THE UNIVERSITY OF CALGARY

Dining Philosophers and Other Process Coordination Problems: Algorithms, Tools and Stabilization

by

Lixiao Wang

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

August, 2003

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Dining Philosophers and Other Process Coordination Problems: Algorithms, Tools and Stabilization" submitted by Lixiao Wang in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

Lisa Higham

Supervisor, Dr. Lisa Higham
Department of Computer Science

Christiane Lemieux

Dr. Christiane Lemieux
Department of Computer Science
and of Mathematics and Statistics

Michael P. Lamoureux

Dr. Michael Lamoureux
Department of Mathematics and
Statistics

2003 August 26

Date

Abstract

The Process Coordination problem, also called concurrency control, is one of the classic problems in distributed computing.

This thesis has two major contributions. It reviews some significant research results on the Process Coordination problems with the contribution of unifying different kinds of problems. Two common frameworks are devised to describe different problems and solutions, so that the similarities and differences among them can be highlighted. The Dining Philosophers problem is demonstrated to be a fundamental representative of the set of Process Coordination problems.

The second contribution is to develop a robust algorithm for the Dining Philosophers problem that can withstand transient system failures. Self-stabilization is a strong model that handles transient faults in distributed systems. A randomized self-stabilizing mechanism that assigns distinct labels up to distance 3 in a network is presented. This mechanism, when generalized to assign distinct labels up to distance k, for any positive integer k, can solve the self-stabilizing Dining Philosophers problem, and it may have other independent applications. Construction and proof of this mechanism rely upon a new tool for combining randomized self-stabilizing algorithms even when they both update common variables.

Acknowledgments

First of all, I would like to thank my supervisor, Dr. Lisa Higham, for her guidance, support, and encouragement.

I would also like to thank Dr. Christiane Lemieux and Dr. Michael Lamoureux for their valuable comments and corrections.

Finally, I would like to thank my family, my mother Qinglan Xu, my father Bingheng Wang, and my sister Liwei Wang. They have always been very supportive of my pursuit of advanced education.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

A distributed system [24, 1] is a collection of individual computing devices that can communicate with each other. Many kinds of systems belong to the distributed setting, such as communication networks, multiprocessor computers, and a multitasking single computer. All these systems have similar fundamental coordination requirements among the communicating entities, whether they are computers, processors, or processes. Here we use the term process to indicate any computing device.

1.1 Process Coordination problem

In this thesis we study a classic problem in distributed settings, *the Process Coordination problem.* The Process Coordination problem (also called *concurrency control*) consists of a set of processes communicating with each other to execute some coordination activities, which normally happen as a result of agreements.

Usually Process Coordination problems need to deal with mutual exclusion and synchronization. Mutual exclusion requires that two concurrent activities do not

access shared data at the same time. Synchronization provides a condition to coordinate the actions of concurrent activities.

Difficulty arises in symmetric and asynchronous systems, where processes are all identical and no assumption is made on relative process speeds or the number of processes. One undesirable situation occurs when two activities are waiting for each other and neither can proceed. This sort of circular waiting is called a *deadlock*. For example, suppose processes A and B each need two resources to continue, but only one resource has been assigned to each of them. If the system has only two such resources, neither process can ever proceed. This situation can also be generalized to a sequence of processes, $p_1, ..., p_n$, where p_i is waiting for p_{i+1} for $i = 1, ..., n-1$ and p_n is waiting for p_1. None of these processes can make progress. Another situation related to the Process Coordination problem is *starvation*, which occurs when a blocked activity is consistently passed over and not allowed to run. For example, consider two cpu bound jobs, one running at a higher priority than the other. The lower priority process could never be allowed to execute.

Various problems arise based on different properties of the system, requirements on mutual exclusion and synchronization, and ability to prevent deadlock and starvation. Processes in some problems are tightly or centrally controlled by other entities. Other problems allow individual processes to make independent decisions and be notified of changes. Some problems require that at least one process in the system does not starve, while others require that every process cannot starve.

In this thesis six typical problems that belong to the set of Process Coordination problems are considered: the Dining Philosophers problem, the Drinking Philosophers problem, the Resource Allocation problem, the Committee Coordination prob-

lem, the Multiway Rendezvous problem, and the Multiparty Interactions problem. Based on the problem specification and the way to solve them, these problems can be divided into two classes:

General Resource Allocation problems: including Dining Philosophers, Drinking Philosophers, and Resource Allocation problems.

Process Synchronization problems: including Committee Coordination, Multiway Rendezvous, and Multiparty Interaction problems.

Understanding the differences among these distinct problems will help one make an appropriate choice to solve specific issues. Also understanding the similarities among these problems will help one implement one problem in terms of another.

One of the objectives of our research is to study different Process Coordination problems by giving a literature survey of some significant results.

Most problem descriptions existing in the literature are informal, ambiguous, or even self-contradictory. Another objective is to overcome these difficulties by devising two uniform frameworks for specifying problems, the *Object Oriented model* and the *Graph model.* First each problem is introduced by paraphrasing its informal description and requirements. Then the ambiguities or contradictions are pointed out. Finally the problem and its solution are recast using the Object Oriented model and Graph model. The more formal descriptions provide the unambiguous specifications often missing in the general literature. This also makes it possible to highlight the similarities and differences among various problems.

The Dining Philosophers problem has been considered as a classic Resource Allocation problem since presented and solved by Dijkstra in 1965. In this thesis the

Dining Philosophers problem is demonstrated to be a fundamental problem of the set of Process Coordination problems, because other Process Coordination problems can either be mapped directly onto the Dining Philosophers problem or can be solved by using its solution as a subroutine. A robust and fault-tolerant solution to the Dining Philosophers problem is one way to provide a robust and fault-tolerant tool to solve each of the other Process Coordination problems.

The last contribution of this thesis is to design a particular kind of fault-tolerant solution (a self-stabilizing solution) to the Dining Philosophers problem in *fully distributed* and *completely symmetric* systems. In fully distributed systems, no central memory or central process is used. In completely symmetric systems, all processes are identical and have the same initial state. In particular, they do not have distinct identifiers.

1.2 Modeling Distributed Systems

Because there are so many assumptions, issues, and alternatives in distributed systems, an abstract representation of the distributed system, in which different Process Coordination problems are described, should be given. These representations are often called *models*.

When modeling a distributed system, several components needs to be described to capture the variants of distributed systems caused by different process behaviors and communication behaviors. In this section, some common alternative choices for several different such components are given.

1.2.1 Communication

The communication model describes the mechanisms, through which processes exchange information with each other. Typically two models are used: the *message passing* model and the *shared memory* model. For each of these general classifications, the model must also specify which of several other possible variants is being assumed.

In the message passing model, processes communicate by exchanging messages through unidirectional or bidirectional communication channels. Normally each communication channel is modeled as a queue. A process sends a message by adding it to the appropriate outgoing channel(s), and receives a message by removing it from one of its incoming channel. A channel may have bounded or unbounded size. A process may send one message to a specified neighbor in one step, broadcast a message to a subset of neighbors in one step, or even send different messages to a set of neighbors in a single step. A communication channel may deliver messages in the same order as they were added (a FIFO queue), or deliver them in an arbitrary order.

In the shared memory model, processes communicate by accessing one or more shared objects. Frequently, these objects are atomic variables. An atomic variable can be either read or written in one single step. If a variable can be written by one process but read by several processes, it is called a single-writer multi-reader variable. A variable can also be multi-writer multi-reader or even single-writer single-reader. Sometimes stronger objects are used, such as test-and-set, or fetch-and-add, or other read-modify-write objects such as queues and stacks. With some reasonable additional assumptions, algorithms designed for shared memory settings can usually

be transformed into algorithms for message passing settings.

A variant of the shared memory model is called the *link register* model. Communication between two processes p and q is modeled by two single-writer single-reader registers R_q^p and R_p^q. Register R_q^p is written by p and read by q. Similarly, register R_p^q is written by q and read by p. The link register model is commonly used in the self-stabilizing setting, which will be introduced in Section 1.2.9. In this thesis, a slightly modified link register model is used. Each register is assumed to have a single writer but two readers. The extra reader is the process that also writes to it. Thus in the above example both process p and q can read R_p^q and R_q^p. This modified model is no more powerful than the standard link register model, because one can always assume that every process keeps a local copy of everything it wrote to its shared registers. A process can simulate the reading operation on its shared registers by reading the corresponding local copies. Normally, a message passing model can be simulated by a link register model with certain restrictions.

1.2.2 Communication Topology

For any distributed system, construct the underlying communication graph (also called the network of the system) as follows: assign a vertex to every process, and put an edge between two processes if they can communicate with each other directly. Two processes are called neighboring processes, if there is an edge between their corresponding vertices. Another variant models communication with a directed graph (see page 22), when a message passing system has unidirectional channels.

Some problems are only defined in networks with restricted topologies such as rings, trees, or complete graphs. Other problems may have constraints on processes'

knowledge of the whole network. For example, there may be a restriction on the number of vertices or the maximum degree of any vertex in the system. As a consequence, some solutions are only designed and correct in systems with special topologies. The topology of the communication graph can be fixed. It can also be dynamic, which means the addition or deletion of nodes or communication channels are allowed as the application executes.

1.2.3 Network Labels

Some applications require that the processes of the network have labels. Different problems may have different assumptions about how the vertices are labeled. For example, the neighboring processes may require distinct labels, or any two processes within distance k may require distinct labels. Sometimes there is one special process (often called the *leader*), whose label is distinct from all the other processes, which may all have the same label. If none of the vertices are labeled, then the system is called *anonymous*. Solving problems in a labeled setting[1] is typically easier than in an anonymous setting. Normally, problems become more general (and interesting) in anonymous systems.

Edges in a network could also have labels to reflect some properties. For example, the weight of an edge could represent the cost of communication on the corresponding link.

Even in the anonymous setting, every process usually has some way to distinguish between its neighbors. This is accomplished by providing a process with a local name

[1]Here we assume that not all labels are identical; if they were, then the system cannot be distinguished from an anonymous system.

for every communication channel or every shared register it uses to communicate with each of its neighbors. Systems with this feature are *locally oriented.*

1.2.4 Timing

Two basic models of timing in distributed systems are the *synchronous* model and the *asynchronous* model.

In the synchronous model, processes take steps simultaneously. The execution proceeds in synchronous rounds. Typically this is achieved by using a global clock pulse, which triggers the next step of each process. The synchronous model is the simplest model to describe, to program and to analyze. Even though this model is not very realistic, understanding how to solve a problem under the synchronous setting is often helpful for developing solutions for more complex and realistic settings.

In the asynchronous model, processes take steps at arbitrary speeds. Both the absolute speed of each process and the relative speed between processes may vary arbitrarily during the computation. Asynchrony makes it hard to predict the state of the system at any particular time. The asynchronous model is more general and practical. However it is harder to solve problems without timing constraints.

1.2.5 Scheduler

In an asynchronous system, the *scheduler* (also called the *daemon*) manages the activities of processes. At each step a scheduler determines which processes execute the next operation of their program. Two common schedulers are the *central scheduler* and the *distributed scheduler.* A central scheduler activates only one process at a time. The distributed scheduler selects a nonempty set of processes and activates

them all simultaneously. Some algorithms work correctly under a central scheduler but not under a distributed scheduler. The requirement of a central scheduler is usually considered an unreasonable assumption for a truly distributed system. This is because the central scheduler is implemented by either using global information, which is unrealistic, or using a mutual exclusion technique, which defeats the potential concurrency of the system.

A scheduler produces a computation of the system by interleaving operations of all processes. The fairness assumption on a scheduler captures the behaviors of this interleaving. There are many different strengths of fairness. A *weakly fair* scheduler ensures that each process takes an infinite number of steps in any infinite execution. A *k-fair* scheduler ensures that in any interval where a process takes $k+1$ steps, every other processes takes at least one step. A *round robin* scheduler activates processes in a fixed order under a 1-fairness assumption.

1.2.6 Algorithm Type

An algorithm can be *deterministic* or *randomized.* A deterministic algorithm provides a transition function for each process. When a process is selected by the scheduler, its next state is determined by this function. A randomized algorithm provide a probability space for the next state for each current state of every process. When a process is selected, its state is randomly chosen from the probability space.

A deterministic algorithm *solves a problem P under a set S of schedulers*, if all possible computations produced by any element of S satisfy the specification of problem P.

Two kinds of randomized algorithms are often considered: *Las Vegas randomized*

algorithms and *Monte Carlo randomized algorithms.* An algorithm is called a Las Vegas randomized algorithm for problem P under a set S of schedulers, if with probability 1, all possible computations produced by any element of S satisfy the specification of P. An algorithm is called a Monte Carlo randomized algorithm for problem P under a set S of schedulers, if with probability at least p, any computation produced by an element of S satisfies the specification of P, where $0 < p < 1$. Randomization is often exploited by an algorithm in order to break symmetry in an anonymous setting.

1.2.7 Atomicity

An *atomic step* is the computation performed by a process as one indivisible action. There are two typical kinds of atomicity: *composite atomicity* and *read/write atomicity*, which differ in the size of the atomic step. In composite atomicity, an atomic step consists of several operations of a process. For example, a process can read the state of all its neighbors and change its own state in one atomic step. In read/write atomicity, an atomic step is only a single read or a single write operation.

Clearly algorithms designed under read/write atomicity still work correctly in any systems with composite atomicity, but not vice versa. Furthermore, solving a problem and proving the correctness of a solution are typically much easier by assuming composite atomicity than read/write atomicity.

In randomized algorithms, random choices, such as coin flips, are often made. If the random choice is not separated from the following read or write operation, the model is called *coarse atomicity.* In the *fine atomicity* model, however, an atomic step contains only a single random choice, a single read operation or a single write

operation.

1.2.8 Fault Models

Real systems consist of many varied components arranged in complex interconnections. Even when individual components are very reliable, a failure somewhere in the system is likely simply because of number of components and the complexity of their interconnections. A realistic system model (such as the asynchronous, distributed scheduler model) should capture most kinds of process and communication failures that might happen. Commonly considered failures include:

- **Process failures:**
  - Stop failures: Any process might stop executing its program forever after a certain point in the execution.
  - Transient faults: Any process may stop executing its program for a while and then recover from the fault later.
  - Byzantine failures: Any process can act arbitrarily and maliciously without being identified as faulty by the rest of the system.
- **Message passing communication failures:** The message might be lost, duplicated, reordered, or even corrupted.
- **Shared memory communication failures:** The shared data can be corrupted.

In the following section, a technique used to handle transient faults is introduced.

1.2.9 Self-Stabilization

Many of the distributed algorithms discovered so far have a shortcoming: they are correct only under unrealistic simplifying assumptions about the distributed system, where no transient faults happen. Normally when errors occur in distributed systems, they tend to be "bursty" since errors tend to create more errors. A distributed algorithm is more useful when it is fault-tolerant. A strong fault-tolerant model to handle transient faults is the self-stabilizing distributed system.

When something goes wrong, a self-stabilizing distributed system can automatically return to an error-free configuration without being manually reset, or shut down and rebooted. Also a self-stabilizing distributed system can start in any arbitrary configuration and after some bounded amount of computation reach a predefined *legitimate configuration.* If the arbitrary configuration includes arbitrary topologies, then the self-stabilizing protocol can be used for dynamic systems. Once a self-stabilizing system reaches a legitimate configuration, it stays in a legitimate configuration during any fault-free computation.

Since 1974, when Dijkstra [10] introduced the idea of self-stabilization, and more intensely, since 1983, after Lamport [20] highlighted its importance, there has been substantial research on self-stabilizing distributed algorithms. Self-stabilization is an elegant feature for distributed systems, because the manageability and stability of distributed systems decrease rapidly with the increase of the number of components or the complexity of the interconnection among components. Distributed systems also present challenging research topics for self-stabilization due to their heterogeneous computing environments, mismatching computing power of system

components, and various computing constraints.

1.3 Overview of Thesis

In this thesis, we study six different Processes Coordination problems, demonstrate the fundamental role of the Dining Philosophers problem in this set of problems, and design a self-stabilizing solution to the Dining Philosophers problem. Chapters 2 and 3 review several significant papers on the set of General Resource Allocation problems and the set of Process Synchronization problems, respectively, and unify different problems using two common frameworks that are introduced at the beginning of Chapter 2. Chapter 4 compares the similarities and differences between the General Resource Allocation problem and the Process Synchronization problem by comparing two representatives from each class, the Dining Philosophers problem and the Committee Coordination problem. Chapter 5 gives the formal definitions of self-stabilization and a new enriched fair composition technique that can be used to design and prove correct complex self-stabilizing algorithm. This technique is used in Chapter 6 together with other ideas to devise a self-stabilizing solution to the Dining Philosophers problem. In Chapter 7, we summarize the contributions of this thesis, discuss further comments and describe the future work.

CHAPTER 2

General Resource Allocation Problems

A General Resource Allocation problem consists of a set of resources and a set of potential users of these resources. From time to time every user may require a set of resources. Upon being granted all the requested resources, the user will do some work with the resources and relinquish them eventually. The problem is to devise a protocol that satisfies the following two requirements:

Exclusion: no resource can be used by more than one user simultaneously.

Lockout-Freedom: every requesting user will eventually acquire all the resources she needs.

If each user uses a fixed set of resources, then the problem is a *Static Resource Allocation problem.* Otherwise, if a user requires different sets of resources each time, then the problem is a *Dynamic Resource Allocation problem.* The Dining Philosophers problem and the Drinking Philosophers problem are fundamental versions of the Static Resource Allocation problem and the Dynamic Resource Allocation prob-

lem, respectively, where the philosophers represent users and forks or beverages represent resources.

In this chapter, we discuss some previous work on the Dining Philosophers problem and the Drinking Philosophers problem. Most problem descriptions existing in the literature are informal and unclear. First we paraphrase the original descriptions and requirements, then we recast both problems and their solutions using two frameworks, the Object Oriented model and Graph model. This makes it easy to clarify the ambiguities and to illustrate the similarities and differences between these two problems.

2.1 Two Frameworks for Modeling Distributed Systems

In this section, two models are introduced, which will be used to define the distributed systems used in this thesis: the Object-Oriented Model and the Graph Model. We will use the Dining Philosophers problem as a running example while describing both models. Here is an informal description of the Dining Philosophers problem.

The Dining Philosophers problem, which is first defined by Dijkstra [9] and later generalized by several authors, consists of a set of philosophers sharing a set of forks. Each philosopher goes indefinitely through a cycle: thinking, hungry, and eating. When a philosopher thinks, she does not interact with others. From time to time, she may get hungry and want to eat. To eat she needs a fixed set of forks. A philosopher may only pick up one fork at a time and she cannot pick up a fork that is already in the hand of another philosopher.

The exclusion property is that no two philosophers eat simultaneously if they

use a common fork. *Deadlock-freedom* is the property that if any philosopher is hungry, then eventually there is a philosopher that eats. *Lockout-freedom* is the property that if a philosopher is hungry, then eventually that philosopher will eat. The Dining Philosophers problem requires exclusion and one of two progress properties. Either the progress is deadlock-freedom or the progress is lockout-freedom under the assumption that every eating philosopher will eventually finish and release all her forks.

2.1.1 Object Oriented Model of a Distributed System

In a distributed system that communicates through shared memory, the computing devises can be represented as *processes* that communicate via shared *objects*. In this thesis such a distributed system is modeled by a set of processes P operating on a set of objects J and is denoted by (P, J). We will specify both components, and give the meaning of an implementation of a distributed system.

Objects

Every object supports a nonempty set of operation(s) applicable to it. To define an object, one can describe a collection of states and a collection of operations, and for a pair of operation and state, give a resulting state. This specifies the precondition and effect of each operation, and gives rise to a collection of allowable sequence of operations on the object. Lynch used this way to describe algorithms in her book "Distributed Algorithms" [24].

Equivalently, Herlihy and Wing [17] introduced another way to define an object by giving a set of allowable sequences of operations on it. An arbitrary sequence of

operations applied to an object x is *valid* if and only if it is in the specification of x. An arbitrary sequence of operations on a collection of objects X is valid, if and only if for each object $x \in X$, the subsequence of operations applied only to x is valid.

In this thesis, an operation by process p is denoted by $\langle \text{operation\_name}(I) \rangle_p{:}O$, where I is the input parameter of the operation and O is the output parameter. When the process's name is not required or obvious from context, the subscript, p, is omitted.

For example, in the Dining Philosophers system, the object is a set of forks $\mathcal{F} = \{f_1, f_2, ..., f_k\}$ that supports two operations, $\langle \text{grab}(F) \rangle$ and $\langle \text{release}(F) \rangle$, where $F \subseteq \mathcal{F}$. Informally any interleaving of $\langle \text{grab}(\cdot) \rangle$ and $\langle \text{release}(\cdot) \rangle$ operations on $\mathcal{F}$ is valid if and only if: 1) for every $\langle \text{grab}(F) \rangle$, all forks in F are available before the operation and will become unavailable after the operation; and 2) for every $\langle \text{release}(F) \rangle$, all forks in F are not available before the operation and will become available after the operation.

More formally, let $S = o_1, o_2, ...$ be any sequence of $\langle \text{grab}(\cdot) \rangle$ and $\langle \text{release}(\cdot) \rangle$ operations on $\mathcal{F}$. To give the validity condition on S we define another variable $\mathcal{CF}_i$ inductively by:

1) $\mathcal{CF}_0 = \mathcal{F}$

2) $$\mathcal{CF}_i = \begin{cases} \mathcal{CF}_{i-1} \cup F_i & \text{if } o_i = \langle \text{release}(F_i) \rangle \\ \mathcal{CF}_{i-1} \backslash F_i & \text{if } o_i = \langle \text{grab}(F_i) \rangle \end{cases}$$

Then S is valid provided

1) if $o_i = \langle \text{release}(F) \rangle$ for $F \subseteq \mathcal{F}$, then $F \cap \mathcal{CF}_{i-1} = \emptyset$, and

2) if $o_i = \langle \text{grab}(F) \rangle$ for $F \subseteq \mathcal{F}$, then $F \subseteq \mathcal{CF}_{i-1}$.

The following three sequences contain operations on $\mathcal{F}$, where $F_1 = \{f_1, f_2, f_3\}$, $F_2 = \{f_4, f_5\}$, and $F_3 = \{f_3, f_6\}$. It is easy to see that S_1 is valid, but S_2 and S_3 are not because property 1) and 2) are violated, respectively.

$$S_1 : \langle \text{grab}(F_1) \rangle, \langle \text{grab}(F_2) \rangle, \langle \text{release}(F_1) \rangle, \langle \text{grab}(F_3) \rangle, \langle \text{release}(F_3) \rangle, \langle \text{release}(F_2) \rangle$$

$$S_2 : \langle \text{release}(F_2) \rangle, \langle \text{grab}(F_1) \rangle, \langle \text{release}(F_1) \rangle, \langle \text{grab}(F_3) \rangle, \langle \text{release}(F_3) \rangle$$

$$S_3 : \langle \text{grab}(F_1) \rangle, \langle \text{grab}(F_3) \rangle, \langle \text{release}(F_3) \rangle, \langle \text{release}(F_1) \rangle$$

Processes

A process in a distributed system is just program of operations applicable to objects in the system. The order in which the operations of a process are invoked by the program is called *program order.*

Processes in the Dining Philosophers problem are the set of philosophers $\mathcal{P} = \{p_1, p_2, ..., p_n\}$. Associated with each $p_i \in \mathcal{P}$ is an nonempty and static set of forks $F(i) \subseteq \mathcal{F}$, which indicates the set of forks used by philosopher p_i. The program for each $p_i \in \mathcal{P}$ is:

Do Forever:

$\langle \text{think} \rangle$

$\langle \text{grab}(F(i)) \rangle$

$\langle \text{eat} \rangle$

$\langle \text{release}(F(i)) \rangle$

End Do

The $\langle \text{think} \rangle$ and $\langle \text{eat} \rangle$ are arbitrary operations that do not act on $\mathcal{F}$.

Computations

Each process's *individual computation* is a sequence of operations executed by the process, such that the order of the operations agrees with the program order. A *computation* of a distributed system is some valid interleaving of the operations of each process as determined by the scheduler. More formally, a *computation of a distributed system* (P, J) is a total order of the operations of all processes in P that extends program order and is valid.

The computation of the Dining Philosophers system is an interleaving of operations of each process's program that extends program order and satisfies the validity conditions for $\langle \text{grab}(\cdot) \rangle$ and $\langle \text{release}(\cdot) \rangle$. For example, suppose p_1, p_2, and p_3 are philosophers and use a set of forks $F_1 = \{f_1, f_2, f_3\}$, $F_2 = \{f_4, f_5\}$, and $F_3 = \{f_3, f_6\}$, respectively. Then S_4 is a computation of the Dining Philosophers system, but S_5 is not because even though it preserves program order, it does not satisfy the validity conditions.

S_4: $\langle \text{think} \rangle_{p_1}, \langle \text{grab}(F_1) \rangle_{p_1}, \langle \text{think} \rangle_{p_2}, \langle \text{eat} \rangle_{p_1}, \langle \text{think} \rangle_{p_3}, \langle \text{grab}(F_2) \rangle_{p_2}, \langle \text{eat} \rangle_{p_2},$
$\langle \text{release}(F_1) \rangle_{p_1}, \langle \text{grab}(F_3) \rangle_{p_3}, \langle \text{eat} \rangle_{p_3}, \langle \text{release}(F_3) \rangle_{p_3}, \langle \text{release}(F_2) \rangle_{p_2}$

S_5: $\langle \text{think} \rangle_{p_2}, \langle \text{eat} \rangle_{p_2}, \langle \text{release}(F_2) \rangle_{p_2}, \langle \text{think} \rangle_{p_1}, \langle \text{think} \rangle_{p_3}, \langle \text{grab}(F_1) \rangle_{p_1}, \langle \text{grab}(F_3) \rangle_{p_3},$
$\langle \text{eat} \rangle_{p_3}, \langle \text{release}(F_3) \rangle_{p_3}, \langle \text{eat} \rangle_{p_1}, \langle \text{release}(F_1) \rangle_{p_1}$

Implementation of a Distributed System

A *distributed problem* requires that a specified system (P, J) be *implemented* using a different set of objects, $\hat{J}$. For example, the objects in the specification level Dining Philosophers system are sets of forks. When implemented in a shared memory setting, objects are usually atomic read/write variables and test-and-set objects. In

message passing settings, objects are messages or tokens. Figure 2.1 illustrates the relation between a specification level system (P, J) and the corresponding implementation level system $(\hat{P}, \hat{J})$.

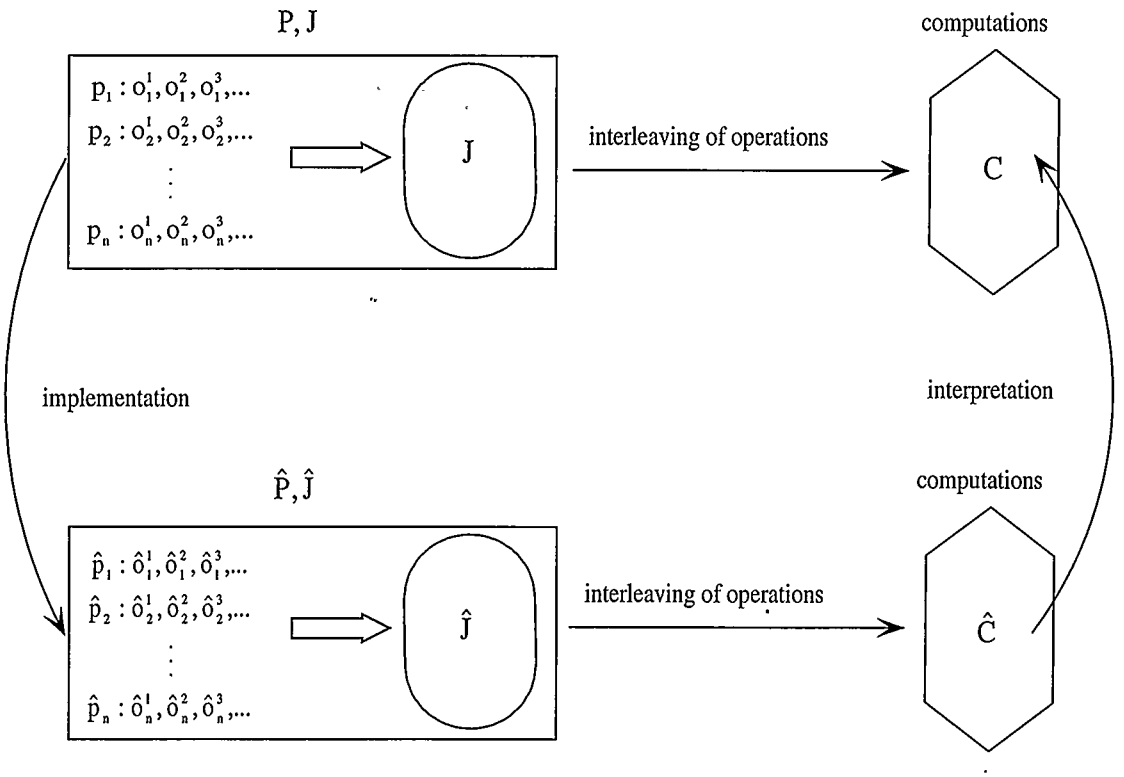


Figure 2.1: Specification Level System vs. Implementation Level Systems

For each operation on each object in J at the specification level, an implementation provides a procedure call containing operations on the lower level objects in $\hat{J}$. Each such procedure invocation must return a response of the same type as the specification level operation that it implements. For example, an implementation of the Dining Philosophers system must provide programs for $\langle \text{grab}(\cdot) \rangle$ and $\langle \text{release}(\cdot) \rangle$ that eventually return successfully.

The processes $\hat{P}$ of the implementation level are constructed by replacing each operation of each process in P by the procedure that implements that operation.

The system $(\hat{P}, \hat{J})$ so constructed has a set of computations as well. Each computation is just a valid sequence formed by some interleaving of the operations of the implementation level processes, $\hat{P}$. Any such implementation level computation can be *interpreted* as a computation of the specified system (P, J) by assuming that each high level operation occurred atomically at some point between the invocation and the response of its implementing procedure.

For the implementation to be correct, we require that every possible computation of $(\hat{P}, \hat{J})$ must have an interpretation that is valid for the specified system[1].

2.1.2 Graph Model of a Distributed System

A distributed system can also be presented in graph theoretical terms. In this section, we first give some basic concepts in graph theory, which can be found in most introductory graph theory texts (for example [4]). Then we show how to represent a distributed system by a graph.

Graph Theory Preliminaries

A *graph* $G = \{V, E\}$ consists of a nonempty set of *vertices* (or *nodes*) denoted by V and a set of *edges* denoted by E. An edge is an unordered pair of vertices in V. If there is an edge (v_0, v_1) between vertices v_0 and v_1, then v_0 and v_1 are *adjacent vertices.* If two edges are incident to the same vertex, then they are *adjacent edges.* An edge with identical ends is a *loop.* If two or more edges are incident with the

[1]This is equivalent to the correctness condition called *linearizability* introduced by Herlihy and Wing [17].

same pair of vertices, then they are *multiple* edges. A graph is *simple* if it does not contain loops or multiple edges. A graph is *complete* if there is an edge between every pair of distinct vertices.

A (simple) *path* from a vertex v_0 to vertex v_k in graph G is a sequence $(v_0, ..., v_k)$ of vertices such that $(v_i, v_{i+1}) \in E$ for $i = 0, ..., k-1$ and $v_0, ..., v_k$ are distinct. A *cycle* is formed by a path $(v_0, ..., v_k)$ and $(v_0, v_k) \in E$. The *length* of a path or a cycle is the number of edges in it. The *distance* between two vertices $v_0, v_1 \in V$ in a graph, denoted $d(v_0, v_1)$, is the length of the shortest path between the v_0 and v_1. For the sake of completeness, $d(v_0, v_0)$ is defined as 0. The *diameter* of a graph, denoted D is,

$$D = max\{d(v_0, v_1) \mid v_0, v_1 \in V\}.$$

A graph is *connected* if there exists a path between every pair of vertices in the graph. The *neighborhood* of a vertex $v \in V$ in a graph, denoted $N(v)$, is the set of nodes adjacent to v. Formally

$$N(v) = \{x \in V | (v, x) \in E\}.$$

The *degree* of a vertex v, denoted $\delta(v)$, is the size of v's neighborhood, $|N(v)|$. The maximum degree of a graph, denoted Δ, is

$$\Delta = max\{\delta(v) | v \in V\}.$$

A graph $G = (V_1, V_2, E)$ is *bipartite* if its vertex set can be partitioned into two sets V_1 and V_2, such that every edge $(x, y) \in E$ is incident to one vertex x in V_1 and another vertex y in V_2.

A *directed graph* (or a *digraph* for short) G consists of a set of vertices V and a set of edges E, where an edge is a ordered pair of vertices $\langle v_i, v_j \rangle$ for $v_i, v_j \in V$. In a

digraph an edge is not only incident on a vertex, but is also *incident out of* a vertex or *incident into* a vertex. An edge, which is incident out of a vertex v_i, is called an *outgoing edge* of v_i. An edge, which is incident into a vertex v_i, is called an *incoming edge* of v_i.

A (simple) *directed path* from a vertex v_0 to vertex v_k in digraph G is a sequence $\langle v_0, ..., v_k \rangle$ of vertices such that $\langle v_i, v_{i+1} \rangle \in$ E for $i = 0, ..., k-1$ and $v_0, ..., v_k$ are distinct. A *directed cycle* is formed by a directed path $\langle v_0, ..., v_k \rangle$ and $\langle v_k, v_0 \rangle \in E$.

Modeling a Distributed System as a Labeled Graph

The graph model we use in this thesis is a labeled graph, where the vertices of the graph represent the components of the system and edges represent a relation among the components.

Each individual component of the system is a state machine, which changes state based on its current state and the state of one or more of its neighbors. The *state transition function* of a component is defined by the program of the state machine. In the graph model of a distributed system, each node is labeled with the current state of the corresponding system component. The global configuration of the graph is the combination of labels of all vertices. For example, in a graph that contains n vertices $\{v_1, ..., v_n\}$ where the set of possible labels for vertex v_i is L_i, the global configuration of such a graph is an n-tuple in $L_1 \times L_2 \times \cdots \times L_n$.

In cases where the relationship between components is dynamic, this model is generalized to allow labels on edges. In this case the global configuration includes the labels on edges as well.

There are two possible ways to model the Dining Philosophers problem as a

graph.

Bipartite Graph Model When modeled by a node-labeled bipartite graph $G = (P, F, E)$, where vertices sets P and F represent a collection of philosophers and forks, respectively. An edge (p, f) is in E, if and only if $p \in P$, $f \in F$ and philosopher p uses fork f.

Every philosopher has a dynamic label called state. The state of a philosopher $p \in P$, denoted by *state*(p), is in {thinking, hungry, eating}. The only state transitions are thinking $\rightarrow$ hungry $\rightarrow$ eating $\rightarrow$ thinking. Transitions from thinking to hungry and eating to thinking are spontaneous.

The global configuration of the graph is required to satisfy the following exclusion property all the time:

- Exclusion: In any configuration, $\forall p \in P$ if *state*(p)=eating, then $\nexists q \neq p \in P$ such that *state*(q)=eating and $(p, f) \in E$ and $(q, f) \in E$ for some $f \in F$.

Under the assumption that any philosopher with state eating will change her state to thinking later on, the global configuration of the graph is required to satisfy one of the following progress properties:

- Deadlock-Freedom: In any configuration, if *state*(p)=hungry, for $p \in P$, then there exists a subsequent configuration, in which *state*(q)=eating, for $q \in P$.

- Lockout-Freedom: In any configuration, if *state*(p)=hungry, for $p \in P$, then there exists a subsequent configuration, in which *state*(p)=eating.

General Graph Model When a fork is shared by k philosophers[2], where $k \geq 2$, we can use $\frac{k(k-1)}{2}$ virtual forks shared between every pair of these philosophers to represent the real fork. To acquire the real fork a philosopher has to obtain the $k-1$ virtual forks incident to her. If a fork is shared by exactly 2 philosophers, then the real fork is the same as the virtual fork. To model this with a graph, consider philosophers as nodes, and make a clique among a set of nodes if the corresponding philosophers share the same fork. The edges denote virtual forks. It is easy to see that every virtual fork is shared by exactly two philosophers. Therefore any Dining Philosophers problem can be characterized by a node-labeled general graph $G = (P, VF)$, where the vertex set P represents a collection of philosophers and the edge set VF represents a collection of virtual forks.

In this graph model every philosopher has the same set of states and the same transition function as in the Bipartite Graph Model. The progress requirement (deadlock-freedom or lockout-freedom) remains unchanged. The definition of the exclusion changes as follows:

- Exclusion: In any configuration, if there exists an edge between p and q, then $\neg(state(p)\text{=eating} \wedge state(q)\text{=eating})$

2.2 Dining Philosophers Problem

The Dining Philosophers problem is to implement the Dining Philosophers system such that no two philosophers eat simultaneously if they use a common fork and one

[2]We are not interested in the case where $k = 1$, which implies the fork is used by only one philosopher, since no conflict needs to be solved in this case. Therefore we exclude such forks in our model.

of the progress requirements is satisfied.

The Dining Philosophers system consists a set of philosophers operating on a set of forks. Both components are defined in Section 2.1.1. We also gave the graph theoretical description of the Dining Philosophers problem in Section 2.1.2. In the object oriented specification, the validity condition only captures the exclusion property, which guarantees that no two philosophers can use the same fork simultaneously. Normally a progress property is also required under the assumption that any philosopher's $\langle \text{eat} \rangle$ operation always terminates. Deadlock-freedom is guaranteed, if for any $\langle \text{grab}(F(i)) \rangle$ operation by philosopher $p_i \in \mathcal{P}$, there exists a subsequent operation $\langle \text{eat} \rangle$ by a philosopher $p_j \in \mathcal{P}$. Similarly, lockout-freedom is guaranteed, if for any $\langle \text{grab}(F(i)) \rangle$ operation by philosopher $p_i \in \mathcal{P}$, there exists a subsequent operation $\langle \text{eat} \rangle$ by the same philosopher p_i.

In the rest of this section, we present some typical variants of the Dining Philosophers problem based on different assumptions on system models. