to the single resource nature of the algorithm. Also, the issue of site failure is not dealt with here. We just assume that sites do not fail while the algorithm is running.

1.6 ORGANIZATION

The remainder of this thesis is divided into 6 chapters numbered 2 through 7. Chapter 2 gives an overview of what work has been done in the field of distributed mutual exclusion. Chapter 3 develops a basic algorithm in order to demonstrate the underlying concepts of our three algorithms. Chapters 4 through 6 are each devoted to one of our algorithms. Chapter 7 gives some concluding remarks and outlines some possible future extensions to our work.

CHAPTER 2

RELATED RESEARCH

The problem of Distributed Mutual Exclusion has been studied much in the past ten years. There are two types of mutual exclusion algorithms; Token based algorithms and Response based. Token based algorithms use a special message type called the token. Only one token message exists in the system. It is passed from site to site. The site that has possession of the token is allowed to execute the critical section. Since only one token message exists, mutual exclusion is guaranteed.

Response based algorithms require a site interested in entering the critical section to send requests to either all or a subset of all the other sites. A response must be received from all the sites to which a request was made before the requesting site is allowed to enter the critical section.

This chapter examines the classic algorithms in both the Token based and Response based areas.

2.1 TOKEN BASED

In this section we present a survey of the major contributions in the area of Token Based Distributed Mutual Exclusion algorithms.

2.1.1 SUZUKI AND KASAMI

In Suzuki and Kasami's algorithm [10] each site interested in the critical section sends requests to all other sites. These requests are numbered from 1 to n . Each site maintains an array of n entries which contain the most recent request received for each of the n sites. The token has a similar array of n entries which contains the most recent request fulfilled for each of the n sites. The token also has a queue of requests that need to be fulfilled.

When a site makes a token request, it increments its request number and sends out a request with that number to all other sites.

When a site receives a request, it checks its array of requests to see if the sequence number of this request is greater than the last request sequence number for that site. If so, it updates the current request sequence number for that site. If not, it ignores the request.

When a site is done with the token, it goes through its array of request sequence numbers for all n sites and compares them with the token array of the sequence numbers fulfilled. If a site, S_i has a greater sequence number in the request array than in the token fulfilled array, and S_i is not on the token queue, S_i is appended to the token queue. If a site, S_i has a lower sequence number in the request array than in the token fulfilled array, then S_i 's entry in the request array is updated to the value in the token fulfilled array.

The token is sent to the first site on the token queue. When received, that site removes itself from the front of the token queue and enters the critical section.

As can be easily seen the number of messages required per critical section entry is $n-1$ requests plus 1 token message or n messages per each critical section entry.

2.1.2 RICART AND AGRAWALA

Ricart and Agrawala's algorithm [6] is an improvement to Suzuki and Kasami's. The main difference is that there is no token queue. Each site maintains an array of n token requests, and the token queue maintains an array of n request fulfillments.

When a site makes a token request, it increments its request number and sends out a request with that number to all other sites.

When a site receives a request, it checks its array of requests to see if the sequence number of this request is greater than the last request sequence number for that site. If so, it updates the current request sequence number for that site. If not, it ignores the request.

When a site is done with the token, it can easily determine which sites are requesting the token by comparing its array of request sequence numbers with the token's array of fulfilled sequence numbers. Each site that has a higher request sequence number than fulfilled sequence number is requesting the token. Since the token no longer has a queue, the local site must select one site which is requesting the token to send the token to. One method for selecting the site to get the token next is to pick the lowest numbered site that is higher than that of the current site. This approach achieves a high degree of fairness. In essence, site S_i checks sites $S_{i+1}, S_{i+2}, S_{i+3}, \dots, S_n, S_1, S_2, \dots, S_{i-1}$ for the first one that is requesting the token. Another method for selecting the site to send the token next is to pick the lowest numbered site that is requesting the token. This is obviously not fair since it can greatly favor the low numbered sites.

As can be easily seen the number of messages required per critical section entry is $n-1$ requests plus 1 token message or n messages per each critical section entry.

2.1.3 SINGHAL

Singhal's algorithm [8] is an improvement on Ricart and Agrawala's. The main difference is that each node maintains an array of the probable states of all n sites. Each site can be "R"equesting the token, "E"xecuting the critical section, "N"ot requesting the token, or "H"olding the token. When a site becomes interested in the token, it sends requests to all the sites that it believes are "R"equesting the token. The token also maintains an array of the possible states of all n sites.

When a site sends a token request it also sends its information about the possible states of sites. The receiving site can then determine on a site by site basis, which information is more up to date, the information attached to the token request or its own local state array. This is done with sequence numbers in the same way as both the Ricart and Agrawala algorithm and the Suzuki and Kasami algorithm.

When a site wants to request the token, it sends requests to all sites that it currently thinks are "R"equesting the token. Along with these requests, it sends its own array of possible states of all n sites.

When a site receives a request, it uses the array of possible states to update its local array of possible states. If the receiving site is also "R"equesting the token, but has not

sent a request to the requesting site, it then sends out a request along with its array of possible state information.

When a site receives the token, it uses the token array of possible states to update its local array of possible states, sets its state to "E"xecuting the critical section and then executes the critical section. After finishing with the critical section, it checks to see if there are any states that are "R"equesting the token. If so, it uses one of the two methods described in Ricart and Agrawala's algorithm to select the next site to receive the token. Before the token is sent, it attaches its own array of possible states to the token. If there are no sites requesting the token, it keeps the token and sets its own state to "H"olding the token.

When the system is initialized, each site thinks that all the lower numbered sites are "R"equesting and all the higher numbered sites are "N"ot requesting.

The performance of Singhal's algorithm turns out to be about $(n+1)/2$ messages per critical section. This is due to the fact that requests are sent to only about half of the sites instead of all of the sites with the two previous algorithms.

2.1.4 RAYMOND

Raymond's algorithm [4] is a tree based algorithm. The root of the tree holds the token.

Each node communicates only with its immediate neighbors. Each node knows which of its neighbors is on the path to the root where the token is. This node is referred to as *Near*. A site requests the token by sending a request message to its *Near* node.

If a site becomes interested in the token and it has not received a request from any of its children, it sends a request to *Near* and places itself in its local request queue. While the site is waiting for the token, it places any requests that it might receive from its children in the same queue without making further requests.

If a site receives a token request from one of its children with the site itself not being interested in the token, then the child is placed in the local queue and a token request is sent to *Near*. While the site is waiting for the token, it places any more requests that it might receive from its children in the same queue without making further requests. Furthermore, if the site itself becomes interested in the token, it just places itself in the local queue without any further requests.

When a site receives the token from its *Near* neighbor, the direction of the edge between itself and its *Near* neighbor is changed. The site becomes the root of the tree. The first entry on the local queue is removed. If that entry is the site itself, then the critical section is executed. If that entry is one of the sites children, then the token is sent to that child, and the direction of the edge between the site and the child is reversed

effectively making the child the new *Near* node. If the sites local queue is not empty, then a token request is sent to the node that just received the token; the new *Near* node.

Raymond's algorithm requires $\log(n)$ messages per critical section entry due to the tree nature of the algorithm. Also, no special sequence numbers or queue structures are needed in this algorithm.

2.1.5 TREHEL AND NAIMI

In Trehel and Naimi's algorithm [11] each site has two variables, *NewRoot* and *Next*. *NewRoot* is the site that is believed to hold the token and where requests are sent to. *Next* is the next site to enter the critical section after the local site has entered the critical section.

When a site becomes interested in the token, it sends a request to *NewRoot*. When *NewRoot* receives the request, it will either set its *Next* to the requesting site or forward the request to its *NewRoot*. If it is requesting the token and has no site currently in its *Next* variable, it will set *Next* to the requesting site. If it is not requesting the token or if it is requesting but already has a site in its *Next* then it will forward the request to its *NewRoot*. In both cases, it changes its *NewRoot* to the requesting site since it is a site that will be receiving the token soon.

When a site receives the token, it uses the critical section. When done, it checks to see if its *Next* has a site in it. If so, it sends the token to that site and clears the variable. If not, it holds the token until it receives a request.

This algorithm provides simplicity of local data structures with no token queue while requiring only $\log(n)$ messages per critical section entry.

Trehel and Naimi's improved algorithm [12] tries to reduce the number of requests forwarded by changing *Next* to a request queue and adding a token queue. Instead of forwarding multiple requests, they are stored in the queue. When a site is finished with its critical section, it appends the local queue onto the token queue, sets its *NewRoot* to the last site on the token queue, and sends the token to the first site on the token queue. This algorithm also requires $\log(n)$ messages per critical section entry with a lower multiplying coefficient.

2.2 RESPONSE BASED

In this section we present a survey of the major contributions in the area of Response Based Distributed Mutual Exclusion algorithms. By Reply Based Algorithms we generally mean algorithms in which a site wishing to enter its critical section can only do so once it has received a response to every one of its requests.

2.2.1 LAMPORT

In all fairness, Lamport's algorithm [2] was never intended to be a serious algorithm at all. It was intended only as an illustration of a possible use for logical time clocks. The topic of logical time clocks is the primary focus of the paper. With this in mind, we explore the algorithm.

Lamport proposes a strategy whereby all events in a distributed system can be given a timestamp as to when they occurred with a reasonable degree of accuracy. This timestamp strategy is shown to have a useful application in distributed mutual exclusion among other things.

In the distributed mutual exclusion example that Lamport gives, each message is given a timestamp. This includes token requests, token passes, and other messages.

When a site requests the token it sends requests to all other sites. A site receiving a request places the request in a priority queue in order of the timestamp. The request at the front of the queue has the earliest timestamp. The receiving site also sends a response or acknowledgment message to the requesting site.

