1896

FAULT TOLERANCE OF COTERIES

A
Dissertation submitted to
JAWAHARLAL NEHRU UNIVERSITY, New Delhi
in partial fulfillment of the requirements
for the award of the degree of

Master of Technology
In
Computer Science & Technology

By
SOMITRA KUMAR SANADHYA

Under the Guidance of
PROF. P. C. SAXENA
&
PROF. C. P. KATTI

SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI – 110067
JANUARY – 2002

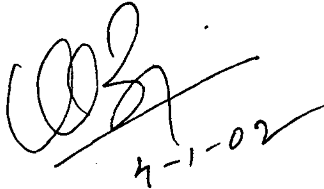
SCHOOL OF COMPUTER & SYSTEMS SCIENCES

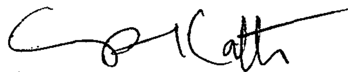
जवाहरलाल नेहरू विश्वविद्यालय
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI-110067 (INDIA)

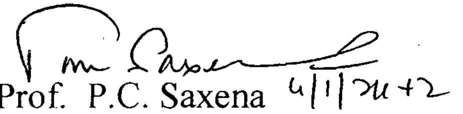
CERTIFICATE

This is to certify that that the dissertation entitled *"Fault Tolerance of Coteries"* which is being submitted by Mr. Somitra Kumar Sanadhya to the School of Computer & Systems Sciences, JAWAHARLAL NEHRU UNIVERSITY, NEW DELHI for the award of Master of Technology in Computer Science & Technology is a bonafide work carried out by him under my supervision.

This is an original work and has not been submitted in part or in full to any university or institution for the award of any Degree.







Prof. (Dr.) K.K. Bharadwaj
Dean
SC & SS, JNU

Prof. C.P. Katti
Supervisor
SC & SS, JNU

Prof. P.C. Saxena
Supervisor
SC & SS, JNU

Professor- K. K. Bharadwaj
DEAN
School of Computer & Systems Sciences
Jawaharlal Nehru University
New Delhi- 110067

ACKNOWLEDGEMENTS

I wish to convey my heartfelt gratitude and sincere acknowledgements to my Supervisors Prof. P. C. Saxena and Prof. C.P. Katti for their constant encouragement, guidance and affection throughout my work.

I am grateful to them for providing me enough infrastructure through Data Communication and Distributed Computing Group (DCDCG) laboratory to carry out my work. It is a great honour for me for being a member of DCDC Group and I am thankful to all the members of the group for their cooperation, encouragement and understanding.

I would like to thank all my faculty members for their help and useful suggestions during my work. I am also thankful to all my classmates for their constructive criticism and useful suggestions.

I am indebted to Mr. Jagmohan Rai for his constructive criticisms and help throughout this work. I could not have completed this work without his help. It will not be fair if I do not mention the help rendered by my batch mates during the course of this work. I am particularly thankful to Mr. Anant Jayswal, Mr. Deepak Nigam, Mr. Praveen Tripathi, Mr. Rakesh Mohanty, Mr. Umar Siddiqui, Mr. Rajesh Kumar and Ms. Nita Gupta. I sincerely thank them all for the help and encouragement.

SOMITRA KUMAR SANADHYA

CONTENTS

1. Introduction

1.1Distributed Computing System

1.2 The Mutual Exclusion Problem

1.2.1 Multiple Applications

1.2.2 Structured Applications

1.3 Types of Concurrent Processes

1.3.1 Process Unaware Of Each Other

1.3.2 Processes Indirectly Aware Of Each Other

1.3.3 Processes Directly Aware Of Each Other

1.4 Competition Among Processes For Resources

1.5 Cooperation Among Processes By Sharing

1.6 Cooperation Among Processes By Communication

1.7 Requirements For Mutual Exclusion

2. Mutual Exclusion In Distributed Systems

2.1 Centralized Algorithm With Control Node

2.2 Characteristics Of A Fully Distributed Algorithm

2.3 Lamport's Time Stamping Scheme

2.4 Lamport's Algorithm (Using Distributed Queue)

2.5 A Token Passing Scheme Based On Time Stamping

3. A Taxonomy Of Distributed Mutual Exclusion

3.1 Background

3.1.1 System Model

3.1.2 Token Vs Non-Token Dichotomy

3.1.3 Performance Measures

3.2 A Classification Of Token-Based Algorithms

3.2.1 Broadcast-Based Algorithms

3.2.2 Logical-Structure Based Algorithms

3.3 Non-Token Based Algorithms
3.3.1 The Information Structure Of Non-Token Based Algorithms
3.3.2 The Generalized Non-token Based Algorithm
3.3.3 Static vs. Dynamic Information Structures
3.4 A Classification Of Non-Token Based Algorithms
3.4.1 Ricart Agrawala Type Permission
3.4.2 Maekawa-Type Permission
3.4.3 Two Special Cases Of The Generalized Algorithm
3.4.4 Ricart-Agrawala Type Algorithms
3.4.5 Maekawa-Type Algorithms
3.4.6 A Centralized Mutual Exclusion Algorithm
3.4.7 A Fully Distributed Mutual Exclusion Algorithm
3.4.8 Agrawal-Abbadi Algorithm
3.4.9 Deadlock Problem in Maekawa-Type Algorithms
3.4.10 Deadlock-Free Maekawa-Type Algorithms
3.4.11 Lamport's Algorithm
3.5 Hybrid Algorithm

4. Coteries And Their Properties
4.1 Introduction
4.2 Definitions
4.3 Properties of coteries

5. A Fault Tolerance Algorithm
5.1 Preliminaries
5.2 The Algorithm
5.3 Correctness Of The Algorithm
5.4 Illustration

6. Conclusions
References

Abstract

A distributed system is a collection of processors that do not share memory or a clock. Owing to the complexities of a distributed system, it has to provide mechanism for dealing with a variety of situations and failures that are not encountered in a centralized system. One of the principle issues in the design of a distributed system is the enforcement of mutual exclusion.

In the event of multiple concurrent processes, there exists a possibility that more than one process might be trying to access a shared object or a non-shareable resource. The "Mutual Exclusion Problem" is ensuring that no more than one process enters into the "critical region" simultaneously.

Coteries were introduced in a by Garcia-Molina H. and Barbara D. [6] as a mathematical abstraction to model mutual exclusion in distributed systems. A coterie on a set of nodes S is a collection of mutually intersecting, minimal, non-empty subsets of S. The elements of a coterie are called quorums. Any node that wishes to enter into the critical section has to take permission for doing so from all the nodes comprising a quorum. It can be shown that this condition ensures mutual exclusion. The performance of a coterie can be measured in terms of its message complexity, delay, fault tolerance or availability. The fault tolerance of a coterie is the maximal number of node failures that the coterie can tolerate so that even after these failures at least one of its quorums is available. It is a worst case performance indicator. The objective of the present project is to firstly outline various approaches used to implement mutual exclusion in distributed systems. Next we proceed to study properties of coteries and finally present an algorithm to find the fault tolerance of a coterie. We also use the algorithm to find the fault tolerance of some well-known coteries.

CHAPTER 1
INTRODUCTION

1.1 Distributed Computing System

A distributed computing interconnects many autonomous computers to satisfy the information processing needs of modern software applications and business enterprises.

Umar [19] defines it as a collection of autonomous computers interconnected through a communication network to achieve business functions. Technically, the computers do not share main memory so that the information cannot be transferred through global variables. The information between the computers is exchanged only through messages over a network.

The absence of shared main memory and the requirement that all information passes only through messages is what distinguishes distributed computing systems from centralized/multi processor system. A system is centralized if its components are restricted to one site, 'decentralized' if its components are autonomous mechanisms with no or limited coordination, and distributed if its components are autonomous mechanism which also coordinate their operations through a global mechanism [8].

1.2 The Mutual Exclusion Problem

The mutual exclusion problem was originally considered in centralized systems for the synchronization of exclusive access to the shared resource. In this problem, concurrent access to a shared resource of the critical section of a process must be synchronized so that at any time only one process can access the critical section. Concurrent process may come in to picture in thee different contexts.

1.2.1 Multiple Applications

The management of multiple processes within a single processor system is called Multiprogramming. Most personal computers, workstations, single processor systems and modern operating systems for those computers support multiprogramming. It was invented to allow the processing time of the computer to be dynamically shared among a number of active jobs or applications. Concurrency is thus inherent in multiprogramming.

1.2.2 Structured Applications

As an extension of the principles of modular design and structured programming, some applications can be effectively implemented as a set of concurrent processes.

1.2.3 Operating System Structure

The same structuring advantage applies to the systems programmer and, in fact some of the operating systems are themselves implemented as a set of processes. The basic requirement for the support of concurrent processes is Mutual Exclusion.

1.3 Types of Concurrent Processes

The process concurrency can exist as a result of the multiprogramming of independent applications, of multiple process applications (including on distributed systems) and of the use of a multiple-process structure in the operating system. With these possibilities we can classify the processes into the following three different types, depending on the way processes interact with each other.

1.3.1 Process Unaware Of Each Other

These are independent processes that are not intended to work together. Multiprogramming of multiple independent processes, or processes running on distributed systems are examples of this type. These can be even batch jobs or interactive sessions, or a mix of both. Although, the processes are not working together, they may cause the problem of 'competition'. For example, two independent applications may both want to access the same disk or file or printer. These accesses need to be regulated.

1.3.2 Processes Indirectly Aware Of Each Other

These are processes that are not necessarily aware of each other by name but share access to some object, such as I/O buffer. Such processes exhibit 'cooperation' in sharing the common object.

1.3.3 Processes Directly Aware Of Each Other

These are processes that are able to communicate with each other by name and that are designed to work jointly on some activity. Many threads of a single process would fall in this category. Such processes also exhibit 'cooperation'. However as against the previous type of processes which cooperate by sharing these processes cooperate by communicating with each other.

1.4 Competition Among Processes For Resources

Concurrent processes come into conflict with each other when they are competing for the use of same resource. Two or more processes need to access a resource during the course of their execution. Each process is unaware of the existence of the other processes, and each is to be unaffected by the execution of the other processes. Thus each process must leave the state of any resource that it uses unaffected. Examples of resources include I/O devices, memory, processor time and the clock.

With competing processes, there is no exchange of information between them. However, the execution of one process may affect the behavior of other processes. In particular, if one resource gets allocated to a process, then the other one will have to wait. Therefore, the waiter process will be slowed down. In the most extreme case, the blocked process may never get access to the resource and hence will never terminate successfully.

In the case of competing processes, three control problems must be faced. First is the need of mutual exclusion. Supposing that two or more process require access to a non-sharable resource, such as a printer. During the course of execution, each process will be sending commands to the I/O device receiving status information, sending data, and/or receiving data. Such a resource is called a critical resource and the portion of the program that uses it as a critical section of the program. It is important that only one program at a time be allowed in its critical section. In the case of the printer, for example, we wish any individual process to have control of the printer while it prints an entire file. Otherwise, lines from competing processes will be interleaved.

The enforcement of mutual exclusion creates two additional problems. One is that of deadlock. It is a situation in which more than one process is in waiting state while holding on to some resources, and a process waits infinitely because the resources that it needs are never released by other waiting processes.

The third control problem is starvation. It is the situation of a process never gaining access to a resource because that resource so always being allocated to some other waiting process. Thus even though there is no deadlock, the process may be starved of the resource.

1.5 Cooperation Among Processes By Sharing

This case covers processes that interact with other processes without being explicitly aware of them. For example, multiple processes may have a success to a shared variable or to shared files or databases. Processes may use

and update the shared data without reference to other processes but know that other processes may have access to the same data. Thus, the processes must cooperate to ensure that the data they share are properly managed. The control mechanisms must ensure the integrity of the shared data.

Because data are held on resource (devices, memory etc.) the control problems of mutual exclusion, deadlock and starvation are again present. The only difference is that data items may be accessed in two different modes, reading and writing, and only writing operations must be mutually exclusive. However, over and above these problems a new requirement is introduced, that of data coherence. Raynal [14] gives example of two items of data a and b to be maintained in the relationship $a = b$. That is, any program that updates one value must also update the other to maintain the relationship. Next, we consider the following two processes.

P1 = $a := a+1;$

$b := b+1;$

P2 = $b := 2 * b;$

$a := 2 * a;$

If the state is initially consistent, each process taken separately will leave the shared data in a consistent state. However, if the two processes P1 and P2 respect mutual exclusion on each individual data item a and b, then the following concurrent execution results:

$a := a+1;$

$b := 2 * b;$

$b := b+1;$

$a := 2 * a;$

At the end of this execution sequence, the condition $a = b$ is no longer valid. The problem can be avoided by declaring the entire sequence in each process to be a critical section, even though strictly speaking, no critical resource is involved.

1.6 Cooperation Among Processes By Communication

In the previous two cases, each process has its own isolated environment that does not include the other processes. The interactions among processes are indirect. In both cases, there is sharing. In the case of competition, they are sharing resources without being aware of the other processes. In the second case, they are sharing values, and although each process is not explicitly aware of the other processes, it is aware of the need to main data integrity. When processes cooperate by communication, however, the various processes participate in a common effort that links at the process. The communication provides a way to synchronize, or coordinate the various activities.

Because nothing is shared between processes in the act of passing messages, mutual exclusion is not a control requirement for this sort of cooperation.

1.7 Requirements For Mutual Exclusion

The successful use of concurrency among processes requires the ability to define critical sections and enforce mutual exclusion. This is fundamental for any concurrent-processing scheme. Any facility or capability that is to provide support for mutual exclusion should meet the following requirements.

1. Mutual exclusion must be enforced. Only one process at a time is allowed into its critical section among all processes that have critical sections for the same resource or shared object.
2. A process that halts in its non-critical section must do so without interfering with other processes.
3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely; no starvation can be allowed.
4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay; i.e. deadlock should not be allowed.

5. No assumptions should be made about relative process speeds or number of processes.
6. A process remains inside its critical section for a finite time only.

There are a number of ways in which the requirements for mutual exclusion can be satisfied. Various approaches to implement mutual exclusion in a distributed environment are discussed in the next chapter.

CHAPTER 2
MUTUAL EXCLUSION IN DISTRIBUTED SYSTEMS

In a distributed environment, algorithms for mutual exclusion may be centralized or distributed. We start with the most obvious algorithm, which however is not the best available.

2.1 Centralized Algorithm With Control Node

In this algorithm, one node is designated as the control node and this node controls access to all shared objects. When any process requires access to a critical resource, it issues a request to its local resource - controlling process. This process in turn sends a request message to the control node, which returns a replay (permission) message when the shared object becomes available. When a process has finished with a resource, a release message is sent to the control node. Such a centralized algorithm has two key properties; Only the control node makes resource-allocation decisions.

All necessary information is concentrated in the control node, including the identity and location of all resources and the allocation status of each resource.

The centralized approach is straightforward, and it is easy to see how mutual exclusion is enforced. The control node will not satisfy a request for a resource until that resource has been released. However, such schemes suffer several drawbacks. If the control node fails, then the mutual exclusion mechanism brakes down, at least temporarily. Furthermore, every resource allocation and de-allocation requires an exchange of messages with the control node. Thus, the control node may become a bottleneck.

2.2 Characteristics Of A Fully Distributed Algorithm

Because of the problems associated with centralized algorithms, there has been ore interest in the development of distributed algorithms. Maekawa et. al [12] have identified the following properties which characterize a fully distributed algorithm:

1. All nodes have an equal amount of information on average.
2. Each node has only a partial picture of the total system and must make decisions based on this information.
3. All nodes bear equal responsibility for the final decision.
4. All nodes expend equal effort, o average, in effecting a final decision.
5. Failure of a node, in general, does not result in a total system collapse.
6. There exists no system wide common clock with which to regulate the timing of events.

Points 2 and 6 can be further elaborated. Some distributed algorithms require that all information known to any node be communicated to all other nodes. Even in this case at any given time, some of that information will be in transit and will not have arrive at all of the other nodes. Thus, because of time delays in message communication, a node's information is usually not completely up to date and is in that only partial information.

Because of the delay in communication among systems, it is impossible to maintain a system-wide clock that is instantly available to all systems. Furthermore it is also difficult to maintain one central clock and to keep all local clocks synchronized precisely to that central clock; over time, there will be some drift among the various local clocks that will cause a loss of synchronization.

2.3 Lamport's Time Stamping Scheme

Fundamental to the operation of most distributed algorithms for mutual exclusion and deadlock is the temporal ordering of events. The lack of a common clock or a means of synchronizing local clocks is thus a major constraint. The problem can be expressed in the following manner. We would like to be able to say that event a at system i occurred before (or after) event b at system j, and we would like to be able to consistently arrive at this conclusion at all systems in the network. Unfortunately, this statement is not precise for two reasons. First, there may be a delay between the actual occurrence of an event and the time that it is observed on some other system. Second, the lack of synchronization leads to a variance in clock readings on different systems.

To overcome these difficulties, a method referred to as time-stamping has been proposed by Lamport [9] that orders events in a distributed system without using physical clocks. This technique is so efficient and effective that many distributed mutual exclusion algorithms use it as well.

Lamport associated an event with messages. A local event can be bound to a message very simple; for example, a process can send a message when it desires to enter its critical section or when it is leaving its critical section. To avoid ambiguity, event may be associated with sending of messages only, not with the receipt of messages. Thus, each time that a process transmits a message, an event is defined that corresponds to the time that the message leaves the process.

The time stamping scheme is intended to order events consisting of the transmission of messages. Each system i in the distributed system maintains a local counter C_i which functions as a clock. Each time a system transmits a message, it first increments its clock by 1. The message is sent in the form: (m, T_i, i).

Where m = contents of the message

Ti = time stamp for this message, set to equal Ci.

i = Numerical identifier of this site.

When a message is received the receiving system j sets its clock to one more than the maximum of its current value and the incoming time stamp:

$Cj = 1 + \max [Cj, Ti]$.

At each site, the ordering of events is determined by the following rules. For messages x from site i and y from site j, x is said to precede y if one of the following conditions holds:

if $Ti < Tj$, or

if $Ti = Tj$ and $i < j$.

The time associated with each message event is the time stamp accompanying the message, and the ordering of these times is determined by the preceding two rules. That is, two message events with the same time stamp are ordered by the number of their sties. Because the application of these rules is independent of site, this approach avoids any problems of drift among the various clocks of the communicating process.

The ordering imposed by this scheme does not necessarily correspond to the actual time sequence. However, for the algorithms based on this scheme, it is not important which event actually happened first. The only important fact is that all processes that implement the algorithm agree on the ordering that is imposed on the events.

2.4 Lamport's Algorithm (Using Distributed Queue)

This algorithm, which is based on the time-stamping scheme, is based on the following assumptions:

1. A distributed system consists of N nodes, uniquely numbered from 1 to N. Each node contains one process that makes requests for mutually exclusive access t resources on behalf of the processes; this process also serves as an arbitrator to resolve incoming requests that overlap in time.

2. Messages sent form one process to another are received in the same order in which they are sent.
3. Every message is correctly delivered to its destination in a finite amount of time.
4. The network is fully connected i.e. any process can send messages to any other process directly.

The algorithm attempts to generalize an algorithm which would work in a straight-forward manner in a centralized system. If a single central process manage the resource, it could queue incoming requests a grant request in a first in, first out manner. To achieve this same algorithm in a distributed system, all the sites must have a copy of the same queues. Time stamping is used to assure that all sites agree on the order in which resource requests are to be granted. Because it takes some finite amount of time for message to transit a network, there is a danger that two different sites will not agree on which process is at the head of the queue. This could lead t a failure of the algorithm. To avoid such problems, the following rule is imposed: for a process to make an allocation decision that is based on its own queue, it needs to have received a message from all other sites that guarantees that no message earlier than its own head-of-queue message is still in transit.

At each site, a data structure is maintains that keeps a record of the most recent message received from each site. At any instant, entry q[j] in the local queue contains a message from Pj. The queue is initialized as follows:

q[j] = (Release, 0, j) j = 1 ....., N

Three types of messages are used in this algorithm:

- (Request, Ti, i) : A request for access to a resource is made by Pi.
- (Reply, Tj, j) : Pj grants access to a resource under its control.
- (Release, Tk, k) : Pk releases a resource previously allocated to it.

The algorithm is as follows:

1. When P_i requires access to a resource, it issues a request (Request, T_i, i), time stamped with the current local clock value. It puts this message in it own local queue at q[j] and sends the message to all other processes.
2. When P_j receives (Request, T_i, i), it puts this message in its own queue at q[i] and transmits (Reply, T_j, j) to all other processes. It is this action, that implements the rule described earlier, that assures that no earlier Request message is in transit at the time of a decision.
3. P_i can access a resource (enter its critical section) when both of the following conditions hold:
   (i) Pi's own Request message in queue q is the earliest Request message in the array; because messages are consistently ordered at all sites, this rule permits one and only one process to access a resources at any instant.
   (ii) All other messages in the local queue are later than the message in q[i]; this rule guarantees that P_i has learned about all requests that preceded its current request.
4. P_i releases a resource by issuing a release (Release, T_i, i) which it puts is its own queue and transmits to all other processes.
5. When P_i receives (Release, T_j, j) it replaces the current contents of q[j] wit this message.
6. When P_i receives (Reply, T_j, j), it replaces the current contents of q[j] with this message.

It can be easily shown that this algorithm enforces mutual exclusion, is fair, avoids deadlock and avoids starvation.

2.5 A Token Passing Scheme Based On Time Stamping

A number of investigators have proposed a quite different approach to mutual exclusion that involves passing a token among the participating processes. The token is an entity that at any time is held by one process. The

process holding the token may enter its critical section without asking permission. When a process leaves its critical section, it passes the token to another process.

Suzuki, I. and Kasami, T. [18] proposed an algorithm based on token passing scheme based on time stamping scheme of Lamport. For this algorithm two data structures are needed. The token, which is passed from process to process, is actually an array, token, whose K*th* element records the time-stamp of the last time that the token visited process P*k*. In addition, each process maintains an array, request, whose j*th* element records the time-stamp of the last Request received from P*j*.

Initially, the token is assigned arbitrarily to one of the processes by setting token-present to true for that process. When a process wishes to use its critical section, it may do so if it currently possesses the token; otherwise, it broadcasts a time-stamped request message to all other processes and waits until it receives the token. When a process P*i* leaves its critical section, it must transmit the token to some other process. It chooses the next process to receive the token by searching the request array in the order i+1, i+2, . . ., 1,2,. . ., i-j for the first array entry request [*j*] such that the time-stamp for P*j*'s last request for the token is greater than the value recorded in the token for P*j*'s last holding of the token request [*j*] > token [*j*].

The algorithm requires either of the following;

- N messages (N-1 to broadcast the request and 1 to transfer the token) when the token is not held by the requesting token.
- No message if the process already holds the token.

Over the last decade, the problem of mutual exclusion has received considerable attention and several algorithms to achieve mutual exclusion in distributed systems have been proposed. The next chapter presents a taxonomy of distributed mutual exclusion algorithms.

CHAPTER 3
A TAXONOMY OF DISTRIBUTED MUTUAL EXCLUSION

3.1 Background

3.1.1 System Model

The distributed system we consider consists of N geographically distributed sites (S*1*,S*2*,...,S*N*) which are connected by a communication network. The sites do not share a memory and communicate solely by message passing. Thus, we exclude shared memory systems. Unless specifically mentioned, the underlying network is assumed to be logically fully connected (i.e. each site can communicate directly with every other site), to be reliable, and to deliver messages in finite time. For the ease of illustration, we assume that there is only one critical section in the system.

3.1.2 Token vs Non-Token Dichotomy

Mutual exclusion algorithms have employed two approaches to achieve mutual exclusion and can be divided into two broad classes: token-based and non-token-based. A hybrid algorithm combines the techniques of two algorithms.

In *token-based* algorithms, a unique token is shared among the sites. A site is allowed to enter its critical section if it possesses the token. Therefore, difficult tasks are (i) to insure fair scheduling of token among competing sites without excessive message overhead and (ii) to detect loss of the token and regenerate a unique token.

Non-token-based algorithms require one or more successive rounds of message exchanges among the sites to obtain the permission to execute critical section. In these algorithms, a site enters its critical section only after an

assertion defined on its local variables becomes true. Mutual exclusion is achieved because the assertion becomes true only at one site any given time.

There is an orthogonal classification of mutual exclusion algorithms: static and dynamic. A mutual exclusion algorithm is static if its actions do not depend upon the system state (or history). The actions of a dynamic mutual exclusion algorithm are influenced by how the system has evolved.

3.1.3 Performance Measures

The performance of mutual exclusion algorithms is generally measured by the following three metrics:

(i) *Message complexity,* The number of messages necessary per critical section invocation.

(ii) *synchronization delay*, which is the number of sequential message exchanges required after a site leaves the critical section and before next site enters the critical section, and

(iii) *response time*, which is the time interval a request waits to execute critical section after its request messages have been sent out.

Performance of a mutual exclusion algorithm depends upon loading conditions of the system and has been studied under two special loading conditions, viz., "*low load*" and "*high load*". In *low load* conditions, there is seldom more than one request for mutual exclusion simultaneously in the system. Consequently, there is hardly any interference among the requests. In *heavy load* conditions, there is always a pending request for mutual exclusion at a site. Thus, after having executed a request, a site immediately initiates activities to enable the execution of next request. The notations used in expressing the performance of mutual exclusion algorithms are as follows:

N = total no of sites in the system

T = average message propagation delay

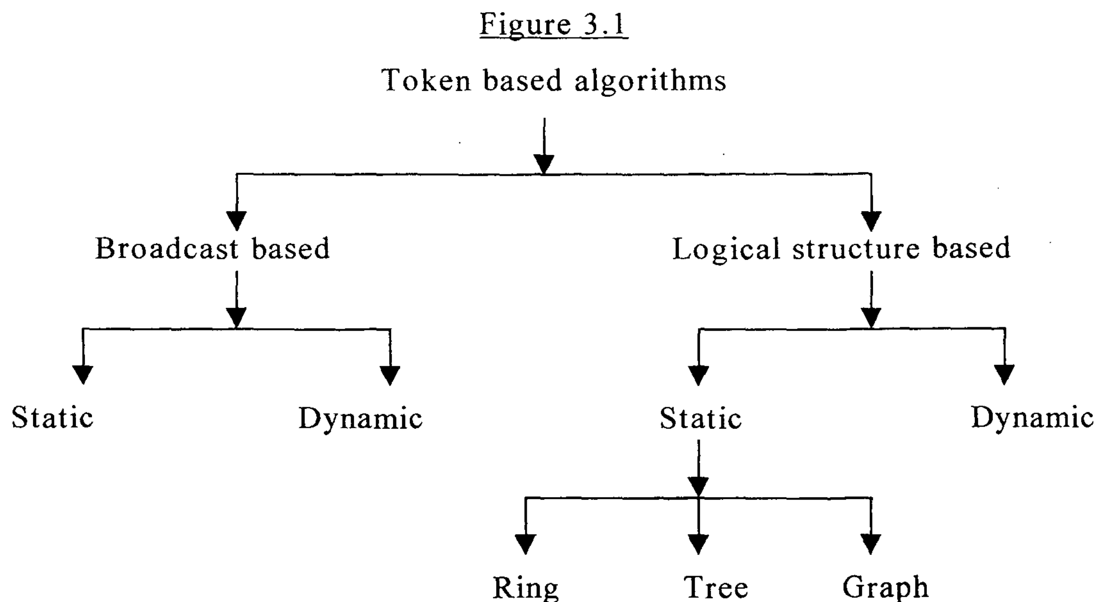
E = average critical section execution time

When the value of a performance metric has statistical fluctuations, we generally talk about the average value of that metric.

3.2 A Classification Of Token-Based Algorithms

Figure 3.1 depicts a classification of token-based algorithms. Two major categories of are broadcast-based and logical-structure-based.

Figure 3.1



3.2.1 Broadcast-Based Algorithms

In broadcast-based mutual exclusion algorithms, no structure is imposed on sites and a site sends token request messages to other sites in parallel. Broadcast-based token-based algorithms are divided into two classes: static and dynamic. Static algorithms are memory-less because they do not remember the history of critical section executions. In these algorithms, a requesting site sends token requests to *all* sites.

Dynamic algorithms, on the other hand, remember a recent history of the token locations and send token request messages only to dynamically

selected sites which are likely to have the token. An important consideration in these algorithms is that a requesting site must send a token request to a site which either has the token or is going to get it in near future. For example, in Singhal's algorithm [17], data structures at sites are initialized and updated such that for any two sites, when any one of them requests the token, that site does send a request message to the other site. Thus, no site is isolated from another site. The key idea behind the algorithm is that a site's request reaches a site which has the token even though request messages are not sent all sites.

Performance:

Broadcast based mutual exclusion algorithms are fast (have low delays) because of parallelism in message transfer. When the load is low, response time is equal to a round-trip message delays (27). (Strictly speaking, it is 2(1-1/N) T because with probability 1/N, a requesting site will have the token and thus, will have zero response time). As the load increases, interference due to concurrent requests increases causing the response time to rise. (A requesting site gets the token after a number of other sites have executed critical section). At high loads, on the average all other sites execute their critical section between two successive executions of the critical section by a site and response time asymptotically converges to N*(T+E).

Broadcast based algorithms have high message traffic: Static algorithms require N messages per critical section execution. Dynamic algorithms on the average require N/2 messages per critical section execution at low loads and this number monotonically reaches to N at high loads.

3.2.2 Logical-Structure Based Algorithms

In logical-structure-based mutual exclusion algorithms, sites are weaved into a logical configuration. These mutual exclusion algorithms can be static or dynamic.

Static Algorithms:

In static algorithms, the logical structure remains unchanged and only the direction of edge changes. Static algorithms have used three logical configurations, namely, tree, graph and ring. In ring based algorithms, sites are arranged in a ring fashion. A token circulates on the ring serially from site to site. A site must wait to capture the token before entering its critical section. In tree-based algorithms, sites are arranged as a directed tree whose root site holds the token. A request for token propagates serially in the tree from a requester node to the root node. Token is passed serially from root to the requester and as token traverses the edges, the direction of the edges is reversed so that the requester becomes the new root of the directed tree.
In graph-based algorithms, sites are arranged as a directed graph with a sink node which holds the token. Requests for token and token propagation are handled in the same way as in tree-based algorithms. An advantage of graph-based algorithms is that they are fault-tolerant to communication link and sites failures (because a graph has multiple paths between nodes while a tree does not). However, a cost for this fault-tolerance is increased message traffic because additional messages need to be exchanged to prevent cycles in the graph structures.

Dynamic algorithms:

In dynamic algorithms, the logical structure changes as sites request and execute the critical section. The shape of the structure, relative position of sites in the structure, and connectivity all undergo changes. An interesting feature of the dynamic logical structure algorithms is that the logical structure can remember the history of critical section execution (i.e. recent token locations) and can adjust to maximize the performance.

Performance:

Tree-and graph-based algorithms generate low message traffic. This is because the average diameter of randomly generated trees of N nodes is "log N". A randomly constructed tree typically has a diameter on O(log N). (The

diameter is the maximum distance between any two nodes in the tree). Thus, on the average in log N messages, a token request will reach the root node. In structure-based algorithms, token request message propagation is highly focused towards the site which has the token. (The logical structure helps guide the token request messages. However, these algorithms suffer from large delays because a token request message must propagate serially from a requester to the root. At low loads, the average response time is T*log N. At low loads, the average response time is N*(T+E).

Structure-based algorithms have very good performance in message traffic at high loads. This is because when a site receives a token request message when that site itself is requesting the token, it blocks the further propagation of the received request (because that site has already sent out a request message). This type of sharing of request messages, which is very frequent at high loads, considerably reduces the message traffic.

3.3 Non-Token Based Algorithms

In non-token-based algorithms, a site can be in one of the following three states: requesting the critical section, executing the critical section, or idling (i.e. neither requesting nor executing the critical section). A request for the critical section is assigned a timestamp which is generated according to Lamport's scheme [9]. Timestamps of requests are used to prioritize requests and to resolve conflicts among concurrent requests.

The concept of information structures has been introduced to unify different non-token based mutual exclusion algorithms in distributed systems. We organize discussion of non-token-based algorithms in the following way: we first discuss the information structures and a generalized non-token-based mutual exclusion algorithm. We then show how several known non-token-based algorithms are special cases of this algorithm and classify these algorithms.

3.3.1 The Information Structure Of Non-Token Based Algorithms

The information structure of a non-token-based mutual exclusion algorithm defines the data structure needed at a site to record the status of other sites. The information structure at a site S_i consists of the following three sets

(i) request set R_i,

(ii) inform set I_i, and

(iii) status set St_i.

These sets consist of the identification numbers of the sites of the system. A site must acquire permission from all the sites in its request set before entering the critical section. When the state of site changes (due to wait to enter the critical section or due to exit from the critical section), it informs about the change to all sites in its inform set. The status set S_i contains the sites about which Si maintains status information. Note that $\forall i::S_i \in I_j \Rightarrow S_j \in St_i$.

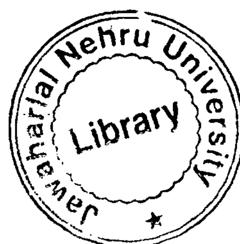
TH-9445

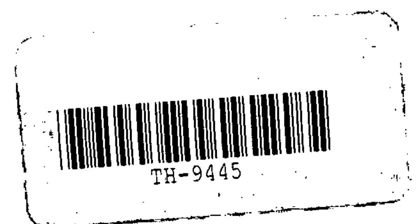
3.3.2 The Generalized Non-token Based Algorithm

A site maintains a variable which indicates that site's knowledge of the status of the critical section. To execute the critical section, a site takes the following actions: The site sends time-stamped REQUEST messages to all the sites in its request set. The site executes the critical section only after it has received a GRANT message from all the sites in its request set. On exit from the critical section, the site sends a RELEASE message to every site in its inform set.

Every site maintains a queue which contains REQUEST messages, in the order of their timestamps, for which no GRANT message has been sent.

- On the receipt of a REQUEST message, a site takes the following actions: Place the REQUEST on its queue. If the state variable indicates that critical section is free, then send a GRANT message to the site at the top





of the queue and remove its entry from the queue. If the recipient of the GRANT message is in its status set, then set the state variable to indicate that the recipient site is in the critical section.

- On the receipt of a RELEASE message, a site takes the following actions: state variable is set to free. If the queue is non-empty, then send a GRANT message to the site at the top of the queue and remove its entry from the queue. If the recipient of the GRANT message is in its status set, then set the state variable to indicate that the recipient site is in, the critical section. Repeat the last two steps until the state variable indicates that a site is in the critical section or the queue becomes empty.

Correctness Condition

If $\forall i :: I_i \subseteq R_i)$ (1)

$(\forall i \forall j :: (I_i \cap I_j \neq \phi) \vee (S_i \in R_j \wedge S_j \in R_i)$ (2)

The correctness condition states that for every pair of sites, either they request each other's permission or they request permission of a common site (which maintains the status information of both and arbitrates conflicts between them).

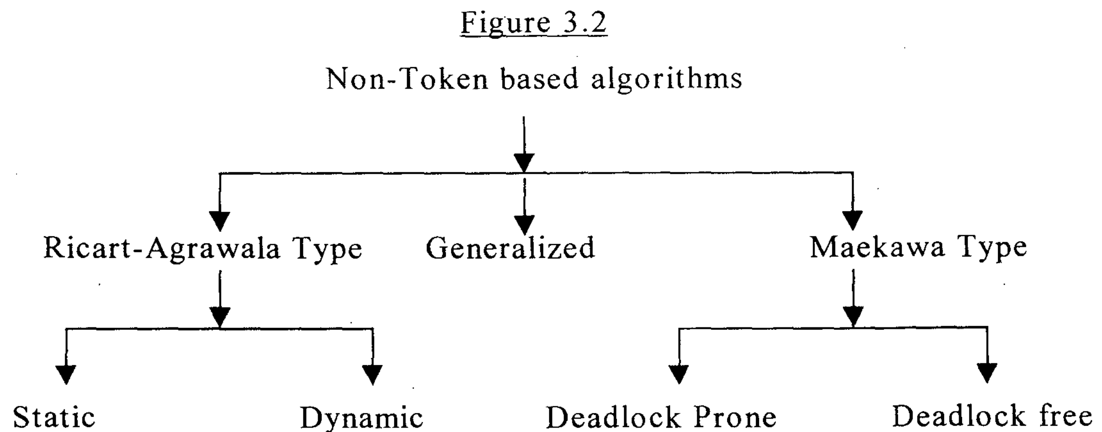
3.3.3 Static vs. Dynamic Information Structures

In static information structure algorithms, the contents of request sets remain fixed and do not change as the sites execute critical section. In dynamic information structure algorithms, the contents of these sets change as the sites execute the critical section. Design of dynamic information-structure mutual exclusion algorithms is much more complex because it requires coming up with rules for updating the information structures such that correctness conditions (1) and (2) for mutual exclusion are satisfied dynamically.

3.4 A Classification Of Non-Token Based Algorithms

Figure 3.2 shows a classification of non-token-based algorithms. In non-token-based algorithms, a site exchanges one or more successive rounds of messages among the sites to obtain their permissions to execute the critical section. A permission is a consent to enter the critical section and plays an important role in enforcing mutual exclusion.

Figure 3.2



3.4.1 Ricart Agrawala Type Permission

In Ricart Agrawala-type permission, a site grants permission to a requesting site immediately if it is not requesting the critical section or its own request has lower priority. Otherwise, it defers granting permission until its execution of the critical section is over. Semantics of Ricart Agrawala-type permission is: "*As far as I am concerned, it is OK for you to enter the critical section.*" That is, while granting a permission, a site looks into its own conflict. A site can grant permission to many requesting sites simultaneously.[16]

3.4.2 Maekawa-Type Permission

In Maekawa-type permission, on the contrary, a site can grant permission only to one site at a time. A site grants permission t a requesting site only if it has not currently granted permission to another site. Otherwise, it delays ranting permission until the currently granted permission has been released. Thus, acquiring permission is like locking the site in "exclusive' mode. Semantics of Maekawa-type permission is "As far as all the sites in my status set are concerned, it is OK for you at enter the critical section". By granting permission to a site, the site guarantees that no other sites in its status set can be executing the critical section concurrently.[11]

3.4.3 Two Special Cases Of The Generalized Algorithm

The generalized algorithm combines the strategies of Ricart-Agrawala's [16] and Maekawa's [11] algorithms. In a generalized algorithm, a site S, acquires Maekawa-type permission from all the sites in its in form set I, (since $I \subseteq R_i$ for correctness) and acquires Ricart-Agrawala-type permission from all the sites in set $R_i - I_i$. A site sends Maekawa-type permission to sites in its status set and sends Ricart-Agrawala-type permission to all other sites. Once a site has granted permission to a site in its status set, it cannot grant permission to any other site unless the previous permission has been released.

If the first predicate of condition (2) is false for all *i* and *j*, then the resulting algorithm is Ricart-Agrawala-type. If the second predicate of condition (ii) is false for all i and j, then the resulting algorithm is Maekawa-type.

3.4.4 Ricart-Agrawala Type Algorithms

In Ricart-Agrawala type algorithms, the request sets of sites are formed satisfying the following two conditions:

$\forall i :: S_i \varepsilon I_i$ (3)

$\forall i \forall j :: (S_i \varepsilon R_j) \wedge (S_j \varepsilon R_i)$ (4)

A site requests Ricart-Agrawala-type permission from all the sits in its request set. Note that condition (4) implies that $\forall S_i :: R_i = \{S_1, S_2, \ldots S_N\}$. This assignment of request sets results in the classical Ricart-Agrawala algorithm. The Ricart-Agrawala algorithm is static because the contents of the request sets do not change as the algorithm executes.

Performance:

Ricart-Agrawala-type mutual exclusion algorithms are fast (have low delays) because of parallelism in transfer of request and reply messages. When load is low, response time is equal to a round-trip message delays (2T). As load increases, interference due to concurrent requests increases causing response time to rise. At high loads, on the average all other sites execute their critical sections between two successive executions of the critical section by a site and response time asymptotically converges to N*(T+E).

Ricart-Agrawala-type mutual exclusion algorithms, however, have high message traffic: Ricart-Agrawala algorithm requires 2*(N-1) messages per critical section execution.

3.4.5 Maekawa-Type Algorithms

In Maekawa-type mutual exclusion algorithms, the request set, R_i, of a site S_i is identical to its inform set, Ii and the request sets of sites satisfy the following conditions:

$\forall i :: S_i \subseteq R_i$ (5)

$\forall i \forall j :: R_i \cap R_j \neq \phi$ (6)

A site S_i executes its critical section only after it has acquired Maekawa-type permission from all the sites in its request set. Condition (5) and (6) define a class of mutual exclusion algorithms referred to as Maekawa-type mutual exclusion algorithms.

3.4.6 A Centralized Mutual Exclusion Algorithm

In the centralized mutual exclusion algorithm, the request sets satisfy the following condition, in place of condition (6).

$\forall i \forall j :: R_i \cap R_j = \{S_k\}$ (8)

That is, a single site (here S_k) acts as the sole arbitrator to resolve conflicts among all the sites. This algorithm is highly skewed as far as symmetry (distribution of responsibility) is concerned.

3.4.7 A fully Distributed Mutual Exclusion Algorithm

Another extreme of Maekawa-type algorithms is the fully distributed mutual exclusion algorithm, where a site requests Maekawa-type permission from all other sites:

$\forall S_i :: R_i = \{S_1, S_2, \ldots S_N\}$.

This algorithm is impractical because of excessive message traffic and is covered for completeness.

3.4.8 Agrawal-Abbadi Algorithm

In the Agrawal-Abbadi algorithm, request sets are formed based on a binary tree configuration of system sites. Sites are logically organized as a binary-tree and the request set of a site contains all the sites on a path from root to the leaf, containing that node/site. Thus, the size of the request sets is O(log N) and the message complexity of the algorithm is O(log N). However, the algorithm violates the spirit of "equal responsibility" because some sites appear in more request sets than others. The algorithm achieves fault-tolerance to site failures and network partitioning by assigning multiple request sets to sites -one set to be used in a particular failure.

3.4.9 Deadlock Problem in Maekawa-Type Algorithms

Maekawa type algorithms are in general prone to deadlocks because other sites exclusively lock a site and lock requests are not prioritized by their

timestamps. Maekawa-type algorithms handle deadlocks by requiring a site to yield a lock if the timestamp of its request is larger than the timestamp of some other request waiting for the same lock (unless the former has succeeded in locking all the needed sites). A site suspects a deadlock (and initiates message exchanges to resolve it) whenever a higher priority request finds that a lower priority request has already locked the site. Maekawa-type algorithms require exchange of INQUIRE, FAILED and YIELD message whose sole purpose is to break deadlocks. Thus, Maekawa-type algorithms require extra messages even though there is no deadlock. The maximum number of message in this case is 5* [R*i*].

Performance

Maekawa-type algorithms have low message complexity. At low loads, site S*i* exchanges 63*[R*i*] messages per critical section execution. As load increases, the number of messages transferred to handle deadlocks increases and at high loads message complexity becomes 5*[R*i*]. However, reduced message complexity comes at a cost of higher synchronization delays. The synchronization delays in Maekawa-type algorithms are 2T as opposed to T in Ricart-Agarwala type algorithms. This is because in Maekawa-type algorithms, a site exiting the critical section must first unlock the arbiter site which in turn sends a GRANT message to the next site to enter the critical section. (Two serial message delays between the exit of the critical section by a site and enter into the critical section by the next site.

3.4.10 Deadlock-Free Maekawa-Type Algorithms

Deadlocks free Maekawa-type algorithms are free from deadlocks in the sense they do not require phases of INQUIRE, FAILED and YIELD message exchanges to detect and recover from deadlocks. The primary reason Maekawa-type algorithms are prone to deadlocks is that REQUEST messages may arrive at a site in wrong order. It may happen that a site has already been

locked by a lower priority request when a higher priority request arrives at it. Such condition is called an antagonistic conflict.

In deadlock free Maekawa-type algorithms the problem of antagonistic conflicts is solved by having the higher priority request convert the exclusive lock into a shared lock (i.e. locks on sites are mutable). A request succeeds in locking a site if its priority is higher than the priority of all the requests which currently hold the lock on that site. By making the locks mutable, essentially the same effect is achieved that Maekawa's algorithm achieves by pre-empting the lock from a site and granting it to a higher priority site.

Since a site can be locked by several sites concurrently (due to mutable locks), request sets must satisfy the following stronger condition for mutual exclusion: (condition $Ri \cap Ri \neq \phi$ is not strong enough).

$$\forall i \forall j : i \neq j :: Si \in Rj \vee Sj \in Ri \qquad (9)$$

Condition (9) essentially states that the amount of communication among the sites should be increased. This condition defines a class of deadlock free mutual exclusion algorithms because a large number of algorithms can be formed satisfying it. Next, we discuss examples of deadlock free Maekawa type mutual exclusion algorithms.

3.4.11 Lamport's Algorithm

Well-known mutual exclusion algorithm of Lamport is a fully distributed version of deadlock free Maekawa type mutual exclusion algorithms. In Lamport's algorithm, the request sets of sites satisfy the following conditions:

$$\forall Si :: Ri = \{S1, S2, \ldots SN\}$$

Compared with the fully distributed Maekawa type algorithm, the fully distributed Maekawa type algorithm is deadlock prone and thus can require up to 5*(N - 1) messages per critical section execution.

3.5 Hybrid Algorithm

In general, there is a trade off between the speed and message complexity of distributed mutual exclusion algorithms. No single mutual exclusion algorithm can optimize both the speed and the message complexity. Concept of hybrid mutual exclusion algorithms has been purported to simultaneously minimize both the time delay and the message complexity. Hybrid algorithms are capable of combining the advantages of two mutual exclusion algorithms and offer potential of providing improved performance over an extended range of loads.

Sites in the system are divided among several groups and two different mutual exclusion algorithms are used to resolve intra-group and inter-group conflicts. Sites use one algorithm to resolve conflicts with sites in the same group and use a different algorithm to resolve conflicts with sites in different groups. By carefully controlling the interaction between the local and the global algorithms, one can minimize both message traffic and synchronization delay simultaneously.

CHAPTER 4

COTERIES AND THEIR PROPERTIES

4.1 Introduction

In many distributed systems it is necessary to have a mutual exclusion mechanism that works even when nodes fail or the communication lines are broken. For example, consider a system that manages replicated data. Owing to a network partition, the system maybe divided into isolated groups of nodes. It is not wanted that users at isolated groups update the database concurrently since this would cause the copies to diverge. So, if a group is going to perform updates, it must be able to guarantee that no other group is performing this activity. This mutual exclusion has to be enforced without communication between groups.

One well-known solution is to assign a priori a number of votes (or points) to each node in the system, and a group whose members have a majority of the total votes is allowed to perform the restricted operation. The mutual exclusion is achieved because at most a single group can have a majority of votes at a time. (It is possible that at a given time no group has a majority and can perform the operation. There seems to be no way to avoid this problem. Even giving one node all votes does not help since that node may fail).

Votes are used to achieve mutual exclusion in a number of other algorithms. For example, in the so-called Byzantine Generals problems, nodes may fail and yield incorrect or even misleading results. The computation being performed must be replicated, and if nodes with a majority of votes agree on a result, it is considered correct. Votes are also used in some commit protocols. After a failure, nodes must decide whether to commit or abort a transaction,

and the protocol must ensure that at most one group of nodes makes such a decision.

Lamport suggested a second solution to the mutual exclusion problem. The idea is to define a priori a set of groups that may perform the restricted operation. Each pair of groups should have a node in common to guarantee mutual exclusion. For example, if we have nodes a, b and c we may defined the set $\{\{a,b\}, \{b,c\}, \{a,c\}\}$. Nodes a and b can perform the operation together, knowing that neither group $\{b,c\}$ or $\{a,c\}$ can be formed notice that this set of groups is equivalent to assignment one vote to each node (for n votes to each node).

The assignment of votes or the choice of set of groups can have a critical effect on the reliability of a distributed system. Consider, for example, a system with nodes a, b, c, d and an assignment that gives one vote to each this seems like a natural choice because it gives each node equal weight. Since three votes are needed for a majority, this is equivalent to the set of groups.

$$S = \{\{a,b,c\}, \{a,b,d\}, \{a,d\}, \{b,c,d\}\}.$$

But now, consider an assignment that gives node a two votes and the rest a single vote. The majority is still three votes, so this is equivalent to:

$$R = \{\{a,b\}, \{a,c\}, \{a,d\}, \{b,c,d\}\}$$

Set R and its associated vote assignment is clearly superior to S because all groups of nodes that can operate under S can operate under R, but not vice versa. For instance, a and b can form a group under R but not under S. So, if the system splits into groups $\{a,b\}$ and $\{c,d\}$, there will be one active group under R but none under S. So clearly, no system designer should ever select set S (or its equivalent vote assignment), in spite of the fact it seems "natural".

The term {R dominates S} means that R is always superior to S. Obviously, dominated sets are to be ignored. But this is not an easy task. In the previous example, e.g., which of the nodes should get the two votes? There are many choices, but many are duplicates. For example, giving a four

votes, *b* three votes and *c* and *d* two each, yields exactly the same set of groups that was given by R. So again, in the selection process, duplicate vote assignments are to be ignored.

Once the number of choices is narrowed down, the system designer will have to consider each one in light of the failure characteristics of the system. The set or assignment that maximizes the probability that the system is in operation would be selected and used in practice.

4.2 Definitions

According to Webster's dictionary a coterie is a 'close circle of friends who share a common interest... a set refers t a group, usually larger and, hence, less exclusive than a coterie". Hector Garcia-Molena and Daniel Barbara introduced coteries as a mathematical tool to implement distributed mutual exclusion. If we refer to a set of nodes as a group, then coteries are a set of groups (i.e. set of set of nodes) that are 'well formed". The exact definition can be stated as:

Definition 4.1 : Coterie

Let U be the set of nodes that compose the system. A set of groups S is coterie under U, if and only if $G \in S$ IMPLIES THAT $G \neq \phi$ AND $G \subseteq U$.

- (Intersection property) if G, H $\in$ S, then G and H must have atleast one common node i.e. $G \cap H \neq \phi$
- (Minimality) There are no G, H $\in$ S such that $G \subset H$.

When U is understood by the content, then it is dropped from the discussion.

It is not necessary that all nodes appear in a coterie. For instance $\{\{ a \}\}$ is a coterie under $\{ a, b, c\}$.

Some more definitions and properties related to coteries are now stated. On the basis of these definitions, we state some theorems on coteries in the next section.

Definition 4.2: Domination for Coteries:

Let R, S be coteries (under U). R dominates S if $R \neq S$ and, for each $H \in S$, there is a $G \in R$ such that $G \subseteq H$. (We say that G is the group that dominates H.)

Definition 4.3 : Dominated and nondominated Coteries:

A coterie S (under U) is dominated iff there is another coterie (under U), which dominates S. If there is no such coterie, then S is nondominated (ND).

Definition 4.4: Hypergraph:

Let $X = [X_1, \ldots. X_n]$ be a finite set and $E = (E_1 \backslash i=1\ldots..m)$ a family of subsets of X. If $E_1 \neq \phi$ $(I = 1\ldots.m)$ and $U_1\ E_1 \subseteq X$,[1] the couple $H = (X, E)$ is called a hypergraph. The value $[X] = n$ is the order of the hypergraph, the element $X_1,\ldots.X_n$ are called the vertices, and the set $E_1\ldots, E_m$ are called the hyperedges.

Definition 4.5: Transversal:

A transversal of a hypergraph $H = (X: E_1,\ldots. E_m)$ is defined to be a set $T \subset X$ such that $T \cap E_1 \neq \phi$ for $(I = 1, \ldots. M)$. A minimal transversal is a transversal such that no proper subset of it is a transversal the same concept is applicable to coteries.

Definition 4.6: Coloring of a Hypergraph:

A coloring of a hypergraph is a coloring of the nodes so every hyperedge has at least two colors.

Definition 4.7: Chromatic Number of a Hypergraph:

The chromatic number is defined to be the smallest number of colors needed for a hypergraph coloring. A hypergraph for which there exists a k coloring is said to be k-colorable.

Definition 4.8: Critical Hypergraph:

An edge E of a hypergraph H is critical if the chromatic number of H – [E] is less than the chromatic number of H, that is, if deleting the hyperedge reduces the chromatic number, A hypergraph is critical if it is connected and each edge of its is critical.

4.3 Properties of coteries

Wherever possible we should not use a dominated coterie because there is a coterie that provides more protection against partitions.
For instance, the coterie{{a, b, c}, {c,d,e } } should be replaced by { c}, and the coterie { { a,b}, { b,c} } should be replaced by {{a,b},{ b,c}, { c,a} } or by { { b} }.

Theorem 4.1:

Let S be a coterie under U, S is dominated iff there exists a group $G \subseteq U$ such that

(i) G is not a superset of any group in S.

(ii) G has the intersection property, i.e., for all $H \in S$, $G \cap H \neq \phi$.

Proof:

First we show that condition (i) and (ii) imply S is dominated. There are two cases to consider. If there are one or more $H_1, H_2, \ldots H_n$, then construct set $R = (S - H_1 - H_2 - \ldots H_n) \in S$ such that $G \subset H_1, H_2, \ldots H_n$, then construct set $R = (S - H_1 - H_2 - \ldots - H_n) \cup G$ is the dominating coterie.

Now, assume that S is dominated by coterie R. We show that conditions (i) and (ii) hold by considering two cases. In the first case, $S \subset R$. Let G be

one of the elements in R – S. Set G must satisfy conditions (i) and (ii) or else R would not be a coterie. For the second case, $S \not\subset R$ and there must be an H $\in$ S and a G $\in$ R such that $G \subset H$. If condition (i) is false for G, then $G \supseteq H'$ for some H' $\in$ S and S is not a coterie because $H \supset G \supseteq H'$. Similarly, if condition (ii) does not hold for G, R would not be a coterie. (If H' $\in$ S and $H' \cap G' = \phi$, then $G \cap G = \phi$, where G' is the group in R that dominates H'.) So in either case, the conditions hold.

Checking domination of coteries seems to be a hard problem. The best algorithm that known at this point is the one suggested by previous theorem. It generates all the possible subsets of the universe of n nodes, and for each one, checks if it can be added to the coterie. This algorithm is clearly exponential in n.

Theorem 4.2:

The maximum number of groups in a coterie under a universe of n element is bounded by 2^{n-1}.

Proof:

Since all the groups in the coterie must intersect with each other, no group and its complement may be present. Thus, a coterie can have almost half of the possible subsets of the set of n elements.

Theorem 4.3:

There are coteries that have an exponential number of groups on n.

Proof:

Consider the coteries with groups of size [(n+1)/2]. There are $[\,^{n}C_{(n+1)/2}]$ such groups. Using the definition of combinations, it can be proved that $[\,^{n}C_{(n+1)/2}] > 2^{n}/n$.

As an example of the coteries discussed in the above theorem, we have

R = {{a, b, c}, {a, b, d}, {a, b, e}, {a, c, d}, {a, c, e}.

{a, d, e}, {b, c, d}, {b, c, e}, {b, d, e}, {c, d, e}}
which consists of all the groups of size 3 in a universe of five nodes.

We now establish the relationship between coteries and combinatorial object called hypergraph. This connection proves to be useful in showing some properties of coteries and in establishing another characterization of ND-Coteries.

A coterie can be viewed as hypergraph where the coterie groups are the hyperedges. However, not all hpergraphs represent coteries, since clearly the properties of intersection and minimality must be present.

Theorem 4.4

A coterie is dominated iff the corresponding hypergraph is 2-colorable.

Proof:

If a coterie S is dominated then according to Theorem 4.1 there exists a group G not in S that is not a superset of any group in S and that has the intersection property and, therefore, that can be added to S. It is enough to view G as one color class and the complement of it as the other.
Recognizing 2-colorable hypergraphs is known to be NP-complete, which reinforces the belief that the same is true for coterie domination. However, since coteries are special hypergraphs, this result cannot be directly extended to the problem of coterie domination.

Having seen the properties of coteries we now present an algorithm to find the fault tolerance of a coterie in next chapter.

CHAPTER 5
A FAULT TOLERANCE ALGORITHM

The fault tolerance of a coterie is a measure of quality of the coterie. It is the maximal number of node failures that the coterie can tolerate so that even after these failures at least one of its quorums is available. It is a worst case indicator. Before stating an algorithm for fault tolerance of a coterie we first state the preliminaries.

5.1 Preliminaries -

Consider a distributed system with N nodes numbered 1 through N. Let S denote the set of nodes of this system. We begin with the definition of a coterie.

Definition 5.1:

A coterie C on S is a collection of non-empty subsets of S satisfying the conditions:

i) $A,B \in C \Rightarrow A \cap B \neq \phi$ (Intersection Property)

ii) $A,B \in C \Rightarrow A \not\subset B, B \not\subset A$ (Minimality Condition)

Elements of a coterie are called quorums or quorum groups. A node seeking access to a shared resource must seek permission from each and every node of some quorum in the assigned coterie. Since a node grants permission to only one node at a time and since any two quorums in a coterie have at least one node in common, mutual exclusion is guaranteed.

Let C be a coterie on S, consisting of m quorums Q_1, Q_2, ..., Q_m. We define the following:

Definition 5.2:

For each i = 1,2,...,m, define a function $bQ_i: S \rightarrow \{0,1\}$ as

$$(bQ1)(x) = \begin{cases} 1, \text{if } x \in Qi \\ 0, \text{if } x \in Qi \end{cases}$$

Definition 5.3:

Define a function bool_quorum : $C \rightarrow \{0,1\}^N$ as bool_quorum(Qi) = bQi, for each quorum Qi, in C, i.e. (bool(Qi))(j) = [(bQi)(j)], for j = 1,2,....,N.

Definition 5.4:

Define bool_coterie(C), the Boolean coterie corresponding to C, as an m x N array [bool(Q_i)], where Q_i, i = 1,2,....,m are quorums of C.

Thus, if $\boldsymbol{C_m}$ denotes the set of all coteries, having m quorums, on S (|S| = N) and if $\boldsymbol{B_{mxN}}$ denotes the set of all mxN boolean arrays, then bool_coterie can be regarded as a function from $\boldsymbol{C_m}$ to $\boldsymbol{B_{mxN}}$. Note that, while the function bool converts each quorum of **C** into a one-dimensional Boolean array of size N, the function bool_coterie converts each coterie **C** in $\boldsymbol{C_m}$ into an mxN Boolean array [q_{ij}], with a 1 at the position (i, j) if the only if the node j belongs to the quorum number i.

Definition 5.5:

For the coterie C with [q_{ij}] as the corresponding bool_coterie, define

a) freq: $S \rightarrow S \cup \{0\}$ as freq (j,C) = $\sum_{i=0}^{m} q_{ij}$, and

b) max_freq (S, C) = $\max_{x \in S} \{freq(x)\}$

When there is no question of doubt, we can write freq(j,C) as freq (j) and max_freq(S,C) as max_freq(S).

Definition 5.6:

A node $p \in S$ is called most_frequent_node in C if freq (p,C) = max_freq(S,C). Thus, for any $x \in S$, freq(x,C) denotes the frequency of occurrence of x in the coterie C, i.e.,the number of quorums of C to which x belongs; max_freq(S,C) is the maximum of these frequencies, taken over all the nodes in S.

Definition 5.7:

For any coterie C and for any node $p \in S$, let

a) (p) = $\{Q \in C: p \in Q\}$ and

b) $C \sim (p) = \{Q \in C: p \in Q\}$

With these definitions in hand, we present, in the next section, an algorithm to find the fault-tolerance of a given coterie.

5.2 The algorithm:

The fault-tolerance of a coterie C is the maximal number of node-failures that will cause the failure of each and every quorum-group of C. It is a worst-case measure. As defined in [13], the fault-tolerance of a quorum system is measured by the maximum number of processors that can fail before all the quorums are hit, in the worst possible configuration of failures. A quorum is hit if at least one of its members has failed. We follow this definition to obtain the fault-tolerance of a given coterie. We actually obtain a worst possible configuration of nodes such that the failure of the nodes in the configuration causes the failure of each and every quorum of the coterie. Let C_0 be the given coterie and let q be an integer variable. The procedure is as follows:

Step 1: Rewrite the given coterie C_0 as a Boolean array bool_coterie. Set q=0.

Step 2: Find frequency freq(x) for each node x of the system.

Step 3: Find maximum frequency max_freq and the most_frequent_node p_q.

(If there are more than one most_frequent_nodes, then choose that node for which row sum in the bool_coterie is minimum.)

Step 4: If max_freq $\neq 0$, set q=q+1. Find $C_q+_1=C_q \sim (p_q)$.

Step 5: Repeat steps 2 to 4 until max_freq = 0.

Step 6: If max_freq = 0, end. Fault-tolerance of C_0=q-1.

The working of the algorithm is simple. In step 1, each quorum in the given coterie C_0 is converted into a N bit stream of 0s and 1s, placing a 1 at the jth place in the quorum Qi if the node j belongs to it and 0, if it does not. We obtain the bool_coterie corresponding to C_0 as an mxN Boolean array. In step 2, the entries in each and every column of the bool_coterie are added. For any node k in S, this gives us freq(k) , the number of quorums of C_0 to which k belongs. In step 3, the maximum value of freq(k) is obtained for k=1,2,.....,N. This gives us max_freq and a node p_0 with frequency equal to max_freq is singled out as the most_freqent_node. In such a case, that node must be chosen for which row sum in the bool_coterie is minimum. Any one of such nodes may be chosen as p_0. In step 4, a new coterie (call it new_coterie) $C_1 = C_0 \sim (p_0)$ is obtained form bool_coterie by deleting all its quorums, which contain the node p_0. This new_coterie, in fact, contains only those quorums of bool_coterie (or, the given coterie C_0) that will survive, if the node p_0 fails. Again, the frequency of each node in C_1 is checked up. If max_freq > 0, the process is repeated. The node p1 with the maximum frequency is deleted form C_1 and another coterie (new coterie), $C_2 = C_1 \sim (p_1)$ is obtained. The process is repeated again and again, till max_freq = 0. This signifies the failure of each and every quorum of the given coterie. A count is kept on the number of times the cycle (freq→max_freq→new_coterie→ freq) is repeated. If max_freq becomes 0 after deleting k nodes, the fault-tolerance of the given coterie C_0 is taken as k-1.

5.3 Correctness of the algorithm:

To see the correctness of the algorithm, let us see exactly what is being done. To find the fault-tolerance of a given coterie C_0, we choose its most_frequent_node, delete all the quorums containing this node and obtain a coterie with left-out quorums. This process is repeated till all the quorums

wiped out. We claim that, if the process completes in k stages, the fault-tolerance of C_0 is k-1. The reason is as follows :

The function freq gives us the frequency of occurrence of each and every node of the system in the coterie C_0. If freq (x, C_0) is r for some node $x \in S$, it means that if the node x fails, exactly r quorums of C_0 will be hit, i.e., the failure of the node x will cause the failure of exactly r quorums of the coterie C_0. We obtain $C_1 = C_0 \sim (p_0)$, by deleting from C_0, all its quorums which contain p_0, a most_frequent_node of S. This is the node whose failure will cause the maximum damage to the coterie C_0. If more than one maximum frequency exists then we choose the node with minimum row sum. This way we choose a node whose failure affects minimum number of processes. Again, we find the frequency of occurrence of nodes of S in this new coterie, choose one of the most_frequent_nodes, p_1, of S and delete all the quorums of C_1 containing this node. This gives rise to another coterie $C_2 = C_1 \sim (p_1)$ consisting of quorums that are still surviving. And so on. The process is repeated again and again, till all the quorums of C_0 are hit. Every new coterie we obtain is, a proper sub-coterie of the previous coterie and as such $|C_{i+1}|<|C_i|$ for every i. Therefore, the process must come to an end after a finite number of steps. That is, there must exist a positive integer k (less than N, of course) such that $C_{k-1} \neq \phi$ but $C_k = \phi$. This means that it is takes deletion of k most_frequent_nodes $p_0, p_1, \ldots\ldots, p_{k-1}$ to destroy the given coterie completely and every time we delete a node, we choose a node that takes the maximum possible toll of quorums of C_0. Thus, P $= \{p_0, p_1 \ldots, p_{k-1}\}$ is a worst possible configuration of failures. Therefore the fault-tolerance of C_0 is k-1, because at least one quorum in C_0 survived even after the failure of k-1 nodes but no quorum in it could survive when k such nodes failed.

5.4 Illustration:

Let us see with the help of an example how does the algorithm work. Let S = {1,2,...., 10} and suppose that the coterie C on S, consists of 9 quorums, given by

Q_1 = {1,2,3}, Q_2 = {2,4,5}, Q_3 = {1,5,6}, Q_4 = {2,6,7}, Q_5 = {3,5,7}, Q_6 = {2,3,5}, Q_7 = {3,4,6,7} Q_8 = {3,5,6,8}, Q_9 = {2,5,6,8,9,10}.

We rewrite each of these quorums as a Boolean string of length 10, putting a 1 at the jth place iff the node j belongs to the quorum. Thus, the coterie C can be rewritten as

Quorums	*Node1*	*Node2*	*Node3*	*Node4*	*Node5*	*Node6*	*Node7*	*Node8*	*Node9*	*Node 10*
Q_1	1	1	1	0	0	0	0	0	0	0
Q_2	0	1	0	1	1	0	0	0	0	0
Q_3	1	0	0	0	1	1	0	0	0	0
Q_4	0	1	0	0	0	1	1	0	0	0
Q_5	0	0	1	0	1	0	1	0	0	0
Q_6	0	1	1	0	1	0	0	0	0	0
Q_7	0	0	1	1	0	1	1	0	0	0
Q_8	0	0	1	0	1	1	0	1	0	0
Q_9	0	1	0	0	1	1	0	1	1	1
Freq (x)	2	5	5	2	6	5	3	2	1	1

This is the bool_coterie corresponding to the coterie C. the last row is the table gives the value of freq(x) for node x in S. therefore,

$$max_freq=max\{freq(x):x \in S\} = \{2,5,5,2,6,5,3,2,1,1\}$$

Since, frequency of node 5 equals max_freq, we choose p_0 = 5 and delete from bool_coterie, all the quorums which contain the node 5. This gives us the new_coterie C1={Q_1, Q4, Q7} as shown below, because failure of node 5 causes failure of the remaining quorums.

Quorums	*Node1*	*Node2*	*Node3*	*Node4*	*Node5*	*Node6*	*Node7*	*Node8*	*Node9*	*Node 10*	*Row Sum*
Q_1	1	1	1	0	0	0	0	0	0	0	3
Q_4	0	1	0	0	0	1	1	0	0	0	3
Q_7	0	0	1	1	0	1	1	0	0	0	4
Freq (x)	1	2	2	1	0	2	2	0	0	0	

Therefore, max_freq = 2 and there are 4 nodes in S_1 having frequency 2 in C_1. We can choose node 2 or 3 or 6 or 7 (looking at the minimum row sum). Let us choose $p_1 = 2$. Then, the new_coterie, C_2, obtained from C_1 by deleting all quorums containing the number 2, consists of just one quorum Q_7. But, the coterie C_2 is not in a position to tolerate any more failures. Deletion of any node will make max_freq equal to 0. Thus, fault-tolerance of C_0 is 2.

Having seen how to use the algorithm, we give the fault-tolerance of some well-known coteries.

For a system consisting of 7 nodes, if the nodes are arranged as a binary tree of depth 2 and the quorums are obtained using the Tree quorum protocol [1], the resulting coterie has 15 quorums and its fault-tolerance is 2.

For a system consisting of 9 nodes, if the nodes are arranged as a square grid and quorums obtained by using the technique given by Maekawa [11], the resulting coterie has 9 quorums and its fault-tolerance is 2. For the same grid, if quorums are obtained using grid quorum protocol [4], the resulting coterie has 27 quorums but its fault-tolerance is still 2. In Maekawa's algorithm, the quorum for the node at position (i, j) is obtained by collecting all the nodes in row i and column j. In the grid protocol, a quorum is formed by collecting one node each from every column and all the nodes in any one column. For a system consisting of 16 nodes also, the two coteries exhibit the same fault-

tolerance, 3, though the number of quorums in the coterie obtained using grid protocol is 256 as compared to Maekawa's coterie, which has just 16 quorums.

For a system consisting of 10 nodes, if the nodes are arranged in a triangular grid and the quorums are formed using the method suggested in [10], the fault-tolerance of the resulting coterie (with just 5 distinct quorums) turns out to be 2. On a set of 15 nodes, the coterie obtained using this technique consists of 6 distinct quorums and has the same fault-tolerance 2.

If the nodes of a system consisting of 12 nodes are arranged in a modified grid and quorums formed using broken billiard paths as suggested in [2], the resulting coterie has 12 quorums and its fault-tolerance is 1. For a system consisting of 24 nodes the corresponding coterie obtained using this technique has fault-tolerance 3.

Chapter 6

Conclusions

In this dissertation, we study distributed mutual exclusion algorithms with special emphasis on permission based distributed mutual exclusion algorithms. We survey a number of token-based as well as permission based algorithms. Coterie based systems form the central idea of this dissertation. Therefore we study the properties of coteries and performance measures for them in detail. We also present an algorithm for finding fault tolerance of a given coterie. During the course of development of this algorithm, we also developed a brute force algorithm for generating all possible coteries for a given node set. However, this approach becomes computationally too costly as the size of the distributed system increases. The development of an algorithm for generation of all possible coteries without resorting to brute force method could be an interesting area for future research.

Another area of future research could be the development of algorithms for other performance measures of coteries, for example message complexity. The behaviour of coterie based algorithms under varying load conditions could also be studied in a future work.

References

1. Agrawal D. and Abbadi A. El. The generalized tree quorum protocol: An efficient approach for managing replicated data. *ACM Transactions on database systems*. 17(4),1992.
2. Agrawal D., Omer E. and Abbadi A. El. Billiard quorums on the grid. *Information Processing Letters*.64, 1997.
3. Amir Y. and Wool A. Optimal availability quorum systems: Theory and practice. *Information Processing Letters*.65 , 1997.
4. Cheung S.Y., Ammar M.H. and Ahmed M. The Grid Protocol: A high performance scheme for maintaining replicated data. *IEEE Transactions on knowledge and data engineering*. 4(6), 1992.
5. Diks K., Kranakis E., Krizanc D., Mans B. and Pelc A. Optimal coteries and voting schemes. *Information Processing Letters*.51, 1994.
6. Garcia-Molina H. and Barara D. How to assign votes in a distributed system. *Journal for the Association for Computing machinary*. 32(4), 1985.
7. Ibaraki T, Nagamochi H and Kameda T. Optimal coteries for rings and related networks. *Distributed Computing*. 8, 1995.
8. Kleinsrock L. Distributed systems, *Communications of the ACM*. Nov 1985. Vol 18 No 11.
9. Lamport L. Time, Clocks and the ordering of events in a distributed system. *Communications of the ACM*. July 1978.
10. Luk W. and Wong T. Two new quorum based algorithms for distributed mutual exclusion. *Proceedings of the 17th International conference on distributed computing systems(ICDCS 1997)*.1997.
11. Maekawa Mamorou. A √N algorithm for mutual exclusion in decentralised systems. *ACM Transactions on computer systems*. 3(2), 1985.
12. Maekawa M., Oldenhoeft A. and Oldenhoeft R. *Operating systems: Advanced Concepts, Benjamin Cummings*. 1987.
13. Pelag D. and Wool A. The availability of Quorum Systems. *Information and Computation*.123, 1995.

14. Raynal M. Algorithms for mutual exclusion, *Cambridge MA, MIT Press.* 1986.
15. Raynal M. A simple taxonomy for the distributed mutual exclusion algorithms, *ACM Operating systems Rev.* 23(2). 1991.
16. Ricart G. and Agrawala A. K. An optimal algorithm for distributed mutual exclusion in computer networks. *Communications of the ACM.* Jan1981.
17. Singhal M. A clas of deadlock free Maekawa type mutual exclusion algorithms for distibuted systems. *Distibuted Computing.* 4, Feb 1991.
18. Suzuki I. and Kasami T. An optimality theory for the mutual exclusion algorithms in Computer Networks. *Proceedings of the third International Conference on Distributed Computing Systems.* Oct 1982.
19. Umar Amjad. *Distributed Computing, Prentice Hall.* 1993.