A Study on Distributed k-Mutual Exclusion Algorithms

Hirotsugu Kakugawa

February 1995

Abstract

The mutual exclusion problem is a problem of arbitrating access conflicts for resources. The problem has been considered as a fundamental problem in computer science and extensively studied from the first minute operating systems started providing multi-tasking or multi-programming feature. Recently, a large number of computers are connected to a computer network. Such a system is called *a distributed system*. In a distributed system, several processes do their jobs by communicating with other processes on remote computers. When they share resources, processes may request the same resource at the same time. If the resource requires mutually exclusive access, then some regulation is needed to access it. This is *the distributed mutual exclusion problem*.

Most of previous works for the distributed mutual exclusion problem treat the case in which only one resource exists in a distributed system. This model may be suitable for modeling, e.g., access control of a distributed database. However, there are other cases in which more than one identical resources exist in a distributed system. The problem of arbitrating identical k resources is called *the distributed k-mutual exclusion problem*.

Distributed systems consist of many components such as computers and communication links. In general, the probability that all components are simultaneously in operational is smaller than the probability that a component is in operational. This implies that when we design a distributed system, we should expect that some components may fail. Fault tolerance is therefore regarded as one of the most important issues in designing distributed systems. Unlike parallel computers, distributed systems are loosely coupled, so that it is easy to add redundant components to increase the availability of distributed systems in such a way that even if several computers and/or communication links may fail, the rest of system is still in operational and alive components work correctly.

This dissertation investigates the distributed k-mutual exclusion problems. We discuss two approaches: *the coterie approach* and *the self-stabilization approach*. In Chapter 1, we give a general introduction to the distributed k-mutual exclusion problem, and address the objectives of this dissertation.

Part I contains the coterie approach. The concept of *coterie* is introduced to reduce the number of messages a process to enter a critical section and to increase the availability of systems. In Chapter 2, we give an introduction to the coterie-based distributed mutual exclusion and introduce the concept of *k-coterie* as an extension of coterie. In Chapter 3, the availability of k-coterie is investigated. In Chapter 4, a distributed k-mutual exclusion algorithm using k-coterie is proposed and its correctness

is proven. In Chapter 5, to demonstrate the efficiency of the proposed algorithm, computer simulations of the proposed algorithm are done.

In Part II, the self-stabilization approach is discussed. A self-stabilizing system is a system which converges to a legitimate (stable) system state without centralized control even if any transient errors happen. In Chapter 6, we give an introduction to the self-stabilization approach. Formal definitions of computational models are described. In Chapter 7 we propose several self-stabilizing mutual exclusin algorithms. Forst, we propose a self-stabilizing k-mutual exclusion algorithm for unidirectional and bidirectional ring networks whose sizes are prime. The proposed algorithm does not require process identifiers, i.e., it is a uniform system. Thus, it works for anonymous ring networks. Next, we investigate the self-stabilizing 1-mutual exclusion problem as a special case of the self-stabilizing k-mutual exclusion problem. We propose a randomized self-stabilizing 1-mutual exclusion algorithm for unidirectional ring networks.

In Chapter 8, we summlize the results in this dissertation and discuss future tasks.

Acknowledgments

I wish to thank my advisor Professor Masafumi Yamashita for his advice, suggestions and encouragements. He taught me a research field of distributed algorithms and suggested me to study the distributed mutual exclusion problem.

I am very grateful to Professor Keiichi Yamagata, Professor Toshimasa Watanabe, Professor Ken'ichi Hagihara, and Associate Professor Shin'ichi Wakabayashi for their invaluable discussions and comments on this dissertation.

I would like to thank Professor Tadashi Ae, Associate Professor Reiji Aibara, and Research Associate Satoshi Fujita for their invaluable comments and discussions. I also thank Research Associate Mr. Yasuhide Ito for his invaluable support.

I thank Professor Katsushi Inoue at Yamaguchi University, Professor Itsuo Takanami at Iwate University, and Associate Professor Hiroshi Matsuno at Oshima National College of Maritime Technology for their encouragement. Especially, Professor Inoue introduced me the theoretical computer science and taught me how to do research.

I would like to thank Associate Professor Masaaki Mizuno and Kyoko Mizuno at Kansas State University and Research Associate Yoko Kamidoi at Hiroshima City University for their encouragement.

A special thank goes to Miss Yayoi Ushio for her encouragement and delicious tea-breaks.

Contents

1 Introduction **15**

1.1 The Mutual Exclusion Problem . . . 15

1.2 Distributed Systems . . . 16

1.3 The Distributed Mutual Exclusion Problem . . . 16

1.4 The Distributed k-Mutual Exclusion Problem . . . 17

1.5 Fault Tolerance of Distributed Systems . . . 18

1.6 Organization of This Dissertation . . . 18

I The Coterie Approach **21**

2 The Coterie Approach for the Distributed k-Mutual Exclusion **23**

2.1 Previous Works for the Distributed 1-Mutual Exclusion . . . 23

2.1.1 The first distributed 1-mutual exclusion algorithm by Lamport . . . 23

2.1.2 Majority and voting . . . 24

2.1.3 Coterie . . . 25

2.1.4 Study on fault tolerance . . . 26

2.1.5 Token-based algorithms . . . 26

2.2 Previous Works for the Distributed k-Mutual Exclusion . . . 27

2.3 Models and k-Coteries . . . 28

3 Availability of k-Coterie **33**

3.1 Assumptions and Definitions . . . 33

3.2 k-Majority Coteries . . . 35

3.3 k-Singleton Coteries . . . 38

3.4 Concluding Remarks . . . 39

4 A Distributed k-Mutual Exclusion Algorithm using k-Coterie **41**

4.1 The Distributed k-Mutual Exclusion Algorithm . . . 41

4.2 Correctness proofs . . . 43

4.3 Message complexity . . . 44

4.4 Concluding Remarks . . . 45

5 Experimental Evaluation of the k-Mutual Exclusion Algorithm 47

5.1 Assumptions and the Simulation Model . . . 47
5.2 Outline of the Simulation System . . . 48
5.3 The Distributed k-Mutual Exclusion Algorithm by Kerry Raymond . . . 50
5.4 Simulation and Results . . . 50
5.5 Concluding Remarks . . . 57

II The Self-Stabilization Approach 59

6 The Self-Stabilization Approach for the Distributed k-Mutual Exclusion 61

6.1 Computational Models . . . 61
6.2 Previous Works . . . 62
6.3 Preliminaries . . . 64
6.3.1 The process and network model . . . 64
6.3.2 Scheduling of processes . . . 65
6.3.3 The self-stabilizing k-mutual exclusion problem . . . 65

7 Self-Stabilizing Mutual Exclusion Algorithms 67

7.1 Self-Stabilizing k-Mutual Exclusion Algorithms . . . 67
7.1.1 Burns and Pachl's Algorithm . . . 67
7.1.2 Unidirectional Uniform Rings . . . 69
7.1.3 Bidirectional Uniform Rings . . . 72
7.2 A Self-Stabilizing 1-Mutual Exclusion Algorithm with Randomization . . . 79
7.2.1 The self-stabilizing system under a c-dragon . . . 79
7.2.2 The randomized self-stabilizing system under a c-daemon . . . 80
7.2.3 The randomized self-stabilizing 1-mutual exclusion algorithm . . . 81
7.3 Concluding Remarks . . . 89

8 Conclusion 91

A Local Coteries and a Distributed Resource Allocation Algorithm 93

A.1 The Resource Model . . . 93
A.2 The Resource Allocation Problem . . . 94
A.3 Local Coteries . . . 95
A.4 A Distributed Resource Allocation Algorithm . . . 96
A.5 Correctness Proof . . . 99
A.6 Concluding Remarks . . . 101

B Implementations of Distributed k-Mutual Exclusion Algorithms 103
B.1 Our Distributed k-Mutual Exclusion Algorithm using k-Coterie 104
B.2 Raymond's Distributed k-Mutaul Exclusion Algorithm 108
B.3 The Behavior of a Process 110

List of Figures

5.1 The behavior of a process 48
5.2 The simulation system (in the case $n = 5$) 49
5.3 The average number of messages ($k = 2$, $n = 5$). 52
5.4 The average number of messages ($k = 2$, $n = 8$). 53
5.5 The average number of messages ($k = 2$, $n = 11$). 54
5.6 The average number of messages ($k = 3$, $n = 7$). 55
5.7 The average number of messages ($k = 4$, $n = 9$). 56

List of Tables

3.1 $p_u(n,r,k)$ for some n ($k=1,...,6,\ r=1,...,k$). 36

3.2 $p_l(n,r,k)$ for some n ($k=1,...,6,\ r=1,...,k$). 39

3.3 (k,r)-availabilities of Maj$_k$and Sgl$_k$($k=4,\ n=14$). 40

4.1 Message complexities of disributed k-mutual exclusion algorithms 46

5.1 Cross over probabilities for $k=2$ 52

Chapter 1

Introduction

1.1 The Mutual Exclusion Problem

The mutual exclusion problem first arised when the concept of concurrent processes was introduced in operating systems. When more than one processes share memory cells, undesirable situations may happen: Suppose that two processes P_1 and P_2 which share a variable, say x, wish to increment x by one. To increment the value of x, a process loads the value of x into a register in CPU, increments the value of the register by one, and then stores it back into x. If P_2 starts executing the above procedure after P_1 finishes its execution, the result is correct, i.e., the value of x is incremented by two. However, what if their executions are interleaved? Consider, for example, the following interleaved execution sequence. P_1 loads x, P_1 increments the register, P_2 loads x, P_2 increments the register, P_2 stores the register into x, and then P_1 stores the register into x. x is incremented by only one !

To guarantee such an undesirable situation does not happen, the concept of *critical section* is introduced. A program text can be partitioned into two kinds of sections: sections in which there are no accesses to shared resources (e.g., shared variables) and sections in which shared resources are accessed. The latter sections are called *critical sections* or *critical regions*. Then it is easy to see that by synchronizing processes in such a way that at most one of them is in a critical section, we can achieve one aim of avoiding undesirable situations. For instance, by encapsulating the three steps of increment procedure, (1) loading x into a register, (2) incrementing the value of register, and (3) storing the value of the register, in a critical section, we always get a correct result.

To make executions of critical sections mutually exclusive, a process wishing to enter a critical section must issue an operation to get a permission. Dijkstra introduced an abstract data type called *semaphore* in [Dij68]. To enter a critical section, a process must issue a P operation. If there is a process being in a critical section at the time instant, the execution of the process is suspended until no process is in a critical section. When a process exits a critical section, it issues a V operation to permit another process to enter a critical section.[1] Modern CPUs support P and V or similar instructions (e.g.,

[1]To speak rigidly, operations P and V are defined as follows: When a process performs a P operation, it executes next instruction (i.e., it enters in a critical section) if there is no processes in a critical section. Otherwise, it is blocked until no

test-and-set instruction) in order to solve the mutual exclusion problem.

In this dissertation, we discuss the mutual exclusion problem in a computer network (not in a single computer).

1.2 Distributed Systems

Recently, a large number of computers are connected to a computer network. A set of computers connected by a set of communication links is called a *distributed system*. We characterize distributed systems by the absence of shared memory. In a distributed system, processes on a computer do their tasks with other processes on remote computers. To achieve cooperative tasks (or competitive tasks), processes must communicate with other processes via communication links since there is no shared memory.

The following motivates distributed systems[Hag90, Hag93]:

- **High performance** — Since the system consists of several computers, independent tasks can be processed in parallel. Load balancing is easy.
- **Distribution of users** — When users of the system are geometrically distributed, it is natural to process tasks distributedly.
- **Extensiveness** — In general, addition of computers and communication links can be done easily with small modification of the current system. Replacement of computers and communication links is also easy. This property comes from the nature that distributed systems are loosely coupled.
- **Fault-tolerance** — A centralized system cannot provide services when the central machine stops by failure. Distributed systems may provide services if there are several alive components.

Distributed systems have many advantages compared with centralized systems. However, designing distributed algorithms to control distributed systems is by no means easy because of the following reasons: Computers must send/receive messages to other computers to get enough information to do their tasks. Messages are delivered with delay and therefore in principle there is no way to capture the global state of the system. In addition, there is no process which controls the entire distributed system. Therefore, to achieve fault-tolerance, algorithms must consider failures such as process stops and message losts.

1.3 The Distributed Mutual Exclusion Problem

When processes in a distributed system share a resource which must be accessed exclusively, the access to the resource must be controlled as in the case of concurrent processes in stand-alone operating

processes are in a critical section. A process performs a V operation when it exits from critical section. The operating system unblocks a process after a performance of a V operation.

systems. To enter a critical section, a process must assure that there is no process which is being in a critical section in the distributed system.

Many algorithms have been proposed to solve the distributed mutual exclusion problem. They are classified into two types[Ray91b]:

- **Permission-based principle** — A process P wising to enter a critical section requests some other processes to permit it to enter a critical section. If a permission is given from each process P is asking, P can enter the critical section.
- **Token-based principle** — There is an object called a *token* in a distributed system and it travels among processes. A process can enter a critical section while it is holding the token. The mutual exclusion is guaranteed because there is only one token in the system and there are no two processes having a token at the same time.

Several algorithms are surveyed in Chapter 2.

Consider a distributed system having two magnetic tape drives A and B. Suppose that two processes P and Q wish to use two magnetic tape drives. In such a case, we must be careful to avoid the state in which P reserves A and Q reserves B, since both P and Q are stuck forever if both of them request another tape drive. *Deadlock* is the terminology to denote such situations.

We also avoid a starvation situation in which a request is not satisfied forever (i.e, a magnetic tape drive cannot be allocated forever).

In designing a mutual exclusion algorithm, guaranteeing the *deadlock free property* and the *starvation free property* are important issues. Note that once a deadlock happen, it cannot be solved; while starvation can.

1.4 The Distributed k-Mutual Exclusion Problem

In the example of a distributed database described above, only one item is shared by processes. However, there are systems such that k identical resources are shared by processes.

For example, consider a Ethernet local area network and many computers executing processes are connected to it. Since Ethernet is a CSMA/CD (Carrier Sense Multiple Access with Collision Detect) type local area network, the performance of the network becomes bad suddenly when computers send packets frequently. To avoid such situation, a distributed k-mutual exclusion can be applied. The bandwidth of a network can be considered as resources and a program fragment in which a process sends a large amount of data via network can be considered as a critical section. When a process wishes to send data, it must enter a critical section. Then the total amount of traffic of a network can be controlled.

The simplest way of solving this problem is to solve the 1-mutual exclusion problem for each resource, i.e., we distinguish each resource by labeling a unique name and provide a mutual exclusion algorithm for each resource. This is a simple solution, however, a process must choose which resource it wish to use even if the k resources are identical. By this solution, many processes may be waiting for a resource even if there are free resources. This motivates a study of distributed k-mutual exclusion algorithms. The distributed k-mutual exclusion problem is the main theme of this dissertation.

1.5 Fault Tolerance of Distributed Systems

Fault tolerance is an important issue and it is desirable that distributed systems can tolerate from any failures. But implementations of fault tolerance are difficult or sometimes impossible. For instance, it is shown that there is no consensus algorithm in totally asynchronous system even if the number of faulty process is one [FLP85, Tau91]. Thus, it is common to classify the failures into several classes and fault tolerant systems are discussed by assuming failure classes.

For instance, failures are classified as follows [Hag90, Hag93]:

- **Crash failure** — Processes (or links) completely stop when an error occur. If a failure occurs, it never send any message.
- **Send-omission failure** — Messages may be lost when sending.
- **General-omission failure** — Messages may be lost when sending and/or receiving.
- **Byzantine failure** — Processes may send strange messages to cheat other processes.

Although several computers and/or links may stop by power down and the value of memory cells or messages on links may be lost, they have complete functionality and may work correctly again if power is supply recovers. Such failures are called *transient failures*.

A system which tolerates against any transient failures is called a *self-stabilizing system* and was first discussed by Dijkstra [Dij74]. A self-stabilizing system is a system which converges without centralized control to a legitimate (stable) system state even if any transient errors occur. In the latter half of this dissertation, we propose several self-stabilizing mutual exclusion algorithms.

1.6 Organization of This Dissertation

This dissertation consists of two parts. We discuss the coterie approach of the distributed k-mutual exclusion problem in Part I. The self-stabilization approach is discussed in Part II. Part I includes Chapter 2 to Chapter 5 and Part II includes Chapter 6 to Chapter 7.

In Chapter 2, we discuss the coterie approach. Previous works for distributed mutual exclusion are also reviewed in this chapter. Coterie is a set of process groups such that a process wishing to use a resource must get permission from all processes of a process group. We propose a concept called k-coterie as an extension of coterie. In Chapter 3, the availability of coterie is analyzed. Intuitively, the *availability* is the probability that at least one process can use a resource in spite of process and/or link failures. Since there exists k resources, the definition of availability is not enough. We introduce a new measure called *(k, r)-availability*. The (k, r)-availability is the probability that at least r processes can use resources at a time. If $k = r = 1$, the (k, r)-availability is the conventional availability. We show a necessary and a sufficient conditions for a class of coteries called k-majority coterie to be optimal in the sense of (k, r)-availability. In Chapter 4, we propose a distributed k-mutual exclusion algorithm using a k-coterie and its correctness is shown. To demonstrate the efficiency of the proposed algorithm, the average message complexity of the algorithm is examined by computer simulations. The simulation results is shown in Chapter 5. In the simulation, each process is executed on different workstations connected to a local area network.

In Chapter 6, we discuss the self-stabilization approach. A self-stabilizing algorithm is an algorithm which tolerates from any transient failures and therefore, initialization is not necessary for the system; it converges to a stable state automatically. In this dissertation, we consider a uniform self-stabilizing systems on ring networks. A system is called *uniform* if all processes are identical and do not have process identifiers. In Chapter 7, we propose several self-stabilizing mutual exclusion algorithms. First, we propose a self-stabilizing k-mutual exclusion algorithm on rings whose sizes are primes. Next, we consider the self-stabilizing 1-mutual exclusion problem as a special case. In [BP89], Burns and Pachl showed that there exists no uniform deterministic self-stabilizing 1-mutual exclusion algorithm if the number of processes on a ring is composite. We show that there exists a uniform probabilistic self-stabilizing mutual exclusion algorithm when the number of processes is composite.

In Chapter 8, we summarize the results in this dissertation and present open problems and future tasks.

Part I

The Coterie Approach

Chapter 2

The Coterie Approach for the Distributed k-Mutual Exclusion

In Part I, we investigate the distributed k-mutual exclusion problem by taking the coterie approach. First, we discuss the distributed mutual exclusion (i.e., the distributed 1-mutual exclusion) based on coterie and survey previous works. Then, we motivate a study of the distributed k-mutual exclusion. Finally, we introduce a concept k-coterie to solve the distributed k-mutual exclusion problem.

2.1 Previous Works for the Distributed 1-Mutual Exclusion

The distributed 1-mutual exclusion problem is one of the fundamental distributed problems and many algorithms to solve the problem have been proposed. In this section, we survey previous works of the distributed 1-mutual exclusion.

2.1.1 The first distributed 1-mutual exclusion algorithm by Lamport

The first distributed mutual exclusion algorithm is proposed by Lamport [Lam78]. To guarantee mutual exclusion, no deadlock, and no starvation, distributed mutual exclusion algorithms must have some arbitration mechanism. To this end, he proposed a *logical clock* in totally asynchronous distributed systems. A logical clock is defined as follows [Lam78]:

- Initially, a logical clock of every process is zero.
- When an internal (local) event (e.g., update of a variable) occurs at a process P, a logical clock of P is incremented by one.
- When a process P_s sends a message M to P_d, the value of P_s's logical clock, say c_s, is attached to M, i.e., a pair $\langle M, c_s \rangle$ is sent. When P_d receives a message, it retrieves a clock value of P_s $(= c_s)$ and compares with its own logical clock c_d. Then, P_d's logical clock is updated by taking maximum of these two logical clocks. i.e., $c_d := \max(c_d, c_s)$.

Note that this logical time has no relation to the physical time.

The priority among mutual exclusion requests is defined by a pair of a logical time at which a request is issued and a process identifier of a requesting process. The pair of a logical time and a process identifier is call a *timestamp*, and it is assumed that every request message contains a timestamp. Since total ordering is defined on timestamps, processes can tell which request has the highest priority. Thus, by usage of timestamps, his algorithm avoids starvations and deadlocks.

In his algorithm, a process which enters a critical section sends request messages to all the other processes. When a process receives a request message, the request is put into a priority queue and it sends a reply message to the requesting process. The requesting process enters a critical section if it receives reply messages from the other processes and its request is the highest among items is its priority queue. To exit from a critical section, it sends a release message to the other processes and deletes its request from its queue. A process receiving a release message deletes the corresponding item from the priority queue. For every invocation of a mutual exclusion, it must send messages to the other processes in a distributed system. So, this algorithm is based on the *unanimous* consensus method and requires $3(n-1)$ messages per invocation of a mutual exclusion. If a process stops by a failure then other alive processes cannot enter their critical sections; thus it is not a good algorithm from the view point of the fault tolerance.

Ricart and Agrawala proposed an improved algorithm [RA81] which requires $2(n-1)$ messages per invocation of mutual exclusion, but it sends a request message to every process like Lamport's algorithm. Carvalho and Roucairol further improved the algorithm to reduce the number of messages [CR83, RA83].

2.1.2 Majority and voting

In Lamport's algorithm and Ricart and Agrawala's algorithm, a process must communicate with all processes. To guarantee mutual exclusion, however, the unanimous consensus method is not necessary. Thomas proposed the *majority* consensus algorithm to guarantee mutual exclusion [Tho79]. A process which enters a critical section must get permissions from a majority of all processes. Assuming that more than a half processes are alive, alive processes can enter their critical sections, i.e., they can continue their tasks even if at most half of the system components stop. This algorithm is definitely more resilient than Lamport's algorithm [Lam78].

As a generalization of the majority method, Gifford proposed the *weighted voting* [Gif79]. Each process is assigned a number of votes. A process must collect a majority of total votes to enter a critical section. Note that the majority method by Thomas is a special case when each process has one vote. Each computer has different reliability, in general. If more votes are assigned to more reliable computers then it is expected that the availability of system increases. (Recall that the availability of mutual exclusion is the probability that at least one process in a distributed system can enter a critical section.) In addition, the number of processes that a process must exchange messages on an invocation of a mutual exclusion can be controlled by changing vote assignments. As an extension of the weighted voting, In [CAA90], Cheung, Ahamad and Ammar proposed the multi-dimensional voting method as an extension of the voting method. The vote assigned to a process is a multi-dimensional vector.

2.1.3 Coterie

To decrease the number of messages per mutual exclusion invocation and to increase the availability, the concept of *coterie* is proposed by Garcia-Molina and Barbara [GMB85]. The definition of coterie is as follows.

Definition 1 *Let U be the set of all processes. A set $\mathcal{C} = \{Q_1, Q_2, ..., Q_m\} \neq \emptyset$ is a* coterie *if and only if the following conditions hold:*

1. *Non-emptiness: For each i, $Q_i \neq \emptyset$.*
2. *Intersection property: For each i, j, $Q_i \cap Q_j \neq \emptyset$.*
3. *Minimality: For each i, j ($i \neq j$), $Q_i \not\subseteq Q_j$.*

Elements of a coterie is called quorums. □

A process wishing to enter a critical section sends a request message to every process in a quorum $Q \in \mathcal{C}$. If it can get permission from every processes in a quorum, it can enter a critical section. Mutual exclusion is guaranteed because every two quorums has non-empty intersection and processes in an intersection of quorums serve as an arbiter of mutual exclusion requests. It is shown that (1) every voting assignment in the weighted voting scheme can be expressed in terms of coterie and (2) there exists a coterie which cannot be expressed in terms of the vote assignment[GMB85]. Therefore, the majority method [Tho79] and the centralized method are also expressed in terms of coterie. Coterie is thus more powerful than the vote assignment method.

Garcia-Molina and Barbara [GMB85] proposed the concept *domination* of coteries.

Definition 2 *Let $\mathcal{Q}$ and $\mathcal{R}$ be coteries. $\mathcal{Q}$* dominates *$\mathcal{R}$ if and only if a condition*

$$\forall R \in \mathcal{R} \exists Q \in \mathcal{Q}[Q \subseteq R] \wedge \mathcal{Q} \neq \mathcal{R}$$

holds. A coterie $\mathcal{Q}$ is a non-dominated coterie *if and only if there is no coterie which dominates $\mathcal{Q}$.*

A coterie $\mathcal{Q}$ which dominates $\mathcal{R}$ is better than $\mathcal{R}$ because of the following reasons:

- **Availability**: Suppose that a set of alive processes is S. By definition of domination, if there exists $R \in \mathcal{R}$ and $R \subseteq S$ then there exists $Q \in \mathcal{Q}$ and $Q \subseteq S$. Intuitively, if a system using $\mathcal{R}$ is operational at the presence of failures then a system using $\mathcal{Q}$ is also operational, but the opposite is not always true.

- **Message complexity**: Assume that a system uses $\mathcal{R}$ and that a process communicates with processes in $R \in \mathcal{R}$. By definition of domination, there is a quorum in $Q \in \mathcal{Q}$ such that $Q \subseteq R$, which implies that a process can use Q instead of R if a system uses $\mathcal{Q}$. Because $Q \subseteq R$, the number of messages a process must send is smaller than or equal.

Several algorithms using coterie has been proposed. Maekawa proposed an algorithm using coterie constructed from finite projective planes. The size of quorums of the coterie is approximately $\sqrt{n}$. He showed that coteries based on finite projective planes are the optimal coteries in the sense that each process has equal amount of responsibility to the mutual exclusion control. A process wishing to enter a critical section sends a request message to every process in a quorum. It waits until permission is granted by all process in the quorum. After exiting a critical section, it releases the permission. To avoid deadlock, permissions are preempted according to the priority defined by Lamport [Lam78]. (Sanders pointed out that Maekawa's algorithm may cause deadlocks [San87].) Each process requires $O(\sqrt{n})$ messages per mutual exclusion invocation because the size of quorums is $\sqrt{n}$. Singhal proposed a Maekawa-type deadlock free algorithm without additional messages for deadlock resolution [Sin91].

Not only mutual exclusion algorithms but also properties of coteries and construction methods are investigated by many researchers.

In [AA89], Agrawal and Abbadi proposed a coterie constructed by binary tree. The size of quorums of a coterie varies from $\log n$ to $\lceil \frac{n+1}{2} \rceil$. Kumar proposed a hierarchical quorum consensus and a coterie with multilevel hierarchies whose quorum size is $n^{0.63}$ [Kum91]. Ibaraki and Kameda investigated properties of coteries from the point of view of boolean functions [IK91] and showed a characterization of non-dominated coteries. Neilsen, Mizuno and Raynal proposed a method for constructing a complex coterie from simple coteries [NM92, NMR92].

2.1.4 Study on fault tolerance

Barbara and Garcia-Molina discussed the availability of mutual exclusion [BGM87]. They showed several heuristics for vote assignment to increase the availability of mutual exclusion for arbitrary network topology. When the network topology is complete, the communication links never fail, and reliability of each process is $p > 0.5$, then the majority method [Tho79] is shown to be optimal in the sense of availability. Rangarajan and Tripathi proposed a variation of finite projective planes based coteries to increase the availability. The quorum size of the coterie is $\sqrt{n \log n}$.

2.1.5 Token-based algorithms

The above algorithms are based on he permission-based principle, i.e., a process can enter its critical section only if certain permission is granted.

Algorithms based on the token-based principle have also been proposed. Suzuki and Kasami proposed an algorithm which requires at most n messages per invocation on mutual exclusion [SK85]. An imaginary object called *token* is provided in the system and a process which holding the token is the process which has the privilege to enter its critical section. If a process holds a token then it is not necessary to send any request messages. Otherwise, it sends a request message to every process. In their algorithm, the sequence number is used to guarantee deadlock freedom and starvation freedom. Suzuki and Kasami also showed an algorithm with bounded sequence number. The algorithm proposed by Ricart and Agrawala [RA81] also uses the sequence number but the value is unbounded.

Raymond proposed another token-based algorithm [Ray89b]. Her algorithm dynamically maintains a directed spanning tree of a network. The direction of an edge of a spanning tree indicates the direction of a token. A request message is forwarded along directed edges of a spanning tree. This method does not require a process sending its request message to all processes. The number of messages required per invocation of mutual exclusion depends on the topology of tree but typically $O(\log n)$ under light demands of mutual exclusions. In the case that the demands of mutual exclusions are heavy, approximately four messages are necessary. Satyanarayanan and Muthukrishnan proposed a modification of Raymond's algorithm such that it can provide least executed criterion as a fairness policy of mutual exclusion by processes [SM92].

Mizuno, Neilsen and Rao proposed an algorithm based on token-based principle using coteries [MLR91]. A process which is requesting to enter a critical section sends a request message to a process of a quorum of a coterie.

2.2 Previous Works for the Distributed k-Mutual Exclusion

In this section, we review previous works for the distributed k-mutual exclusion problem. An algorithm for distributed k-mutual exclusion can be constructed from k mutual exclusion algorithms. That is, we name k resources distinct names and a process wishing to use a resource chooses a resource name among k resources and issue a request for the mutual exclusion algorithm for the resource. This is a simple solution but has a drawback. Suppose that every process specifies the same resource, they must wait a long time even if there are free resources. By this reason, several distributed k-mutual exclusion algorithms have been proposed.

The first distributed k-mutual exclusion algorithm is proposed by Raymond [Ray89a]. Her algorithm is a modification of Ricart and Agrawala's distributed 1-mutual exclusion algorithm [RA81]. According to her algorithm, a process must send a request message to every process in a distributed system. It can enter a critical section if it receives $n - k$ reply messages, where n is the number of processes. The algorithm requires $2n - k - 1$ messages in the best case and $2(n - 1)$ in the worst case. This algorithm tolerates from failures of arbitrary $k - 1$ processes. In [BC94], Baldoni and Ciciani proposed a modification of Raymond's algorithm [Ray89a] so that it can provide priorities (e.g., short job first) for mutual exclusion requests. To avoid starvations, they used gated batch priority queues.

Raynal proposed a resource allocation algorithm in [Ray91a]. He discussed allocation of any amount of resources among M identical resources. This is a generalization of k-mutual exclusion because k-mutual exclusion can be considered requesting one resource among k resources. The algorithm proposed by Raynal also sends a request message to every processes.

In [SR92], Srimani and Reddy proposed another algorithm which is a modification of Suzuki and Kasami's algorithm [SK85]. The number of messages necessary for each mutual exclusion invocation is a half of that for Raymond's algorithm. The algorithm is token-based and k tokens are circulated to guarantee k-mutual exclusion.

2.3 Models and k-Coteries

In this section, the computational model we assume in Part I is described. A *distributed system* consists of n *processes* and bidirectional *communication links* connected between all pairs of processes. (That is, the network topology is a complete graph.) We assume that the structure of a program that each process executes is as follows:

```
Process $P_i$;
begin
    while true do
        begin

            $\boxed{\text{Non-Critical Section}}$

            ⟨Enter a Critical Section⟩

            $\boxed{\text{Critical Section}}$

            ⟨Exit from a Critical Section⟩

            $\boxed{\text{Non-Critical Section}}$

        end
end.
```

Each process executes the same program, but has unique process identifier. Without loss of generality, we assume that process identifiers are positive integers, which every process knows. Processing speed of processes may be different. Some processes may execute a program fast and others may do really slow; the processing speed of processes may change even during the execution of a program. But it is guaranteed that a process can execute its next instruction within a finite time unless the execution of its algorithm has been terminated.

Each process has its own *local clock*. Each local clock may indicate different time, and no processes can tell the global time.[1] Therefore, processes cannot make use of their local clocks to synchronize

[1]The definition of the distributed k-mutual exclusion problem requires the existence of the global time.

with other processes.

Since there is no centralized control to solve the problem and the only mechanism provided in the system for information exchange between processes is the *message passing*, i.e., processes do not have shared memory, processes must collect enough information from other processes through communication links. We assume that links are error-free.

Each process has a message queue of infinite length, which stores messages arrived to it. Operations provided for the message passing are as follows.

- **SEND** operation
  SEND is used to send a message. To send a message, a destination process must be specified. Messages sent by a process are eventually put into the message queue of the destination process in a finite time.

- **RECEIVE** operation
  As described, each process maintains a message queue. The first message in the queue is retrieved by issuing RECEIVE. We assume that a process can tell if the queue is empty or not.

The order of messages is kept unchanged during the delivery. That is, if a process P_1 sends messages m_1 and m_2 in this order to P_2 then P_2 receives m_1 and m_2 in the same order. It is guaranteed that each message is delivered in a finite time. But the message delivery delay is unpredictable; the delay may vary during the execution of a program.

Consider extending the concept of coterie for k-mutual exclusion. (The definition of coterie is shown in definition 1.) The 1-mutual exclusion is guaranteed because there are no two distinct quorums in a coterie. Thus, k processes can be in their critical sections if there are k distinct quorums, and $k+1$ processes cannot be in their critical sections at a time if there are no $k+1$ distinct quorums. By this intuition, we have the concept of k-coterie. The formal definition is as follows.

Definition 3 *A non-empty set $\mathcal{C}$ of non-empty subsets q of U is called a k-*coterie *if and only if all of the following three conditions hold:*

1. **Non-intersection property:**
   For any $h(< k)$ elements $Q_1, ..., Q_h \in \mathcal{C}$ such that $Q_i \cap Q_j = \emptyset \ \ (i \neq j)$ for $1 \leq i, j \leq h$, there exists an element $Q \in \mathcal{C}$ such that $Q \cap Q_i = \emptyset$ for $1 \leq i \leq h$.

2. **Intersection property:**
   For any $k+1$ elements $Q_1, ..., Q_{k+1} \in \mathcal{C}$, there exists a pair Q_i and Q_j such that $Q_i \cap Q_j \neq \emptyset$.

3. **Minimality property:**
   For any two distinct elements Q_i and Q_j in $\mathcal{C}$, $Q_i \not\subseteq Q_j$.

An element q of a k-coterie $\mathcal{C}$ is called a quorum. □

Note that a 1-coterie is a coterie, and therefore, the concept of a k-coterie is an extension of a coterie.

Example 1 *Let $U = \{1, 2, ..., 6\}$. The following $C1, ..., C5$ are k-coteries ($k = 1, 2, 3$) under U. Note that a condition $\cup_i Q_i = U$ does not have to be true by the definition of k-coterie.*

- $k = 1$

$$\mathcal{C}_1 = \{\{1\}\}$$

$$\mathcal{C}_2 = \{\{1, 2\}, \{2, 3\}, \{3, 1\}\}$$

- $k = 2$

$$\mathcal{C}_3 = \{\{1\}, \{2\}\}$$

$$\mathcal{C}_4 = \{\{1, 2\}, \{3, 4\}, \{3, 4\}, \{4, 1\}\}$$

- $k = 3$

$$\mathcal{C}_5 = \{\{1, 4\}, \{2, 5\}, \{3, 6\}, \{1, 5\}, \{2, 6\}, \{3, 4\}, \{1, 6\}, \{2, 4\}, \{3, 5\}\}$$

□

A majority method can be defined for k-mutual exclusion. The following k-coterie, a *k-majority coterie*, is a coterie that each quorums consists of any $W = \lceil (n+1)/(k+1) \rceil$ processes. This is called k-majority since W is approximately n/k and there are no $k + 1$ groups of W processes.

Definition 4 *Let $W = \lceil (n + 1)/(k + 1) \rceil$, where n is the number of processes. The set* $\text{Maj}_k = \{Q_i \mid Q_i \subseteq U,\ |Q_i| = W\}$ *is called a* k-majority coterie. □

A majority coterie is defined when $n \geq k^2$.[2] A k-coterie which corresponds to primary the site method is called a *k-singleton coterie*. A k-singleton coterie is a k-coterie which consists of k quorums such that each quorum consists of one process.

Definition 5 *A* k-singleton coterie Sgl_k *is a set $\{\{P_1\}, \ldots, \{P_k\}\}$, where $P_i \in U$ for $i = 1, \ldots, k$, and P_i's are distinct.* □

[2]In [MYKC94], Yuang and Chang claimed that n and k must satisfy following two conditions so that the k-majority coterie is a k-coterie:

- $kW \leq n$,
- $(k + 1)W > n$

where W is an integer.

Fujita et al. proposed a construction algorithm of a k-coterie whose quorum size is $O(\sqrt{n}\log n)$ in [FYA91]. Like a concept domination for coteries, a concept domination for k-coteries can be defined. Nielsen and Mizuno extended the concept of non-domination for k-coteries [NM94]. They also proposed a composition method for k-coteries.

Huang, Jiang and Kuo also reached k-coterie independently, which is slightly different from ours, and investigated availability [STHK93]. Baldoni proposed k-coterie [Bal94b, Bal94a], which is completely different from ours. His k-coterie requests that 'intersection of any k quorums is non-empty'. This idea is based on the following: every process has k permissions and a process wishing to enter a critical section gets a permission from each process in a quorum. If k processes are in their critical sections then another process wishing to enter a critical section cannot get permissions since the intersection of any k quorums is non-empty, which implies that there exists a process which passed all its permission to other process. The message complexity of their algorithm is $3\lceil n^{k/(k+1)} - 1\rceil$ in the best case and $5\lceil n^{k/(k+1)} - 1\rceil$ in the worst case.

Chapter 3

Availability of k-Coterie

In this chapter, we investigate the availability of the distributed k-mutual exclusion by k-coterie. In [BGM87], Barbara and Garcia-Molina showed that if the network topology is complete and communication links never fail and the reliability of each process is $p > 0.5$ then the majority method [Tho79] is optimal in the sense of availability. It is conjectured that a k-majority coterie is an optimal coterie under some conditions because a k-majority coterie is a natural extension of majority coterie (a coterie corresponding to the majority method). In this section, we investigate the optimality of k-majority coteries. Not only k-majority coterie but also k-singleton coterie is investigate in this chapter.

3.1 Assumptions and Definitions

Before investigation of availability of k-coteries, we describe assumptions and define several concepts. We investigate the availability of k-coteries under the following assumptions:

1. The network topology of a distributed system is a complete graph; between each pair of processes, there is a bidirectional communication link.
2. The communication links never fail.
3. For all processes P, the reliability of P, i.e., the probability of P being in operation, is the same constant $0 \leq p \leq 1$.

Availability is a probability that at least one process can achieve mutual exclusion in the case of the 1-mutual exclusion. For the purpose of investigation of fault-tolerance of the k-mutual exclusion, we extend this concept. Since k processes may enter their critical sections, the probability that r processes can enter their critical sections can be considered as a measure of fault-tolerance of the k-mutual exclusion, where r is an integer such that $1 \leq r \leq k$. This idea is formalized as a concept of the (k, r)-availability.

Definition 6 *Let $\mathcal{C}$ be a k-coterie over U, and r $(1 \leq r \leq k)$ be an integer. The* (k, r)-characteristic function $F_{\mathcal{C},k,r}$ *of $\mathcal{C}$ is a function from 2^U to $\{0, 1\}$ defined as follows:*

For each $S \subseteq U$, $F_{\mathcal{C},k,r}(S) = 1$ if and only if there exist r quorums $Q_1, ..., Q_r \in \mathcal{C}$ satisfying both of the following two conditions;

$$Q_i \cap Q_j = \emptyset \text{ for } 1 \le i, j \le r,\ i \ne j, \text{ and}$$
$$\text{for all } i,\ Q_i \subseteq S.$$

□

That is, $F_{\mathcal{C},k,r}(S) = 1$ if and only if r processes can enter their critical sections, provided that all processes in S are being up.

Definition 7 *Let $\mathcal{C}$ be a k-coterie, and r $(1 \le r \le k)$ be an integer. The (k, r)-*availability *$R_{k,r}(\mathcal{C})$ of $\mathcal{C}$ is the probability that at least r processes can enter a critical section.*

More formally, let $G = (V, E)$ be the topology of the distributed system under consideration. Let V' and E' be, respectively, the sets of processes and links in operation, and $P_r(V', E')$ denote the probability that this situation occurs. The topology of the distributed system in operation is the graph $G' = (V', (V' \times V') \cap E')$. We say a quorum $Q \in \mathcal{C}$ is available with respect to G' if Q is a subset of the vertex set of a connected component of G'. If there are r distinct available quorums $Q_1, \ldots, Q_r \in \mathcal{C}$ with respect to G' such that $Q_i \cap Q_j = \emptyset$ for $1 \le i, j \le r,\ i \ne j$, we say that G' is r-available. Then the (k, r)-availability of $\mathcal{C}$ on G is defined as follows :

$$R_{G,k,r}(\mathcal{C}) = \sum_{G' \text{is } r\text{-available}} P_r(V', E')$$

The (k, r)-availability depends on G. Because we assume that G is complete in this dissertation, we omit G from $R_{G,k,r}$. □

Note that the (1,1)-availability coincides with the availability.

Let S be a set of processes being in operation. Then, $F_{\mathcal{C},k,r}(S) = 1$ if and only if at least r processes can enter a critical section (i.e., $G' = (S, (S \times S) \cap E)$ is r-available) since the topology of the distributed system is a complete graph. On the other hand, the probability that the set of processes being in operation is exactly S is $p^{|S|}(1-p)^{n-|S|}$. Thus, the (k, r)-availability of a coterie $\mathcal{C}$ can be calculated using the following formula:

$$R_{k,r}(\mathcal{C}) = \sum_{S \subseteq U} F_{\mathcal{C},k,r}(S) p^{|S|}(1-p)^{n-|S|}.$$

Let $\mathcal{C}$ be a k-coterie, and r $(1 \le r \le k)$ be an integer. Now, we construct a new k'-coterie $\mathcal{C}'$ as follows:

First, let

$$\mathcal{C}' = \{Q \quad | \quad Q = Q_1 \cup \cdots \cup Q_r,\ Q_i \in \mathcal{C} \text{ for } 1 \le i \le r, \text{ and } Q_i \cap Q_j = \emptyset \text{ for } 1 \le i, j \le r,\ \ i \ne j\}.$$

Next, we remove all elements Q from $\mathcal{C}'$ such that $Q' \subseteq Q$ for some $Q' \in \mathcal{C}'$, in order for the resultant $\mathcal{C}'$ satisfying the minimality property. Then $\mathcal{C}'$ has the following properties.

Property 1 *$\mathcal{C}'$ is a $\lfloor \frac{k}{r} \rfloor$-coterie.* □

Property 2 *Let $k' = \lfloor \frac{k}{r} \rfloor$. Then,*

$$F_{\mathcal{C},k,r} = F_{\mathcal{C}',k',1}.$$

Hence,

$$R_{k,r}(\mathcal{C}) = R_{k',1}(\mathcal{C}').$$

□

We call $\mathcal{C}'$ *the r-contracted coterie* of $\mathcal{C}$.

3.2 k-Majority Coteries

We investigate k-majority coteries Maj_k in terms of the (k,r)-availability.

Theorem 1 *Let n be the number of processes, k be an integer such that $(n+1)$ is a multiple of $(k+1)$, and r $(1 \leq r \leq k)$ be an integer. Then, there is a constant $p_u(n,k,r)$ such that for any process reliability p $(p_u(n,k,r) \leq p \leq 1)$, Maj_k achieves the maximum (k,r)-availability. Hence, Maj_k is the best k-coterie in terms of the (k,r)-availability if $p \geq p_u(n,k,r)$, where*

$$p_u(n,k,r) = \frac{c(n,k,r)}{c(n,k,r)+1},$$

and

$$c(n,k,r) = \sum_{i=0}^{rW-1} \binom{n}{i}.$$

(Proof) Let $\mathcal{C}$ $(\neq \text{Maj}_k)$ be any k-coterie. We show that $R_{k,r}(\text{Maj}_k) \geq R_{k,r}(\mathcal{C})$ for any $p \geq p_u(n,k,r)$. Let $W = \lceil (n+1)/(k+1) \rceil$ (i.e., W is the size of each quorum in Maj_k).

Let $\mathcal{C}$ be any k-coterie such that $R_{k,r}(\mathcal{C}) > R_{k,r}(\text{Maj}_k)$. If every quorum Q in $\mathcal{C}$ had size $\geq W$, then $R_{k,r}(\text{Maj}_k) \geq R_{k,r}(\mathcal{C})$ would hold, because if $F_{\mathcal{C},k,r}(S) = 1$ then $F_{\text{Maj}_k,k,r}(S) = 1$ for any $S \subseteq U$, since $|S| \geq rW$. Therefore, there exists a quorum Q_0 with size $< W$ in $\mathcal{C}$.

First, we show that there exists a set S $(\subseteq U)$ with size rW such that $F_{\mathcal{C},k,r}(S) = 0$. Suppose that for any S with size rW, $F_{\mathcal{C},k,r}(S) = 1$ holds. Let $U_1 = U - Q_0$. Since $|U_1| \geq n - W + 1$, $|U_1| \geq kW$. Arbitrarily choose a set S $(\subseteq U_1)$ with size rW. Since $F_{\mathcal{C},k,r}(S) = 1$, there is a quorum Q_1 $(\subseteq S)$ in $\mathcal{C}$ whose size is at most W. Then we repeat this procedure for $U_2 = U_1 - Q_1$. In this way, we repeat this procedure $(k-r)$ times and can find a sequence of quorums $Q_0, ..., Q_{k-r}$ in $\mathcal{C}$. Clearly, $Q_i \cap Q_j = \emptyset$ for $0 \leq i,j \leq (k-r)$, $i \neq j$. Since $|Q_i| \leq W$ for $0 \leq i \leq (k-r)$,

$k,\ W,\ n$	$r=1$	$r=2$	$r=3$	$r=4$	$r=5$	$r=6$
$k=1,\ W=5,\ n=9$	0.9961089	—	—	—	—	—
$k=2,\ W=5,\ n=14$	0.9993207	0.9999329	—	—	—	—
$k=3,\ W=4,\ n=15$	0.9982669	0.9999390	0.9999689	—	—	—
$k=4,\ W=3,\ n=14$	0.9906542	0.9997121	0.9999226	0.9999386	—	—
$k=5,\ W=3,\ n=17$	0.9935484	0.9998937	0.9999847	0.9999918	0.9999924	—
$k=6,\ W=3,\ n=20$	0.9952830	0.9999539	0.9999962	0.9999987	0.9999990	0.9999990

Table 3.1: $p_u(n,r,k)$ for some n ($k=1,...,6,\ r=1,...,k$).

$|U_{k-r+1}| \geq rW$. Thus, there exist r quorums $Q_{k-r+1},...,Q_k(\in \mathcal{C})$ in U_{k-r+1} such that $Q_i \cap Q_j = \emptyset$ for $k-r+1 \leq i,j \leq k, i \neq j$. It is a contradiction, since $Q_i \cap Q_j = \emptyset$ for $0 \leq i,j \leq k,\ \ i \neq j$.

Then, there exists a set $S\ (\subseteq U)$ with size rW such that $F_{\mathcal{C},k,r}(S) = 0$. Let $\Delta = R_{k,r}(\text{Maj}_k) - R_{k,r}(\mathcal{C})$. Since $F_{\text{Maj}_k,k,r}(S') = 1$ for every S' with size rW, by definition,

$$\begin{aligned}\Delta &\geq p^{|S|}(1-p)^{n-|S|} - \sum_{i=0}^{rW-1}\binom{n}{i}p^i(1-p)^{n-i} \\ &\geq p^{rW}(1-p)^{n-rW} - c(n,k,r)p^{rW-1}(1-p)^{n-rW+1},\end{aligned}$$

where

$$c(n,k,r) = \sum_{i=0}^{rW-1}\binom{n}{i}.$$

It is easy to show that $\Delta \geq 0$ if

$$p \geq \frac{c(n,k,r)}{1+c(n,k,r)} = p_u(n,k,r).$$

□

Since $c(n,k,r) < c(n,k,r+1)$, the following corollary holds.

Corollary 1 *If $p \geq p_u(n,k,k)$, then* Maj$_k$ *is optimal in the sense of (k,r)-availability for all $1 \leq r \leq k$.* □

Table 3.1 shows $p_u(n,k,r)$ $(k=1,\ldots,6$ and $r=1,\ldots,k)$ for some n.

Theorem 2 *For any non-negative integer m, $(2m+1)$-majority coterie* Maj$_{2m+1}$ *achieves the maximum $(2m+1,m+1)$-availability, if the process reliability $p \geq \frac{1}{2}$ and $(n+1)$ is a multiple of $2(m+1)$.*

(Proof) Let $\mathcal{C}$ ($\neq$ Maj$_{2m+1}$) be any $(2m+1)$-coterie, and assume that $\mathcal{C}$ achieves a better $(2m+1, m+1)$-availability than Maj$_{2m+1}$ for some $p \geq \frac{1}{2}$. By $\mathcal{C}'$, we denote the $(m+1)$-contracted coterie of $\mathcal{C}$. Then by Property 1, $\mathcal{C}'$ is a 1-coterie. By definition of Maj$_k$, the $(m+1)$-contracted coterie of Maj$_{2m+1}$ is 1-majority coterie Maj$_1$, since $(n+1)$ is a multiple of $2(m+1)$. Since Maj$_1$ (i.e., majority coterie) achieves the maximum $(1,1)$-availability (i.e., availability) for all $p \geq \frac{1}{2}$ (Theorem 3.1 in [BGM87]), the $(1,1)$-availability of Maj$_1$ is not smaller than that of $\mathcal{C}'$, a contradiction by Property 2. $\square$

So far, we have derived a sufficient condition for k-majority coterie to be optimal in terms of the process reliability p. Now, we proceed to state a lower bound on the process reliability p for k-majority coterie to be optimal. We first present how to construct a new k-coterie $\mathcal{C}$ from k-majority coterie Maj$_k$, and then by comparing their (k,r)-availabilities, derive the necessary condition.

Arbitrarily choose n, k, and r (such that $(n+1)$ is a multiple of $(k+1)$, and fix them. We construct a k-coterie $\mathcal{C}$ from Maj$_k$ as follows: Let Q_0 be any quorum in Maj$_k$, and P_0 be any element in Q_0. Let Q_1 = $Q_0 - \{P_0\}$. Then,

$$\begin{aligned}\mathcal{C} &= \text{Maj}_k + \{Q_1\} - \{Q \in \text{Maj}_k \mid Q = Q_1 + \{P\},\ P \in U - Q_1\} \\ &\quad -\{Q \in \text{Maj}_k \mid Q \cap Q_0 = \{P_0\}\}.\end{aligned}$$

We compare their availabilities. Observe that $F_{\mathcal{C},k,r}(S) = 1$ for all $S \subseteq U$ with size at least $rW+1$, and that $F_{\mathcal{C},k,r}(S) = 0$ for all $S \subseteq U$ with size at most $rW-2$, where $W = (n+1)/(k+1)$ (i.e., the size of quorum in Maj$_k$). On the other hand, by definition, $F_{\text{Maj}_k,k,r}(S) = 1$ if and only if $|S| \geq rW$. Define Γ^+ and Γ^- as follows:

$$\begin{aligned}\Gamma^+ &= \{S \subseteq U \mid F_{\text{Maj}_k,k,r}(S) = 0 \ \&\ F_{\mathcal{C},k,r}(S) = 1\} \\ \Gamma^- &= \{S \subseteq U \mid F_{\text{Maj}_k,k,r}(S) = 1 \ \&\ F_{\mathcal{C},k,r}(S) = 0\}\end{aligned}$$

Note that by the observations, $|S| = rW-1$ if $S \in \Gamma^+$, and $|S| = rW$ if $S \in \Gamma^-$. Since Q_1 is the only quorum with size $W-1$ in $\mathcal{C}$, $S \in \Gamma^+$ if and only if $Q_1 \subseteq S,\ P_0 \notin S$, and $|S| = rW-1$, by definition of $\mathcal{C}$. Therefore,

$$\begin{aligned}|\Gamma^+| &= \binom{n-W}{rW-1-(W-1)} \\ &= \binom{kW-1}{(r-1)W}.\end{aligned}$$

Next, we show that $S \in \Gamma^-$ if and only if $Q_1 \cap S = \emptyset,\ P_0 \in S$ and $|S| = rW$. To show if part, assume that $F_{\mathcal{C},k,r}(S) = 1$ holds (since $F_{\text{Maj}_k,k,r}(S) = 1$). Since $P_0 \in S$, there is a quorum

Q containing P_0 in $\mathcal{C}$, a contradiction since $Q \cap Q_0 = \{P_0\}$. As for only if part, if either $P_0 \notin S$ or $Q_1 \cap S \neq \emptyset$, then one can easily find r quorums $G_1, ..., G_r$ in $\mathcal{C}$ such that $S = \bigcup_{i=1}^{r} G_i$ and $G_i \cap G_j = \emptyset$ for $1 \leq i, j \leq r, \ \ i \neq j$. Therefore,

$$\begin{aligned} |\Gamma^-| &= \binom{n-W}{rW-1} \\ &= \binom{kW-1}{rW-1}. \end{aligned}$$

By definition,

$$\begin{aligned} \Delta &= R_{k,r}(\mathcal{C}) - R_{k,r}(\text{Maj}_k) \\ &= |\Gamma^+| p^{rW-1}(1-p)^{n-(rW-1)} - |\Gamma^-| p^{rW}(1-p)^{n-rW} \\ &= p^{rW-1}(1-p)^{n-rW} \times \left\{ \binom{kW-1}{(r-1)W}(1-p) - \binom{kW-1}{rW-1} p \right\}. \end{aligned}$$

Therefore, $\Delta > 0$ if and only if

$$p > \frac{\binom{kW-1}{(r-1)W}}{\binom{kW-1}{(r-1)W} + \binom{kW-1}{rW-1}}.$$

□

Theorem 3 *Let n be the number of processes, k be an integer such that $(n+1)$ is a multiple of $(k+1)$, and r $(1 \leq r \leq k)$ be an integer. Then, there is a constant $p_l(n,k,r)$ such that for any process reliability p $(0 < p < p_l(n,k,r))$,* Maj$_k$ *does not achieve the maximum (k,r)-availability. Hence,* Maj$_k$ *is not the best k-coterie in terms of (k,r)-availability if $0 < p < p_l(n,k,r)$, where*

$$p_l = \frac{\binom{kW-1}{(r-1)W}}{\binom{kW-1}{(r-1)W} + \binom{kW-1}{rW-1}}.$$

□

Table 3.2 shows $p_l(n,k,r)$ ($k = 1, \ldots, 6$, $r = 1, \ldots, k$) for some n.

3.3 k-Singleton Coteries

This section shows a sufficient condition for k-singleton coteries to be optimal in terms of the process reliability p.

Theorem 4 *Let n be the number of processes, and k $(\leq n)$ and r $(1 \leq r \leq k)$ be integers. Then, there exists a constant $q(n,k,r) > 0$ such that (any) k-singleton coterie* Sgl$_k$ *is optimal for all process*

$k,\ W,\ n$	$r=1$	$r=2$	$r=3$	$r=4$	$r=5$	$r=6$
$k=1,\ W=5,\ n=9$	0.5000000	—	—	—	—	—
$k=2,\ W=5,\ n=14$	0.0078740	0.9921260	—	—	—	—
$k=3,\ W=4,\ n=15$	0.0060241	0.5000000	0.9939759	—	—	—
$k=4,\ W=3,\ n=14$	0.0178571	0.2631579	0.7368421	0.9821429	—	—
$k=5,\ W=3,\ n=17$	0.0108696	0.1538462	0.5000000	0.9891304	0.9891304	—
$k=6,\ W=3,\ n=20$	0.0072993	0.0099010	0.3373494	0.6626506	0.9009901	0.9927007

Table 3.2: $p_l(n,r,k)$ for some n ($k=1,...,6,\ r=1,...,k$).

reliability p $(0 \leq p \leq q(n,k,r))$. *Hence,* Sgl_k *is the best* k*-coterie in the sense of* (k,r)*-availability if* $p \leq q(n,k,r)$.

(Proof) Let $\mathcal{C}$ be any k-coterie which is not a k-singleton coterie. We show that there exists a constant $t > 0$ such that for all process reliability p $(0 \leq p \leq t)$, the (k,r)-availability of Sgl_k is larger than or equal to that of $\mathcal{C}$. The proof here is similar to that of Theorem 1.

Let $\Delta = R_{k,r}(\text{Sgl}_k) - R_{k,r}(\mathcal{C})$. By definition, for all S with size at most $r-1$, $F_{\text{Sgl}_k,k,r}(S) = F_{\mathcal{C},k,r}(S) = 0$. Define

$$\begin{aligned} m_0 &= \left|\{S \mid F_{\text{Sgl}_k,k,r}(S) = 1, |S| = r\}\right|, \text{ and} \\ m_1 &= \left|\{S \mid F_{\mathcal{C},k,r}(S) = 1, |S| = r\}\right|. \end{aligned}$$

Then, clearly, $m_0 > m_1$, since $\mathcal{C}$ is not a k-singleton coterie. Therefore, by definition,

$$\Delta \geq p^r(1-p)^{n-r} - \sum_{i=r+1}^{n} \binom{n}{i} p^i (1-p)^{n-i}.$$

It is easy to see that there is a constant t such that $\Delta \geq 0$ for all p $(0 \leq p \leq t)$.

Since the number of different k-coteries are finite, the theorem follows. □

3.4 Concluding Remarks

In this chapter, we investigated the goodness of two typical k-coteries, k-majority coteries and k-singleton coteries, in terms of the (k,r)-availability. Intuitive interpretation of the (k,r)-availability of a k-coterie is the probability that r processes can enter their critical sections (in spite of process failures).

We derived a necessary and a sufficient conditions on the process reliability p for k-majority coterie to achieves the maximum (k,r)-availability. We also showed that there is a constant q (> 0) such that for any process reliability $0 < p < q$, (any) k-singleton coterie achieves the maximum (k,r)-availability. The investigation revealed that k-majority ($k \geq 2$) is no longer optimal for all $p > \frac{1}{2}$. (As a matter of a result, 3-majority is not optimal even if $p = 0.9939$ for $n = 15$ and $r = 3$.)

Table 3.3 shows the (k,r)-availability of k-majority and k-singleton coteries when $n = 14$ and $k = 4$. It can be observed that as r increases, the process reliability p at which the (k,r)-availabilities

$k,\ r$		p										
		0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
$4,\ 1$	Maj$_k$	0.0000	0.1584	0.5519	0.8392	0.9602	0.9935	0.9994	1.0000	1.0000	1.0000	1.0000
	Sgl$_k$	0.0000	0.3439	0.5904	0.7599	0.8704	0.9375	0.9744	0.9919	0.9984	0.9999	1.0000
$4,\ 2$	Maj$_k$	0.0000	0.0015	0.0439	0.2195	0.5141	0.7880	0.9417	0.9917	0.9996	1.0000	1.0000
	Sgl$_k$	0.0000	0.0523	0.1808	0.3483	0.5248	0.6875	0.8208	0.9163	0.9728	0.9963	1.0000
$4,\ 3$	Maj$_k$	0.0000	0.0000	0.0004	0.0083	0.0583	0.2120	0.3373	0.6405	0.8883	0.9985	1.0000
	Sgl$_k$	0.0000	0.0037	0.0272	0.0837	0.1792	0.3125	0.4752	0.6517	0.8192	0.9477	1.0000
$4,\ 4$	Maj$_k$	0.0000	0.0000	0.0000	0.0000	0.0006	0.0065	0.0398	0.1608	0.4481	0.8416	1.0000
	Sgl$_k$	0.0000	0.0001	0.0016	0.0081	0.0256	0.0625	0.1296	0.2401	0.3164	0.5220	1.0000

Table 3.3: (k, r)-availabilities of Maj$_k$and Sgl$_k$($k = 4,\ n = 14$).

of Sgl$_k$ and Maj$_k$ reverse also increases. For example, the $(4, 4)$-availability of Sgl$_4$ is larger than that of Maj$_4$ even if $p = 0.7$, but the $(4, 1)$-availability of Maj$_4$ has already been larger than that of Maj$_4$ when $p = 0.3$. (This tendency can be shown formally.) Therefore, when we choose appropriate k-coteries in practical applications, we should take into account parameter r as an important one.

For simplicity of analysis, throughout the chapter we assume that $(n + 1)$ is a multiple of $(k + 1)$, when k-majority is investigated. It is strongly conjectured that the tendencies of k-majority in this chapter should hold for general k, and an analysis of this case is left as a future work.

Chapter 4

A Distributed k-Mutual Exclusion Algorithm using k-Coterie

In this chapter, we propose a distributed k-mutual exclusion algorithm which uses a k-coterie. Different from algorithms proposed in [Ray89a, Ray91a, SR92], the number of messages sent by processes can be smaller. Another advantage of this algorithm is that it provides so-called the graceful degradation property; since a critical section entrance request is granted if all members in a quorum grant it, even though a large part of the system are being down, there is a possibility that a process can enter a critical section.

4.1 The Distributed k-Mutual Exclusion Algorithm

To avoid deadlocks and starvations, the timestamp introduced by Lamport[Lam78] is used. Let t be the logical time at which a process P initiates a request. Then, the pair (t, P) is the timestamp attached to the request. Note that since an identifier of a process is unique, so is pair (t, P). As usual, we define a total order among timestamps by the lexicographical order assuming that the identifiers are non-negative integers.

Now, we present a detailed description of our algorithm. Our algorithm and Maekawa's algorithm [Mae85][1] are the same, except the following difference:

In Maekawa's algorithm, for each process P, a quorum Q is statically determined, and insists on gathering permission from all members in Q. This approach may be reasonable for solving the 1-mutual exclusion problem, since failing to gather permission from Q likely suggests that another process is being in a critical section, i.e., P cannot gather permission from any quorum. On the other hand, when the k-mutual exclusion problem is considered, insisting on Q does not seem to be a good idea, since although Q is busy, P may be able to find another quorum from which it can gather permission,

[1]It is shown that Maekawa's algorithm[Mae85] cannot avoid deadlocks [San87]. We adopt the version suggested by Sanders[San87].

because there are $(k-1)$ quorums which do not intersect with Q. Thus, our algorithm tries to find such a quorum.

Let $\mathcal{C}$ be a k-coterie. Each process P has local variables YES, NOTNOW, and PERM. Variables YES (resp. NOTNOW) keeps the set of processes which have agreed (by message OK) (resp. disagreed (by message WAIT)) on P entering a critical section, and variable PERM keeps the process (i.e., more rigorously, the REQUEST it initiates) that P has agreed on entering a critical section (by message OK) but has not yet received a message RELEASE stating that it has left the critical section, if there is such a process. Since P never give permission to two processes at a time, PERM is either empty or a singleton set. Initially, YES, NOTNOW, and PERM are the empty set. Note that P may receive OK messages from processes in NOTNOW. In such cases, these processes are moved from NOTNOW to YES. The process P also maintains a priority queue QUEUE for keeping REQUESTs in the order of their timestamps.

The algorithm is given in English as in many literatures (e.g., [Mae85]) to save space.

The Algorithm

- *When P wishes to enter a critical section:*

  It selects a quorum Q from $\mathcal{C}$, and sends REQUEST(t, P) to every member P_j in Q (including P itself), and waits for a reply (OK or WAIT) from P_j, where (t, P) is the timestamp (i.e., t is the current logical local time in P). If every P_j answers an OK, P can enter the critical section.

  If some processes answer WAITs, P adds the processes answering OK (resp. WAIT) to YES (resp. NOTNOW), selects another quorum Q' which minimizes $|Q \cap \text{YES}|$ from quorums in $\mathcal{C}$ not intersecting with NOTNOW (if there is such a quorum), and repeats the procedure from the first, except that this time, P sends REQUEST(t, P) only to members in $(Q' - \text{YES})$. (Hence, each process receives at most one REQUEST message from P.) If P cannot find a quorum satisfying the condition, then P waits for receiving OK messages.

  During the above procedure, P may receive an OK from a process P_j in NOTNOW. Then, P tests if a quorum is included in YES after moving P_j from NOTNOW to YES, and P can enter the critical section if the test succeeds.

- *When P leaves the critical section:*

  It sends a RELEASE message to each process in YES$\cup$NOTNOW.

- *When P receives REQUEST(t, P_j) from a process P_j:* Process P sends back OK, if PERM is empty, and adds REQUEST(t, P_j) to PERM.

  If PERM is $\{$REQUEST$(t_s, P_s)\}$, i.e., not empty, then P acts as follows. Process P inserts REQUEST(t, P_j) in QUEUE. Let REQUEST(t_r, P_r) be the request having the smallest timestamp (i.e., the one having the highest priority) among those in QUEUE. If $(t, P_j) > min\{(t_s, P_s), (t_r, P_r)\}$, then P sends back a WAIT to P_j. Otherwise, i.e., if REQUEST(t, P_j)

has the highest priority, P sends a message QUERY to resume the permission from P_s unless P_s is being in a critical section, and waits for a reply (RELINQUISH or RELEASE) from P_s. (If P has already sent a QUERY to P_s and is waiting for a reply, then no further QUERYs are necessary to send.) If P receives a RELINQUISH, then it exchanges REQUEST(t_s, P_s) and REQUEST(t, P_j), i.e., it moves REQUEST(t_s, P_s) from PERM to QUEUE and REQUEST(t, P_j) from QUEUE to PERM, sends a WAIT to all processes in QUEUE to which P has not sent a WAIT since the last QUERY was issued, and finally sends an OK to P_j.

- *When P receives a RELEASE message from P_j:*

  P removes the request from P_j in PERM. If QUEUE is not empty, then let REQUEST(t_r, P_r) be the request having the highest priority in QUEUE. Then, P moves it from QUEUE to PERM, sends an OK to P_r, and sends a WAIT to all processes in QUEUE to which P has not sent a WAIT since the last QUERY was issued.

- *When P receives a QUERY message from P_j:*

  If P is not in a critical section and P_j is in YES, then P moves P_j from YES to NOTNOW and sends back a RELINQUISH message to P_j. If either P is being in a critical section or P_j is not in YES, then P does nothing.

An example of implementation of this algorithm is shown in Appendix B.

4.2 Correctness proofs

Now, we show the correctness of the proposed algorithm. We show that the algorithm guarantees k-mutual exclusion, deadlock free, and starvation free.

Theorem 5 *The algorithm guarantees k-mutual exclusion.*

(Proof) Any process P can enter a critical section if and only if there is a quorum Q such that $Q \subseteq$YES. If more than k processes are being in critical sections at a time, then by definition of k-coterie, there are processes P and P_j such that YESs of P and P_j have a process P_r in common, a contradiction since if YES of a process P_s includes P_r then PERM of P_s contains a REQUEST from P as its only element. □

Theorem 6 *The algorithm is deadlock free.*

(Proof) Assume that a deadlock happens. Consider a directed graph whose nodes are processes and links are edges defined as follows: there exists an edge from P to P_j in the graph if and only if P_j has the permission of a process P_r and P is requesting it, i.e., P is waiting for its release. Since the system is in a deadlock state, there exists a cycle in the graph. Let $P_0, P_1, ..., P_{m-1}$ be processes that forms a cycle such that

$$P_0 \to P_1 \to \cdots \to P_{m-1} \to P_0,$$

and t_i be the priority (the timestamp at which a mutual exclusion request was issued) of process P_i. Note that without loss of generality, we can assume that no process P_i is in a critical section. (If such process P_i exists, it eventually exits from a critical section and releases the permissions it keeps and the cycle of the graph is broken in a finite time.)

Each process P preempts its permission having sent to a process P_j if a new request whose priority (defined by timestamp) is higher than P_j's. Since a cycle is formed, the permission which is kept by $P_{i+1 \bmod m}$ is not preempted by P_i for all i. But $t_i > t_{i+1 \bmod m}$ holds for each i, we have $t_0 > t_0$; a contradiction. □

Theorem 7 *The algorithm is starvation free.*

(Proof) Assume that there exists a process P which starves. In general, more than one process may starve. Without loss of generality, we assume that P's REQUEST is the one having the earliest (i.e., smallest) timestamp. Since the system is deadlock-free by Theorem 6, non-starving processes wishing to enter their critical sections will eventually enter them and therefore the timestamps they attach to REQUEST increase. Since REQUESTs are discarded when the corresponding RELEASEs arrive, the system will eventually reach a configuration such that the timestamp of P's REQUEST is the smallest one among those existing in the system not only now but also forever.

Let Q be the quorum that P selects. Then P sends a REQUEST to all members $P_j \in Q$, and all P_j will eventually receive the REQUEST and store them in their QUEUEs. As showed above, the system will eventually reach a configuration such that the timestamp of P's REQUEST is the smallest one in the system, and therefore P's REQUEST will eventually be moved to the head of QUEUE at each $P_j \in Q$. Process P_j returns an OK to P if its PERM is empty. Suppose that PERM contains a REQUEST from another process P_r. Then P_j sends a QUERY to P_r, it will eventually reach P_r, P_r will return either a RELINQUISH or a RELEASE, and finally it will eventually reach P_j, since P_r's REQUEST has a timestamp larger than P's REQUEST. In either case, P_j returns an OK to P. At P, a QUERY never arrive after an OK since the timestamp of P's REQUEST is the smallest even in a future. Now, a contradiction is derived since P will eventually receive OKs from all members $P_j \in Q$ and can enter its critical section. □

4.3 Message complexity

Let $\mathcal{C}$ be the k-coterie used in the algorithm. The number of messages required per mutual exclusion entrance is $3|Q|$ in the best case, since a process sends REQUEST, receives OK, and sends RELEASE, to and from all members of Q, where Q is the quorum in $\mathcal{C}$ selected by the process, as [Mae85]. Since there proposed an algorithm for constructing a k-coterie whose quorum size is $O(\sqrt{n}\log n)$ [FYA91], the message complexity of our algorithm become $O(\sqrt{n}\log n)$, in the best case.

When a process P fails to gather permission from all members in a quorum Q (i.e., when a WAIT message arrives), unlike Maekawa's algorithm, the algorithm selects another quorum and tries to gather permission from members of another quorum. Therefore, the algorithm is by no means efficient, as far as the worst case message complexity is concerned; $6n$ messages per critical section entrance is required, where n is the number of processes. (For example, the worst case occurs in a process P, when for all process P_j $(\neq P)$, P sends REQUEST to P_j, P_j sends QUERY to some process P_r, P_r sends RELINQUISH to P_j, P_j sends OK to P, P sends RELEASE to P_j, and P_j sends OK to P_r.)

This is definitely a serious problem, and in order to avoid such situations, we must bound the number of "retries" so that the total number of processes that P can send request messages is bounded by a reasonable function $c(n)$. It is easy to see that Theorems 5 – 7 hold, even if we bound the number of retries in terms of bounding function $c(n)$, provided that $c(n) \geq c$, where c is the maximum quorum size of C, and therefore, the number of messages required per critical section entrance is bounded from above by $6c(n)$, in the worst case. For instance, if we take $c(n) = |Q|$, where $|Q|$ is the size of a quorum, then the message complexity is $6|Q|$. But by bounding the number of retries, processes may be required to wait a longer time than our original algorithm, since processes may be able to find a free quorum by further retries.

4.4 Concluding Remarks

In this chapter, we proposed a distributed k-mutual exclusion algorithm based on the concept of k-coterie. The message complexity of our algorithm is $3c$ in the best case, and $6n$ in the worst case, where c and n are the maximum quorum size and the number of processes, respectively. The worst case message complexity, $6n$, is extremely bad, but by introducing a bounding function $c(n)$ $(\geq c)$ which bounds the number of processes to which a process can send a request, the worst case message complexity can be reduced to $6c(n)$, at the expense of the increase of waiting time for entering a critical section. An obvious open question is what $c(n)$ should be used for the purpose here.

In [Bal94b], Baldoni proposed a distributed algorithm for the k-out of-M resources allocation problem which requires $3\lceil n^{k/(k+1)} - 1\rceil$ in the best case and $5\lceil n^{k/(k+1)} - 1\rceil$ in the worst case. Manabe and Aoyagi also proposed the same definition of k-coterie independently [MA93] and proposed a distributed k-mutual exclusion algorithm which require $5|Q| + 3$ messages in the worst case and $3|Q| + 3$ in the best case where $|Q|$ is the size of quorum used in their algorithm.

In appendix A, we consider more general case in such a way that a set of resources avaiable to a process is different from processes. To this end, we introduce a concept of local coterie and propose a distributed resources allocation algorithm.

As a final remark, we would like to stress that there can be many different metrics to measure the goodness of k-mutual exclusion algorithm, besides the message and the time complexities. For example, from the view of fault tolerance, availability is considered to be a good measure for measuring the goodness of a k-coterie and investigated in the previous chapter. However, investigation of k-mutual exclusion algorithm using other metrics is still remained open, and this is left as a future work.

Algorithms	**Message Complexity**	
	the best case	the worst case
Raymond [Ray89a]	$2n - k - 1$	$2(n - 1)$
Raynal [Ray91a]	0	$3(n - 1)$
Srimani and Reddy [SR92]	0	$n + k - 1$
Baldoni [Bal94b]	$3\lceil n^{k/(k+1)} - 1\rceil$	$5\lceil n^{k/(k+1)} - 1\rceil$
Ours	$3\|Q\|$	$6n$

Table 4.1: Message complexities of disributed k-mutual exclusion algorithms

Chapter 5

Experimental Evaluation of the k-Mutual Exclusion Algorithm

In the previous chapter, a k-mutual exclusion algorithm using a k-coterie is proposed and its message complexities in the best and worst case are discussed. It is difficult to evaluate the average message complexity of distributed algorithms by analysis, in general. In this chapter, we evaluate the message complexity of the average case of the proposed distributed k-mutual exclusion algorithm by computer simulations. We also evaluate an algorithm by Raymond proposed in [Ray89a] and show the advantages of our algorithm.

5.1 Assumptions and the Simulation Model

In Chapter 2, we assumed that the distributed system assumed in Part I is totally asynchronous. To evaluate the average behavior of distributed algorithms, such assumption is not appropriate; we assume that each process shares the same time flow, i.e., the distributed system is synchronous. Note that the algorithm on the system is designed under the assumption of asynchrony. Because we assume a global clock, we can define a common time unit; a *quantum time* is a unit time used in this chapter.

The model of behavior of each process is as follows: Each process has four states (Normal, Requesting, In-CS and Exiting) and changes its states according to conditions.

- Normal state — When a process is in this state, it does not do active task, i.e., it is passive. If it receives a message from another process then it processes the message. But a mutual exclusion request happens with probability p $(0 \leq p \leq 1)$ every quantum time. If a mutual exclusion request happen, the state become Requesting state.

- Requesting state — This is the state that a process is executing a procedure for mutual exclusion request (e.g., sending request messages, waiting permissions, etc.). When a process successfully enters a critical section, the state become In-CS state.

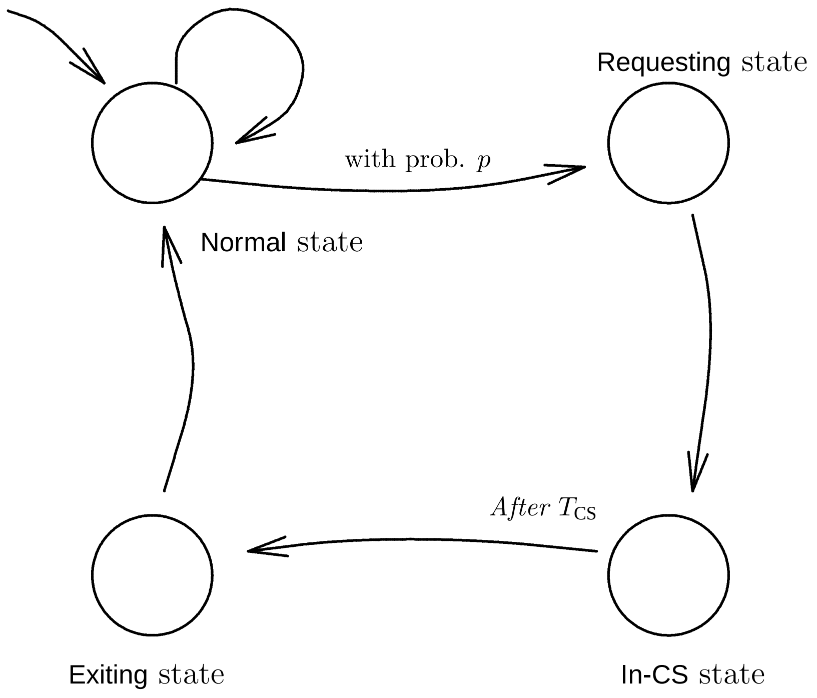

Figure 5.1: The behavior of a process

- In-CS state — When a process is in a critical section, it is in this state. After some specified time T_{CS} is passed after entering a critical section, the process comes out a critical section and its state become Exiting state.

- Exiting state — A process is in this state when it is executing a procedure of exiting a critical section such as returning permissions. After finishing an exiting procedure, the state become Normal state.

The behavior of a process is illustrated in Figure 5.1

5.2 Outline of the Simulation System

In this section, the simulation system is briefly described. Since the purpose of this chapter is not discussing a simulation method itself, we describe the outline of the design and implementation of the system.

The simulation system is executed on several workstations that are interconnected by a local area network. Processes are executed on different workstations, i.e., when a distributed system which consists of n processes is simulated, n workstations are used. (See Figure 5.2.) Therefore, each process is executed truly in parallel.

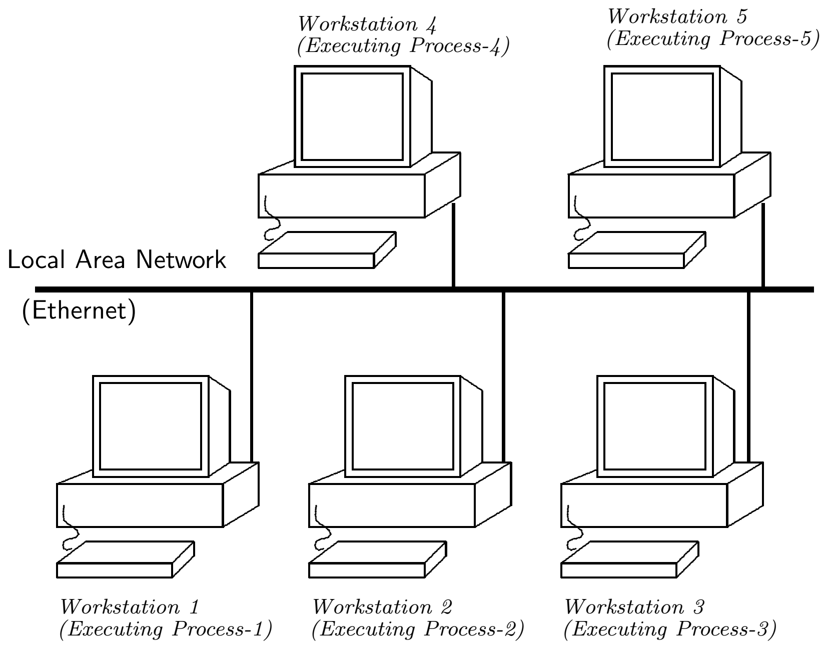

Figure 5.2: The simulation system (in the case $n = 5$)

As described above, we are assuming that the speed of time flow at each process is the same. To implement such situation, one of solutions is letting the time flow of a process be the same as (or proportional to) that of real time. We let the time unit at processes be T_Q second. (In our experiment, one unit of time, T_Q is 1 second.) Therefore, the speed of time flow at a process does not depend on the processing speed of workstations, i.e., the same time flow is guaranteed. Each workstation has real time clock; therefore implementation is easy. Since 1 second is enough long time for CPUs, the local computation time at processes is negligibly short.

The message exchange between processes are implemented by inter process communication facilities[Sun90]. Since *stream* communication is synchronous, if two processes try to send message at the same time then these processes fail into deadlock state; a process waits for message reception of the other process, and the other process waits for message reception of another one. Therefore, message passing must be asynchronous. Thus, message exchange between processes is implemented by using asynchronous *datagram* communication.

The simulation program is written in programming language C. An executable file is placed at each workstation and executed by remote execution feature. Program fragments of implementation of the proposed algorithm and Raymond's algorithm are shown in Appendix B.

5.3 The Distributed k-Mutual Exclusion Algorithm by Kerry Raymond

In this section, we briefly explain the distributed k-mutual exclusion algorithm proposed by Raymond [Ray89a].

In her algorithm, sequence number ([Lam78]) is used to avoid deadlock and starvation. A process X wishing to enter a critical section sends a `REQUEST` message to the other $n-1$ processes, where n is the number of processes in the distributed system. When a process Y receives a `REQUEST` message, it sends a `REPLY` message unless it is in a critical section or requesting a mutual exclusion with higher sequence number than X's sequence number. Otherwise, Y defers sending a `REPLY` message to X.

The process X can enter its critical section if it receives $n-k$ `REPLY` messages. Since $n-k = (n-1)-(k-1)$, receiving $n-k$ `REPLY` messages guarantees that the number of processes which are not in their critical sections nor are requesting with higher priority is less than k. Thus, X can enter its critical section.

Since a process enters a critical section if it receives only $n-k$ `REPLY`, it may receive `REPLY` messages when it is in a critical section, after exiting a critical section, or when it is requesting next mutual exclusion, and so on. The algorithm is designed to ignore such delayed messages. See [Ray89a] in detail.

It is easy to see that the algorithm require at least $2n-k-1$ messages per mutual exclusion invocation. In the worst case, $2(n-1)$ messages are necessary. This method is not fault-tolerant comparing with our algorithm because alive processes are not in operational if arbitrary k processes are stopped.

5.4 Simulation and Results

Conditions of the experiment are as follows:

- a quantum time T_Q is 1 second,
- T_{CS}, the time that a process is in a critical section, is 1 quantum time,
- a k-coterie used by our algorithm is the k-majority coterie, and
- the experiment is done for 500 quantum time.

Because k-majority coterie is a coterie whose quorum size is not small, the message complexity of our algorithm become smaller if we use a coterie whose quorum sizes are smaller. We use k-majority coterie because it is simple.

The experiment is done for:

- $k = 2$, $n = 5, 8, 11$,
- $k = 3$, $n = 7$, and

- $k = 4, n = 9$.

For each experiment, p, the probability of mutual exclusion request, is varied from 0.01 to 1.0. Workstations used for the experiment are 7 AV-300's (Nippon Data General) and 4 DS-7400's (Nippon Data General) on which the DG/UX operating system (version 4.32 for AV-300, version 4.02 for DS-7400) is available.

Under conditions as described above, the total number of messages sent during the experiment and the number of entrance of critical sections are counted. From these two data, the average number of messages per mutual exclusion invocation is calculated. Let this value be μ, which is computed by the following formula.

$$\mu = \frac{\sum\limits_{1 \le i \le n} M_i}{\sum\limits_{1 \le i \le n} C_i},$$

where M_i is the number of messages that process i sends and C_i be the number of times that process i enters a critical section during the experiment $(1 \le i \le n)$.[1]

Results of the experiment are shown in Figure 5.3 – Figure 5.7.

In case that p is small (for instance, in case of $k = 4,\ n = 9$; see Figure 5.7), μ (the number of messages which our algorithm requires to enter a critical section) is much smaller than that of Raymond's algorithm, as expected. Figure 5.7 shows that it achieves the best case $3|Q| = 6$ when $p = 0.01$. We can see from figures that μ gradually increases with the increase of p if p is small (for instance, $p < 0.2$ in case of $k = 4,\ n = 9$). But when p become larger, μ suddenly increases and when p comes near to 1.0, μ saturates. This observation is described as follows. When p is enough small, mutual exclusion requests do not collide often. In addition to it, even if a process fails to get permissions from a quorum, the probability that it gets permissions from a next quorum is large. Therefore, the number of additional messages is rather small. But p increases, collisions often happen and the probability that processes choose another coterie but fails to get permissions become large and preemption also happens often; this cause a sudden increase of μ.

Consider the case that k is fixed and n increases (see Figure 5.3, Figure 5.4, and Figure 5.5). In this case, the increase of n causes the increase of the probability of collision of mutual exclusion requests. Therefore, μ increases. Let p_{xover} be a probability that the number of message of our algorithm become larger than that of Raymond's. We call p_{xover} *cross over probability*. In the case of $k = 2$, the cross over probabilities can be found from figures and shown them in Table 5.1 It is interesting that the product of the number of processes and cross over provability is almost the same. From this observation, the message complexity of our algorithm depends of the total probability of mutual exclusion requests in the distributed system. It is easily guessed that the product of n and p_{xover} depends of the k-coterie the algorithm uses, however, we use this observation to guess the range of p such that our algorithm is more efficient than Raymond's algorithm in the sense of message complexity.

[1] For convenience, let process identifier be an integer between 1 and n.

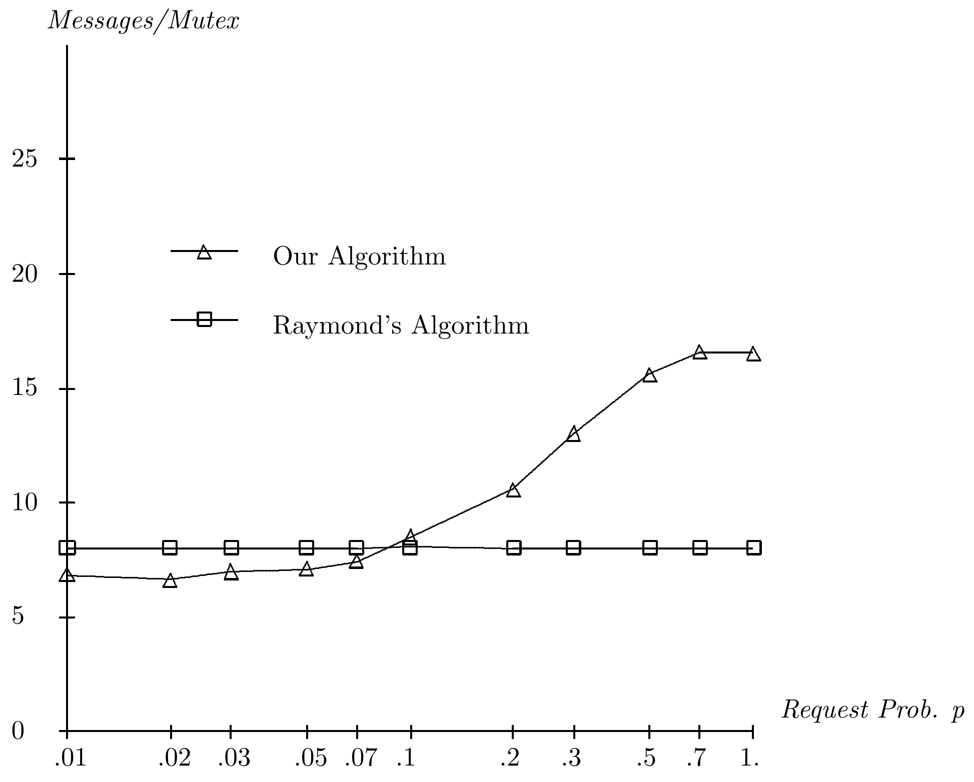

Figure 1. The Number of Messages

Figure 5.3: The average number of messages ($k = 2$, $n = 5$).

n	p_{xover} (cross over probability)	$n \cdot p_{\text{xover}}$
5	$\approx$ 0.9	$\approx$ 0.45
8	$\approx$ 0.6	$\approx$ 0.48
11	$\approx$ 0.4	$\approx$ 0.44

Table 5.1: Cross over probabilities for $k = 2$

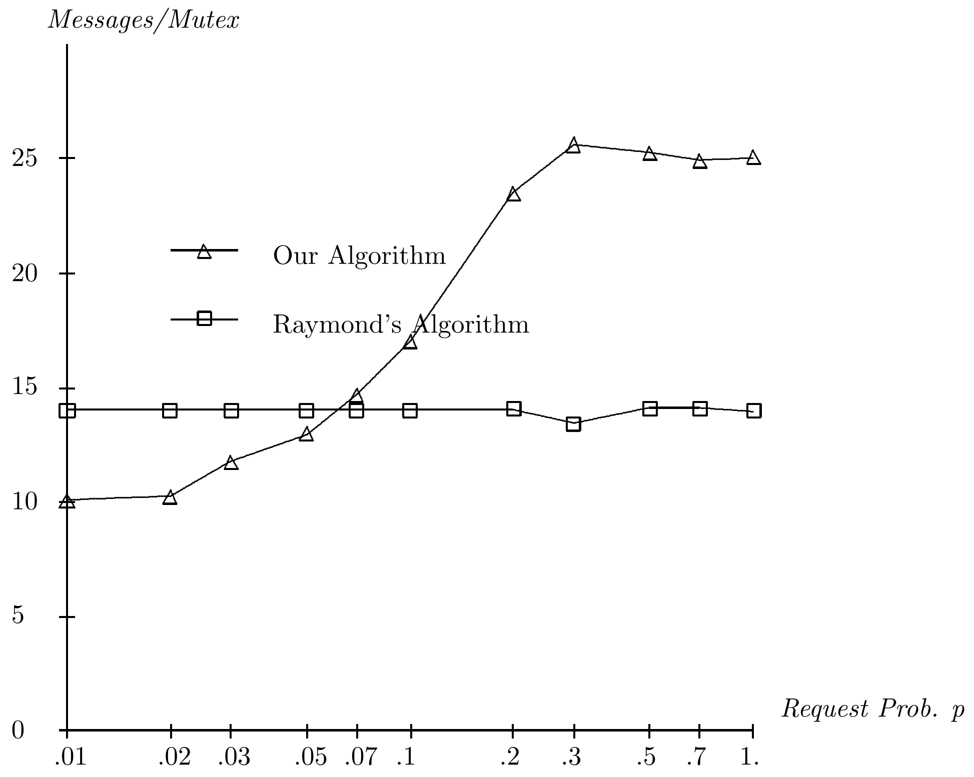

Figure 1. The Number of Messages

Figure 5.4: The average number of messages ($k = 2$, $n = 8$).

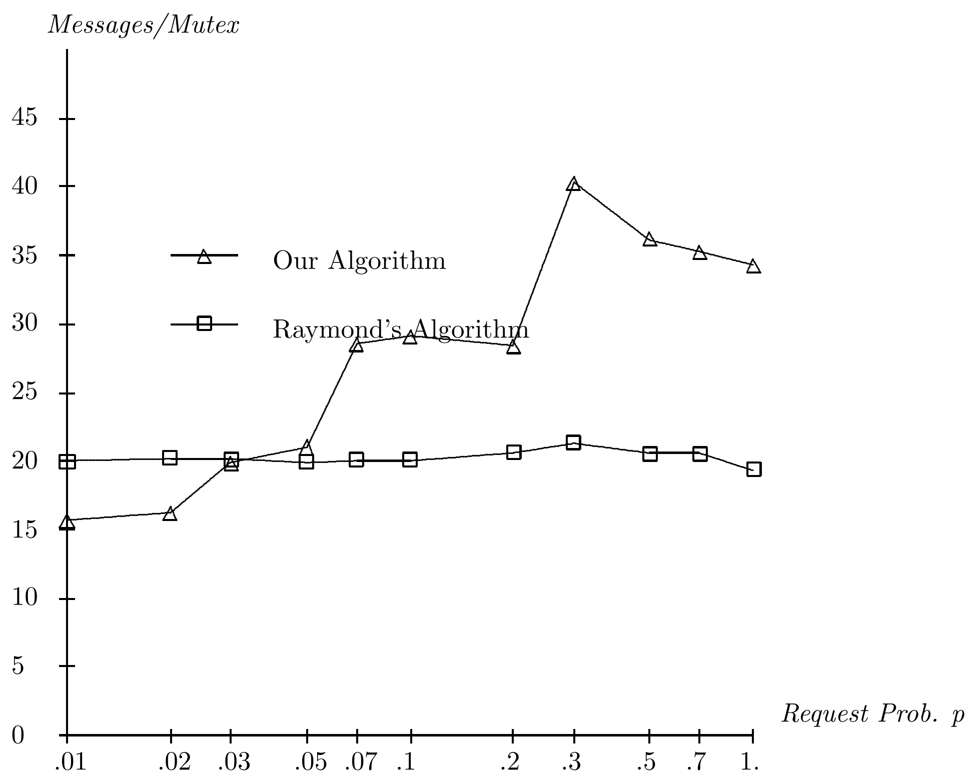

Figure 1. The Number of Messages

Figure 5.5: The average number of messages ($k = 2$, $n = 11$).

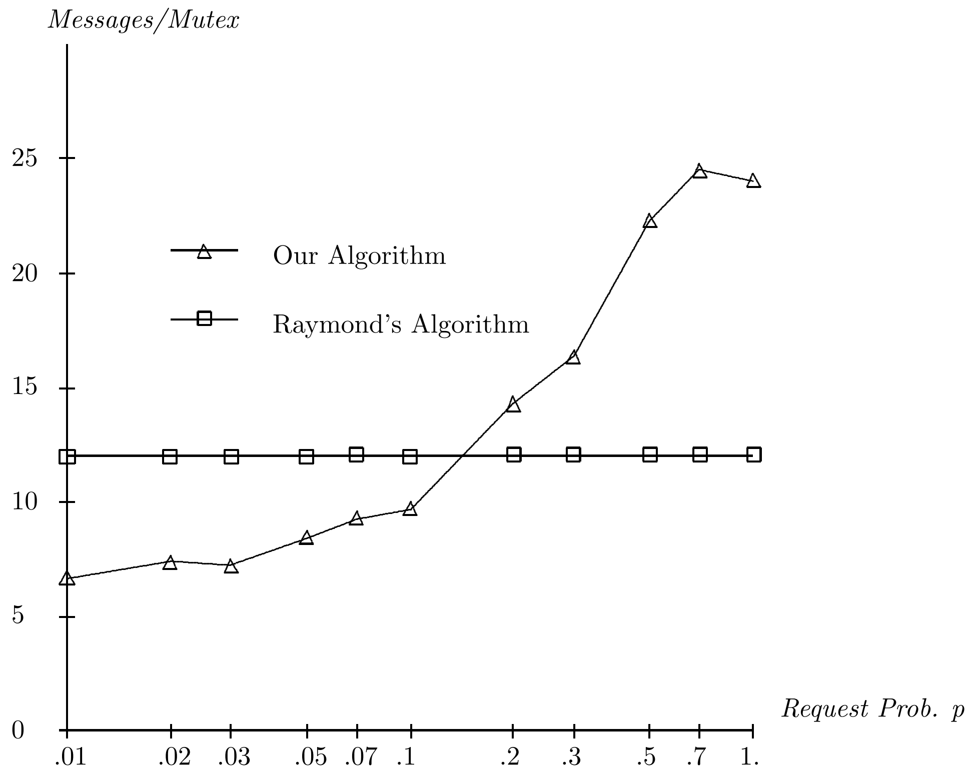

Figure 1. The Number of Messages

Figure 5.6: The average number of messages ($k = 3$, $n = 7$).

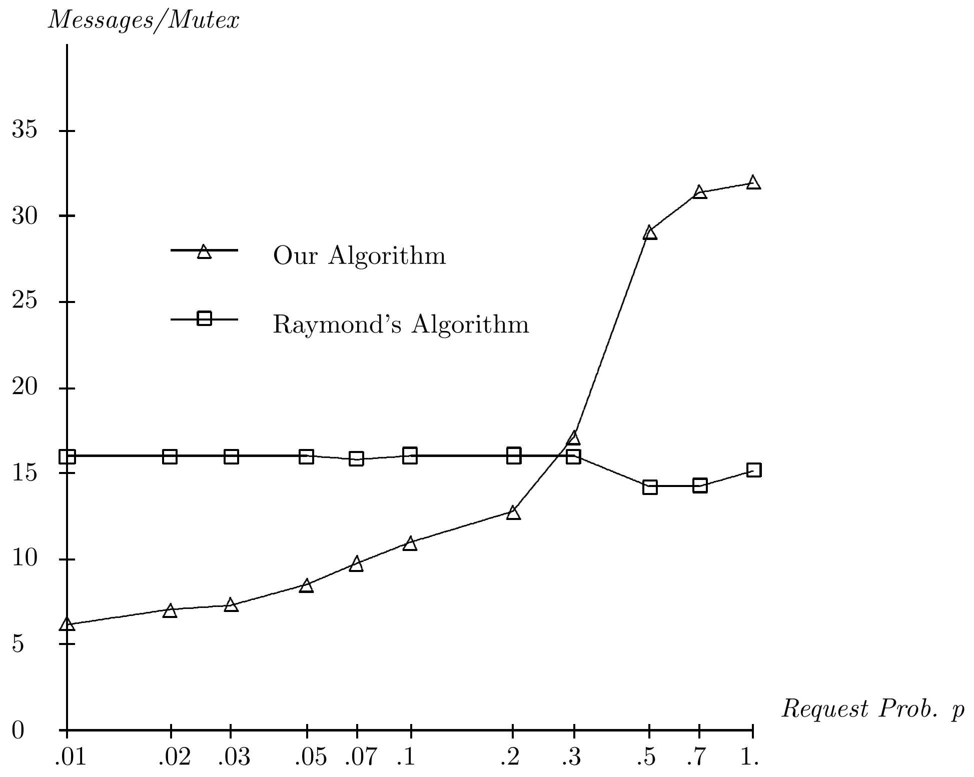

Figure 1. The Number of Messages

Figure 5.7: The average number of messages ($k = 4$, $n = 9$).

The larger k becomes (for instance, compare cases $k = 2,\ n = 5$ and $k = 4,\ n = 9$; see figure 5.3 and figure 5.7), the smaller μ becomes if p is small. This is why that the size of quorum become smaller if k become larger. Note that the k-majority coterie is used in this experiment. If we use another coteries whose quorum size is small, μ becomes smaller.

It is shown that our algorithm requires $6n$ messages per mutual exclusion invocation. Even if $k = 2$ and $p = 1.0$, the number of messages per mutual exclusion invocation is approximately $3n$ which is half of the worst case message complexity $6n$.

5.5 Concluding Remarks

In this chapter, we evaluated our distributed k-mutual exclusion algorithm which uses k-coteries. As a drawback of our algorithm, the number of messages become much larger than Raymond's algorithm requires. But the probability of mutual exclusion is small and k is large, our algorithm require less messages. Since we can choose a k-coterie whose quorum size is small, the number of required messages can be reduced.

The time between the time of mutual exclusion request happen and the time of entrance of critical section can be considered as a measure of evaluation of mutual exclusion algorithm. But the simulation model we adopted is not appropriate to evaluate the time because the delivery time of messages are much smaller that time unit. To evaluate such measure, we need another simulation model and a simulation system implementing such model. This is left as a future task.

Part II

The Self-Stabilization Approach

Chapter 6

The Self-Stabilization Approach for the Distributed k-Mutual Exclusion

A self-stabilizing system is a system which converges to a legitimate (correct) system state even if the system starts from an arbitrary system state. The concept of self-stabilizing systems is proposed by Dijkstra in [Dij74]. Even if a system state changes from a legitimate state to a non-legitimate state by transient failures (e.g., message omission, restart of process, etc.), the system starts the execution of a self-stabilizing algorithm from the state and eventually reaches to a legitimate state again. Thus, self-stabilizing systems are resilient to any transient failures. Since the fault-tolerance of distributed systems is an important issue, the study of self-stabilizing systems getting more active.[1]

In this chapter, we summarize computational models used in studies of self-stabilizing systems. Next, we give a review of previous works for the self-stabilizing mutual exclusion problem which are related to this dissertation. Finally, we give formal definitions of computational models and the self-stabilizing k-mutual exclusion problems used in Part 2.

6.1 Computational Models

Usually, distributed algorithms adopt an asynchronous message passing model for information exchange between processes. Self-stabilizing algorithms, however, adopt the following models for communications.

- **State communication model** — A communication model such that every process can know its neighbors' states. There is no explicit message sending/receiving steps in the description of an algorithm based on this model. It is assumed that neighbors' states can be known without time delay.

[1]A term *self-stabilizing algorithm* formally refers to just an algorithm which has self-stabilizing property executed by processes and a term *self-stabilizing system* formally refers to a system consisting of a network (processes and communication links) and a self-stabilizing algorithm executed by a process. In this dissertation, we use terms "self-stabilizing systems" and "self-stabilizing algorithms" interchangeably.

- **Register communication model** — A communication link between process P_A and P_B is assumed to consist of two registers $R_A and R_B$. When P_A sends a message to P_B, P_A writes data into the register R_A. To receive a message from P_A, P_B reads the register R_A. To send a message to P_A, P_B writes data into the register R_B and then P_A read from it.

- **Message communication model** — Processes use an asynchronous message passing to exchange information. Since it is asynchronous, the delay for message delivery is finite but it cannot be predicted.

Since distributed systems consist of more than one processes, a scheduling of executions of processes is one of an important issues in designing distributed algorithms. The following models are proposed as schedulers (adversaries).

- **Central daemon (or, c-daemon)** — A scheduler such that only one process is chosen to be executed at each step. A process can read states of all its neighbor processes and updates its state in one step.

- **Distributed daemon (or, c-daemon)** — A scheduler such that arbitrary number of processes are chosen to be executed at each step. A process can read states of all its neighbor process and updates its state in one step.

- **Read/write daemon (or, r/w-daemon)** — This model can be adopted if a communication model is the register communication model. At each step, only one process is chosen to be executed and each process can take an action such that an internal transition followed by reading from or writing to a register.

Many distributed algorithms assume the existence of unique process identifier. The following models related to process identifier have been considered.

- **Uniform** — There is no process identifier and every process has the same algorithm. Thus, all processes are completely identical.

- **Semi-uniform** — All process except one or several (constant) number of processes are identical. Special process(es) has different algorithm from other processes.

- **Unique identifier** — Every process has a unique process identifier.

6.2 Previous Works

In this section, we review the previous works for the self-stabilizing mutual exclusion problem.

The first paper in which self-stabilization is proposed is [Dij74] by Dijkstra in 1974. He proposed a self-stabilizing algorithm on bidirectional rings which solves the mutual exclusion problem. In his paper, he introduced daemons as models of scheduler and his algorithm based on state communication, c-daemon, and semi-uniform model. In a ring network, he assumed a special process called the "bottom" process. The idea of the algorithm is as follows. Depending on a relation with neighbors' states, a process is said to have an token if a predicate holds. By the execution of a process, a token circulates along a ring. If a token arrives at a bottom process, the moving direction of a token is reflected by the bottom process. If two tokens collide, one token disappears and the other token remains. Thus, if there are more than one tokens on the ring then the number of tokens decrease by collisions of tokens and eventually the number of tokens become one. Since the number of tokens is at least one by the construction of the algorithm, the number of tokens in a ring become one which is a legitimate configuration.[2] He showed algorithms which require K states (where $K > n$ and n is the number of processes), 4 states, and 3 states.

It is desirable that there is no exceptional process in a distributed system. A distributed system is *uniform* if every process has the same algorithm and no process identifier. Dijkstra showed that there is no uniform deterministic self-stabilizing mutual exclusion algorithm on a ring network whose size is composite[Dij82]. (The same result can be seen in [BP89].) Burns and Pachl proposed a uniform deterministic self-stabilizing mutual exclusion algorithm on unidirectional ring networks whose size is prime assuming state communication model under c-daemon. The proposed algorithm require $O(n^2)$ states for each process, and then, they showed a method of reducing the number of states. Finally, they obtained an algorithm requiring approximately $n^2/\ln n$ states. It is shown by Ceger that a deterministic self-stabilizing mutual exclusion algorithm assuming state communication model under c-daemon require at least $n-1$ state [Bur94]. Burns and Pachl pointed out that there is a gap between lower bound and upper bound of the number of states and this is still an open problem. Recently, Huang proposed a uniform deterministic self-stabilizing mutual exclusion algorithm on bidirectional rings [Hua93]. His algorithm is a composition of a leader election algorithm and Dijkstra's self-stabilizing mutual exclusion algorithm. Since the ring is uniform, a distinguished process assumed by Dijkstra's algorithm is elected by a leader election algorithm. The number of states that Huang's algorithm requires is $3n$.

Since determinism and uniformity are strong requirements for self-stabilizing systems, self-stabilizing systems with relaxed requirements has been proposed. In addition, not only ring networks but general networks are also considered in other researches.

Israeli and Jalfon proposed a self-stabilizing mutual exclusion algorithm on general networks by random walk of token [IJ90]. A process which have a token can be considered as having a privilege to enter a critical section and it sends a token to a neighbor process which is randomly chosen. A token is eliminated when two tokens collide. Even if there are more than one tokens in a network, it is expected that they collide with high probability. They showed that (1) the upper bound of the expected steps that the number of tokens converges to one, and (2) the (exact) expected steps that the number of tokens converges to one in the case that the network is a bidirectional ring.

Dolev, Israeli and Moran proposed a semi-uniform self-stabilizing mutual exclusion algorithm on general networks [DIM90, DIM93]. They assumed a special process in networks. Their algorithm is dynamic in the sense that it tolerates changes of networks (addition and/or removal of processes and links) during execution of the algorithm provided that a special process never removed. Their

[2]A system state (tuple of states of all processes) is called a *configuration*. Formal definition is given in the next section.

algorithm is composed of two self-stabilizing algorithms: a self-stabilizing spanning tree algorithm and a self-stabilization mutual exclusion algorithm based on random walk of a token on a spanning tree.

Nishikawa, Masuzawa and Tokura proposed a uniform self-stabilizing probabilistic leader election algorithm on tree networks and complete networks [NMT92]. It is observed that the self-stabilizing mutual exclusion cannot be solved deterministically on symmetry networks [Dij74]. The proposed algorithm by Nishikawa et al. uses randomization to break a symmetry. They showed that a composition of their uniform self-stabilizing leader election algorithm and semi-uniform mutual exclusion algorithm proposed in [DIM90] yields a uniform mutual exclusion algorithm.

Herman proposed a uniform self-stabilizing probabilistic mutual exclusion algorithm on ring network whose size is odd in [Her90]. He assumed that every process is executed synchronously. Each process has only one bit as a state, i.e., the number of states is two.

Not only using randomization to provide self-stabilizing property, a special network topology is proposed. For instance, Ghosh proposed a deterministic self-stabilizing mutual exclusion algorithm and on a special network topology in [Gho91].

6.3 Preliminaries

In this section, we give formal definitions of concepts and terms used in the self-stabilization approach.

6.3.1 The process and network model

A *unidirectional uniform ring system* is a triple, $R = (n, \delta, Q)$ where n is the number of processes in the system, δ is a transition algorithm, Q is a finite set of state of process. The processes are arranged on a ring, i.e., processes $P_0, P_1, ..., P_{n-1}$ are arranged in a clockwise manner. (Right is clockwise direction and left is counterclockwise direction.) Let Q_i be the state set of process P_i. Note that $Q_i = Q_j$ for all i, j, but we use this notation for the simplicity of explanation. The systems is called *uniform* since the δ and Q are the same for every process.

A *configuration* of R is an n-tuple of a state of processes; a state of process P_i is $q_i \in Q_i$ then a configuration of the system is $\gamma = (q_0, q_1, ..., q_{n-1})$. Let Γ be the set of all configurations, i.e., $\Gamma = Q_0 \times Q_1 \times \cdots \times Q_{n-1}$. The transition algorithm δ of a process is given by a set of *guarded commands*:

$$\begin{array}{l} \text{IF } \langle \text{guard}_1 \rangle \text{ THEN } \langle \text{command}_1 \rangle \\ \text{IF } \langle \text{guard}_2 \rangle \text{ THEN } \langle \text{command}_2 \rangle \\ \qquad \vdots \\ \text{IF } \langle \text{guard}_m \rangle \text{ THEN } \langle \text{command}_m \rangle \end{array}$$

Guards are predicates $g_j(q_i, q_{i-1})$ and commands are assignment statements $q_i := f_j(q_i, q_{i-1})$. A uniform ring system is called a *randomized uniform ring system* if random bit generator is used in a command. To describe randomized behavior of processes when we write algorithms, we assume that a random bit generator is provided as a primitive function. Especially, the uniform ring system is called a *deterministic uniform ring system* if random bit generator is not used in any commands.

A *bidirectional uniform ring system* is defined similarly. Guards are predicates $g_j(q_i, q_{i-1}, q_{i+1})$ and commands are assignment statements $q_i := f_j(q_i, q_{i-1}, q_{i+1})$.

6.3.2 Scheduling of processes

It is said that P_i has a *privilege* at a configuration γ if and only if $g_j(q_i, q_{i-1})$ for some $1 \leq j \leq m$. P_i can execute (change state) only when it has a privilege. In general, there exists more than one process which have privilege. In this dissertation, we consider the following types of scheduler in order to choose processes to be executed:

- c-daemon (central daemon) — the scheduler chooses any process among privileged processes and let the process execute.
- c-dragon (central dragon) — the scheduler chooses a process among privileged processes with uniform probability and let the process execute.

A scheduler chooses a process which has a privilege and executes a command whose guard is true. Even if more than one one guard is true, only one command is chosen and be executed. After the execution, assume that the state of P_i is changed to q. Then, the configuration becomes

$$\gamma' = (q_0, q_1, ..., q_{i-1}, q, q_{i+1}, ..., q_m)$$

This relation between configurations is denoted by $\gamma \to \gamma'$. The transitive closure of the relation $\to$ is denoted by $\to^*$. To explicitly describe that the transition is made by process P, we write $\xrightarrow{P}$. A *computation* or a *transition sequence* Δ starting from $\gamma_0 \in \Gamma$ is an infinite sequence of configuration $\gamma_0, \gamma_1, ...$, where $\gamma_j \to \gamma_{j+1}$ for all $j \geq 0$.

6.3.3 The self-stabilizing k-mutual exclusion problem

Let Λ be a set of configurations of a uniform ring system $R = (n, \delta, Q)$. A deterministic uniform ring system R is a *deterministic self-stabilizing mutual exclusion system* for Λ if and only if all of the following conditions hold:

- **No Deadlock:** For any configuration $\gamma \in \Gamma$, there exists at least one $\gamma' \in \Gamma$ such that $\gamma \to \gamma'$.
- **Closure:** For any $\gamma \in \Lambda$ and $\lambda' \in \Gamma$, $\gamma \to \gamma'$ implies that $\gamma' \in \Lambda$.
- **No Livelock:** For any $\gamma_0 \in \Gamma$ and any (infinite) computation $\Delta = \gamma_0, \gamma_1, ...$, there exists a j such that $\gamma_j \in \Lambda$.
- **Fairness:** For any $\lambda_0 \in \Lambda$ and any (infinite) transition sequence $\Delta = \lambda_0, \lambda_1, ...$ and any process P_i $(0 \leq i < n)$, there exists infinite transitions made by P_i.

- **k-mutual Exclusion:** For each configuration $\lambda \in \Lambda$, the number of processes which have a privilege at λ is exactly k.

The set of configurations Λ is called a set of *legitimate configurations* since the system takes a configuration in Λ when the system is stabilized.

A randomized uniform ring system R is a *randomized self-stabilizing mutual exclusion system* for Λ if and only if all of the following conditions hold:

- **No Deadlock:** (same as deterministic version)
- **Closure:** (same as deterministic version)
- **No Livelock:** For any $\gamma_0 \in \Gamma$, let $\mathcal{D}$ be the set of all possible (infinite) computation $\Delta^i = \gamma_0^i, \gamma_1^i, ...$ and d_i be the smallest index of configuration such that $\gamma_{d_i}^i \in \Lambda$ for each Δ^i. Then, the expected value of d_i for each $\Delta^i \in \mathcal{D}$ is finite.
- **Fairness:** (same as deterministic version)
- **Mutual Exclusion:** (same as deterministic version)

When a ring system $R = (n, \delta, Q)$ is self-stabilizing mutual exclusion systems for Λ, the system S is denoted by four tuple $S = (n, \delta, Q, \Lambda)$.

We define a self-stabilizing k-mutual exclusion problem with additional requirement. Let $\Pi(\lambda)$ be a set of processes which have a privilege at a configuration λ and V be a any set of k processes. *Type-2 self-stabilizing k-mutual exclusion problem* is a problem such that there exists a computation starting from any legitimate configurations $\lambda \in \Lambda$ which reaches a configuration λ', where $\Pi(\lambda') = V$. *Type-1 self-stabilizing k-mutual exclusion problem* is a problem without this requirement. Note that type-1 and type-2 are the same when $k = 1$.

Type-2 problem requires that there must exist a computation which reaches any arrangement of privilege from any legitimate configuration. As we will show, there is no algorithm which solves type-2 problem on unidirectional rings.

Chapter 7

Self-Stabilizing Mutual Exclusion Algorithms

7.1 Self-Stabilizing k-Mutual Exclusion Algorithms

In this section, we propose (deterministic) self-stabilizing k-mutual exclusion algorithms under a c-daemon on unidirectional and bidirectional ring networks. The solution is not trivial by the following reasons. (1) If the number of tokens is less than k, the number of tokens must be increased. This implies that token collision scheme cannot be applied simply. (2) When the number of tokens is exactly k, collision of tokens must be avoided. (3) Otherwise, the number of tokens must be decreased.

Since it is easy to show that there is no self-stabilizing k-mutual exclusion algorithm under a c-daemon, we assume a fair schedule of a c-daemon.[1] The proposed algorithms are based on the algorithm by Burns and Pachl's uniform deterministic self-stabilizing 1-mutual exclusion algorithm [BP89]. First, we cite their algorithm and explain it because it is necessary in the proofs of our algorithms.

7.1.1 Burns and Pachl's Algorithm

The self-stabilizing k-mutual exclusion algorithms proposed in 7.1.2 and 7.1.3 is based on the self-stabilizing mutual exclusion algorithm proposed by Burns and Pachl in [BP89]. Before describing our k-mutual exclusion algorithms, we cite Burns and Pachl's algorithm $S_0 = (n, \delta_0, Q_0, \Lambda_0)$ first. In the rest of this section, we call Burns and Pachl's algorithm as *BP*.

Let $n \geq 5$ be the prime number of processes. A set of states is Q_0 and a state $q_i \in Q_0$ of each process P_i is a tuple $l_i.t_i$, where $l_i \in \{0, 1, ..., n-2\}$ and $t_i \in \{0\} \cup \{2, 3, ..., n-2\}$. The first field l_i is called *label* and the second fields t_i is called *tag*.

For the simplicity of description of the algorithm, we define the following predicates:

$$R_A(i) \quad \equiv \quad (l_i \neq l_{i-1} + 1) \wedge (l_i \neq 0 \vee t_{i-1} = 0 \vee t_{i-1} \neq l_i - l_{i-1} \vee t_{i-1} < t_i),$$

[1] A schedule is *fair* if a process which have a privilege is executed within a finite steps.

$$R_B(i) \quad \equiv \quad (l_i = l_{i-1} + 1) \wedge (t_{i-1} \neq t_i) \wedge (l_i \neq 0)$$

A set of guarded commands δ_0 of S_0 is defined as follows.

Rule BP-A:
IF $R_A(i)$ THEN $l_i.t_i := (l_{i-1} + 1).(l_i - l_{i-1})$.

Rule BP-B:
IF $R_B(i)$ THEN $l_i.t_i := l_i.t_{i-1}$.

Arithmetic operations on labels and tags are computed modulo $n - 1$.

Legitimate configurations are configurations taking the following forms.

$$..., l - 2.0, l - 1.0, l.0, \underline{l.0}, l + 1.0, l + 2.0, ...$$

where l is any label and underlined state is a state of which a process that has a privilege.

After execution of the privileged process, the configuration become the following configuration.

$$..., l - 2.0, l - 1.0, l.0, l + 1.0, \underline{l + 1.0}, l + 2.0, ...$$

Note that the privilege is moved to the right process.

The next lemma holds for S_0 [BP89].

Lemma 1 *The no deadlock property holds, i.e., there exists a process P which has a privilege by Rule BP-A or Rule BP-B at any configuration.* □

Now, we define several terms used in this algorithm. (These terms are also used in the rest of this section.) Let $l_1.t_1, l_2.t_2$ be states of two consecutive processes P_1, P_2 in clockwise oder on a ring respectively. We say that P_2 has a *gap* if and only if $l_2 \neq l_1 + 1 (\bmod\ n - 1)$ is true and its *gap size* is defined by $l_2 - l_1$. A *segment* is a maximal sequence of processes $s = (P_i, P_{i+1}, ..., P_j)$ which does not include a process having a gap and P_i (P_j) is called the *head* (*tail*) process of s. For a segment $s = (P_i, P_{i+1}, ..., P_j)$, we say that the segment is *well formed* if and only if $t_x = l_{j+1} - l_j (\bmod\ n - 1)$ holds for every x $(i \leq x \leq j)$.

We describe the way of stabilization of the BP algorithm briefly. At a legitimate configuration, the number of segments is 1 and the segment is well formed. For any initial configuration, the number of segments is at most n at the configuration. The application of Rule BP-A and Rule BP-B does not increase the number of segments. Rule BP-A works as a movement of a privilege and decreasing the number of segments if there are more than one. Rule BP-B works to make a segment well formed. Even if a c-daemon try to keep the number of segments, every segments become well formed and there is at least one process which cannot make a move. Thus, the number of segments decreases within a finite steps until it become one.

7.1.2 Unidirectional Uniform Rings

Now, we show a type-1 self-stabilizing k-mutual exclusion algorithm on unidirectional ring $S_1 = (n, \delta_1, Q_1, \Lambda_1)$. We assume that $n \geq 5$ is prime.

Algorithm *SSUUR(k)*

Let a state set be $Q_1 = \{l.t\},\ l \in \{0, 1, ..., n-2\},\ t \in \{0\} \cup \{2, 3, ..., n-2\}$. Each field l, t are called *label* and *tag* respectively. A set of guarded commands is as follows. Note that a set of guarded commands is given to each P_i but it is identical for all processes.

First, we define the following predicates.

$$\begin{aligned} R_A(i) &\equiv (l_i \neq l_{i-1}+1) \wedge (l_i \neq 0 \vee t_{i-1} = 0 \vee t_{i-1} \neq l_i - l_{i-1} \vee t_{i-1} < t_i), \\ R_B(i) &\equiv (l_i = l_{i-1}+1) \wedge (t_{i-1} \neq t_i) \wedge (l_i \neq 0) \end{aligned}$$

```
Rule Uni-A:
    IF $R_A(i)$ THEN
        $l_i.t_i := (l_{i-1}+1).(l_i - l_{i-1})$.
Rule Uni-B:
    IF $R_B(i)$ THEN
        $l_i.t_i := l_i.t_{i-1}$.
Rule Uni-C:
    IF $(n-k \leq l_i \leq n-2) \wedge \neg(R_A(i) \vee R_B(i))$ THEN
        do nothing.
```

The arithmetic operation for labels and tags are computed modulo $n-1$.

A set of legitimate configurations is a set of following configurations.

$$..., l-2.0, l-1.0, l.0, l.0, l+1.0, l+2.0, ...,$$

for any label l. □

Now, the correctness of this algorithm is presented.

Lemma 2 *At any legitimate configuration, there are exactly k processes have privileges, i.e., k-mutual exclusion property holds. In addition, closure property also holds.*

(Proof) Let $\lambda \in \Lambda_1$ be any legitimate configuration, which can be expressed as

$$..., l-2.0, l-1.0, l.0, l.0, l+1.0, l+2.0, ...$$

for some l. Let P_0 be a process which has a privilege at λ by Rule Uni-A and $l_0.t_0$ be the state of P_0. (Note that $t_0 = 0$.)

- If $l_0 \in \{n-k, ..., n-2\}$:
  For each $l \in \{n-k, ..., l_0-1, l_0+1, ..., n-2\}$, There exists exactly one P_i such that $l_i = l$ and it has a privilege by Rule Uni-C. There exists two processes such that $l_i = l_0$ holds. One of them has a privilege by Rule Uni-C and another has a privilege by Rule Uni-A. Thus, the number of processes having a privilege by Rule Uni-C is $k-1$, and none of these is P_0. Thus, exactly k processes have a privilege.

  Since a configuration never change by executions of Rule Uni-C, we consider only executions of Rule Uni-A by P_0. Let γ' be the next configuration. γ' takes a form of

  $$..., l-2.0, l-1.0, l.0, l+1.0, l+1.0, l+2.0, ...$$

  which is a legitimate configuration. Therefore, closure property holds.

- If $l_0 \notin \{n-k, ..., n-2\}$:
  For each $l \in \{n-k, ..., n-2\}$, there exists only one P_i such that $l_i = l$. Thus, the number of processes which have a privilege by Rule Uni-C is $k-1$ and none of them is P_0. Thus, exactly k processes have a privilege. The closure property can be shown by the same proof given above. □

Lemma 3 *The fairness property holds.*

(Proof) By the definition of S_1, there exists exactly one process P which has a privilege by Rule Uni-A at any configuration $\lambda \in \Lambda_1$. By the assumption of the fairness execution of processes by a c-daemon, P is executed within a finite steps. Then, a process which has a privilege by Rule Uni-A moves to the right process. □

Lemma 4 *The no deadlock property holds.*

(Proof) Since each guard of Rule Uni-A and Rule Uni-B is the same as that of Rule BP-A and Rule BP-B by Burns and Pachl[BP89]. Thus, by the same proof for no deadlock property shown in [BP89], the no deadlock property of proposed algorithm is shown. □

Now, we have the following theorem.

Theorem 8 *S_1 is a type-1 self-stabilizing k-mutual exclusion system.*

(Proof) Since Rule Uni-C never change the configuration, we do not consider executions of Rule Uni-C without loss of generality. (Note that we assume a fair c-daemon.) There is no configuration such that every privilege is a privilege by Rule Uni-C and there exists at least one process which have a privilege by Rule Uni-A by the discussion in Lemma 4. Thus, for any configuration γ_0, the configuration reaches γ_1 such that

$$..., l-2.0, l-1.0, l.0, l.0, l+1.0, l+2.0, ...$$

by the same proof given in [BP89]. This is a legitimate configuration. □

Next theorem claims that there exists no algorithm for the type-2 k-mutual exclusion problem on unidirectional rings.

Theorem 9 *For each $n \geq 6$ and $k,\ (3 \leq k \leq n-3)$, there exists no type-2 self-stabilizing k-mutual exclusion algorithm on unidirectional ring of size n.*

(Proof) Assume that there exists a type-2 self-stabilizing k-mutual exclusion algorithm on a unidirectional ring. Let Λ be a set of legitimate configurations and $\lambda_0 \in \Lambda$. Then, there exists a legitimate configuration $\lambda_1 \in \Lambda$ and a computation $\lambda_0 \to^* \lambda_1$ such that consecutive k processes have a privilege at λ_1.

Without loss of generality, processes $P_0, P_1, ..., P_{k-1}$ have a privilege at λ_1. It is easy to see that locations of privileges never change by executions of $P_i\ (0 \leq i \leq k-2)$. (Otherwise, the number of privileges become less than k.) Thus, the movement of privilege happens only when P_{k-1} loses a privilege after several executions of P_{k-1}. Then, P_k has a privilege next. By the same way, P_k loses a privilege and then P_{k+1} has a privilege. This is repeated until P_{n-1} has a privilege. Note that any executions of $P_i\ (0 \leq i \leq k-2)$ do not cause a movement of privileges, since the ring is unidirectional.

Now consider the following two cases.

- If $k \leq \lfloor n/2 \rfloor$:
  A configuration such that any two privileges among k privileges are not adjoining each other is not reachable.
- If $k > \lfloor n/2 \rfloor$:
  A configuration such that any processes which do not have a privilege are not adjoining each other is not reachable. □

Corollary 2 *There is no self-stabilizing k-mutual exclusion algorithm under a c-daemon for $k \geq 2$ but there exists algorithm under a fair c-daemon.*

(Proof) Assume that there exists an algorithm under a (non-fair) c-daemon. Let $P_0, P_1, ..., P_{n-1}$ be a processes in clockwise on a ring. Consider a legitimate configuration at which process P_0 has a privilege. Execute P_0 until it loses a privilege. It does lose a privilege because the fairness property is not satisfied (consider a schedule executing only P_0). A privilege moves to its right process P_1. Do the same thing for P_1. Then the privilege moves to the right. Repeat this procedure until a privilege does not move any more. (A privilege do not move at some process P_i within a finite steps; otherwise privileges collide and the number of privileges decrease.) Then, execute only P_i. Other process do not have a chance enjoying privilege; the fairness property is not satisfied. □

Note that for $n \geq 5$ is prime, S_1 is a self-stabilizing system for the type-2 problem if $k = 2, n-2, n-1$,

7.1.3 Bidirectional Uniform Rings

We propose a type-2 self-stabilizing k-mutual exclusion algorithm S_2 on bidirectional rings for size $n \geq 5$ is prime. The proposed algorithm is based on the following idea. Consider a bidirectional ring consisting k *tracks* (rings). Each process execute the 1-mutual exclusion algorithm for unidirectional rings proposed by Burns and Pachl[BP89] in parallel for each track. A process has a privilege when it has a privilege at least one track in the sense of Burns and Pachl's algorithm. If each track are executed infinitely often, each track stabilizes and the number of privileges become one for each track. Then, the number of privilege become at most k in the ring. To satisfy a k-mutual exclusion property, the number of privileges must be exactly k, which implies that no process has a privilege at most one track.

The definition of $S_2 = (n, \delta_2, Q_2, \Lambda_2)$ is shown below.

Algorithm *SSBUR(k)*

Let the state set be $Q_2 = \{(l_j^1.t_j^1, l_j^2.t_j^2, ..., l_j^k.t_j^k) \mid l_j^i \in \{0, 1, ..., n-2\}, t_j^i \in \{0\} \cup \{2, 3, ..., n-1\}\}$, and Γ_2 be a set of all configuration.

A set of guarded commands δ_2 is defined as follows. Let a configuration be $(q_0, q_1, ..., q_{n-1})$, $q_j = (l_j^1.t_j^1, l_j^2.t_j^2, ..., l_j^k.t_j^k)$. To make the description simple, we define the following functions and predicates:

$$\begin{aligned}
R_A(i,j) &\equiv (l_j^i \neq l_{j-1}^i + 1) \wedge (l_j^i \neq 0 \vee t_{j-1}^i = 0 \vee t_{j-1}^i \neq l_j^i - l_{j-1}^i \vee t_{j-1}^i < t_j^i),\\
R_B(i,j) &\equiv (l_j^i = l_{j-1}^i + 1) \wedge (t_{j-1}^i \neq t_j^i) \wedge (l_j^i \neq 0),\\
S_p(j) &\equiv \{i \mid (1 \leq i \leq k) \wedge (l_j^i = l_{j-1}^i)\},\\
\pi_j &\equiv |S_p(j)|, \text{ and}\\
R_S(j) &\equiv \prod_{1 \leq i \leq k} \{((l_j^i = l_{j-1}^i + 1) \wedge (l_{j+1}^i = l_j^i)) \vee ((l_j^i = l_{j-1}^i) \wedge (l_{j+1}^i = l_j^i + 1))\\
&\qquad \vee ((l_j^i = l_{j-1}^i + 1) \wedge (l_{j+1}^i = l_j^i + 1))\}\\
&\quad \wedge \forall i, j' (1 \leq i \leq k, j-1 \leq j' \leq j+1)[t_{j'}^i = 0]\\
&\quad \wedge \pi_j \geq 1.
\end{aligned}$$

A transition rules δ_2^j for a process P_j is as follows. (Though process identifiers $j-1, j, j+1$ appears, $\delta_2^j = \delta_2^{j'}$ for any j, j'.)

```
Rule Bi-A:
    IF $\neg R_S(j) \wedge \exists i(1 \leq i \leq k)[R_A(i,j)]$ THEN
      For each $i'$ such that $R_A(i',j)$ is true:
        $l_j^{i'}.t_j^{i'} := (l_{j-1}^{i'} + 1).(t_j^{i'} - t_{j-1}^{i'})$,
        For each $i'$ such that $R_B(i', j)$ is true:
          $l_j^{i'}.t_j^{i'} := l_j^{i'}.t_{j-1}^{i'}$.

Rule Bi-B:
    IF $R_S(j) \wedge (\pi_j = 1) \wedge (\pi_{j+1} \geq 1)$ THEN
        do nothing.

Rule Bi-C:
    IF $R_S(j) \wedge (\pi_j = 1) \wedge (\pi_{j+1} = 0)$ THEN
        For $i' = \min S_p(j)$:
          $l_j^{i'}.t_j^{i'} := (l_{j-1}^{i'} + 1).(t_j^{i'} - t_{j-1}^{i'})$.

Rule Bi-D:
    IF $R_S(j) \wedge (\pi_j \geq 2) \wedge (\pi_{j+1} \geq \pi_j - 1)$ THEN
        do nothing.

Rule Bi-E:
    IF $R_S(j) \wedge (\pi_j \geq 2) \wedge (\pi_{j+1} < \pi_j - 1)$ THEN
        For $i' = \min S_p(j)$:
          $l_j^{i'}.t_j^{i'} := (l_{j-1}^{i'} + 1).(t_j^{i'} - t_{j-1}^{i'})$.

Rule Bi-F:
    IF $\exists i (1 \leq i \leq k)[R_B(i, j)]$ THEN
        For each $i'$ such that $R_B(i', j)$ is true:
          $l_j^{i'}.t_j^{i'} := l_j^{i'}.t_{j-1}^{i'}$.
```

A legitimate configuration $\lambda \in \Lambda_2$ is as follows: (1) Each track is in a legitimate configuration in the sense of BP, (2) each process P_j has a privilege of BP at most one track. A set of legitimate configurations Λ_2 is the set of the following configurations. Let $\gamma = (q_0, q_1, ..., q_{n-1})$ be a configuration such that $q_j = (l_j^1.t_j^1, l_j^2.t_j^2, ..., l_j^k.t_j^k)$. Then, $\gamma \in \Lambda_2$ if and only if the next condition holds.

- $\forall i, j[t_j^i = 0]$,
- For each i, $(l_0^i, l_1^i, ..., l_{n-1}^i)$ is a cyclic shift of $(l^i, l^i, l^i + 1, l^i + 2, ..., l^i + n - 3, l^i + n - 2)$ for some l^i. (Arithmetic operation is computed modulo $n - 1$.)
- For each j, $|\{i \mid l_j^i = l_{j-1}^i\}| \leq 1$. □

Now, the correctness proof of S_2 is presented below.

Lemma 5 *The number of processes which have a privilege is exactly k at any legitimate configuration.*

(Proof) It is clear by the definition of the set of legitimate configurations. □

Lemma 6 *The closure property holds.*

(Proof) Let $\lambda \in \Lambda_2$ be any legitimate configuration. The guards of Rule Bi-B or Rule Bi-C are true at processes which have a privilege at λ . (If the right process has a privilege then it has a privilege by Rule Bi-B. Otherwise it has a privilege by Rule Bi-C.) Assume that a process P_j which has a privilege.

- The case that P_j had a privilege by Rule Bi-B:
  The next configuration is the same as λ.

- The case that P_j had a privilege by Rule Bi-C:
  After application of Rule Bi-C, P_j does not have a privilege and P_{j+1} has a privilege at a track in the sense of BP. This configuration is in Λ_2. □

Lemma 7 *The fairness property holds.*

(Proof) Let $\lambda \in \Lambda_2$ be any legitimate configuration. At a configuration λ, there exists processes $P_a, P_b, ...$ such that their right processes do not have a privilege since $k < n$. These processes $P_a, P_b, ...$ have a privilege by Rule Bi-C. (Other processes which have a privilege is by Rule Bi-C.)

The application of Rule Bi-B does not change the configuration. By the fairness assumption of a c-daemon, a process P among $P_a, P_b, ...$ is executed within a finite steps. After execution of process P, it loses a privilege and the right process of P has a privilege instead. Thus, the movement of privilege within a finite steps is guaranteed by the fairness of a c-daemon. Therefore, every process has a privilege infinitely often in a infinite computation. Note that fairness property does not hold without fairness of a c-daemon. □

Lemma 8 *The no deadlock property holds.*

(Proof) Assume that a configuration $\gamma \in \Gamma_2$ is a deadlock configuration, i.e., guards of rules are false at every process. Since the number of process is prime, the same proof of no deadlock property of Burns and Pachl's algorithm (Lemma 4.3 in [BP89]) can be applied to this lemma. Thus, at least one of guards of Rule BP-A or that of Rule BP-B is true at some process. If the guard part of Rule BP-B is true at some tracks, a privilege by Rule Bi-F exists; a contradiction. Therefore, $\forall i \exists j [R_A(i, j)]$ is true. Let i_0, j_0 be integers such that $R_A(i_0, j_0)$ is true.

- When $R_S(j_0)$ is true:
  Since $\vee_{r \in \{\text{B, C, D, E}\}}(\text{Guard of Rule Bi-}r) = R_S(j_0)$, one of guards of Rule Bi-B, Bi-C, Bi-D and Bi-E is true; a contradiction.

- When $\neg R_S(j_0)$ is true:
  The guards of Rule Bi-A is true; a contradiction. □

Next lemma is shown in [LS92].

Lemma 9 *Let m_i be the number of gaps of the i-th track and g_i ($i = 0, 1, ..., m-1$) be gaps of the i-th track. Then, $\Sigma_{j=0}^{m-1} g_j = m - 1 (\bmod\, n-1)$.*

Lemma 10 *There is no configurations such that all privileges are privileges by Rule Bi-B and/or Rule Bi-D.*

(Proof) Assume that there is a configuration γ at which all privileges are privileges by Rule Bi-B and/or Bi-D. Since algorithm BP is livelock free, there exists j such that the guards of Rule BP-A ($= R_A(i,j)$) or that of Rule BP-B ($= R_B(i,j)$) is true for each track i. If we assume that the guard of BP-B is true for some i, j, then the guard of Bi-F become true and it is a contradiction. Thus, the guard of BP-B is not true at each track, i.e., $\neg R_B(i,j)$ is true for all i, j.

- The case that the guard of Bi-B is true at P_j:
  Since $R_S(j)$ is true, there exists i such that $l^i_{j+1} = l^i_j$ is true and $t^i_{j+1} = t^i_j = 0$ is true. Thus we have $R_A(i, j+1)$ is true. If we assume that $\neg R_S(j+1)$ is true then the guard of Rule Bi-A is true; a contradiction. Thus, $R_S(j+1)$ is true, which implies that one of guards of Rule Bi-B, Bi-C, Bi-D and Bi-E is true. (Note that logical-OR of guards of Rule Bi-B, Bi-C, Bi-D and Bi-E is $R_S(j)$.) Therefore, P_{j+1} has a privilege by Rule Bi-B or Rule Bi-D by assumption.

- The case that the guard of Rule Bi-D is true at P_j:
  There exists i such that $l^i_{j+1} = l^i_j$ is true since $\pi_j \geq 1$, and $R_A(i, j+1)$ is true since $t^i_{j+1} = t^i_j = 0$. By the same reason discussed above, P_{j+1} has a privilege by Rule Bi-B or Rule Bi-D.

By above discussion, every process has a privilege by Rule Bi-B or Rule Bi-D. Thus, $\forall i, j [R_S(j) \wedge \neg R_B(i,j)]$ is true. Since every gap size is 0 for each track, the sum of all gap sizes is 0 for each track. By the definition of $R_S(j)$, each track has at most $n-1$ segments. By this fact and by lemma 9, the number of segments at each track is 1. Thus, we have $\Sigma_j \pi_j \leq k$. On the other hand, if $R_S(j)$ is true then $\pi_j \geq 1$ is true, which implies $\Sigma_j \pi_j \geq n$; a contradiction. □

The next lemma shows that if each track is legitimate then the entire ring will reach a legitimate configuration within a finite steps.

Lemma 11 *For each i ($1 \leq i \leq k$), assume that the i-th track is in a legitimate configuration in the sense of BP at a configuration γ_0. Then, for any computation Δ starting from γ_0 such that $\Delta = \gamma_0 \rightarrow \gamma_1 \rightarrow \cdots$, there exists a finite integer τ such that $\gamma_\tau \in \Lambda_2$.*

(Proof) The behavior of S_2 when each track is legitimate in the sense of BP is the same as the behavior of the following (self-stabilizing) system S_3. The system S_3 consists of a bidirectional ring of size n and its algorithm is described below. Each process P_j takes the following state: π_j ($0 \leq \pi_j \leq k$, $\Sigma_{j=0}^{n-1} \pi_j = k < n$).[2] The algorithm of S_3 works to make π_j be at most one at any configuration.

[2]Although the network of S_3 is a bidirectional ring, the next state of a process is determined by its state and the state of its right process. In a strict sense, definition of S_3 does not match the definition of the self-stabilizing system defined in Chapter 6. The (another) definition of self-stabilization for S_4 is omitted because it can be defined similarly.

The legitimate configurations are configurations such that $0 \leq \pi_j \leq 1$ for each j. We say that P_i has a *token* if and only if $\pi_i \geq 1$ and π_i is called *the number of tokens* of P_i. The algorithm (transition relation) of S_3 is as follows.

```
Rule Bi-B':
    IF $(\pi_j = 1) \wedge (\pi_{j+1} \geq 1)$ THEN
      do nothing.

Rule Bi-C':
    IF $(\pi_j = 1) \wedge (\pi_{j+1} = 0)$ THEN
      $\pi_j := 0,\ \pi_{j+1} := 1.$

Rule Bi-D':
    IF $(\pi_j \geq 2) \wedge (\pi_{j+1} \geq \pi_j - 1)$ THEN
      do nothing.

Rule Bi-E':
    IF $(\pi_j \geq 2) \wedge (\pi_{j+1} < \pi_j - 1)$ THEN
      $\pi_j := \pi_j - 1,\ \pi_{j+1} := \pi_{j+1} + 1.$
```

It is easy to see that $\Sigma_{j=0}^{n-1}\pi_j = k$ always holds by the definition of the algorithm. Now, we show that self-stabilizing properties of S_3.

No deadlock property:
At a initial configuration, $\Sigma_{j=0}^{n-1}\pi_j = k,\ 2 \leq k < n$ holds and Bi-r is true at process P_j for some j and some $r \in \{$B',C',D',E'$\}$ since $\vee_{r\in\{\text{B',C',D',E'}\}}$(The guard of Rule Bi-$r$) $= (\pi_j \geq 1)$. Next, we show that a configuration such that all privileges are privileges by Rule Bi-B' and/or Rule Bi-D' does not exist. Assume the contrary. Let P_j be a process which has a privilege by Rule Bi-B' or Rule Bi-D'. Then, we have $\pi_{j+1} \geq 1$. Thus, P_{j+1} also has a privilege. By assumption, the privilege of P_{j+1} is also a privilege by Rule Bi-B' or Rule Bi-D'. By repeating this argument, we conclude that every process has a privilege by Rule Bi-B' or Rule Bi-D', which contradicts the fact that $\Sigma_{j=0}^{n-1}\pi_j = k < n$.

Closure Property:
Every privilege at a legitimate configuration λ is a privilege by Rule Bi-B' and/or Rule Bi-C'. It is easy to see that a configuration after λ is also a legitimate configuration.

No livelock property:
Assume that a livelock happens. By the proof of no deadlock property, we can conclude that at least one process have a privilege by Rule Bi-C' or Rule Bi-E'. For a configuration $\gamma = (q_0, ..., q_{n-1})$, we defined $M(\gamma) = \max\{\pi_j \mid 0 \leq j < n\}$. For any γ' such that $\gamma \rightarrow^* \gamma'$, it is clear that $M(\gamma') \leq M(\gamma)$. By assumption, there exists a configuration γ' and a computation Δ starting from γ' such that $M_0 = M(\gamma') = M(\gamma'') \geq 2$ for all $\gamma' \rightarrow^* \gamma''$. More over, there exists γ'' such that $\gamma' \rightarrow^* \gamma''$ in the computation Δ, the number of processes P_j such that $\pi_j = M_0$ is the same at all configurations after

γ''. Let J be a set of j such that $\pi_j = M_0$ and $\pi_{j+1} < M_0$ at γ''. Since $k < n$, we have $\pi_i \neq M_0$ for some i $(0 \leq i < n)$. In addition, consecutive processes at which $\pi_i = M_0$ holds except P_j $(j \in J)$ do not have a privilege by Rule Bi-C' nor Rule Bi-E'.

- The case that P_j has a privilege by Rule Bi-D' for each $j \in J$ at any configuration after γ'':
  If we assume that some process P_i $(0 \leq i < n)$ applied Rule Bi-C' or Rule Bi-E' after γ'' then $\pi_{i+1} < \pi_i$ was true before the execution of the rule. That is, a token is moved if the right process has less tokens. Since a set of processes such that $\pi_i = M_0$ never change, the number of applications of Rule Bi-C' and Bi-E' is finite and no process will have a privilege by Rule Bi-C' nor Rule Bi-E' within a finite steps; a contradiction.

- Otherwise, i.e., there exists a configuration after γ'' and $j \in J$, P_j has a privilege by Rule Bi-E' at the configuration:
  In this case, we have $\pi_{j+1} < \pi_j - 1 = M_0 - 1$. Unless P_j apply a rule, π_{j+1} does not increase. Thus, once P_j has a privilege by Rule Bi-E', the privilege is not lost by the execution of P_{j+1}. Therefore, P_j applied Rule Bi-E' within a finite steps by the assumption of fairness of a c-daemon. After application of Rule Bi-E' by P_j, the number of tokens of P_j become $\pi_j - 1 = M_0 - 1$ and that of P_{j+1} become $\pi_{j+1} + 1 < M_0$. Since the number of tokens of other processes is the same, the number of processes which have M_0 tokens decreases by the execution of P_j; a contradiction.

Therefore, we conclude that livelock never occurs and the S_3 system reaches a legitimate configuration within a finite steps. □

Lemma 12 *No livelock property holds for S_2.*

(Proof) By lemma 10, a livelock such that the same configuration is repeated does not occur. Thus there is at least one process which have a privilege by Rule Bi-A, Bi-C, Bi-E or Bi-F. By the assumption of fairness of a c-daemon, one of such process is executed with in a finite steps and the configuration changes. Thus, we do not take the executions of Rule Bi-B and Rule Bi-D into consideration.

Assume that there exists a configuration $\gamma \in \Gamma_2$ and a computation Δ which is a livelock computation. If every track reaches a legitimate configuration in the sense of BP, the entire ring also reaches a legitimate configuration by lemma 11. Thus, we assume that there exists a configuration γ' $(\gamma \rightarrow^* \gamma')$ and the I-th track such that the I-th track is not a legitimate configuration in the sense of BP at every configuration γ'' $(\gamma' \rightarrow^* \gamma'')$ in the computation Δ and the I-th track never change after γ'. (Otherwise, the track will be legitimate.) By the definition of S_2, an application of Rule Bi-A or Rule Bi-C or Rule Bi-E implies an application of Rule BP-A (and BP-B depending on the condition) for some track, and an application of Rule Bi-F implies an application of Rule BP-B for some tracks.

Since at least one of Rule Bi-A, Bi-C or Bi-E is applied infinitely often in Δ, there is a track which is infinitely often changes its configuration; thus there exists a track which become legitimate in the sense of BP within a finite steps. Let I_0 be such a track with the smallest suffix. At the I_0-th track, a privilege is moved from left to right. If a process P has a privilege by Rule BP-B at the I-th track then Rule BP-B is applied when P executes Rule BP-A at the I_0-th track; a contradiction. Thus, there exists no privilege by Rule BP-B at the I-th track and exists only privileges by Rule BP-A.

Let J be a set of indices of processes which have a privilege at the I-th track in the sense of BP. Then, $j \in J$, a process P_j has a privilege by Rule Bi-E when it has a privilege by Rule BP-A at the I_0-th track. Note that if the privilege of P_j is a privilege by Rule Bi-A or Rule Bi-C then Rule BP-A is applied at the I-th track. This contradicts the assumption. (Since the I_0-th track is in a legitimate configuration in the sense of BP and a privilege is circulating by Rule BP-A, P_j has a privilege except Rule Bi-B or Rule Bi-D when P_j has a privilege by Rule BP-A at the I_0-th track.) In addition, $I_0 < I$ holds by the definition of Rule Bi-E. Since $R_S(j)$ is true at P_j, the number of segments at the I-th track is at most $n-1$. Because $t^I_{j-1} = t^I_j = t^I_{j+1} = 0$ is true for each $j \in J$ and there is no privilege at the I-th track, if we assume that the number of segments at the I-th track is 1 then all tags at the I-th track are 0, which implies that the track is well formed in the sense of BP and it is a contradiction. Thus the number of segments of the I-th track s is $2 \leq s \leq n-1$ and the gap size at P_j is 0 for each process P_j which has a privilege by Rule BP-A at the I-th track.

By lemma 9 and the fact $2 \leq s \leq n-1$, there exists a gap whose size is not 0. In other words, if $g_0, g_1, ..., g_{s-1}$ is a sequence of consecutive gap sizes in clockwise order then there exists h such that $g_h \neq 0$. If P_H has a gap g_h then P_H does not have a privilege by Rule BP-A. The reason of this fact is described as below. Consider a configuration at which a privilege by Rule BP-A at the I_0-th track. If P_H has a privilege by Rule BP-A at the I-th track then $R_S(H)$ become false since the gap size is not 0 and Rule Bi-A is applied which implies that the configuration of the I-th track is modified. This contradicts the assumption.

Thus, $\neg R_A(I, H) = (l^I_H = 0) \wedge (t^I_{H-1} \neq 0) \wedge (l^I_{H-1} \neq 0) \wedge (t^I_{H-1} \geq t^I_H)$ is true at P_H. By this fact, a label of the leftmost process in a segment whose gap size is not 0 is 0.

Let a segment S be a segment whose gap size is not 0 and its left segment S_{-1} has a gap size 0. (It is clear that such segment exists by the above discussion.) By the assumption, the head process P_0 of S has label 0 and its left process P_{-1} (i.e., the tail process of S_{-1}) has non-zero label and non-zero tag. Thus, S_{-1} contains a process whose label is 0 and the length of S_{-1} is more than one since the guard of Rule BP-B is false at every process. The sequence of labels of the I-th track (starting from the label of P_H) is as follows.

$$l_{1,1}, l_{1,2}, l_{1,3}, ..., l_{2,1}, l_{2,2}, l_{2,3}, ..., l_{3,1}, l_{3,2}, l_{3,3}, ...$$

where $l_{i,1} = 0$ and $l_{i,j} \leq l_{i,j+1}$ holds for each i, j. That is, the sequence of labels is sequences of non-decreasing sequences starting from 0.

Assume that the head process of P_L of the segment S_{-1} has label 0. Then, the guard of Rule BP-A is true since the gap size of S_{-1} is 0. Thus, $L \in J$ and the tag of P_L is 0. Each process does not has a privilege by Rule BP-B and the number of processes which have a label 0 in each segment is at most one since the number of segments is more than one. Thus, each process in the segment S_{-1} has a tag 0; a contradiction. Therefore, the label of P_L is 0. But there is no such sequence of non-decreasing sequences since the number of segments is more than one; a contradiction.

By above discussion, we conclude that each track reaches a legitimate configuration within a finite steps. This fact and lemma 11, this lemma holds. □

We have the following theorem:

Theorem 10 *S_2 is a type-2 self-stabilizing k-mutual exclusion system for $n \geq 5$ is prime.* □

7.2 A Self-Stabilizing 1-Mutual Exclusion Algorithm with Randomization

In this section, we investigate the 1-mutual exclusion problem as a special case of the k-mutual exclusion problem. We consider the 1-mutual exclusion problem on unidirectional rings assuming state communication under a c-daemon and a c-dragon and propose uniform 1-mutual exclusion algorithms.

Since the ring is unidirectional, the solution is not trivial. If the ring is bidirectional, random walk of tokens can be used to stabilization of the number of tokens as described in [IJ90]. However, a unidirectional ring under a c-daemon cannot use random walk method because the movement of token is one direction (i.e., the choice for a processes is moves the token right or not) and a c-daemon may choose a schedule of processes not to collide tokens.

It is shown that there is no self-stabilizing 1-mutual exclusion algorithm if the number of process is composite[BP89]. We propose a uniform randomized self-stabilizing 1-mutual exclusion algorithm for any size of ring. The proposed algorithm can escape the malicious schedule of a c-daemon and it self-stabilizes with high probability without deadlock.

Before proposing an algorithm of randomized version, we propose a self-stabilizing deterministic mutual exclusion algorithm under a c-dragon. Since the scheduler guarantees the probabilistically fair execution of process, the algorithm is much simpler.

7.2.1 The self-stabilizing system under a c-dragon

In this subsection we investigate self-stabilizing mutual exclusion systems on unidirectional ring under a c-dragon.

Theorem 11 *For each $n \geq 1$, there exists a deterministic self-stabilizing system under a c-dragon.*

(Proof) The case $n = 1$ is trivial. Burns and Pachl proposed a deterministic self-stabilizing mutual exclusion system under a c-daemon for $n = 2$ in [BP89]; their system also works correctly under a c-dragon. Thus, we consider the case $n \geq 3$.

The mutual exclusion system we propose is as follows. Let the state set $Q = \{0, 1, ..., n-2\}$. Let $P_0, ..., P_{n-1}$ be processes in the system (in clockwise order) and q_i be the state of process P_i. Note that we show a set of rules for each process P_i, but every process have the same algorithm. The algorithm of S_n is as follows:

Rule: IF $q_{i-1}+1 \neq q_i$ THEN $q_i := q_{i-1}+1 (\mathrm{mod} n-1)$

A legitimate configuration λ is a configuration such that only one process has a privilege at λ. For example,

$$0,1,1,2,3,4,5,6,7,8$$

is a legitimate configuration when $n = 10$.

If a processes P has a privilege, we say that P has a token. It is easy to see that (1) There exists at least one token in the system at any configurations, and (2) The number of tokens never increase by the execution of any set of processes.

By execution of a privileged process, it loses a privilege and a privilege may move to the right process. Therefore, we can consider that a token moves to right. Consider a configuration at which the number of tokens is more than one. Let τ_i and τ_j be any two different tokens. The distance of the two tokens is defined by the minimum distance of processes on which tokens are. If two tokens are on consecutive processes, the distance is one. If two tokens collide, the number of tokens decreases by the definition of the algorithm. Because the scheduler (a c-dragon) chooses a privileged process to be executed, we can regard the movement of tokens as random walk of tokens on a unidirectional ring.

Now consider any two tokens τ_i, τ_j and fix them. Let $d(\tau_i, \tau_j, \gamma)$ be the distance of two tokens τ_i, τ_j at a configuration γ. We consider $d(\tau_i, \tau_j, \gamma)$ as a state of a Markov chain. Note that the state 0 is an absorbing wall and $\lceil n/2 \rceil$ is a reflecting wall. Since the probability of transition from state i to state $i-1$ and from state i to state $i+1$ are both $1/2$ for each $1 \leq i < \lfloor n/2 \rfloor$. Thus, it is easy to see that the expected steps that two tokens τ_i, τ_j collide is finite.

Since a c-dragon chooses a privileged process to be executed with uniform probability, the expected interval steps that one of process corresponding to tokens τ_i, τ_j is executed is finite. Therefore, the expected steps that every tokens collide is finite, which implies that the expected steps that the number of tokens become one is finite. This is a legitimate configuration. $\Box$

7.2.2 The randomized self-stabilizing system under a c-daemon

In this subsection, we a propose randomized self-stabilizing system under a c-daemon. Burns and Pachl [BP89] showed that the number of processes of a ring is *composite* then there exists no deterministic self-stabilizing system under a c-daemon. As we saw above, the self-stabilizing mutual exclusion system for *each* n is easily obtained by assuming a c-dragon. The next interest lies in a self-stabilization assuming a c-daemon: Which additional device is necessary for the existence of self-stabilizing mutual exclusion system for every n under a c-daemon? Our answer is that if each process has a random-bit generator then the expected steps that the number of privileges become one is finite under any schedule.

The outline of reason why there exists no deterministic algorithm is as follows[BP89]: Where $n \geq 4$ is composite, n can be decomposed as $n = xy$, where $x, y \geq 2$. We can construct a x blocks of processes of length y and by choosing i-th process in a block and execute i-th process of all blocks. By this schedule, the number of processes having privilege is at least x. Since the behavior of process is deterministic, a c-daemon can choose a schedule of execution of processes to keep a *symmetry* of configuration. The case explained above, configurations consists of a blocks of length b. To break symmetry of configurations by malicious scheduling of a c-daemon, randomization is added to process behavior.

7.2.3 The randomized self-stabilizing 1-mutual exclusion algorithm

We show a self-stabilizing mutual exclusion algorithm for a ring size n. It is shown that there exists a deterministic self-stabilizing algorithm for a ring of size 2 in [BP89]. The case for $n = 1$ is trivial. Therefore, it is enough to consider the case $n \geq 3$.

The idea of proposed algorithm is based on a algorithm by Burns and Pachl [BP89]. A state set of processes is a 3-tuple $l.t.r$. The first field of states is called *label*, the second is called *tag*, and the last is called *random signature*. To stabilize the ring, we add a toss-a-coin feature to each process to break a symmetry of ring (with high probability) in spite of a c-daemon. The random signature is a signature of a segment which is randomly generated. To break a symmetry of the ring, signatures of segments are compared.

Now we describe a formal definition of proposed algorithm. A state set of processes is $\{l.t.r \mid l \in \{0, 1, 2, ..., n-2\}, t \in \{0, 2, 3, ..., n-2\}, r \in \{0, 1\}\}$. We define following predicates:

$$\begin{aligned}
A_i &= (l_i \neq l_{i-1} + 1) \wedge (l_i \neq 0 \vee t_i = 0 \vee t_i \neq l_i - l_{i-1} \vee t_i \leq t_{i-1}) \\
B_i &= (l_i = l_{i-1} + 1) \wedge (t_i \neq t_{i-1} \vee r_i \neq r_{i-1}) \wedge (l_i \neq 0) \\
C_i &= \neg A_i \wedge (l_i \neq l_{i-1} + 1) \wedge (t_i = t_{i-1}) \wedge (r_i \leq r_{i-1}) \\
\alpha_i &= (l_{i-1} = n - 2)
\end{aligned}$$

The algorithm is described below. The procedure RandomBit() generates a random bit (i.e., 0 or 1) with the same probability $1/2$.

```
Rule A: IF $A_i \wedge \alpha_i$ THEN
            $l_i := l_{i-1} + 1$
            $t_i := l_i - l_{i-1}$
            $r_i :=$ RandomBit()

Rule A': IF $A_i \wedge \neg\alpha_i$ THEN
            $l_i := l_{i-1} + 1$
            $t_i := l_i - l_{i-1}$
            $r_i := r_{i-1}$

Rule B: IF $B_i$ THEN
            $t_i := t_{i-1}$
            $r_i := r_{i-1}$
```

```
Rule C: IF $C_i$ THEN
        $l_i := l_{i-1} + 1$
        $t_i := l_i - l_{i-1}$
        $r_i := r_{i-1}$
```

A legitimate configurations is a configuration such that

$$..., l-2.0.r_{-3}, l-1.0.r_{-2}, l.0.r_{-1}, l.0.r_0, l+1.0.r_1, l+2.0.r_2, ...$$

for some $l \in \{0, 1, ..., n-2\}$. In addition, each legitimate configuration must be the following form as to random signature r_i.

$$n-3.0.r, n-2.0.r, 0.0.r', 1.0.r', ..., l.0.r', l.0.r, l.0.r, ...$$

for $r, r' \in \{0, 1\}$. That is, processes between a process having label 0 and the left process of a privileged process have the same random signature. The other processes (i.e., processes between a privileged process and a process having label $n-1$) have the same random signature. For instance, a configuration

$$5.0.0, 6.0.0, 0.0.1, 1.0.1, 2.0.1, 2.0.0, 3.0.0, 4.0.0,$$

is a legitimate configuration when $n = 8$.

Correctness proof

Before showing the proof, we define several terms used in the following proof. Let $P_0, P_1, ..., P_{n-1}$ be a consecutive processes in a clockwise order on the ring and $l_i.t_i.r_i$ be a state of process P_i. A *segment* s is a sequence of consecutive processes $s = P_a, P_{a+1}, ..., P_b$ such that $l_i = l_{i-1} + 1$ for each $i = a+1, a+2, ..., b$ and $l_a \neq l_{a-1} + 1$ and $l_{b+1} \neq l_b + 1$. Let $\sharp(\gamma)$ be the number of segment at γ. We say that there is a *gap* between processes P_b and P_{b+1} if $l_{b+1} \neq l_b + 1$. The *gap size* of a segment $s = P_a, P_{a+1}, ..., P_b$ is $l_{b+1} - l_b$. A process P_a (P_b) is called a *head process* (a *tail process*) of s. A segment $s = P_a, P_{a+1}, ..., P_b$ is *well formed* if $(t_i = t_{i-1} \wedge r_i = r_{i-1}) \vee (l_i = 0)$ for each $i = a+1, a+2, ..., b$ and $t_b = l_{b+1} - l_b$.

Now, we give the correctness proof of proposed algorithm.

Lemma 13 *For any configuration $\gamma \in \Gamma$, the number of segments is at least one.*

(Proof) It is clear because the possible labels are $n-1$. □

Lemma 14 *The algorithm is deadlock free.*

(Proof) Assume that deadlock happens. Let γ be any deadlock configuration. Since logical OR of guard of all rules is $A_i \vee B_i \vee C_i$, a condition $\neg A_i \wedge \neg B_i \wedge \neg C_i$ holds for every process i at γ. For every head process of a segment, $l_i = 0 \wedge t_i \neq 0 \wedge t_i = l_i - l_{i-1} \wedge t_i > t_{i-1}$ holds because $\neg A_i$ and $l_i \neq l_{i-1} + 1$. Thus, for every segment s at γ, Head(s) has label 0.

The number of segments is at least 1 by lemma 13, we consider following two cases.

- When the number of segments is 1:
  The tail process has label 0 because the number of segment is one and the head process has label 0. Therefore, $t_i = 0 \vee t_i \neq l_i - l_{i-1}$ is true at the head process since $l_i - l_{i-1} = 0$.

  By definition of Rule A and Rule A', the head process has a privilege by Rule A or Rule A'; a contradiction.

- Otherwise:
  Because the number of segments is more than one and every head process has label 0, a process whose label is 0 is a head process. Since no process has a privilege by Rule B, every process in a segment has the same tag and random signature, which contradicts the fact that $t_i > t_{i-1}$ holds for at head process of every segments. □

Lemma 15 *Closure property holds.*

(Proof) Let λ be any legitimate configuration. By the definition of rules, it is clear that the head process of the only segment always have privilege by one of Rule A or Rule A'. It is easy to see that the next configuration of λ is also a legitimate configuration. □

Lemma 16 *Fairness property holds.*

(Proof) Let λ be any legitimate configuration. By the definition of legitimate configurations, the number of processes which have a privilege is one and by lemma 15, the privilege moves to a right process by a execution. Therefore, the privilege circulates the ring. □

Lemma 17 *Mutual exclusion is guaranteed.*

(Proof) It is clear by the definition of legitimate configurations. □

Above lemmas proves four property of self-stabilizing systems. We prove that the expected steps the system stabilize is finite.

Lemma 18 *Let $\gamma_0 \in \Gamma$ be any configuration and $\Delta = \gamma_0, \gamma_1, \gamma_2, ...$ be any infinite computation starting from γ_0. Then, there exists $0 \leq I < \infty$ such that a transition $\gamma_I \rightarrow \gamma_{I+1}$ is an application of Rule A or Rule A' or Rule C.*

(Proof) Assume that there exists $\gamma_0 \in \Gamma$ and an infinite computation $\Delta = \gamma_0, \gamma_1, \gamma_2, ...$ such that a transition $\gamma_j \xrightarrow{\gamma}_{j+1}$ is an application of Rule B for each $j \geq 0$. Application of Rule B never change members of segments and changes only a tag. By definition of Rule B, a tag does not propagate over a gap (and label 0). Thus, for any segment s, the number of applications of Rule B for s is finite if no other rules are applied and there exists a configuration γ_J such that $\gamma_0 \rightarrow^* \gamma_J$ and there is no privilege by Rule B at γ_J.

Because the algorithm is deadlock free (by lemma 14), there exists a process that has a privilege and the privileges are privileges by one of Rule A or Rule A' or Rule C. Therefore, one of these rules are applied. □

Lemma 19 *The configuration reaches a legitimate configuration within a finite steps if the number of segments at an initial configuration γ is one.*

(Proof) Let $\gamma \in \Gamma$ be any configuration of which the number of segments is one. It is easy to see that the number of segments is non-increasing by the definition of algorithm. Thus, for any $\gamma' \in \Gamma$ such that $\gamma \rightarrow^* \gamma'$, the number of segments at γ' is one. By lemma 18, the head process executes one of Rule A, Rule A', or Rule C. By execution of any of these rules, the head process changes and the label of the new head process increases by one. Therefore, within a finite steps from γ, the configuration become a configuration such that the label of a head process of the segment is 0. Let this configuration be γ_0 and let processes be $P_0, P_1, ..., P_{n-1}$ in clockwise order in the ring and P_0 is the head process at γ_0.

By lemma 18, P_0 executes one of Rule A, A' or C and its tag and random signature become the same as P_{n-1}'s. Note that the tag is zero. Let the configuration after P_0 executed a rule be γ_1. Similarly, P_1 executed a rule and its tag and signature become as the same as P_0. Repeating this argument, it is easy to see that the configuration become the legitimate configuration. □

Lemma 20 *For any configuration $\gamma \in \Gamma$ such that the number of segment is n at γ, the number of segments become $n-1$ by an execution of a rule.*

(Proof) There is no privilege by Rule B since the length of every segments is 1. By this fact and by lemma 14, every privilege is a privilege by Rule A or Rule A' or Rule C. The execution of any of these rules makes a segment of length 2 and the number of segments become $n-1$. □

Lemma 21 *Let γ_0 be any segment such that $\sharp(\gamma_0) > 1$ and s be any segment at γ_0. Then, there exists no computation starting γ_0 such that the number of application of Rule B by processes in s is infinite if the processes consisting of s never change.*

(Proof) Let s consists of processes $P_1, P_2, ..., P_m$ in clock wise order of the ring. Assume that there exists a computation such that the number of application of Rule B is infinite. Recall that the processes consisting of s never change during the computation and no process in s applies Rule A, A', nor C by assumption.

Let γ_1 be a configuration just after a process in s applied Rule B after γ_0. Similarly, every time a process in s applies Rule B, define a configuration γ_i. Then we have a sequence of configuration $\gamma_0, \gamma_1, \gamma_2, ...$ For each γ_i, we associate a integer v_i which is represented by m-bit vector whose j-th bit is 1 if and only if P_j has a privilege by Rule B. The most significant bit (1st bit) of v_i corresponds to P_1 and the least significant bit (m-th bit) of v_i corresponds to P_m. Thus, 1st bit of v_i is always 0 because P_1 is a head process.

Then, it is easy to see that the number sequence $v_0, v_1, v_2, ...$ is a decreasing sequence. Because $v_i \geq 0$ for all i, there exists no such number sequence. This is a contradiction. $\Box$

Lemma 22 *Let γ_0 be any segment such that $\sharp(\gamma_0) > 1$. Assume that there exists a computation starting from γ_0 such that the number of segments never change. Then, there exists no segment whose head process never changes.*

(Proof) Assume that there exists such segment s. By lemma 21, there exists no computation such that only Rule B is applied. Thus, Rule A, A' or C is applied within a finite steps, which implies that there exists a segment s' whose member changes infinitely many times during an infinite computation. Let P be a head process of s. Then, P also become a member of s' infinitely often because a segment moves to only one direction on a ring; this is a contradiction. $\Box$

Next lemma shows that any schedule which *try* to keep the number of segments leads to a configuration in which all segments are well formed.

Lemma 23 *Let $\gamma_0 \in \Gamma$ be any configuration such that the number of segments of γ_0, $\sharp(\gamma_0)$, is $2 \leq \sharp(\gamma_0) \leq n-1$. Assume that there exists an infinite computation $\Delta = \gamma_0, \gamma_1, \gamma_2, ...$ such that $\sharp(\gamma_0) = \sharp(\gamma_j)$ for all $j \geq 0$. Then, there exists I such that every segment at γ_i is well formed for all $i \geq I$.*

(Proof) Let $L = \sharp(\gamma_0)$ $(= \sharp(\gamma_1) = \sharp(\gamma_2) = ...)$ and $s_1, s_2, ..., s_L$ be a sequence of segments in clockwise order of the ring. Note that processes consisting segments change with the computation proceeds, but the number of segments is kept by the assumption.

Let l be the label of the tail process of s_1 at γ_0. Then, by lemma 22, the head process of s_2 executes a rule and become the tail process of s_1 within a finite steps and its label is $l+1$. Repeating this discussion, it is easy to see that the configuration reaches a configuration such that the label of the tail process of s_1 become 0 within a finite steps. Let this configuration be γ_1 and $P_1 = \text{Tail}(s_1)$.

Consider a configuration μ_2 such that P_1 become the head process of s_1 for the first time after γ_1. (It is easy to see that such configuration exists by 22.) By the definition of rules, P_1 never change its tag and random signature between γ_1 and γ_2. Thus, the right processes of P_1 in the same signature inherit P_1's tag and random signature. Therefore, the segment s_1 is well formed at γ_2.

The random signature of a segment is generated again when the new tail process takes label 0, but the segment is still well formed. By repeating the same argument, $s_2, s_3, ..., s_L$ become well formed within a finite steps. Therefore, every segment become well formed within a finite steps. $\Box$

The range of labels and definition of gap is the same as the ones in [BP89]. Lin and Simon showed the next lemma in [LS92] for the algorithm in [BP89]. Thus, the next lemma also holds for our algorithm.

Lemma 24 *Let $\gamma \in \Gamma$ be any configuration and $s_1, ..., s_L$ be segments at γ and g_i be a gap of s_i, where $L = \sharp(\gamma)$. Then, $\sum_{1 \leq i \leq L} g_i = L - 1 \bmod n - 1$*

(Proof) Proof can be found in [LS92]. □

Lemma 25 *Let γ_0 be any configuration such that $2 \leq \sharp(\gamma_0) \leq n-1$ and every segment is well formed at γ_0 and $\Delta = \gamma_0, \gamma_1, ...$ be any infinite computation. Then, there exists a head process of a segment, say P, and γ_k such that P does not have a privilege by Rule A nor Rule A' at γ_k,*

(Proof) Assume that every head process has a privilege by Rule A or Rule A' at γ_j for all j. This implies that A_i is true at all head process at γ_j. Although The label of a head process changes with the computation proceeds, the relative relation of labels of tail process and head process which are consecutive processes on a ring is kept (e.g. $t_i \neq l_i - l_{i-1}$). In addition, the tag of each segment never change during the computation. Therefore, a condition $t_i = 0 \vee t_i \neq l_i - l_{i-1} \vee t_i < t_{i-1}$ is always true for all head processes of segment s_i. Otherwise, A_i become false when $l_i = 0$.

Since all segments are well formed, $t_i = l_i - l_{i-1}$ holds. Therefore, the above condition becomes $t_i = 0 \vee t_i < t_{i-1}$. Because $t_i < t_{i-1}$ does not hold for all head processes, there exists j such that $t_j \geq t_{j-1}$. Since above condition is true, $t_j = 0$ holds. Now consider the right segment $s_{j'}$, where $j' = j+1$. To $t_{j'} = 0 \vee t_{j'} < t_{j'-1}$ be true, $t_{j'} = 0$ holds since $t_{j'-1} = t_j = 0$ and tags are non-negative. Repeating this discussion, we have $t_i = 0$ for all i, which contradicts the lemma 24. □

Now, we show that the number of segments decreases with high probability and the expected steps that the ring converges to a legitimate configuration is finite.

Lemma 26 *Let γ_0 be any initial configuration such that $2 \leq \sharp(\gamma_0) \leq n-1$. Then, the expected steps that the number of segments decreases is finite.*

(Proof) Assume that $\Delta = \gamma_0, \gamma_1, \gamma_2, ...$ be any infinite computation such that the number of segments never decrease. By lemma 23, there exists I such that all segments are well formed at γ_i for any $i \geq I$. We consider configurations after γ_I.

Since every segments are well formed and the number of segments are kept, there is no process which has a privilege by Rule B at any configuration γ_i ($i \geq I$). By lemma 25, there exists a process P and a configuration γ_J ($J \geq I$) such that P is a head process of a segment s_j and does not has a privilege by Rule A nor Rule A'. Since every head process has a privilege by assumption, $P =$ Head(s_j) has a privilege by Rule C.

At the computation after γ_J, every process never change its tag, and relative relation of labels at head processes never change (e.g., $l_i \neq l_{i-1}$). Thus, $t_i = 0 \vee t_i \neq l_i - l_{i-1} \vee t_i < t_{i-1}$ (see A_i) is always false at Head(s_j). Otherwise, it does not have a privilege by Rule C at γ_J. Therefore, at configurations γ_k ($k \geq J$), Head(s_j) has a privilege by Rule A or Rule A' when its label is not 0 and it has a privilege by Rule C otherwise. When Head(s_j) has a privilege by Rule C, the label of the left process of Head(s_j) is not $n-2$ nor 0 because, if otherwise, it has a privilege by Rule A or Rule A' (see A_i).

Let s_{j-1} be the left segment of s_j. Then, Head(s_j) has a privilege by Rule A when the label of Tail(s_{j-1}) is $n-2$. By an application of Rule A by Head(s_j), new random signature of s_{j-1} is generated. Therefore, every time the value of label circulates, s_{j-1} has new random signature.

Now consider the right segment s_{j+1} of s_j. If Head(s_{j+1}) has a chance to have a privilege by Rule C at configurations after γ_J, we can conclude that s_j generates new random signature every time the value of label circulates by a similar discussion described above. Otherwise (i.e., s_{j+1} never have a privilege by Rule C), s_j also generates new random signature every time the label of Tail(s_j) is $n-2$ by Rule A.

Let $\tau_0, \tau_1, ...$ be a sequence of indexes of configurations such that s_j and s_{j-1} changed random signatures at least once at some configurations between $\gamma_{\tau_{i-1}}$ and γ_{τ_i}. Then, it is easy to see that there exists a constant T determined by the algorithm such that $\tau_{l-1} - \tau_l < T < \infty$ for all l.

Since random signature is randomly chosen from $\{0, 1\}$, the probability $r_i \leq r_{i-1}$ holds at Head(s_j) is $3/4$. The expected steps that $r_i \leq r_{i-1}$ become false is at most $4T/3$. If $r_i \leq r_{i-1}$ become false, the head process of s cannot make a step by Rule C and it is clear that a daemon cannot choose a schedule that keeps the number of segment. Thus, the number of segment decrease. □

We have the theorem from above lemmas.

Theorem 12 *For each $n \geq 1$, there exists a randomized self-stabilizing mutual exclusion system for a ring of size n under a c-daemon.* □

Note that the algorithm does not work under d-daemon. (Consider a configuration such that a state of every process is $0.0.0$ and a schedule such that all processes are executes at every step. Then, the number of segments never decrease.)

Reduction of the number of states

The proposed algorithm above requires $2(n-1)(n-2) = \Theta(n^2)$ states. By the similar technique proposed in [BP89], we can reduce the number of states of above algorithm.

The number of possible tag value is reduced in the following algorithm, it ranges over $\{0, 1\}$. First, we define following predicates:

$$\begin{aligned}
A_i &= (l_i \neq l_{i-1} + 1) \wedge (l_i \neq 0 \vee t_i = 0 \vee t_i \neq f(l_i - l_{i-1}) \vee t_i \leq t_{i-1}) \\
B_i &= (l_i = l_{i-1} + 1) \wedge (t_i \neq t_{i-1} \vee r_i \neq r_{i-1}) \wedge (l_i \neq 0) \\
C_i &= \neg A_i \wedge (l_i \neq l_{i-1} + 1) \wedge (t_i = t_{i-1}) \wedge (r_i \leq r_{i-1}) \\
\alpha_i &= (l_{i-1} = n - 2)
\end{aligned}$$

The labels range over $\{0, 1, ..., n-2\}$, the random signatures range over $\{0, 1\}$. The function f is a function from $\{0, 2, 3, ..., n-2\}$ and defined as follows.

$$f(k) = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{otherwise} \end{cases}$$

Note that $f(k) = 0$ iff $k = 0$.

The algorithm is the following. The difference is that new tag is given by $f(l_i - l_{i-1})$. The set of legitimate configurations is the same as the set defined in the previous version.

```
Rule RA: IF $A_i \wedge \alpha_i$ THEN
    $l_i := l_{i-1} + 1$
    $t_i := f(l_i - l_{i-1})$
    $r_i :=$ RandomBit()

Rule RA': IF $A_i \wedge \neg\alpha_i$ THEN
    $l_i := l_{i-1} + 1$
    $t_i := f(l_i - l_{i-1})$
    $r_i := r_{i-1}$

Rule RB: IF $B_i$ THEN
    $t_i := t_{i-1}$
    $r_i := r_{i-1}$

Rule RC: IF $C_i$ THEN
    $l_i := l_{i-1} + 1$
    $t_i := f(l_i - l_{i-1})$
    $r_i := r_{i-1}$
```

By modifying the algorithm, the we need a new definition of *well formed*. A segment s_i is well formed iff every process of s has a tag $f(g_i)$, where g_i is the gap size of s_i. The condition for random signature is the same as the original definition.

Lemma 27 *The algorithm satisfies the (1) closure property, (2) fairness property, and (3) mutual exclusion.*

(Proof) Because the behavior of the ring is the same as the original algorithm, the same proof for closure property holds. Thus, the fairness property and mutual exclusion property also hold. □

Lemma 28 *The algorithm satisfies no deadlock property.*

(Proof) The proof is the identical to the proof of lemma 14 except $t_i = l_i - l_{i-1}$ is replaced by $t_i = f(l_i - l_{i-1})$ and $t_i \neq l_i - l_{i-1}$ is replaced by $t_i \neq f(l_i - l_{i-1})$. □

Lemma 29 *The algorithm satisfies no livelock property.*

(Proof) Lemmas 18, 19, 20, 23 hold by the same proof. Lemma 25 is shown by replacing $t_i = l_i - l_{i-1}$ by $t_i = f(l_i - l_{i-1})$ and $t_i \neq l_i - l_{i-1}$ by $t_i \neq f(l_i - l_{i-1})$ in the proof. Note that $t_i = 0$ does not hold at all head processes because $t_i = 0$ implies $l_i = l_{i-1}$ and contradicts the 24. Lemma 26 is also shown by replacing $t_i = l_i - l_{i-1}$ by $t_i = f(l_i - l_{i-1})$ and $t_i \neq l_i - l_{i-1}$ by $t_i \neq f(l_i - l_{i-1})$ in the proof. □

Now we have the following theorem.

Theorem 13 *For each $n \geq 1$, there exists a randomized self-stabilizing mutual exclusion system which requires $4(n-1)$ states per process for a ring of size n under a c-daemon.* □

7.3 Concluding Remarks

In this chapter, we proposed several self-stabilizing mutual exclusion algorithms.

In the first section, we proposed a deterministic self-stabilizing k-mutual exclusion algorithm under a c-daemon on unidirectional and bidirectional ring networks whose size is prime and showed their correctness. It is easy to show that there is no self-stabilizing algorithm whose size n is composite and n does not have a factor k. (The proof can be shown by the similar method used in Theorem 2.1 in [BP89].) In the case that n has factor k, whether there exists an algorithm or not is an open problem.

In the next section, we investigated self-stabilizing mutual exclusion systems under assumptions of a c-daemon, a c-dragon and randomization. We showed that the number of states per process require is $\Theta(n)$ if we assume dragons and randomized behavior of processes under a c-daemon. The known deterministic algorithm for 1-mutual exclusion systems requires $\Theta(n^2/\ln n)$ but our algorithm assuming randomized behavior for each process requires only $O(4(n-1))$.

Carl-Johan Seger proved that any uniform deterministic self-stabilizing 1-mutual exclusion system under a c-daemon for a ring whose size is n requires at least $n-1$ states [Bur94]. He showed that there exists a schedule of processes which cause a livelock if we assume the existence of a self-stabilizing system and the number of states of processes is less than $n-1$. This lower bound by Seger is a bound guaranteeing no livelock. On the other hand, Israeli and Jalfon [IJ90] showed that $\Omega(\log n)$ states is necessary for uniform self-stabilizing system on unidirectional ring. Their lower bound guarantees no deadlock property. Thus, there is a gap for lower bounds of the number of states between achieving no livelock and no deadlock. There is a gap between lower bounds and proposed algorithms. The following problems are left as future tasks.

- Does there exist deterministic self-stabilizing mutual exclusion systems under a c-daemon which requires a state set whose size is less than $\Theta(n^2/\ln n)$?
- Does there exist deterministic self-stabilizing mutual exclusion systems under a c-dragon which requires a state set whose size is less than $n-1$?

Chapter 8

Conclusion

In this dissertation, we have investigated the distributed k-mutual exclusion problem by two approaches: the coterie approach and the self-stabilization approach. We proposed several algorithms for distributed k-mutual exclusion.

In Part I, we have studied the coterie approach.

In Chapter 2, we have proposed a concept called k-coterie as an extension of coterie. To allow k processes be in their critical sections, k-coterie has distinct k quorums but does not have $k+1$ quorums. Processes can enter their critical section without interfering with other processes; on the other hand, processes interfere if we use another definition of k-coterie proposed in [BC94, MA93].

In Chapter 3, the analysis of availability of coterie has been shown. We have shown a sufficient condition and a necessary condition such tat a k-majority coterie is optimal under an assumption that a topology of communication links are complete network and every process fails with the same probability.

In Chapter 4, we have proposed a distributed k-mutual exclusion algorithm using a k-coterie. We have shown that the message complexity of the proposed algorithm is $3|Q|$, where $|Q|$ is the size of a quorum.

In Chapter 5, we have shown the goodness of our distributed k-mutual exclusion algorithm by comparing with Raymond's algorithm by computer simulation. The simulation was done using workstations which are connected to a local area network. Since each process was executed on different workstations, algorithms are simulated in real-time; which can be considered as being close to a real distributed system. The simulation result shows that the proposed algorithm in Chapter 4 is much better than Raymond's algorithm if k is large and mutual exclusion request is not frequent.

In Part II, we have studied the self-stabilization approach. In Chapter 6, we have explained computational models and given a survey for the research area of self-stabilizing mutual exclusion. We also discussed a motivation of self-stabilizing approach. Self-stabilizing systems can tolerate any kind of transient failures. Thus, Self-stabilizing systems are fault-tolerant systems.

In Chapter 7, we have investigated self-stabilzing k-mutual exclusion on ring networks. First, we have investigated two types of self-stabilizing k-mutual exclusion problems on unidirectional and bidirectional rings. We have shown that there exists no type-2 problem (i.e., a configuration of any arrangement of privileges can be reachable from any configuration) does not exist on unidirectional rings. We have proposed a type-1 self-stabilizing k-mutual exclusion algorithm on unidirectional rings and type-2 self-stabilizing k-mutual exclusion algorithm on bidirectional rings under c-daemon. These algorithm require that the number of process of a ring is prime. In the case that the number of process has a factor $f \neq k$, there is a schedule of processes which does not reach a legitimate configuration. Next, we investigated self-stabilizing 1-mutual exclusion problem as a special case of k-mutual exclusion. We have proposed a randomized self-stabilizing 1-mutual exclusion algorithm for any size of unidirectional ring under c-daemon. In the algorithm, randomization is used because there is no algorithm if the number of process is composite.

The coterie approach is an approach that reduces the number of messages for distributed mutual exclusion and increases the availability. The algorithm proposed in Chapter 4, does not consider any failures such as message omissions, process failures, etc. The design of an algorithm which tolerates such failures is the next step of the work.

In [Bal94b, Bal94a], Baldoni proposed k-coterie, which is completely different from ours. The basic idea of his distributed k-mutual exclusion algorithm is that each process has k permissions (or, tokens); a process wishing to enter a critical section must collect tokens from each process in a quorum. To make this scheme work, the requirements for a quorum set $\mathcal{C}$ is as follows:

- Intersection Property: For any $q_1, ..., q_{k+1} \in \mathcal{C}$, $\cap_{i=1}^{k+1} q_i \neq \emptyset$.
- Minimality Property: For any $q_i, q_j \in \mathcal{C}$ such that $q_i \neq q_j$, $q_i \not\subseteq q_j$.

By our definition of k-coterie, each process has one token. On the other hand, each process has k tokens by Baldoni's k-coterie. We believe that there is a unified scheme for these ideas. For example, there may be a condition for a quorum set to achive 4-mutual exclusion when each process has 2 tokens. The investigatin of unified k-coterie scheme is left as a future task.

The self-stabilization approach is a strong approach for transient failures. The design of coterie based mutual exclusion algorithms which tolerates transient failures by using a concept of self-stabilization is a interesting theme. However, designing a self-stabilizing algorithm and proving the correctness are difficult tasks. The automatic construction of self-stabilizing algorithm is an important for realizing self-stabilizing systems.

In this dissertation, we have treated the coterie approach and the self-stabilizing approach separately. A unification of these two approaches is an another task.

Appendix A

Local Coteries and a Distributed Resource Allocation Algorithm

The distributed k-mutual exclusion problem treats a situation such that every process in a distributed system share all resources uniformly. But it is natural to consider that a set of resources available to a process is different by processes. This may happen by limitations of access rights or some geometrical reasons.

Consider a situation such that a process P_1 has access rights to resources r_1, r_2, r_3 and a process P_2 has access rights to resources r_3, r_4 and each process issues a resource request when it requires resources. In such case, the k-mutual exclusion is not suitable to arbitrate the conflicts of resource requests. In addition, if two processes P_3, P_4 do not share any resources then it is desirable that resource allocation is done without interference. The mutual exclusion and the k-mutual exclusion problems are special cases of the resource allocation problem. Such problem is proposed and investigated as "the drinking philosophers problem" by Chandy and Misra [CM84]. In their paper, they showed a token-based resource allocation algorithm for a special case in such a way that each resource is shared by only two processes. The objective of this appendix is to solve the problem under the frame work of coterie and its variants. Generalized resource sharing model is also appear in the paper by Miyamoto [Miy94], in which an allocation problem of *anonymous* resources which are shared by any number of processes is investigated. He used coterie approach to solve the problem.

In this appendix, we consider a problem of allocating *named* resources; a process requests any amount resources and any of free resources are allocated but a process must know the names of allocated resources to use them. We propose a new concept of *local coterie* and a resource allocation algorithm using a local coterie.

A.1 The Resource Model

A distributed system consists of n processes $U = \{P_1, P_2, ..., P_n\}$, bidirectional communication links each connecting two processes, and m resources $R = \{r_1, r_2, \ldots, r_m\}$ shared by processes. Processes

$P \in U$ are allowed to use some of the resources $r \in R$. We denote this relation by function $\alpha : U \to 2^R$. For any $u \in U$,

$$\alpha(P) = \{r \in R \mid P \text{ has an access right to } r\} \in 2^R.$$

When V is a set of processes, with abuse of notation, $\alpha(V)$ denotes $\cup_{P\in V}\alpha(P)$. The triple (U, R, α) is called the *share structure* of the system.

We define a configuration c of the distributed system as a tuple of the states of all processes and communication links. Then a computation π of the system can be described by a (possibly infinite) sequence of configurations starting from the initial configuration. Note that the computation is not determined uniquely in general, even if the initial configuration (including input) is given because of the asynchrony of system.

When the system is at configuration c, processes P may be accessing some resources r. For any $P \in U$, $\rho_P(c)$ denotes the set of resources r which are being accessed by P when the system is at configuration c.

A.2 The Resource Allocation Problem

Consider a distributed system in which each process repeats the local computation and the resource access phases forever. The former phase does not include resource access instructions, and the latter is a series of resource access instructions which starts with a resource request instruction for requesting some resources and ends up with a resource release instruction for releasing all resources it is accessing. Let $S = (U, R, \alpha)$ be its share structure. Each time the resource access phase is executed, the number of resources a process P requests can change between 1 and $|\alpha(P)|$.

The *resource allocation problem* is the problem of implementing the resource requests and release instructions in such a way that whenever a process P requests k $(\leq |\alpha(P)|)$ resources, eventually k resources are allocated to P. Furthermore, as the restriction arising from the share structure, any computation $\pi = c_0, c_1, ..., c_i, ...$ of the resulting distributed system must satisfy the following two conditions:

Allocation Validity: For any configuration c_i and any set $V \subseteq U$ of processes,

$$\bigcup_{P\in V} \rho_P(c_i) \subseteq \alpha(V).$$

Mutual Exclusion: For any configuration c_i and any two different processes $P, P' \in U$,

$$\rho_P(c_i) \cap \rho_{P'}(c_i) = \emptyset.$$

Allocation Validity guarantees that a process P only accesses resources to which it has an access right, and Mutual Exclusion guarantees that every resource is allocated to at most one process at a time.

A.3 Local Coteries

In general, the resource allocation problem treats cases in which resources are shared by different sets of processes unlike the mutual exclusion problem. Consider a case in which two processes P and P' do not share any resource. Then it is a natural requirement that their requests be interference free. (It may or may not be possible, depending on the remaining part of a share structure.) As long as the same quorum set is associated to P and P', the interference inevitably occurs.

In order to take into account the share structure, *For each process P, we associate (possibly different) quorum sets $Q_P \subseteq 2^U$* reflecting the share structure. We call the set $\{Q_P \mid P \in U\}$ a *local coterie* with respect to a share structure (U, R, α). The formal definition of the local coterie is as follows.

Definition 8 *A non-empty set $\{Q_P \mid P \in U\}$ is a* local coterie *with respect to a share structure (U, R, α) if and only if the following conditions are satisfied.*

- **Non-emptiness:** $\forall P \in U[Q_P \neq \emptyset]$.
- **Intersection Property:** $\forall P, P' \in U[\alpha(P) \cap \alpha(P') \neq \emptyset \Rightarrow \forall q \in Q_P, \forall q' \in Q_{P'}[q \cap q' \neq \emptyset]]$.
- **Minimality:** $\forall P \in U, \forall q, q' \in Q_P[q \not\subseteq q']$. □

Note that the definition of local coterie includes that of coterie as a special case when $|R| = 1$ and $\alpha(P) = R$ for all $P \in U$.

First, we show a simple construction algorithm for a local coterie with respect to a share structure (U, R, α).

```
Algorithm LocalCoterie($U, R, \alpha$);
begin
    $q_P$ := $\{P\}$ for all $P \in U$;
    for all $r$ in $R$ do
        for each $P, P'$ in $U$ such that $P \neq P'$ do
            if $r \in \alpha(P) \cap \alpha(P')$ then
                $q_P$ := $q_P \cup \{P'\}$;
                $q_{P'}$ := $q_{P'} \cup \{P\}$
            fi
        od
    od;
    $Q_P$ := $\{q_P\}$ for all $P \in U$;
    return $\{Q_P \mid P \in U\}$
end.
```

Theorem 14 *The algorithm LocalCoterie(U, R, α) correctly computes a local coterie with respects to a share structure (U, R, α), in time $O(|R||U|^2)$.*

(Proof) The non-emptiness and the minimality trivially holds since a quorum set Q_P contains only one quorum for each process $P \in U$. Assume that the intersection property does not hold. Let P_1, P_2 be processes such that $P_1 \neq P_2$ and $(\alpha(P_1) \cap \alpha(P_2) \neq \emptyset) \ \wedge \ (q_{P_1} \cap q_{P_2} = \emptyset)$, where $q_{P_i} \in Q_{P_i}$ for $i = 1, 2$. Since $r \in \alpha(P_1) \cap \alpha(P_2)$ for some $r \in R$, $P_2 \in Q_{P_1}$ and $P_1 \in Q_{P_2}$ by the definition of the algorithm. This implies $\{P_1, P_2\} \subseteq q_{P_1} \cap q_{P_2}$ because $P_i \in Q_{P_i}$ for each $P_i \in U$; a contradiction.

It is easy to see the execution time of the algorithm is $O(|R||U|^2)$. □

Corollary 3 *For any share structure (U, R, α), there exists a local coterie $\mathcal{C}$ with respects to share structure (U, R, α).* □

A.4 A Distributed Resource Allocation Algorithm

Now, we are ready to introduce our algorithm. We first explain an outline of the algorithm, and then describe it in detail.

The processes altogether maintain a distributed database which keeps pairs of a process and a resource it is currently accessing. A process P_u wishing to access k resources sends a query asking whether or not there are k resources available to u.[1] If the answer is yes, then the k resources are allocated to P_u. Hence, the algorithm is assertion-based, as well as quorum-based.

Let $\{Q_u\}$ be a local coterie, where Q_u is the quorum set associated with process P_u. Then an outline of the algorithm is as follows.

In our algorithm, a process P_v is (partially) responsible for the resources which are accessible from a process P_w such that P_v appears as an element of a quorum q in Q_w. Let R_v be the set of resources for which P_v is responsible. For each resource $r \in R_v$, P_v remembers the process which currently accesses r (or the fact it is free, otherwise). A process P_u wishing to access k resources selects an arbitrary quorum $q \in Q_u$, and sends a query message $\langle\texttt{QUERY}\rangle$ to every process P_v in q. A process P_v receiving query $\langle\texttt{QUERY}\rangle$ sends back the names of resources available to P_u. Upon receiving the list of available resource names from every process $P_v \in q$, P_u selects arbitrarily k resource names which appear in every list and sends a lock message $\langle\texttt{LOCK}\rangle$ with the k names to every process P_v to let it update the current states of the k resources. When P_u releases the k resources, it sends an unlock message $\langle\texttt{UNLOCK}\rangle$ with the k names to every process P_v to let it change the states of the resources into free.

The above explanation is just an outline of the algorithm and it does not contain explanations how to avoid deadlocks and starvations and how to treat cases in which P_u cannot find k resources available to P_u. Moreover, in order for the algorithm work correctly, the query step must be carried out in the

[1]We say that a resource r is *available* to P_u if $r \in \alpha(P_u)$ and r is currently free. On the other hand, that P_u is *accessible* to r simply means $r \in \alpha(P_u)$.

mutually exclusive way. Nevertheless, we would like you to observe that if P_u decides to access a set of resources r, then r is currently available to P_u, i.e., P_u has access right to r and r is not used by some process, by definition of local coterie.

The algorithm assumes that each process P_u maintains the following local variables. For convenience of explanation, as in the above rough explanation, define

$$S_v = \{P_w \mid P_v \in q \text{ for some } q \in Q_w\}$$

and

$$R_v = \bigcup_{P_w \in S_v} \alpha(P_w).$$

- C_u – the current logical time at P_u. Initially, it is 0, and is automatically incremented.[2]

- D_u – the array to hold for each resource $r \in R_u$, if r is LOCKed or not. More precisely, for each $r \in R_u$, $D_u(r) = (P_u, t)$ if r is locked by $\langle \texttt{LOCK} \rangle$ message with timestamp t issued by P_u. Otherwise, if r was lastly released by $\langle \texttt{UNLOCK} \rangle$ message with timestamp t, then $D_u(r) = (\perp, t)$. Initially, $D_u = (\perp, 0)$ for all $r \in R_u$.

- W_u – it holds the name of process to which P_u sends the current states of resources held in D_u and is waiting for a reply. In other words, W_u is the process from which P_u is waiting for $\langle \texttt{LOCK} \rangle$ message, after sending $\langle \texttt{RESPONSE} \rangle$ message. If P_u is not in this situation, $W_u = \perp$.

- T_u – it holds the timestamp attached to the $\langle \texttt{QUERY} \rangle$ message that the process held in W_u issued. $T_u = \perp$ if $W_u = \perp$.

- X_u – the priority queue to hold $\langle \texttt{QUERY} \rangle$ messages waiting at P_u for their turns. They are sorted in the order of their timestamps.

We describe our algorithm *AllocResource* in an event driven form.

Algorithm *AllocResource*
Let $\{Q_u\}$ be the local coterie used in the algorithm.

1. **When a process P_u wishes to access k $(\leq |\alpha(u)|)$ resources:**

   Process P_u arbitrarily selects a quorum $q \in Q_u$, and sends $\langle \texttt{QUERY}, P_u, C_u \rangle$ to every process in q.[3] Recall that C_u is the current logical time at P_u and is attached to the message as the timestamp. Then it waits until both of the following two conditions hold:

[2]By using a standard technique that uses unique process identifiers, events occurred in the system are totally ordered by means of the logical time[Lam78].

[3]The number k of requesting resources is not a parameter of $\langle \texttt{QUERY} \rangle$.

- It has received $\langle \texttt{RESPONSE}, P_v, D_v \rangle$ messages at least once from each process $P_v \in q$. Note that P_v sends $\langle \texttt{RESPONSE}, P_v, D_v \rangle$ message carrying the latest version of D_v as soon as D_v is updated, even if it has sent an older version to P_u (see Case 7). Note also that P_u does not need to store old versions. It simply discards them and holds the latest one (see Case 3).
- Recall that every D_v contains the states of all resources in $\alpha(P_u)$ from the view of P_v. Let $A_u \subseteq \alpha(P_u)$ be the set of resources r satisfying $D_{v^*}(r) = (\perp, t^*)$, where t^* is the maximum value occurred in the second (i.e., time) field of $D_v(r)$ among all $P_v \in q$, and P_{v^*} is the process achieving t^*. Intuitively, A_u is the set of resources currently available to P_u, as we will show in the next section. The second condition is that A_u contains at least k resources.

If both of the above conditions hold, P_u then arbitrarily selects a set S_u of k resources from A_u, sends $\langle \texttt{LOCK}, P_u, C_u, S_u \rangle$ message to every process $P_v \in q$, and accesses S_u.

2. **When process P_u releases the set S_u of accessing resources:**

   Process P_u sends $\langle \texttt{UNLOCK}, P_u, C_u, S_u \rangle$ message to every process $P_v \in q$.

3. **When process P_u receives $\langle \texttt{RESPONSE}, P_v, D_v \rangle$ message from a process P_v:**

   Process P_u stores D_v. If it has received an older version of D_v, then it discards it and stores the latest one. Because the message order is assumed to be unchangeable in each communication link, P_u always holds the latest version among versions received so far.

4. **When a process P_v receives $\langle \texttt{QUERY}, P_u, t \rangle$ from process P_u:**

   If $W_v = \perp$, i.e., if process P_v does not wait for $\langle \texttt{LOCK} \rangle$ message from another process, it sends $\langle \texttt{RESPONSE}, P_v, D_v \rangle$ message to P_u, and sets $W_v := P_u$ and $T_v := t$. Recall that t is the logical time at P_u at which the $\langle \texttt{QUERY} \rangle$ message was issued (see Case 1).

   Otherwise, $W_v = P_w$ for some process $P_w \in U$, i.e., P_w waits for the two conditions in Case 1 to hold. If $T_v < t$, i.e., if P_w has higher priority (since T_v is the timestamp attached to P_w's $\langle \texttt{QUERY} \rangle$), P_v stores $\langle \texttt{QUERY}, P_u, t \rangle$ to queue X_v. Otherwise, if $T_v > t$, P_u has the priority. Then in order to preempt the right to lock resources which P_v gave to P_w, P_v sends $\langle \texttt{PREEMPT}, P_v \rangle$ to P_w, and waits for P_w replying either $\langle \texttt{RETURN} \rangle$ or $\langle \texttt{LOCK} \rangle$message (see Cases 1 and 8), after storing the $\langle \texttt{QUERY} \rangle$ messages issued by P_u and P_w to X_v. When P_v again needs to send $\langle \texttt{PREEMPT} \rangle$ to P_w while waiting for a reply from P_w, v ignores it.

5. **When process P_v receives $\langle \texttt{RETURN}, P_w \rangle$ message from process P_w:**

   Process P_v takes the $\langle \texttt{QUERY}, P_x, t \rangle$ message from the top of queue X_v. It is the $\langle \texttt{QUERY} \rangle$ message which has the highest priority. Then P_v sends $\langle \texttt{RESPONSE}, P_v, D_v \rangle$ to P_x, and sets $W_v := P_x$ and $T_v := t$.

6. **When process P_v receives $\langle \texttt{LOCK}, P_w, t, S_w \rangle$ message from process P_w:**

   Process P_v updates its data D_v; it sets $D_v(r) := (P_w, t)$, for each $r \in S_w$. Then it continues (the algorithm fragment for) Case 5 if X_v is not empty.

7. **When process P_v receives $\langle \texttt{UNLOCK}, P_w, t, S_w \rangle$ message from process P_w:**

   Process P_v updates its data D_v; it sets $D_v(r) := (\bot, t)$, for each $r \in S_w$. If $W_u \neq \bot$, then it sends $\langle \texttt{RESPONSE}, P_v, D_v \rangle$ message to W_v. Otherwise, it continues Case 5 if X_v is not empty.

8. **When process P_w receives $\langle \texttt{PREEMPT}, P_v \rangle$ message from process P_v:**

   If it has sent back $\langle \texttt{LOCK} \rangle$ message to P_v, then it simply ignores the $\langle \texttt{PREEMPT} \rangle$ message. Otherwise, process P_w sends back $\langle \texttt{RETURN}, P_w \rangle$ message, and then P_w discards D_v which was sent from P_v. That is, the response $\langle \texttt{RESPONSE}, P_v, D_v \rangle$ is canceled by the $\langle \texttt{PREEMPT} \rangle$ message. □

Although in the above description of *AllocResource*, $\langle \texttt{RESPONSE} \rangle$ carries all data D_v, it is clearly reducible. At a process, say P_u, only the data on the resources in $\alpha(P_u)$ in D_v will be used.

A.5 Correctness Proof

In this section, we show the correctness of our algorithm *AllocResource*, provided that processes accessing resources release them within a finite time.

Theorem 15 *Algorithm AllocResource guarantees Allocation Validity condition.*

(Proof) This theorem holds since each process P_u selects the resources from the candidate set A_u, which is a subset of $\alpha(P_u)$. □

In order to proceed the remaining properties, recall that a process P_u wishing for k resources arbitrarily selects k resources from A_u determined from D_v's for $P_v \in q \in Q_u$, sends $\langle \texttt{LOCK} \rangle$ message carrying the names of k resources to every P_v, and accesses them. On the other hand, process P_v updates D_v responding to the $\langle \texttt{LOCK} \rangle$ message. If two processes which share resources received D_v's simultaneously, they could select the same resources and access them simultaneously. Our algorithm guarantees that such situations never occur. We introduce the notion of *Q-region* to prove it formally.

A process P_u requesting k resources sends $\langle \texttt{QUERY} \rangle$ message to every member P_v of a quorum $q \in Q_u$, and collects D_v's until the two conditions of Case 1 hold. If a $\langle \texttt{PREEMPT} \rangle$ message from $P_w \in q$ arrives in the meanwhile, it discards D_w and waits for new D_w. Recall that receiving a D_v from every $P_v \in q$ is a necessary condition, but is not sufficient. We say that P_u is in the *Q-region* if P_u has received a D_v from every $P_v \in q$, but has neither sent $\langle \texttt{LOCK} \rangle$ message nor received $\langle \texttt{PREEMPT} \rangle$ message since then.

Lemma 30 *Let P_u and P_v be any two processes such that $\alpha(P_u) \cap \alpha(P_v) \neq \emptyset$. Then P_u and P_v are never in their Q-regions simultaneously.*

(Proof) Assume that there exists two processes P_u, P_v such that $\alpha(P_u) \cap \alpha(P_v) \neq \emptyset$ and P_u, P_v are in their Q-regions at a time. Let P_w be a process such that P_w is in both quorums P_u and P_v chose. Note that there exists such P_w since $\alpha(P_u) \cap \alpha(P_v) \neq \emptyset$. Without loss of generality, assume that P_w sent $\langle$RESPONSE$\rangle$ to P_u first. By the definition of algorithm, P_w extracts the request from P_v after sending $\langle$RESPONSE$\rangle$ to P_u. By assumption, P_w sent $\langle$RESPONSE$\rangle$ to P_v before $\langle$LOCK$\rangle$ or $\langle$RETURN$\rangle$ is sent from P_u. This action contradicts the definition of the algorithm. □

Suppose that a resource r has been allocated. If P_v didn't knew this fact, A_u could include r when a process P_v sent D_v for the first time to P_u, which implies that r may be allocated to more than one process since the candidate set A_u is determined from D_v's. The next lemma guarantees that such situations never occur.

Lemma 31 *Let P_u and P_v be any two processes such that $r \in \alpha(P_u) \cap \alpha(P_v) \neq \emptyset$. Assume that r has been allocated to P_u, and P_v is now in its Q-region. Further, assume that P_u used quorum $q_u \in Q_u$ for its resource request and P_v is using quorum $q_v \in Q_v$. By the definition of local coterie, $q_u \cap q_v \neq \emptyset$. Then for any $P_w \in q_u \cap q_v$, $D_w(r) = (P_u, t)$ for some t.*

(Proof) Since P_u is accessing a resource r, it had sent $\langle$LOCK$\rangle$ message to every process in q_u when it exits from Q-region and then it started accessing r. Every $P_w \in q_u \cap q_v$ sends a $\langle$RESPONSE$\rangle$ message to P_v after it receives a $\langle$LOCK$\rangle$ message from P_u. When P_w receives $\langle$LOCK$\rangle$ from P_u, it updates its local database such that r is allocated to P_u with its allocation time. When P_w sends $\langle$RESPONSE$\rangle$ message to P_v, P_w knows that r is already allocated. Thus, $D_w(r) = (P_u, t)$ for some t. □

Theorem 16 *Algorithm AllocResource guarantees Mutual Exclusion condition.*

(Proof) Assume that a resource $r \in \alpha(P_u) \cap \alpha(P_v)$ is allocated to both P_u and P_v simultaneously. The proof is by induction. Mutual Exclusion condition holds at the initial state of the system since no resources are allocated to processes. By lemma 30, any two processes sharing resources are not in their Q-regions simultaneously. Without loss of generality, we assume that P_u leaves its Q-region first by sending $\langle$LOCK$\rangle$ message to allocate r to P_u. Then, P_v can enter its Q-region only after all processes in $q_u \cap q_v$ receiving $\langle$LOCK$\rangle$ message from P_u, where $q_u \in Q_u$ ($q_v \in Q_v$) is the quorum that P_u (P_v) chooses for response request. Since P_u and P_v share resources, $q_u \cap q_v$ is not empty. Let P_w be any process in $q_u \cap q_v$. Then, P_w updates its database so that $D_w(r) = (P_u, t_u)$ holds for some t_u by receiving $\langle$LOCK$\rangle$ message from P_u. By lemma 31, every $\langle$RESPONSE$\rangle$ message sent to P_u from P_w contains data $D_w(r) = (P_u, t_u)$. Therefore P_v cannot choose r; a contradiction. □

Theorem 17 *Algorithm AllocResource is deadlock free.*

(Proof) Since processes request all resources necessary when the resource access phase starts, we do not consider deadlocks caused by nested requests. We consider the deadlocks at the query step.

Assume that a deadlock happens. Since the number of processes is finite, there exists a time such that the number of processes being deadlocked does not increase afterwards. We consider what will happen. Although there may exist processes which do not send and/or receive messages in general, without loss of generality, we can assume that there are no such processes.

Let $V \subseteq U$ be the set of processes being deadlock, and assume that $P_u \in V$ is the process whose timestamp attached to the $\langle \texttt{QUERY} \rangle$ message is the smallest (i.e., highest priority) among V. The $\langle \texttt{QUERY} \rangle$ message by P_u will arrive to every process in a quorum $q \in Q_v$ in a finite time. Since the logical clock monotonically increases, the timestamp of P_u's $\langle \texttt{QUERY} \rangle$ will become the highest among all processes. By the definition of the algorithm, each process P_v in q behaves as follows. If P_v sent $\langle \texttt{RESPONSE} \rangle$ message to a process $P_w \in U$ but it has not received the corresponding $\langle \texttt{LOCK} \rangle$ message, then P_v sends $\langle \texttt{PREEMPT} \rangle$ message to P_w to switch the query right to P_u. If P_v receives $\langle \texttt{RETURN} \rangle$ message from P_w, it will send $\langle \texttt{RESPONSE} \rangle$ message to P_u. Otherwise, it will send $\langle \texttt{RESPONSE} \rangle$ message to P_u, when P_w returns $\langle \texttt{LOCK} \rangle$, since P_u's $\langle \texttt{QUERY} \rangle$ has the highest priority. On the other hand, processes that share resources with P_u does cannot be in their Q-region, and hence, resources are not allocated to them. Therefore, within a finite time, enough number of resources in $\alpha(P_u)$ become free and the request by P_u will be satisfied within a finite time, a contradiction. □

Next theorem can be proved by a similar argument.

Theorem 18 *Algorithm AllocResource is starvation free.* □

Now, we can conclude that the algorithm *AllocResource* correctly solves the resource allocation problem.

Theorem 19 *Algorithm AllocResource solves the resource allocation problem.* □

A.6 Concluding Remarks

In this appendix, we have discussed the resource allocation problem, and proposed a distributed algorithm. Unlike other conflict resolution problems such as the mutual exclusion and the k-mutual exclusion problems, we consider cases in which processes may have access rights to different sets of resources. In order to take into account the resource share relation of the system, we have introduced a new concept called local coterie.

The number of messages necessary to exchange per resource request can be shown to be $4|q|$, where $q \in Q_u$ in the best case and $(7 + |\alpha(P_u)|)|q|$, where $q \in Q_u$ in the worst case. In cases such that each resource is shared by small number of processes, since the quorum size $|q|,\ q \in Q_u$ can be small, our algorithm is suitable. The algorithm by Baldoni [Bal94b] requires $O(n^{M/(M+1)})$ message per resource allocation, where n is the number of processes and M is the number of resources. If M is large, the message complexity of Baldoni's algorithm become approximately $O(n)$, which is less efficient than ours.

Finally, we would like to touch some future works. As a general advantage of quorum-based approach, our algorithm is robust against process and/or link failures; as far as at least one quorum "survives", there is a possibility that resource allocation can be achieved. However, discussing the fault-tolerance aspect of this algorithm in detail is left as a future work. The local coterie construction algorithm proposed in this appendix is simple. However, the local coteries produced are not always good ones. Constructing better local coteries is also left as a future task.

Appendix B

Implementations of Distributed k-Mutual Exclusion Algorithms

The examples of implementation of two distributed k-mutual exclusion algorithms is shown in this appendix. We show the implementation of our algorithm proposed in Chapter 4 and the algorithm proposed by Raymond [Ray89a].

Each program fragment of the implementation of distributed k-mutual exclusion algorithm shown below is a part of the source code which is used in the simulation in Chapter 5 and listed without any modifications.

The template of the implementation of algorithms is as follows:

```
Algorithm();
{
    $\boxed{\textit{Initialization of Variables, etc.}}$

    while (TRUE){
        SiteBehavior();        /* decides the behavior of the process */

        $\boxed{\textit{Do active behavior decided by SiteBehavior().}}$

        if (no messages arrived )
            continue;

        $\boxed{\textit{Receive a message}}$

        $\boxed{\textit{Do passive behavior dependent on the received message.}}$
    }
}
```

The procedure `SiteBehavior()` is a procedure to decide a behavior of a process. For instance, a process is in Normal state, it decides to request a mutual exclusion with specified probability. According to such decision, process do its active behavior. If a mutual exclusion request happen, the process sends request messages, for instance. After finishing active behavior, the process checks message arrival. If a message is arrived, it read the message and process the message according to its message type. This is the passive behavior.

B.1 Our Distributed k-Mutual Exclusion Algorithm using k-Coterie

```
void
KakugawaProcess(k, p, quantum, cycle, tcs)
  int     k;
  double  p;
  int     quantum;
  int     cycle;
  int     tcs;
{
  SiteID       Y,  Z;
  int          Sy, Sz, WaitingY, WaitingSy;
  SiteSet      Quorum, NextSites;
  bool         WaitingOkWait, NoMoreQuorum, WaitingTEmpty,
               WaitingAnswer, WaitingExit;
  Message      Msg;
  char         msgbody[80];
  static bool  LexicoLess();
  static bool  SelectAQuorum(), GetConsensusP();
  extern void  SiteBehavior();

  TransitNormalState();

  WaitingOkWait = false;
  NoMoreQuorum  = false;
  WaitingTEmpty = false;
  WaitingAnswer = false;
  WaitingExit   = false;

  SetSiteSetEmpty(Quorum);
  SetSiteSetEmpty(NextSites);

  for (;;){

    /***
     *** DECISION OF BEHAVIOR OF SITE
     ***/
    SiteBehavior(k, p, quantum, cycle, tcs);

    if (ExitMutexJob)
      return;

    if (EnterCSRequestHappen){
      /**
       ** MUTEX REQUEST HAPPEN
       **/
      TransitRequestingState();
      EnterCSRequestHappen = false;
      RequestingCS = true;
      MaxSeq = MaxSeq + 1;
      Seq = MaxSeq;
      SelectAQuorum(Quorum, Coterie, SetK, SetT);

    L1:
      SiteSetDifference(NextSites, Quorum, SetK);
      SendRequestToSet(NextSites);
      WaitingOkWait = true;
```

```
}

if (RequestingCS
    and (NoMoreQuorum or (WaitingOkWait and SiteSetEmptyP(NextSites)))){
  WaitingOkWait = false;
  if (GetConsensusP(SetK, Coterie)){
    /**
     ** ENTER THE CS
     **/
    TransitInCriticalSectionState();
    RequestingCS = false;
    ExecCS = true;
    NoMoreQuorum = false;
  } else if (not NoMoreQuorum){
    /**
     ** FAILED TO GET A QUORUM ... RETRY!
     **/
    if (SelectAQuorum(Quorum, Coterie, SetK, SetT)){
      goto L1;
    } else {
      NoMoreQuorum = true;
    }
  }
}

if (ExitCSRequestHappen){
  /**
   ** REQUEST OF EXITING THE CRITICAL SECTION HAPPEN
   **/
  ExitCSRequestHappen = false;
  ExecCS = false;
  TransitExitingCriticalSectionState();
  SendReleaseToSet(SetK);
  SetSiteSetEmpty(SetK);
  WaitingTEmpty = true;
}

if (WaitingTEmpty and SiteSetEmptyP(SetT)){
  /**
   ** EXIT THE CS
   **/
  WaitingTEmpty = false;
  TransitNormalState();
}


if (not PendingMessage())
  continue;
Msg = ReceiveMessage();
Y = SenderID(Msg);

sscanf(GetMessageString(Msg), MESSAGE_TYPE_FORMAT, msgbody, &Sy);
MaxSeq = max(MaxSeq, Sy);

/***
 *** MUTUAL EXCLUSION REQUESTING PROCESS
 ***/
if (StrEqual(msgbody, OK_MESSAGE)){
  /**
   ** OK MESSAGE (procedure ReceiptOK)
   **/
  if (WaitingOkWait){
    SiteSetRemove(NextSites, Y);
  }
  if (RequestingCS){
    SiteSetAdd(SetK, Y);
    SiteSetRemove(SetT, Y);
  } else {
    SendRelease(Y);
    SiteSetRemove(SetT, Y);
  }

  DisposeMessage(Msg);
```

```
      continue;
    }

    if (StrEqual(msgbody, WAIT_MESSAGE)){
      /**
       ** WAIT MESSAGE (procedure ReceiptWAIT)
       **/
      if (WaitingOkWait){
        SiteSetRemove(NextSites, Y);
      }
      SiteSetAdd(SetT, Y);

      DisposeMessage(Msg);
      continue;
    }

    if (StrEqual(msgbody, QUERY_MESSAGE)){
      /**
       ** QUERY MESSAGE (Procedure ReceiptQuery)
       **/
      if (ExecCS && SiteSetMemberP(SetK, Y)){
        SendAnswerNo(Y);
      } else {
        if (SiteSetMemberP(SetK, Y)){
          SendAnswerRelease(Y);
          SiteSetRemove(SetK, Y);
          SiteSetAdd(SetT, Y);
        } else {
          /* already released before query arrives */
          ; /* ignore it */
        }
      }

      DisposeMessage(Msg);
      continue;
    }

    /***
     *** TOKEN MANEGER PROCESS
     ***/
    if (StrEqual(msgbody, REQUEST_MESSAGE)){
      /**
       ** REQUEST MESSAGE (procedure ReceiptRequest)
       **/
      if (HaveToken){
        SendOk(Y, Sy);
        HaveToken = false;
      } else {
        if (LexicoLess(LatestTokenHolderSeqNo,LatestTokenHolderSiteID, Sy,Y)){
          SendWait(Y);
          EnPQueue(PQueue, Y,Sy);
        } else {
          if (not WaitingAnswer){
            WaitingY = Y;
            WaitingSy = Sy;
            SendQuery(LatestTokenHolderSiteID);
            WaitingAnswer = true;
          } else {
            if (LexicoLess(WaitingSy,WaitingY, Sy,Y)){
              SendWait(Y);
              EnPQueue(PQueue, Y,Sy);
            } else {
              SendWait(WaitingY);
              EnPQueue(PQueue, WaitingY,WaitingSy);
              WaitingY = Y;
              WaitingSy = Sy;
            }
          }
          /* See ReceiptANSWER_*** for following actions */
        }
      }

      DisposeMessage(Msg);
      continue;
    }

    if (StrEqual(msgbody, RELEASE_MESSAGE)){
      /**
       ** RELEASE MESSAGE (procedure ReceiptRELEASE)
       **/
      if (WaitingAnswer){
        goto GetPseudoAnsRel;
      }

      if (PQueueEmptyP(PQueue)){
        HaveToken = true;
      } else {
        Z  = PQueueHeadItem(PQueue);
        Sz = PQueueHeadPriority(PQueue);
        DiscardPQueueHead(PQueue);
        SendOk(Z, Sz);
        HaveToken = false;
      }

      DisposeMessage(Msg);
      continue;
    }

    if (StrEqual(msgbody, ANSWER_RELEASE_MESSAGE)){
      /**
       ** ANSWER_RELEASE MESSAGE (Token Manager Process)
       **/
      if (WaitingAnswer){
        /* continued action of ReceiptREQUEST */
        EnPQueue(PQueue, LatestTokenHolderSiteID,LatestTokenHolderSeqNo);
      GetPseudoAnsRel:
        WaitingAnswer = false;
        SendOk(WaitingY, WaitingSy);
      } else {
        SendFatalError("ANSWER_RELEASE arrived when !WaitingAnswer");
      }

      DisposeMessage(Msg);
      continue;
    }

    if (StrEqual(msgbody, ANSWER_NO_MESSAGE)){
      /**
       ** ANSWER_NO MESSAGE (Token Manager Process)
       **/
      if (WaitingAnswer){
        WaitingAnswer = false;
        /* continued action of ReceiptREQUEST */
        SendWait(WaitingY);
        EnPQueue(PQueue, WaitingY,WaitingSy);
      } else {
        SendFatalError("ANSWER_NO arrived when !WaitingAnswer");
      }

      DisposeMessage(Msg);
      continue;
    }

    /* ignore bogus msg */
    DisposeMessage(Msg);
  }
}


/***
 *** GetConsensusP() - Check if all sites in a quorum send OK or not.
 ***/
static bool
GetConsensusP(SetK, Coterie)
  SiteSet  SetK;
  kcoterie Coterie;
```

```
{
  int     q, qs, n, i;
  int     f;
  SiteSet quorum;

  n = GetTotalSites();
  qs = HowManyQuorums(Coterie);

  for (q = 0; q < qs; q++){
    f = true;
    NthQuorum(quorum, Coterie, q);
    for (i = 0; i < n; i++){
      if (SiteSetMemberP(quorum, i)){
        if (!SiteSetMemberP(SetK, i)){
          f = false;
          break;
        }
      }
    }
    if (f)
      return(true);
  }
  return(false);
}


static bool
LexicoLess(s1,x1, s2,x2)
  int     s1, s2;
  SiteID  x1, x2;
{
  return((s1 < s2)
          or ((s1 == s2) and (x1 < x2)));
}
```

B.2 Raymond's Distributed k-Mutaul Exclusion Algorithm

```
void
RaymondProcess(k, p, quantum, cycle, tcs)
  int     k;
  double  p;
  int     quantum;
  int     cycle;
  int     tcs;
{
  SiteID       Z;
  Message      Msg;
  SiteID       Y;
  int          Sy;
  int          Count;
  char         msgbody[80];
  int          LexicoLess(), Not_In_CS();
  void         SendRequestMessage(), SendReplyMessage();
  extern void  SiteBehavior();


  TransitNormalState();

  for (;;){

    /***
     *** DECISION OF BEHAVIOR OF SITE
     ***/
    SiteBehavior(k, p, quantum, cycle, tcs);

    if (ExitMutexJob)
      return;

    if (EnterCSRequestHappen){
```

```
    /**
     ** MUTEX REQUEST HAPPEN
     **/
    TransitRequestingState();
    EnterCSRequestHappen = false;
    Requesting_CS = true;
    Our_Seq = Max_Seq + 1;
    for (Z = 1; Z <= N; Z++){
      if (Z != me) {
        SendRequestMessage(Z, Our_Seq);
        Reply_Count[Z] = Reply_Count[Z] + 1;
      }
    }
  }

  if (ExitCSRequestHappen){
    /***
     *** EXIT THE CRITICAL SECTION
     ***/
    ExitCSRequestHappen = false;
    Executing_CS = false;
    TransitExitingCriticalSectionState();
    for (Z = 1; Z <= N; Z++){
      if (Defer_Count[Z] != 0){
        SendReplyMessage(Z, Defer_Count[Z]);
        Defer_Count[Z] = 0;
      }
    }
    TransitNormalState();
  }


  if (!PendingMessage())
    continue;
  Msg = ReceiveMessage();
  Y = SenderID(Msg);
  sscanf(GetMessageString(Msg), MESSAGE_TYPE_FORMAT, msgbody);

  if (StrEqual(msgbody, REQUEST_MESSAGE)){
    /**
     ** REQUEST MESSAGE
     **/
    sscanf(GetMessageString(Msg), REQUEST_FORMAT, msgbody, &Sy);
    Max_Seq = max(Max_Seq, Sy);
    if (Executing_CS
        or (Requesting_CS and LexicoLess(Our_Seq,me, Sy,Y))){
      Defer_Count[Y] = Defer_Count[Y] + 1;
    } else {
      SendReplyMessage(Y, 1);
    }
    DisposeMessage(Msg);
    continue;
  }

  if (StrEqual(msgbody, REPLY_MESSAGE)){
    /**
     ** REPLY MESSAGE
     **/
    sscanf(GetMessageString(Msg), REPLY_FORMAT, msgbody, &Sy, &Count);
    Reply_Count[Y] = Reply_Count[Y] - Count;
    if (Requesting_CS
        and (Not_In_CS() >= N - k)){
      /**
       ** ENTER THE CS
       **/
      Requesting_CS = false;
      Executing_CS = true;
      TransitInCriticalSectionState();
    }
    DisposeMessage(Msg);
    continue;
  }
```

```
    /* ignore bogus msg */
    DisposeMessage(Msg);
  }
}


int
Not_In_CS()
{
  int     Cnt;
  SiteID  Z;

  Cnt = 0;
  for (Z = 1; Z <= N; Z++)
    if ((Z != me)
        and (Reply_Count[Z] == 0))
      Cnt = Cnt + 1;

  return(Cnt);
}
```

B.3 The Behavior of a Process

The program fragment of a process behavior used in Chapter 5 is shown below. Functions whose name end by `Hook` are functions for collecting statistic data. For instance, a function `EnterCSHook()` is called when a process enters a critical section and the number of times a process enters a critical section is counted by this function.

```
float    _ProceedCSAt            = 0.0;
float    _TransitNormalAt        = 0.0;
float    _MutexRequestHappenedAt = 0.0;
static int  InNormalStateTimeCounter = 0;

State    MachineState = STATE_INITIAL;
bool     EnterCSRequestHappen = false;
bool     ExitCSRequestHappen = false;
bool     ExitMutexJob = false;

extern void
  EnterKMutexProcessHook(),
  ExitKMutexProcessHook(),
  SendHook(),
  CSRequestHook(),
  EnterCSHook(),
  ExitCSHook(),
  FinishMutexJobHook();


void
SiteBehavior(k, p, quantum, cycle, tcs)
  int     k;
  double  p;
  int     quantum;
  int     cycle;
  int     tcs;
{
  float    cval;

  cval = CurrentClock();

  if (cval >= (float)cycle){
    ExitMutexJob = true;
    return;
  }

  switch (MachineState){
  case STATE_INITIAL:
  case STATE_NORMAL:
    if (cval >= _TransitNormalAt + (float) InNormalStateTimeCounter){
      InNormalStateTimeCounter += 1;
      if (Random() < (float) p){
        EnterCSRequestHappen = true;
      }
    }
    break;
  case STATE_REQUESTING:
      /* do nothing */
    break;
  case STATE_IN_CRITICAL_SECTION:
    if (cval >= (_ProceedCSAt + (float)tcs)){
      ExitCSRequestHappen = true;
    }
    break;
  case STATE_EXITING_CRITICAL_SECTION:
    /* do nothing */
    break;
  default:
    fprintf(stderr, "Cannot happen in SiteBehabiour()\n");
    exit( -1 );
  }
}


void
TransitNormalState()
{
  State    oldstate;

  _TransitNormalAt = CurrentClock();
  InNormalStateTimeCounter = 0;
  oldstate = MachineState;
  if ((MachineState != STATE_EXITING_CRITICAL_SECTION)
      && (MachineState != STATE_INITIAL)){
    fprintf(stderr, "bogus state transition to NORMAL state\n");
    exit( -1 );
  }

  MachineState = STATE_NORMAL;
  if (oldstate == STATE_EXITING_CRITICAL_SECTION)
    FinishMutexJobHook();
}

void
TransitRequestingState()
{
  _MutexRequestHappenedAt = CurrentClock();
  if (MachineState != STATE_NORMAL){
    fprintf(stderr, "bogus state transition to REQUESTING state\n");
    exit( -1 );
  }
  MachineState = STATE_REQUESTING;
  CSRequestHook();
}

void
TransitInCriticalSectionState()
{
  _ProceedCSAt = CurrentClock();
  if (MachineState != STATE_REQUESTING){
    fprintf(stderr, "bogus state transition to InCriticalSection state\n");
    exit( -1 );
  }
  MachineState = STATE_IN_CRITICAL_SECTION;
  EnterCSHook();
}
```

```
void
TransitExitingCriticalSectionState()
{
  if (MachineState != STATE_IN_CRITICAL_SECTION){
    fprintf(stderr, "bogus state transition to InCriticalSection state\n");
    exit( -1 );
  }
  MachineState = STATE_EXITING_CRITICAL_SECTION;
  ExitCSHook();
}
```

Bibliography

[AA89] Divyakant Agrawal and Amr El Abbadi. An efficient solution to the distributed mutual exclusion problem. In *Principles of Distributed Computing*, pages 193–200, August 1989.

[Bal89] Henri E. Bal. Programming languages for distributed simulation computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[Bal94a] Roberto Baldoni. *Mutual Exclusion in Distributed Systems*. PhD thesis, Universita di Roma “La Sapienza”, 1994.

[Bal94b] Roberto Baldoni. An $O(n^{M/(M+1)})$ distributed algorithm for the k-out of-m resources allocation problem. In *The 14th International Conference on Distributed Computing Systems*, pages 81–88, 1994.

[BC94] Roberto Baldoni and B. Ciciani. Distributed algorithms for multiple entries to a critical section with priority. *Information Processing Letters*, 50:165–172, 1994.

[BGM87] Daniel Barbara and Hector Garcia-Molina. The reliability of voting mechanisms. *IEEE Transactions on Computers*, C-36(10):1197–1208, October 1987.

[BP89] J.E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, April 1989.

[BS91] Rajive L. Bagrodia and Chien-Chung Shen. Midas: Integrated design and simulation of distributed systems. *IEEE Transactions on Software Engineering*, 17(18):1042–1058, October 1991.

[Bur94] J. E. Burns. private communication. July 1994.

[CAA90] Shun Yan Cheung, Mustaque Ahamad, and Mostafa H. Ammar. Multi-dimensional voting: A general method for implementing synchronization in distributed systems. In *Proceedings of 10th International Conference of Distributed Computing Systems*, pages 362–369, 1990.

[CM84] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[CR83] O. S. F. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Communications of the ACM*, 26(2):146–147, February 1983.

[CTW92] Don Coppersmith, Prasad Tetali, and Peter Winkler. Collisions among random walks on a graph. *SIAM Journal of Discrete Mathematics*, 6(3):363–374, August 1992.

[Dij68] E. W. Dijkstra. *Programming Languages*, chapter Sequential Communicating Processes. Academic Press, N.Y., 1968.

[Dij74] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[Dij82] E.W. Dijkstra. Self-stabilization in spite of distributed control. In *Reprinted in Selected Writing on Computing: A Personal Perspective*, pages 41–46. Springer-Verlag, Berlin, 1982.

[DIM90] S. Dolev, Amos Israeli, and S. Moran. Self stabilization of dynamic systems assuming only read/write atomicity. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 103–117. ACM, 1990.

[DIM91] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform dynamic self-stabilizing leader election. *Lecture Notes for Computer Science 579*, pages 167–180, 1991.

[DIM93] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[FDG94] Mitchell Flatebo, Ajoy Kumar Datta, and Sukumar Ghosh. *Readings in Distributed Computing Systems*, chapter Self-Stabilization in Distributed Systems, pages 100–114. IEEE Computer Society Press, Los Vaqueros Circle, Los Alamos, CA, USA, 1994.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FYA91] Satoshi Fujita, Masafumi Yamashita, and Tadashi Ae. A non trivial solution of the distributed k-mutual exclusion problem. *ISA '91 Algorithms LNCS 557*, 1991.

[Gho91] Sukumar Ghosh. Binary self-stabilization in distributed systems. *Information Processing Letters*, 40(3):153–159, November 1991.

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of 7th Symposium on Operating Systems*, pages 150–162. ACM, 1979.

[GMB85] Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, October 1985.

[Hag90] Ken'ichi Hagihara. Distributed algorithms. *Journal of Information Processing Society of Japan*, 31(9):1245–1256, September 1990. (in Japanese).

[Hag93] Ken'ichi Hagihara. Algorithms for fault-tolerant distributed systems. *Journal of Information Processing Society of Japan*, 34(11):1336–1340, November 1993. (in Japanese).

[Her90] Ted Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, June 1990.

[Hua93] Shing-Tsaan Huang. Leader election in uniform rings. *ACM Transactions on Programming Languages and Systems*, 15(3):563–573, July 1993.

[IJ90] Amos Israeli and Marc Jalfon. Token management schemes and random walks yield self stabilizing mutual exclusion. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 119–131. ACM, 1990.

[IK91] Toshihide Ibaraki and Tiko Kameda. Theory of coteries. In *Proc. 3rd Symp. on Parallel and Distributed Systems*, pages 150–157, 1991.

[Kum91] Akhil Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Transactions on Computers*, 40(9):996–1004, September 1991.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LS92] C. Lin and J. Simon. Observing self-stabilization. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 113–123. ACM, 1992.

[MA93] Yoshifumi Manabe and Shigemi Aoyagi. A distributed k-mutual exclusion algorithm using k-coterie. *IEICE Japan, SIG Computation Record*, COMP91-13:11–18, May 1993. (in Japanese).

[Mae85] Mamoru Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, March 1985.

[Mis86] Jayadev Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[Miy94] Hidenori Miyamoto. A study on quorum based approach for solving the anonymous resource conflict resolution problem. Master's thesis, Hiroshima University, February 1994.

[MLR91] Masaaki Mizuno, Mitchell L.Neilsen, and Raghavendra Rao. A token based distributed mutual exclusion algorithm based on quorum agreements. In *Proc. of 11th International Conference on Distributed Computing Systems*, pages 361–368, May 1991.

[MYKC94] Shyan Ming Yuan and Her Kun Chang. Comments on "availability of k-coterie". *IEEE Transactions on Computers*, 43(12):1457, December 1994.

[NM92] Mitchell L. Neilsen and Masaaki Mizuno. Coterie join algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):582–590, 1992.

[NM94] Nitchell L. Neilsen and Masaaki Mizuno. Nondominated k-coteries for multiple mutual exclusion. *Information Processing Letters*, 50:247–252, 1994.

[NMR92] Mitchell L. Neilsen, Masaaki Mizuno, and Michel Raynal. A general method to define quorums. In *Proceedings of 12th International Conference of Distributed Computing Systems*, pages 657–664, 1992.

[NMT92] Naoki Nishikawa, Toshimitsu Masuzawa, and Nobuki Tokura. Uniform self-stabilizing algorithm for mutual exclusion. *IEICE Japan*, J75-D-I(4):201–209, April 1992. (in Japanese).

[RA81] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer network. *Communications of the ACM*, 24(1):9–17, January 1981.

[RA83] Glenn Ricart and Ashok K. Agrawala. Author's response to 'On mutual exclusion in computer networks' by Carvalho and Roucairol. *Communications of the ACM*, 26(2):147–148, February 1983.

[Ray86] Michel Raynal. *Algorithms for Mutual Exclusion*. North Oxford Academic, 1886. (Translated by D. Beeson).

[Ray89a] Kerry Raymond. A distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 30:189–193, February 1989.

[Ray89b] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, February 1989.

[Ray91a] Michel Raynal. A distributed solution to the k-out of-m resources allocation problem. In *Lecture Notes in Computer Science 497*, pages 599–609. Springer-Verlag, 1991.

[Ray91b] Michel Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *ACM Operating Systems Review*, 25(2):47–51, 1991.

[San87] Beverly A. Sanders. The information structure of mutual exclusion algorithms. *ACM Transactions on Computer Systems*, 5(3):284–299, August 1987.

[Sch93] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

[SG92] Abraham Silberchatz and Peter B. Galvin. *Operating Systems Concepts Fourth Edition*. Addison-Wesley, Reading, MA, 1992.

[Sin91] Mukesh Singhal. A class of deadlock-free maekawa-type algorithms for mutual exclusion in distributed systems. *Distributed Computing*, pages 131–138, April 1991.

[SK85] Ichiro Suzuki and Tadao Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 3(4):344–349, November 1985.

[SM92] R. Satyanarayanan and D. R. Muthukrishnan. A note on Raymond's tree based algorithm for distributed mutual exclusion. *Information Processing Letters*, 43(5):249–255, October 1992.

[SR92] Pradip K. Srimani and Rachamallu L.N. Reddy. Another distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 41(1):51–57, January 1992.

[SS94] Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems — distributed, database, and multiprocessor operating systems*. McGraw-Hill, 1994.

[STHK93] Jehn-Ruey Jiang Shing-Tsaan Huang and Yu-Chen Kuo. k-coteries for fault-tolerant k entries to a critical section. In *Proceedings of 13th International Conference of Distributed Computing Systems*, pages 362–369, 1993.

[Sun90] Sun Microsystems, INC. *Network Programming Guide*, part number: 800-3850-10 revision a of 27 edition, 1990.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[Tau91] Gadi Taubenfeld. On the nonexistence of resilient consensus protocols. *Information Processing Letters*, 37:285–289, March 1991.

[Tho79] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2), 1979.

[Tok89] Nobuki Tokura. A tool for distributed algorithm simulation. *Journal of Information Processing Society of Japan*, 30(4):380–386, April 1989.

[Yam93] Masafumi Yamashita. The distributed mutual exclusion problem and coteries. *Journal of Information Processing Society of Japan*, 34(11):1350–1357, November 1993. (in Japanese).