A site enters the critical section when its own request is at the front of the queue and it has received a message from all other sites with a timestamp larger than its own.

When a site determines that it can enter the critical section, it does so. When done, it sends a release message to all other site.

It can easily be seen that Lamport's strategy requires $3(n-1)$ messages per critical section entry. For every entry there are $n-1$ requests, $n-1$ acknowledgments, and $n-1$ releases.

2.2.2 RICART AND AGRAWALA

Ricart and Agrawala's algorithm [6,7] is an improvement to Lamport's. Instead of having both a response and a release message as in Lamport's algorithm, only a response message is used in this algorithm. Implied in the response message is a release.

As with Lamport's algorithm, a site interested in the critical section sends timestamped request messages to all other sites.

A site receiving a request can immediately determine whether itself or the requesting site should enter the critical section first based on the timestamp. If the receiving site should enter the critical section first, then a response message is deferred until after the receiving site has finished the critical section. If the requesting site should enter the critical section first, then a response is sent immediately. Of course, if a receiving site is not requesting the critical section, a response is also sent immediately.

This algorithm reduces Lamport's messages per critical section entry to $2(n-1)$. This is due to the elimination of the release messages.

2.2.3 CARVALHO AND ROUCAIROL

Carvalho and Roucairol's algorithm [1] is an improvement to Ricart and Agrawala's algorithm. Carvalho and Roucairol have noticed that some request messages are not necessary. When a site, S_i , becomes interested in the critical section it only sends requests to all sites who have requested the critical section since S_i last requested the critical section. Sites that have not made a request for the critical section are assumed to not be interested in the critical section, and are not sent requests. It is implied that they give their permission to enter the critical section.

An array is used at each site to record which sites have requested the critical section since the critical section was last entered.

When a site requests the critical section, it sends requests to all sites which have requested the critical section since the last request. This information is stored in the local array. The sites that have not requested the critical section are assumed to give response messages. The array of which sites have requested the critical section is then cleared.

When a site receives a request for the critical section, it acts the same as the Ricart and

Agrawala algorithm. It also makes a note in its array that the site has requested the critical section.

The performance of Carvalho and Roucairol's algorithm depends on the pattern and frequency of requests. It can range from 0 messages per critical section entry to $2(n-1)$.

2.2.4 SINGHAL

Singhal's algorithm [9] is similar to Carvalho and Roucairol's. Each site maintains a Request set, R , and an Inform set I .

When a site becomes interested in the critical section, it must send a request to each site in the request set. A response must be received by each of these sites before the critical section can be entered.

When a request is received, the receiving site determines the order in which it and the requesting site should enter their critical sections based on the timestamp of the request. If the requesting site should execute first, then the requesting site is added to the request set and both a response and a request are sent to the requesting site. If the receiving site should execute first, then the requesting site is added to the inform set, I . If the receiving site is not interested in the critical section, then a response is sent to the requesting site and the requesting site is added to the request set, R .

When a site receives a response, it removes that entry from its request set, R . When the request set becomes empty, the site is allowed to enter its critical section. Upon completion of the critical section, replies are sent to all sites in the inform set, I , and the inform set is copied to the request set.

When the system is initialized, each site places all lower numbered sites in its request set and places only itself in its inform set.

This algorithm has similar performance to that of Carvalho and Roucairol's. The message traffic can range between 0 and $2(n-1)$ messages per critical section depending on the pattern and frequency of requests.

2.2.5 MAEKAWA

Maekawa's algorithm [3] is significantly different from the others. Given a distributed system with n sites, the sites are divided into n subsets numbered 1 to n each containing \sqrt{n} sites. This implies that each site belongs to more than one subset. We can say that a site represents a subset if the site is a member of the subset. Each site, then, represents several subsets. The subsets are chosen so that each subset is represented by at least one member of every subset.

When a site becomes interested in the critical section, it needs only to send requests

to all the sites in the subset with the same number as the site itself. Site S_i sends requests to all sites in subset 1 for example. Since all subsets are represented by these sites, a response from each site in the subset means that all subsets have granted permission to enter the critical section.

When a site finishes with the critical section, it sends a release to all sites in the subset. This in essence sends a release to all subsets.

As can be easily seen, this algorithm requires \sqrt{n} requests, \sqrt{n} replies, and \sqrt{n} releases. The total being $3\sqrt{n}$ messages per critical section entry.

CHAPTER 3

PRELIMINARIES

In this chapter we develop a simple and basic algorithm for solving the distributed mutual exclusion problem. Although this algorithm may have some problems with it, it is used to illustrate the basic techniques of our other algorithms.

3.1 THE DISTRIBUTED SYSTEM

We begin by introducing a distributed system. This system will be used as an illustration for all of our algorithms. It is made up of six nodes or sites labeled S_1 through S_6 . The topology of connections between the sites is not of importance for this discussion and is therefore not shown. We can assume that each node is able to communicate with every other node. That is to say that there are no nodes in the system that are isolated from the rest of the nodes. A diagram of the distributed system is given in figure 3.1.

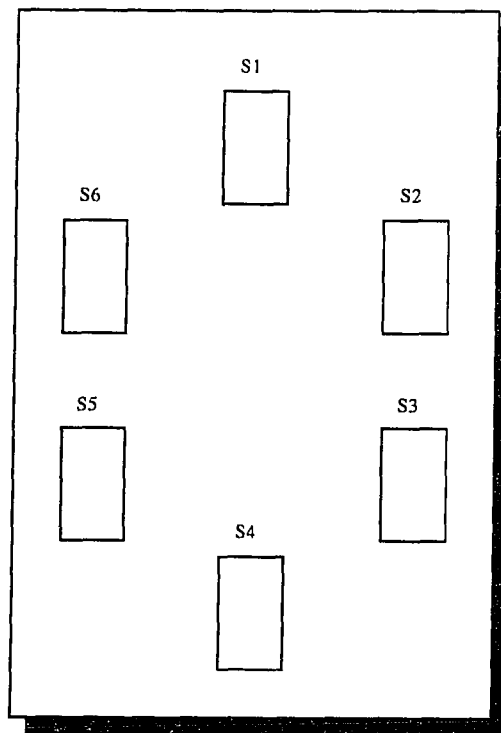


Figure 3.1: Distributed System

3.2 MESSAGES

By definition, sites in a distributed system can only communicate with each other by sending messages. Messages are represented pictorially as arrows. Figure 3.2 shows site S_1 sending a "Hello world" message to site S_2 .

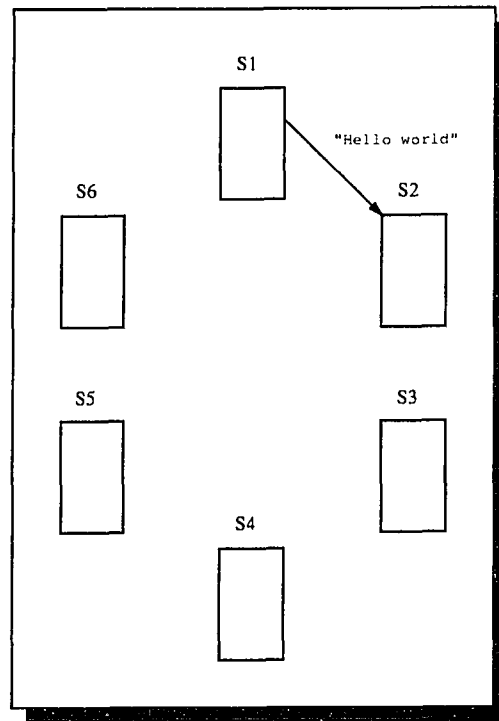


Figure 3.2: Message Passing

3.3 TOKEN

As mentioned earlier, our algorithms use a token. The token is a message that is passed from site to site. There is only one token message in the distributed system. Therefore, mutual exclusion is achieved by allowing only the site that currently has the token to enter the critical section. The site using the token does not send it to the next site until it is done using the critical section. The token is represented in the diagram as a box with a "T" in it. Sending the token from one site to another shown in figure 3.3.

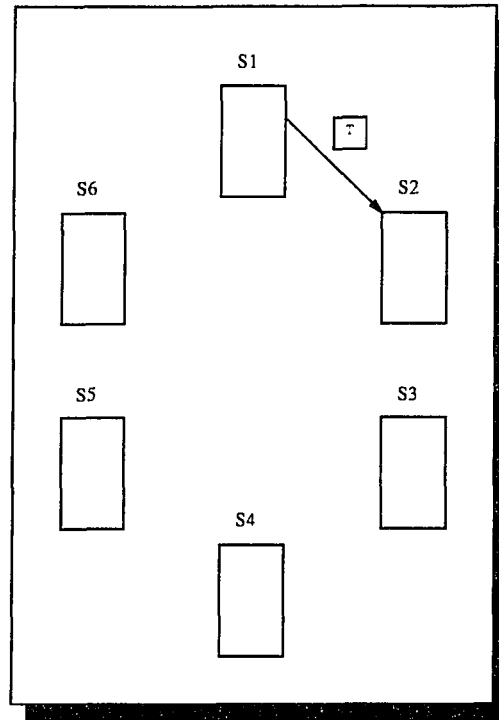


Figure 3.3: Passing The Token

3.4 TOKEN REQUESTS

So far, we have the basics of a distributed system and a token that gets passed around. Only the site that currently has the token is allowed to enter the critical section. This is a good theoretical start, but in a practical situation, how does the site with the token know where to send it next? A similar question might be asked; How does a site that wants the token make its request known to the other sites in a manner that will guarantee that it receives the token?

A first draft at the solution might involve having a site send request messages to all other sites when it becomes interested in the token. There are some obvious problems with this approach. What happens if other sites are making similar requests? The site that currently has the token may receive two different requests. It can only send the token to one of the two requesting sites. Another scenario is that the token may be in transit between sites when a token request is sent or received. No site currently has the token so the requests received at each site are ignored since each site does not have the token. Consider also the number of messages per critical section. Given a system with n sites, a requesting site must send $n-1$ requests. The token being sent to the site that requested it is another message. Already there have been n messages sent and 1 critical section entered. The numbers don't get any better when more than one site is requesting the token at the same time.

We diverge for a moment to a theory in computer architecture. It states that if you want to speed up a machine (CPU), concentrate on the instructions that account for the most execution time. It doesn't make sense to try to speed up an instruction that accounts for 3% of execution time. On the other hand, it makes a lot of sense to try to speed up an instruction that accounts for 35% of execution time. Applying this concept to the message traffic of our first draft, we see that the majority of message traffic is in the requests. Therefore, it makes sense to try to reduce the number of messages required to request the token.

If we knew of some site, S_j , that the token would be going to sometime in the near future, we could send a single request to S_j with special instructions to hold the request until the token is received. This strategy would reduce the number of messages per request from $n-1$ to 1. Still, though, we have the problem of S_j receiving requests from more than one site. To solve this problem, we can create a queue of requests at S_j . That way, all requests that S_j receives can be stored in the queue until the token arrives.

A second problem arises, though. What does S_j do in the event that it has more than one request in its queue when it is ready to send the token to the next site? A solution is to take the queue and incorporate it into the token itself. That way the token can be sent to the first site on the queue. When the first site is done with the token, it removes itself from the queue and sends the token to the next site on the queue. The next site uses the token, removes itself from the queue, and sends the token to the third site, and so on. The queue incorporated into the token is called the token queue. It is an integral part of the token and the two cannot be separated. When the token is sent from one site to the next, the token queue is sent along with it as a single message.

One final problem arises with this strategy. How does a requesting site know where to send its single request in the first place? In order to answer this question, we must back up one cycle. We assume that there is some site, S_i , which has a number of requests in its queue. When S_i receives the token it places its queue of requests on the token queue. We note that the last site on the token queue is S_j . At this point, S_i can send messages to

all sites informing them that the token will eventually end up at S_j . In this way, all sites will know to send their token requests to S_j . When S_j eventually receives the token and places its local queue onto the token queue, it also notes the last site on the token queue, S_k , and informs all other sites of this fact.

Figure 3.4 illustrates the basic algorithm presented so far. In step 1, the token is at site S_4 and one site is on the token queue, S_7 . Two token requests are sent to site S_7 . The reason that the requesting nodes know to send their requests to S_7 is that they have been informed previously that S_7 was the last site on the token queue. In step 2, the token is sent from S_4 to the site at the front of the token queue, S_7 . We also see that the requests from S_2 and S_5 have been placed on the local queue at S_7 . In step 3, the token has arrived at S_7 and S_7 has removed itself from the token queue and appended its local queue to the token queue. We do not show it in the figure, but S_7 is allowed to enter its critical section during this step. The timing of when this is done is not important as long as S_7 holds the token during the entire time that it is in the critical section. In step 4, the token is sent to the first site on the token queue, S_2 . Also, all sites are informed of the new last site on the token queue, S_5 . This enables all sites to now send token requests to S_5 .

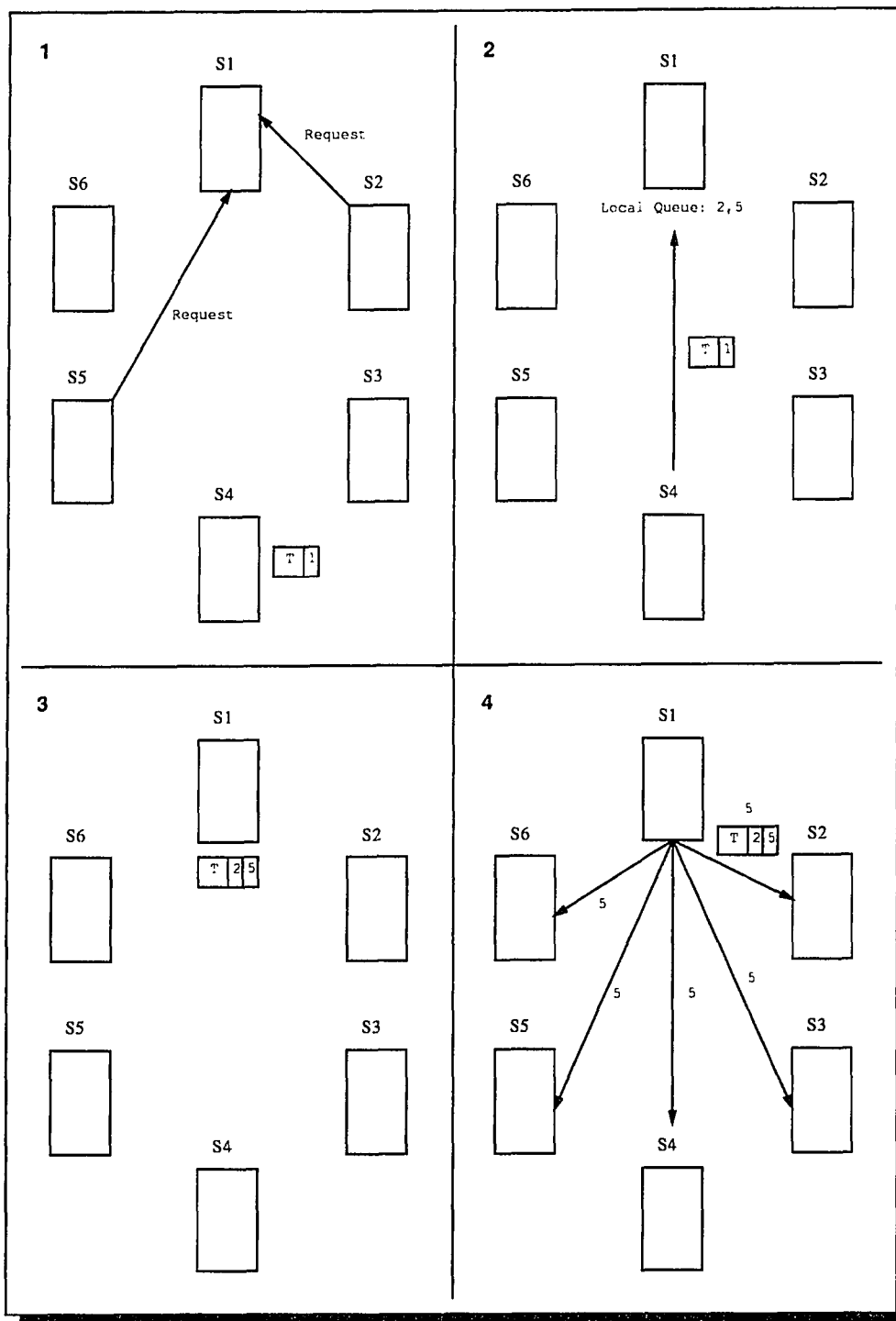


Figure 3.4: One Cycle

As a final development of our basic algorithm we note that it is not necessary to inform any of the sites which are on the token queue of the last site on the token queue. The reason is that the token and token queue will be sent to those sites anyway. When each of the sites on the token queue receive the token, they can look at the token queue to find out which site is at the end. In step 4 of figure 3.4, the two messages to S_2 and S_5 informing them that S_5 was the last site on the queue are not necessary. Both these sites will receive the token and queue during this cycle and can check the token queue themselves.

We also introduce here a formal mechanism used by each site to remember which site is the last on the token queue. Each site has a local variable called *good_site*. *Good_site* holds the site to be reported as the last site on the token queue. It is essentially a "good site" to request the token from.

CHAPTER 4

A SIMPLE DISTRIBUTED MUTUAL EXCLUSION ALGORITHM

This chapter presents an algorithm for solving the mutual exclusion problem. It is given as a preliminary algorithm that is not free from starvation. In section 4.6 we give a modification to the preliminary algorithm that removes the starvation problem.

4.1 DESCRIPTION OF THE SIMPLE ALGORITHM

The preliminary algorithm presented here is not free from starvation and is intended to demonstrate the main concept of how we use the token queue. It is also intended to demonstrate the simplicity and elegance of the algorithm without bogging down in the details of how to avoid starvation. A modification to the preliminary algorithm is presented in section 4.6 which focuses on how to remove the starvation problem.

In the algorithm, we make use of a "Token-Queue". The token queue contains a list of all the sites which have requested the token. The token and token queue are passed to the first site on the token queue, and then to the next, and so on. Each time the token is passed to a new site, that site is removed from the token queue. For example, if sites S_8 , S_2 , and S_4 have requested the token, the token and queue would look like figure 4.1.

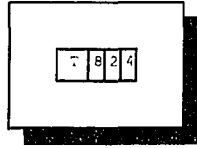


Figure 4.1: Token Queue

Sites which are not on the token queue are sent messages informing them of the last site on the token queue. If a site becomes interested in the critical section, it would send a single request to the last site on the token queue. The last site on the token queue would hold the request until the token arrives, and then places the request at the end of the token queue. Sites only need to be updated about the last site on the token queue when the last site on the token queue changes. For example, if the token queue contains sites S_8 , S_2 , and S_4 , and the token is sent to sites S_8 followed by S_2 without any new sites being added to the token queue then the last site on the token queue is still S_4 . In this situation, no sites need to be informed that the last site on the token queue is still S_4 . If, however, when the token arrived at S_4 three new sites were added to the token queue, S_5 , S_3 , and S_7 , then all sites other than S_4 , S_5 , S_3 , and S_7 would need to be informed that S_7 was the new last site on the token queue. S_4 , S_5 , S_3 , and S_7 will each update themselves that S_7 is the last site on the token queue when they receive the token.

In addition to storing the last site on the token queue, each local site also keeps track of its own state and maintains a local queue of token requests that it has received. A site

may be in one of four states. They are "N"ot requesting the token, "R"equesting the token, "E"xecuting the critical section, and "H"olding the token.

Each site must also maintain a local queue of token requests. For example, let's assume that site S_4 is "R"equesting the token and also happens to be the last site on the token queue. All other sites in the system will send their request for the token to S_4 . Since S_4 knows that the token will eventually be sent to it, it keeps the token requests in its local queue. When the token does arrive at S_4 , S_4 adds its local queue to the token queue.

4.2 INITIALIZATION

In order to initialize the system, the token must be created with an empty token queue, every site must be "N"ot requesting, every site must know the location of the token, and every site must have an empty local queue. This is to say that there have been no requests made for the token, and every site knows the location of the token.

We can arbitrarily say that site S_1 is the site that will create the token with empty token queue, and it must be in the "H"olding state. All other sites will then be in the "N"ot Requesting state and know that site S_1 is a good site to ask for the token.

4.3 DETAILED DESCRIPTION OF THE SIMPLE ALGORITHM

The preliminary algorithm is given as a set of procedures. The procedures represent the necessary responses to make in given situations. It is assumed that all data structures are available to each procedure including the token and token queue.

The preliminary algorithm is not free from starvation. We will present a method for correcting this problem later in the chapter.

```
procedure request_token ;  
begin  
    site_state := "R" ;  
    send ( token_request, good_site ) ;  
end .
```

This procedure is called when a site wishes to enter its critical section. The state of the local site is changed to "R" requesting the token. A request for the token is sent to a "Good Site" to ask for the token. The good site to ask is the last known site on the token queue.

```

procedure receive_good_site_update ;
begin
    receive ( good_site ) ;
end .

```

This procedure is called when a site receives an update about the last site on the token queue. The local variable *good_site* is updated to reflect that the last site on the token queue is a good site to send a token request to.

```

procedure receive_request ;
begin
    if ( ( site_state = "R" ) or
        ( site_state = "E" ) or
        ( site_state = "H" ) ) then
        begin
            enqueue( token_request, local_queue ) ;
            if ( site_state = "H" ) then
                begin
                    call send_token ;
                end if ;
            else
                send ( token_request, good_site ) ;
            end if ;
        end
    end .

```

This procedure is called when a site receives a request for the token. If the site is requesting the token, executing its critical section, or is holding the token, the request is

stored in the local queue. If the site is not requesting the token, then the token request is forwarded to the last known site on the token queue.

If the site is in the "H"olding state this implies that the token queue is empty and the local site is holding the token until it receives some information on where to send it. In this event, a call is also made to send the token to the next site. The *send_token* routine will add the local queue to the token queue before sending the token.

```
procedure receive_token ;  
begin  
  dequeue ( first item, token queue ) ;  
  site_state := "E" ;  
  call critical_section ;  
  call send_token ;  
end .
```

This procedure is called when the token is received. The first item is removed from the token queue. This item should be the local site since the token is sent to the first site on the token queue. Next, the state of the local site is changed to "E"xecuting the critical section and the critical section is entered. Finally, a call to send the token to the next site on the token queue is made.


```

procedure send_token ;
begin
  if ( local_queue not empty ) then
    begin
      enqueue ( local_queue, token_queue ) ;
      for ( i = each site not on token_queue ) do
        send ( token_queue[ last ], site[ i ] ) ;
      end for ;
    end if ;
    if ( token_queue not empty ) then
      begin
        good_site := token_queue[ last ] ;
        send ( token, site[ token_queue[ first ] ] ) ;
        site_state := "N" ;
      else
        site_state := "H" ;
      end if ;
    end .
  end .

```

This procedure is called to send the token to the next site on the token queue. First, if the local queue has requests on it, these requests are added to the token queue. Also, all sites not on the token queue are updated about the last site on the token queue. This is done only if the local queue was not empty which implies that the last site on the token queue has changed. Finally, if the token queue is not empty, the token is sent to the first site on the token queue and the site state is changed to "N"ot requesting. If the token queue happens to be empty, then the token is not sent and the site state is changed to "H"olding which signifies that the local site is waiting for a request to be received in order to send the token.

4.4 DISCUSSION

The basic idea behind the preliminary algorithm is that the token has a queue of sites that must be served. If a site, S_i , knows the last site on the token queue, S_j , it can send its request to that site. If the token queue is long enough, the request from S_i will reach S_j before the token does. Site S_j , being in the "R"equesting state, knows that the token will eventually be sent to it. Site S_j , therefore stores the token request from S_i in its own local queue. When the token reaches site S_j , S_j 's token request is placed at the end of the token queue.

In the event that the token request from site S_i reaches site S_j after the token has come and gone, S_j will forward S_i 's request to the new last site on the token queue.

All the sites that are not on the token queue must be kept informed about the last site on the token queue. Therefore, each time the last site on the token queue changes, each site not on the token queue must be updated. These updates do not necessarily occur in a timely manner. An information packet is sent to each site and the token is then passed on. There is no guarantee that any of the information packets sent will reach their destinations before the token reaches the next site or the next or the next etc.

4.5 CORRECTNESS

We claim that the preliminary algorithm presented achieves mutual exclusion, but we note that it is not free from starvation. We show here that the algorithm achieves mutual exclusion and also show the starvation problem. In section 4.6 we present a way to avoid starvation.

4.5.1 MUTUAL EXCLUSION

In order to achieve mutual exclusion, we must show that at most only one site will be in its critical section at a time. This is trivial due to the nature of token passing. We claim that there is only one token and that a site may enter its critical section only if it possesses the token. Possession of the token by a site, S_i , guarantees that no other site has the token and therefore that no other site is in its critical section.

4.5.2 STARVATION

We claim that the preliminary algorithm, as presented, is not free from starvation. Here we show the starvation problem. Starvation occurs when a site S_i issues a token request to site S_j . By the time the token request is received by site S_j , the token has already come to site S_j and been sent to the next site, site S_k . Site S_j knows this because it is in the "Not requesting state". Site S_j therefore, forwards S_i 's token request to site S_k . By the time

the token request is received by site S_k , the token has again come and gone. As you can see, there is no guarantee that this type of a cycle will ever end. Essentially, site S_i 's token request may never catch up to the token in order to be put on the token queue and therefore, site S_i will never receive the token and will starve. In the next section we discuss a modification to this algorithm which corrects its starvation problem.

4.6 ELIMINATION OF STARVATION

We present a method for modifying the preliminary algorithm to prevent starvation. In this modification to the preliminary algorithm, all token requests are numbered. When site S_i makes a token request it gives the token request a unique number (unique to site S_i). For example, a site could number its requests 0, 1, 2, 3, etc. When site S_j receives S_i 's request, it not only forwards the token request to site S_k , but stores the request in its own local queue along with the unique request number. Likewise, when site S_k receives S_i 's request it does the same.

The basic idea behind this scheme is that the number of sites that the token can be sent to while still eluding S_i 's request is reduced each time S_i 's request is received and forwarded. As site S_i 's request is placed on more and more local queues, the token has fewer and fewer sites that it can travel to which are not aware of S_i 's request. In the worst case, S_i 's request will be sent to $n-1$ sites before it is put on the token queue.

The reason that requests are given unique numbers is to avoid having requests placed on the token queue multiple times. Since multiple sites record copies of a single request, we must ensure that a particular request be placed on the token queue only once. In order to achieve this, the token must also have a history array. The history array records the most recently serviced request for each site in the distributed system. For example, site S_i has made four requests numbered 0, 1, 2, and 3. Each one of these has been serviced. Therefore, the i^{th} entry in the history array contains a 3 indicating that S_i 's request number 3 has been satisfied. However, site S_j may not be aware of this information, and may have S_i 's request number 2 in its local queue. If the token were sent to site S_j in this particular situation, site S_j would compare its own local queue with the history array of the token. Since site S_i 's request number 3 has been satisfied, all previous requests by site S_i must also have been satisfied, including request number 2. Therefore, site S_j knows not to place request number 2 on the token queue.

4.7 PERFORMANCE ANALYSIS

In this section we look at the performance analysis of the preliminary algorithm in an extremely light token request environment and an extremely heavy token request environment. We then make some general comments about the performance.

4.7.1 LIGHT TOKEN REQUEST ANALYSIS

In an environment with very light token requests, n messages are required per critical section execution where n is the number of sites. Lets assume that the token is waiting at some site, S_i , with no sites on the token queue. Another site, S_j , becomes interested in its critical section and sends its request to site S_i . Site S_i places S_j 's request on the token queue and informs all other sites that the last site on the token queue is site S_j . The token is then sent to site S_j . In this example, there was 1 token request, $n-2$ last site update messages, and 1 token transfer. This adds up to exactly n messages.

4.7.2 HEAVY TOKEN REQUEST ANALYSIS

In an environment with very heavy token requests, c messages are required per critical section execution where c is a constant which can be as small as 2. Lets assume that the token queue has been built up to the point where all n sites are on it. The token is sent to the first site on the queue, S_i . It is impossible for S_i to have any sites in its local queue since all the sites are on the token queue. Therefore, site S_i cannot add a last site onto the token queue. Since the last site on the token queue has not changed, no last site update messages are required to be sent. The token is then passed to the next site, S_j . While S_j is using the token, site S_i again becomes interested in the critical section. Since site S_i just had the token, it is aware of the last site on the token queue, S_i . Site S_i sends its token request to site S_i . When site S_j is done with the token, it has no sites on its local queue,

and does not change the last site on the token queue. Therefore, no last site update messages are required to be sent. Site S_j sends the token to the next site on the queue, S_k . While S_k is using the token, site S_j becomes interested in the critical section. Since S_j just had the token, it is aware that S_z is the last site on the token queue, and sends its token request to S_z . As can be seen, all token requests will be sent to site S_z where they will be stored in the local queue until the token arrives. In this situation there will be n token transfers while passing the token to each site on the token queue and n token requests being sent to S_z . When the cycle is complete, the token queue will once again have all n sites on it and n token requests will have been satisfied. This yields 2 messages per critical section execution.

4.7.3 PERFORMANCE IN GENERAL

As we have shown, in light token request environments, our algorithm has no performance improvement over other algorithms. However, as token requests increase, more and more sites are placed on the token queue and subsequently do not need to be sent last site update messages. Our algorithm is designed to perform extremely well with very heavy token requests.

CHAPTER 5

AN EFFICIENT DISTRIBUTED

MUTUAL EXCLUSION ALGORITHM

This chapter presents another algorithm for solving the mutual exclusion problem. It avoids the forwarding of requests that was present in the previous algorithm. It also does not use the sequence numbers that were needed in the previous algorithm. It does, however, introduce a wait at the end of every cycle in order to allow late requests to arrive.

5.1 OUTLINE

Sites which are not on the token queue are periodically sent update messages informing them of the good site. Update messages are sent each time the good site changes. If a site becomes interested in the critical section, it sends a single request to the good site. The good site receives token requests from all sites requesting the token. These requests are stored in a local queue until the token arrives. When the token does arrive, the good site executes its critical section and then appends its local queue to the token queue. A new good site is chosen as the last site on the token queue and update messages are sent to all sites which are not on the token queue. A wait is then executed so that late token

requests can be received and placed on the token queue. Finally, the token and token queue are sent to the first site on the token queue.

5.2 TOKEN AND TOKEN QUEUE

In the algorithm, we make use of a "Token Queue". The token queue contains a list of all the sites which have requested the token. The token and token queue are passed to the first site on the token queue, and then to the next, and so on. Each time the token is passed to a new site, that site removes itself from the token queue. In addition, one site on the token queue is flagged as a good site to send requests for the token to. We will refer to this site as the "good site". For example, if sites S_8 , S_2 , and S_4 have requested the token, and S_4 has been designated as the good site, then the token and queue would look like Figure 5.1. Notice that S_4 has an arrow by it indicating that it is the good site.

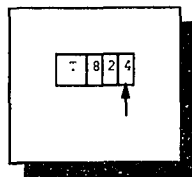


Figure 5.1: Token Queue

5.3 LOCAL QUEUE

All requests for the token are sent to the good site. Therefore, the good site must

maintain a local queue of requests that it has received. The good site's local queue of requests is referred to as the local queue. When the token is eventually received by the good site, it executes its critical section and then appends its local queue onto the token queue.

In actuality, each site has a local queue because it is possible for each site to become the good site. However, only one site at a time is the good site. Therefore only the good site will have entries in its local queue because only the good site receives token requests

5.4 GOOD SITE

The good site is the site specified as a good site to send requests for the token to. The good site is typically the last site on the token queue, although this is not always the case. The good site places all token requests that it receives on the local queue. When the token is eventually received by the good site, it executes its critical section and then appends its local queue onto the token queue. At this point, a new good site must be chosen. The best choice for the new good site is the last site on the token queue. The local site (which is no longer the good site since a new one has been chosen) updates all sites not on the token queue about the new good site. The local site, immediately after sending good site update messages, waits for late token requests to be received before sending the token to the first site on the token queue. The amount of time that it must wait is two time

periods. A time period is defined to be the maximum amount of time that it takes to send a message from any site to any other site.

We claim that two time periods is a necessary and sufficient amount of time to wait in order to collect late token requests. When a site, S_i , sends updates to all sites not on the token queue, it could take at most one time period to reach all destinations. In the worst case, this update will reach the furthest site, S_j , just as S_j is sending a request for the token. This good site update to S_j will take at most one time period. Since S_j has not updated its record of who to request the token from, the request is directed to site S_i . It will take another one time period for S_j 's request to be received at S_i . Therefore, the total amount of time for all late token requests to arrive at S_i after S_i has sent good site update messages can be no more than two time periods. All requests after the two time periods will be sent to the new good site.

Each site in the system keeps track of which site is the good site. Each time the good site changes, update messages are sent to all sites not on the token queue informing them of the new good site. When a good site update is received by a site, S_i , S_i updates its local value of good site.

5.6 MESSAGE TYPES

Three types of messages are used in the algorithm. The token message which includes the token queue, token request messages, and good site update messages.

The token message is used to send the token and token queue from one site to the next site on the token queue.

Token request messages are sent from sites requesting the token to the good site. Included in the token request message is the id of the site that is requesting the token.

Good site update messages are sent when the good site changes. They are sent from the old good site to all sites that are not on the token queue. The reason for not sending good site update messages to sites which are on the token queue are two-fold. First, all sites on the token queue have already made a request for the token and the token will be sent to them. By virtue of this fact, they do not need to make a second request. Second, when the token is sent to a site, S_i , on the token queue, S_i will examine the token queue and update its good site at that time. This is possible since the good site is flagged on the token queue.

5.7 INITIALIZATION

In order to initialize the system, the token must be created with an empty token queue, every site must know the location of the token, and every site must have an empty local queue. This is to say that no sites are requesting the token, and all sites know the location of the token.

There are several ways of achieving this objective, but for simplicity we will arbitrarily say that site S_l is the site that will create the token with empty token queue. Furthermore, all sites will initialize themselves by clearing their local queue and setting their good site to l .

5.8 DESCRIPTION OF THE EFFICIENT ALGORITHM

The algorithm is given as a set of procedures. The procedures represent the necessary responses to make in given situations.

```
procedure request_token ;  
begin  
    send ( token_request, good_site ) ;  
end .
```

This procedure is called when a site wishes to enter its critical section. A request for the token is sent to the good site.

```
procedure receive_good_site_update ;  
begin  
    receive ( good_site ) ;  
end .
```

This procedure is called when a site receives a good site update message. The local variable *good_site* is updated to reflect the new good site. The procedures "receive" and "send" are calls to perform the low level sending and receiving of messages.

```

procedure receive_token_request ;
begin
  receive ( token_request ) ;
  enqueue ( token_request, local_queue ) ;
  if ( have_token ) then
    if ( token_queue not empty ) then
      append ( local_queue, token_queue ) ;
    else
      append ( local_queue, token_queue ) ;
      mark ( good_site, token_queue[ last ] ) ;
      good_site := token_queue[ last ] ;
      for ( i := all sites not on token_queue ) do
        send ( good_site, site[ i ] ) ;
      end for ;
      wait ( 2 time periods ) ;
      have_token := false ;
      send ( token, token_queue[ 1 ] ) ;
    end if ;
  end if ;
end .

```

This procedure is called when the good site receives a request for the token. The token request is placed on the local queue.

If the good site currently has the token then one of two possibilities exists. First, the request could be a late request. A late request is defined to be a request received after the new good site has been chosen. In this case, the late request which has been placed in the local queue is appended to the token queue. It is appended after the good site. One consequence of this is that this request will not be fulfilled during the current cycle. It will be fulfilled during the next cycle. This is to say that the token will be sent to all the

sites preceding the good site, followed by the good site which will pick a new good site and wait for two time periods before the token is sent to the site making the late request. Intuitively, it would seem that this strategy lacks efficiency and that it would be more efficient to place late requests on the queue before the good site forcing the good site to always be the last site on the queue. In reality, however, there is no lack of efficiency in this strategy. The apparent loss of efficiency in placing the late requests after the good site is immediately regained during the next cycle which in effect receives the request.

The second possibility is that there are no entries on either the local queue or the token queue (other than the request just received). In this case, the good site has no sites on the token queue to choose a new good site from and, more importantly, has no site to send the token to. Therefore, the good site is waiting for a request when the request is received. What the good site does in this case is to append its local queue (containing the one request just received) to the token queue, choose the one site on the token queue to be the good site, inform all other sites of the new good site, wait 2 time periods, send the token to the first site on the token queue, and set the local variable *have_token* to false.


```

procedure receive_token ;
begin
  receive ( token ) ;
  have_token := true ;
  call critical_section ;
  dequeue ( token_queue[ 1 ], token_queue ) ;
  get ( good_site, token_queue ) ;
  if ( good_site <> -1 ) then
    begin
      have_token := false ;
      send ( token, token_queue[ 1 ] ) ;
    else
      enqueue ( local_queue, token_queue ) ;
      if ( token_queue not empty ) then
        begin
          mark ( good_site, token_queue[ last ] ) ;
          good_site := token_queue[ last ] ;
          for ( i := all sites not on token_queue ) do
            begin
              send ( good_site, site[ i ] ) ;
            end for ;
          wait ( 2 time periods ) ;
          have_token := false ;
          send ( token, token_queue[ 1 ] ) ;
        end if
      end if ;
    end .
  end .

```

This procedure is called when the token is received. First, a local variable, *have_token*, is set to true. This indicates that the local site has the token. Next, the critical section is executed. Next, the first item is removed from the token queue. This item should be the id of the local site since the token is always sent to the first site on the token queue. Finally, the local *good_site* variable is updated from the token queue.

At this point, we must check to see if the local site is the good site. This is done with a call to "get". "Get" returns the good site from the token queue or -1 if the token queue has no good site specified. If the local site is the good site, the token queue will have no good site on it since the local site id was already removed from the token queue. This is detected by searching the token queue for the good site and not finding one. The result being that the value of *good_site* will be -1.

In the case that the local site is not the good site, the token is sent to the next site on the token queue, and the local variable *have_token* is set to false.

In the case that the local site is the good site, a number of steps must be taken. First, the local queue (which may be empty) is appended to the token queue. Next, if the token queue is not empty then the last site on the token queue is chosen as the new good site and the local value for good site is updated. Next, all sites not on the token queue are sent good site updates. A wait for 2 time periods is begun after which the token is sent to the first site on the token queue and the local variable *have_token* is set to false.

In the case that the local site is the good site and the local queue and token queue are both empty, the local site must just hold the token until a request arrives.

5.9 DISCUSSION

The basic idea behind the algorithm is that the token has a queue of sites that must be served. If a site, S_i , knows the good site, S_j , it can send a **single** request to S_j . If the token queue is long enough, the request from S_i will reach S_j before the token does. Site S_j knows that the token will eventually be sent to it. Therefore, site S_j stores the token request from S_i in its own local queue. When the token reaches site S_j , S_i 's token request is placed at the end of the token queue.

All the sites that are not on the token queue must be kept informed about the good site. Therefore, each time the good site on the token queue changes, each site not on the token queue must be updated. These updates are guaranteed to occur in a timely manner by forcing the site which is sending the update messages to wait for two time periods before sending the token on.

5.10 CORRECTNESS OF THE ALGORITHM

We claim that the algorithm achieves mutual exclusion, is free from starvation and is fair.

5.10.1 MUTUAL EXCLUSION

In order to achieve mutual exclusion, we must show that at most only one site will be in its critical section at a time. This is trivial due to the nature of token passing. We claim that there is only one token and that a site may enter its critical section only if it possesses the token. Possession of the token by a site, S_i , guarantees that no other site has the token and therefore that no other site is in its critical section.

5.10.2 STARVATION

Starvation occurs when a site S_i issues a token request to the good site, S_j . By the time the token request is received by site S_j , the token has already come to site S_j and been sent to the next site, site S_k . We claim that this is an impossible situation since the good site, S_j , must issue new good site updates and then wait for 2 time periods for late requests. In the worst case, the late request from S_i would be received by S_j just before S_j sent the token to the first site on the token queue.

5.10.3 FAIRNESS

This algorithm favors no sites over any other sites. That is to say, no sites are give preferential treatment over any other site. This can be seen in the way requests are placed on the token queue. They are added in the order that they are received. The token requests

are then serviced in the order that they are on the token queue. That is to say that the algorithm works on a first-come-first-served basis and is therefore fair.

5.11 PERFORMANCE ANALYSIS

Performance is measured as the average number of messages required per critical section execution. The number of messages required per critical section is easily derived. We first introduce the notion of a cycle. A cycle begins when the current good site picks a new good site. A cycle ends when the new good site has received the token and finishes executing its own critical section. During a cycle, all sites not on the token queue are sent update messages, the old good site waits 2 time periods for late requests, and the token is passed to all sites on the token queue up to and including the new good site. Notice that the end of one cycle is the beginning of the next cycle and that the algorithm can be viewed as executing one cycle after another.

THEOREM 5.1: The number of messages per critical section execution in the distributed mutual exclusion algorithm is $(n/m)+1$.

PROOF: We examine one cycle and compute the average number of messages required per critical section for that cycle. We assume that there are n sites in the distributed system and we let m be the number of token requests on the token queue at the beginning of

the cycle. We will assume that $m > 0$ since $m = 0$ implies that no sites are on the token queue. It is obvious that $m \leq n$ since each site can only send one request. Site S_i will be designated as the good site that starts this particular cycle and site S_j will be designated as the new good site which will end the cycle.

Site S_i begins the cycle by picking S_j as the new good site. We note that S_j is the last of m sites on the token queue. Therefore, there must have been m requests sent to S_i in order for m requests to be on the token queue. S_i then sends new good site update messages to all sites not on the token queue. Since there are m sites on the token queue, we know that $n - m$ sites are not on the token queue. Therefore $n - m$ new good site update messages are sent by S_i . After S_i waits for 2 time periods, it sends the token to the first site which executes its critical section and that site sends it on. The token is sent in turn to each of the m sites on the token queue and eventually is sent to S_j which executes its critical section and ends the cycle.

At this point we can count the total number of messages that have been sent during this cycle and divide by the number of critical sections that were executed to get the average number of messages

required per critical section. There were m token requests, $n-m$ new site update messages, and m token passes. This adds up to $m+(n-m)+m$ messages or $n+m$ messages. There were m critical sections executed. Therefore, the average number of messages per critical section is $(n+m)/m$ or $(n/m)+1$. \square

Figure 5.2 illustrates the inverse relationship between the number of messages per critical section and the number of sites requesting the critical section.

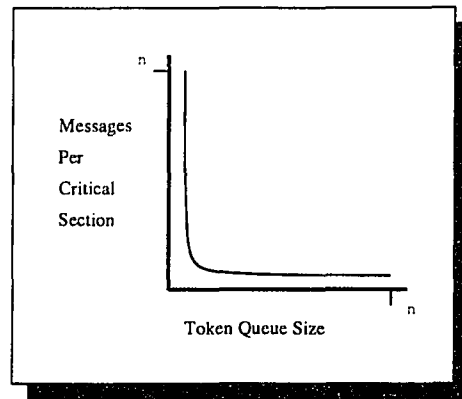


Figure 5.2: Messages Per Critical Section

The explanation of the performance, as shown in Figure 5.2, is that the algorithm improves as the number of requests for the token increase. If very few sites are requesting the token at any given time, the algorithm will require on the order of n messages per

critical section execution. However, as token requests increase, fewer messages per critical section execution are required. For example, if 1 out of every 10 sites are requesting the critical section, the algorithm requires 11 messages per critical section. If 1 out of every 5 sites are requesting the critical section, only 6 messages per critical section are required. In the extreme case of every site requesting the critical section, only 2 messages are required per critical section.

We claim that the performance approaches optimality in an environment with heavy token requests. We base this claim on the fact that if a site, S_i , becomes interested in the critical section, it must send at least one token request. Secondly, the token must be sent to S_i . In an environment with heavy token requests, the majority of message traffic is token requests and token passes with very few good site updates being sent. Each site interested in entering its critical section makes one token request and it takes one token pass to send the token to it.

CHAPTER 6

AN EFFICIENT DISTRIBUTED ALGORITHM FOR THE k MUTUAL EXCLUSION PROBLEM

Most of the work [1-4,6-12] reported in the literature allows at most one site at a time to execute the critical section. Raymond [5] was the first to present a distributed mutual exclusion algorithm which allows up to k sites to simultaneously execute the critical section. His work was an extension to the Ricart and Agrawala [6] algorithm so as to allow up to k sites to execute the critical section concurrently. In this chapter we present a new distributed algorithm which allows up to k simultaneous entries into the critical section.

6.1 INTRODUCTION

This algorithm solves the k mutual exclusion problem in a distributed system using token passing. This is the problem of allowing up to k sites entry into the critical section at the same time. The algorithm makes use of a "Token Queue" and a "Token Semaphore" both of which are part of the token itself. The token queue contains a list of all sites which are requesting the token. The token semaphore is a general semaphore.

The token, queue, and semaphore are sent to the first site on the token queue followed

by the second and the third etc. The token finally stops at a designated wait site. A site may enter its critical section when one of two sets of events occur. (1) A site receives the token, and the semaphore, S , is non-zero. In this case, the site decrements S , removes itself from the token queue, and sends the token to the next site on the token queue, and enters its critical section. (2) A site receives the token, and the semaphore, S , is zero. In this case, the site removes itself from the token queue, sends the token to the next site on the token queue, and waits for a release message from a previous site before entering its critical section. In both cases 1 and 2, when a site receives the token it must note the k^{th} site on the token queue. This is the site that must be sent a release message when the critical section is exited.

A site, S_i , requests the token (puts itself on the token queue) by sending a single token request to a site, S_j , which it believes will be using the token in the near future. When S_j receives S_i 's request it stores the request in a local queue of requests until the token arrives. When the token arrives at S_j , S_j places all the requests in its local queue onto the token queue. Site S_j then removes itself from the token queue, notes the k^{th} site on the token queue, sends the token to the first site on the token queue and enters its critical section. The token is sent in turn to each site on the token queue. When the token is received by site S_i , the semaphore is either positive or zero. If the semaphore is positive, site S_i decrements the semaphore, removes itself from the token queue, notes the k^{th} site on the token queue, sends the token to the first site on the token queue, and enters its critical section. If the semaphore is zero, then S_i removes itself from the token queue,

notes the k^{th} site on the token queue, sends the token to the first site on the token queue, and waits for a release message from a previous site before it can enter the critical section. In either case, when S_i is finished with the critical section, it sends a release message to the k^{th} site that it had previously noted.

6.2 DESCRIPTION OF THE ALGORITHM

Sites which are not on the token queue are periodically sent update messages informing them of the good site. Update messages are sent each time the good site changes. If a site becomes interested in the critical section, it sends a single request to the good site. The good site receives token requests from all sites requesting the token. These requests are stored in a local queue until the token arrives. When the token does arrive, the good site executes its critical section and then appends the local queue to the token queue. A new good site is chosen as the last site on the token queue and update messages are sent to all sites which are not on the token queue. A wait is then executed so that late token requests can be received and placed on the token queue. Finally, the token and token queue are sent to the first site on the token queue.

6.2.1 TOKEN AND TOKEN QUEUE

This algorithm makes use of a "Token Queue" which is part of the token. The token queue contains a list of all the sites which have requested the token. The token and token

queue are passed to the first site on the token queue, and then to the next, and so on. Each time the token is passed to a new site, that site removes itself from the token queue. In addition, one of the sites on the token queue is flagged as the good site to send token requests to. We will refer to this site as the "Good Site".

For example, if sites S_8 , S_2 , and S_4 have requested the token, and S_4 has been designated as the good site, then the token and queue would look like Figure 6.1. Notice that the 4 has an arrow pointing to it indicating that it is the good site.

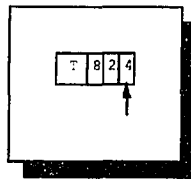


Figure 6.1: Token Queue

6.2.2 TOKEN SEMAPHORE

The algorithm also makes use of a "Token Semaphore" which is part of the token. The token semaphore is a general semaphore. It indicates the number of critical sections that are available to be entered simultaneously. It initially has the value k which is the number of critical sections that can be executed at the same time. Whenever a site begins executing its critical section, the semaphore is decremented by one. When a site completes the critical section, the semaphore is incremented by one. If the semaphore ever

reaches zero, no further sites are allowed to enter their critical section until the semaphore is incremented to a positive number. The token semaphore is passed, with the token and token queue, to each site on the token queue. Each site that receives the token checks the value of the semaphore. If the value is non-zero then the value is decremented by one and the token and semaphore are passed to the next site on the token queue. If the value is zero, then no change is made to the semaphore before passing it to the next site on the token queue.

For example, if sites S_8 , S_2 , and S_4 are currently on the token queue as in figure 6.1, and two more sites are allowed to execute their critical sections, the token would look like figure 6.2. Notice that the first "2" is the general semaphore and indicates that 2 more sites are allowed in the critical section.

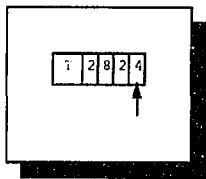


Figure 6.2: Token Queue

6.2.3 LOCAL QUEUE

All requests for the token are sent to the good site. Therefore, the good site must

maintain a local queue of requests that it has received. The good site's local queue of requests is referred to as the local queue. When the token is eventually received by the good site, it executes its critical section and then appends the local queue onto the token queue.

In actuality, each site has a local queue because it is possible for any site to become the good site. However, only one site at a time is the good site. Therefore only the good site will have entries in its local queue because only the good site receives token requests.

6.2.4 GOOD SITE

The good site is the site specified as "a good site to send requests for the token to". The good site is typically the last site on the token queue. The good site places all token requests that it receives on its local queue. When the token is eventually received by the good site, it executes its critical section and then appends its local queue onto the token queue. At this point, a new good site must be chosen. The best choice for the new good site is the last site on the token queue. The local site (which is no longer the good site since a new one has been chosen) sends update messages to all sites not on the token queue informing them of the new good site. The local site, immediately after sending good site update messages, waits for late token requests to be received before sending the token to the first site on the token queue. All late requests are appended to the end of the token queue. The amount of time that it must wait is two time periods. A time period is

defined to be the maximum amount of time that it takes to send a message from any site to any other site.

Two time periods is a necessary and sufficient amount of time to wait in order to collect late token requests. When a site, S_i , sends updates to all sites not on the token queue, it could take at most one time period to reach all destinations. In the worst case, the update will reach the furthest site, S_j , just as S_j is sending a request for the token. Since S_j has not updated its record of the new site, S_j 's token request is sent to site S_i . It will take another one time period for S_j 's request to be received by S_i . Therefore, the total amount of time for all late token requests to arrive at S_i after S_i has sent good site update messages can be no more than two time periods. All requests made after the two time periods have elapsed are guaranteed to be sent to the new good site.

Each site in the system keeps track of which site is the good site. Each time the good site changes, update messages are sent to all sites not on the token queue informing them of the new good site. When a site receives a good site update, it updates its local value of the good site.

The reason for not sending good site update messages to sites which are on the token queue are two-fold. First, all sites on the token queue have already made a request for the token and the token will be sent to them. By virtue of this fact, they will not make a second request until the first request has been satisfied. Second, when the token is sent

to a site, S_i , on the token queue, S_i examines the token queue and updates its good site at that time. This is possible since the good site is flagged on the token queue.

6.2.5 MESSAGE TYPES

Four types of messages are used in this algorithm. The token message which includes the token queue and token semaphore, token request messages, good site update messages, and release messages.

- The token message is used to send the token, token queue, and token semaphore from one site to the next site on the token queue.
- Token request messages are sent from sites requesting the token to the good site. Included in the token request message is the id of the site that is requesting the token.
- Good site update messages are sent when the good site changes. They are sent from the old good site to all sites that are not on the token queue.
- A Release message is sent by each site, S_i , upon exiting its critical section. It is sent to the site that was k^{th} on the token queue after S_i had removed itself. It is guaranteed that if only k sites are allowed in the critical section at one time then

the k^{th} site from S_i will receive the token with the general semaphore being zero. Therefore, the k^{th} site requires a release message also in order to enter its critical section. The exception to this rule is that if the good site on the token queue is before the k^{th} site, then the release message is sent to the good site. This will cause the good site to receive a release message from each of the k previous sites. The k release messages are collected by the good site and used to increment the general semaphore back up to its maximum value of k .

6.2.6 INITIALIZATION

In order to initialize the system, the token must be created with an empty token queue and the general semaphore set to k , the number of sites allowed in the critical section at a time. Every site must know the location of the token, and every site must have an empty local queue. This is to say that no sites are requesting the token, and all sites know the location of the token.

There are several ways of achieving this objective, but for simplicity we will arbitrarily say that site S_1 is the site that will create the token with empty token queue and initialize the general semaphore. Furthermore, all sites will initialize themselves by clearing their local queue and setting their good site to 1.

6.2.7 DESCRIPTION OF THE DISTRIBUTED K MUTUAL EXCLUSION ALGORITHM

The algorithm is given as a set of procedures. The procedures represent the necessary responses to make in given situations.

```
procedure request_token ;  
begin  
    send ( token_request, good_site ) ;  
end .
```

This procedure is called when a site wishes to enter its critical section. A request for the token is sent to the good site.

```
procedure receive_good_site_update ;  
begin  
    receive ( good_site ) ;  
end .
```

This procedure is called when a site receives a good site update message. The local variable *good_site* is updated to reflect the new good site. The procedures "receive" and "send" are calls to perform the low level sending and receiving of messages.

```
procedure receive_token_request ;  
begin  
    receive ( token_request ) ;  
    enqueue ( token_request, local_queue ) ;  
end .
```

This procedure is called when the good site receives a request for the token. The token request is simply placed on the local queue.

```

procedure receive_token ;
begin
  receive ( token ) ;
  get_good_site( token_queue ) ;
  dequeue ( token_queue[ 1 ], token_queue ) ;
  if ( good_site = local_site ) then
    wait_for_all_release_messages ;
    call critical_section ;
    append( local_queue, token_queue ) ;
    clear( local_queue ) ;
    if ( token_queue is empty ) then
      wait_for( token_request ) ;
      append( token_request, token_queue ) ;
    end if ;
    good_site = token_queue[ last ] ;
    for i = all_sites_not_on_token_queue do
      send( good_site, site[ i ] ) ;
    end for ;
    wait( 2 time_periods ) ;
    append( local_queue, token_queue ) ;
    clear( local_queue ) ;
    send( token, token_queue[ 1 ] ) ;
  else
    release_site := min( token_queue[ k ], good_site ) ;
    if ( token_semaphore <> 0 ) then
      decrement( token_semaphore ) ;
      send( token, token_queue[ 1 ] ) ;
      call critical_section ;
      send( release, release_site ) ;
    else
      send( token, token_queue[ 1 ] ) ;
      receive( release ) ;
      call critical_section ;
      send( release, release_site ) ;
    end if ;
  end if ;
end .

```

This procedure is called when the token is received. First, the token is received. Next, the current site finds the good site on the token queue. Next, the current site takes itself off of the token queue. At this point, there are two situations that are possible. Case 1 is that the current site is the good site. Case 2 is that the current site is not the good site.

Case 1: If the current site is the good site then the current Site must:

- 1 Wait for all release messages to catch up
- 2 Call its critical section
- 3 Append its local queue of requests to the token queue and clear its local queue
- 4 Pick a new good site
- 5 Send good site update messages to all sites that are not on the token queue
- 6 Wait for 2 time periods for late requests to catch up
- 7 Append the late requests to the token queue, clear its local queue again and send the token to the first site on the token queue.

It is possible that both the local queue and the token queue are empty after the good site removes itself from the token queue. In this case, the good site waits for a token request to be received and then places this request onto the token queue before picking a new good site and continuing the next cycle.

Case 2: If the current site is not the good site then the release site is calculated as either the k^{th} site on the token queue or the good site on the token queue, whichever one is closer to the beginning of the token queue. Next, a check is made to see if the semaphore on the token is non-zero.

If the semaphore is non-zero, then the semaphore is decremented by one, the token is sent to the next site on the token queue, the critical section is executed and a release message is sent to the release site.

If the semaphore is zero, then the token is sent to the next site on the token queue, and a receive is initiated which waits for a release message.

When the release message is received, the critical section is executed and a release is sent to the release site.

6.2.8 DISCUSSION

The basic idea behind the algorithm is that the token has a queue of sites that must be served. If a site, S_i , knows the good site, S_j , it can send a single request to S_j . Site S_j knows that the token will eventually be sent to it. Therefore, site S_j stores the token request from S_i in it's own local queue. When the token reaches site S_j , S_i 's token request is placed at the end of the token queue.

All the sites that are not on the token queue must be kept informed about the good site. Therefore, each time the good site on the token queue changes, each site not on the token queue must be updated. These updates are guaranteed to occur in a timely manner by forcing the site which is sending the update messages to wait for two time periods before sending the token on.

The token semaphore and release messages are used to allow only up to k sites in their critical sections at one time. When the token is received by the next good site, that site waits for all the release messages to catch up before starting a new cycle.

6.3 CORRECTNESS OF THE ALGORITHM

The algorithm presented achieves mutual exclusion, is free from starvation, and is fair.

6.3.1 MUTUAL EXCLUSION

In order to achieve mutual exclusion, we must show that at most k sites will be in their critical sections at a time. At the beginning of each cycle, only k sites are initially allowed in their critical sections due to the semaphore. After that, release messages are sent to allow further sites into their critical sections. However, one release message is sent each time a site is finished with its critical section. This guarantees that at most k sites will be in their critical sections at any one time.

6.3.2 STARVATION

Starvation occurs when a site S_i issues a token request to the good site, S_j . By the time the token request is received by site S_j , the token has already come to site S_j and been sent to the next site, site S_k . We claim that this is an impossible situation since the good site, S_j , must issue new good site updates and then wait for 2 time periods for late requests. In the worst case, the late request from S_i would be received by S_j just before S_j sent the token to the first site on the token queue.

6.3.3 FAIRNESS

Fairness deals with favoring some sites over other sites in the selection of which sites are allowed to execute their critical sections and in what order. Clearly this algorithm is fair and does not favor any site over any other sites in granting the critical section. This can be seen in the way that requests are received and placed onto the token queue. Token requests end up on the token queue in the exact order that the good site received them. Token requests are serviced in a first-come-first-served manner which is fair.

6.4 PERFORMANCE ANALYSIS

Performance is measured as the average number of messages required per critical section execution.

The number of messages required per critical section is easily derived. We first introduce the notion of a cycle. A cycle begins when the good site picks a new good site and ends when the new good site finally receives the token and executes it's own critical section. During a cycle, good site update messages are sent, the 2 time period wait is executed, and the token is sent to all of the sites on the token queue up to and including the new good site. Notice that the end of one cycle is the beginning of the next cycle and that the algorithm can be viewed as executing one cycle after another.

THEOREM 6.1: The number of messages per critical section execution in the distributed k mutual exclusion algorithm is $(n/m)+2$.

PROOF: We examine one cycle and compute the average number of messages required per critical section for that cycle. We assume that there are n sites in the distributed system and we let m be the number of token requests on the token queue at the beginning of the cycle. We will assume that $m > 0$ since $m = 0$ implies that no sites are on the token queue. It is obvious that $m \leq n$ since each site can only send one request. Site S_i will be designated as the good site that starts this particular cycle and site S_j will be designated as the new good site which will end the cycle.

Site S_i begins the cycle by picking S_j as the new good site. We note

that S_j is the last of m sites on the token queue. Therefore, there must have been m requests sent to S_j in order for m requests to be on the token queue. S_j then sends new good site update messages to all sites not on the token queue. Since there are m sites on the token queue, we know that $n-m$ sites are not on the token queue. Therefore $n-m$ new good site update messages are sent by S_j . After S_j waits for 2 time periods, it sends the token to the first site which executes its critical section and sends it on. The token is sent in turn to each of the m sites on the token queue and eventually is sent to S_j which executes its critical section and ends the cycle.

Each site that received the token during the cycle also is required to send a release message to the k^{th} site on the token queue. Therefore there are m release messages sent.

At this point we can count the total number of messages that have been sent during this cycle and divide by the number of critical sections that were executed to get the average number of messages required per critical section. There were m token requests, $n-m$ good site update messages, m token passes, and m release messages. This adds up to $m+(n-m)+m+m$ messages or $n+2m$ messages. There were m critical sections executed. Therefore, the

average number of messages per critical section is $(n+2m)/m$ or $(n/m)+2$. \square

The explanation of the performance is that our algorithm improves as the number of requests for the token increase. If very few sites are requesting the token at any given time, our algorithm will require on the order of n messages per critical section execution. However, as token requests increase, fewer messages per critical section execution are required. For example, if 1 out of every 10 sites are requesting the critical section, our algorithm requires 12 messages per critical section. If 1 out of every 5 sites are requesting the critical section, only 7 messages per critical section are required. In the extreme case of every site requesting the critical section, only 3 messages are required per critical section.

CHAPTER 7

CONCLUSION

We now conclude this thesis by summarizing the major points and providing some directions for future research.

7.1 SUMMARY

In this thesis we have presented a survey of the major distributed mutual exclusion algorithms as well as three new ones. We have seen that there are two major classes of distributed mutual exclusion algorithms; token based and response based. Token based algorithms use a single token message. The site that possesses the token is allowed to enter the critical section.

Reply based algorithms require a site to receive a response message for each critical section request that they send out. Requests are sent out to all sites or some subset of the sites in the distributed system.

We have presented three new algorithms for achieving mutual exclusion in a distributed system. They are all token based. The first two achieve one mutual exclusion while the third achieves k mutual exclusion.

The algorithms presented make use of a token queue. A site requesting the token sends a single request to a "good site" which stores the request in a local queue of requests. When the token arrives, the local queue is appended to the token queue. All sites that are not on the token queue are informed to update their "good site" to the last site on the token queue. The token is then sent to the first site on the token queue followed by the second and the third and so on. Each site that receives the token removes itself from the token queue, executes the critical section, and sends the token to the first site on the token queue.

The performance of the algorithms improve as the number of requests on the token queue increases. The reason behind this is that fewer good site update messages must be sent while more sites receive the token. The algorithm presented in chapter 5 improves from n messages per critical section in the light token request scenario to 2 messages per critical section in the extremely heavy token request scenario. The k mutual exclusion algorithm in chapter 6 improves to 3 messages per critical section in the extremely heavy token request scenario.

The best algorithm presented in the survey of chapter 2 was $\log(n)$. It must be noted, however, that the algorithms surveyed in chapter 2 consistently performed well while the new algorithms presented perform well in the heavy environment only. The advantage behind this strategy is that when token requests are infrequent, it doesn't matter as much how many messages are being sent. As demand for the token rises, we want to curtail

messages. As it turns out, a fairly constant number of messages are required per cycle in our algorithms. The variable is the number of critical sections that are to be served. With a constant number of messages and an increased number of token requests, the number of requests per critical section drops.

7.2 FUTURE RESEARCH

Research into mutual exclusion Algorithms in distributed systems has proceeded along two fronts, the study of Token-Based algorithms and the study of Response-Based algorithms. Although several algorithms for each group have been suggested in the literature, none of these provide optimal performance in all cases of request traffic. Our work in this area has raised several intriguing open questions.

- 1 Could there be a token based distributed mutual exclusion algorithm which performs optimally in all cases of request traffic?
- 2 Could there be a token based distributed mutual exclusion algorithm which has a constant time in all cases of request traffic?
- 3 Could there be a response based distributed mutual exclusion algorithm which approaches the optimal performance of two messages per critical section as our algorithm does?

Our work leaves much room for future research. Although deadlock detection and correction is fast becoming a separate field by itself, we note that distributed algorithms should be modified to avoid or correct such a condition. Another major area of research involves detection and recovery of site failures. For example, how is site failure detected and quantified? Was the token at the site that failed? Is the site that failed on the token queue? What should a site that has just recovered initialize to? These are all questions that are worthy of further work in this area.

BIBLIOGRAPHY

- [1] Carvalho And Roucairol "On Mutual Exclusion In Computer Networks, Technical Correspondence" Communications Of The ACM, Vol 26, No 2, pp 146-148, Feb 1983
- [2] Lamport "Time, Clocks And Ordering Of Events In Distributed Systems" Communications Of The ACM, Vol 21, No 7, pp 558-565, Jul 1978
- [3] Maekawa "A \sqrt{N} Algorithm For Mutual Exclusion In Decentralized Systems" ACM Transactions On Computer Systems, Vol 3, No 2, pp 145- 159, May 1985
- [4] Raymond "A Tree-Based Algorithm For Distributed Mutual Exclusion" ACM Transactions On Computer Systems, Vol 7 No 1 pp 61-77, Feb 1989
- [5] Raymond "A Distributed Algorithm For Multiple Entries To A Critical Section" Information Processing Letters, Vol 30 No 4 pp 189-193, Feb 1989
- [6] Ricart And Agrawala "An Optimal Algorithm For Mutual Exclusion In Computer Networks" Communications Of The ACM, Vol 24, No 1, pp 9-17, Jan 1981
- [7] Ricart And Agrawala "Author's Response To On Mutual Exclusion In Computer Networks, Technical Correspondence" Communications Of The ACM, Vol 26, No 2, pp 146-1248, Feb 1983
- [8] Singhal "A Heuristically-Aided Algorithm For Mutual Exclusion In Distributed Systems" IEEE Transactions Of Computers, Vol 38, No 5, pp 651-662, May 1989
- [9] Singhal "A Dynamic Information Structure Mutual Exclusion Algorithm For Distributed Systems" Proceedings Of The 9th International Conference On Distributed Computing Systems, pp 70-78, Jun 1989
- [10] Suzuki And Kasami "A Distributed Mutual Exclusion Algorithm" ACM Transactions On Computer Systems, Vol 3, No 4, pp 344-349, Nov 1985
- [11] Trehel And Naimi "A Distributed Algorithm For Mutual Exclusion Based On Data Structures And Fault Tolerance" Proceedings Of The 6th Annual IEEE International Phoenix Conference On Computer And Communications, pp 35-39, Mar 1987
- [12] Trehel And Naimi "An Improvement Of The Log(N) Distributed Algorithm For Mutual Exclusion" Proceedings Of The 7th International Conference On Distributed Computing Systems, pp 371- 375, Sep 1987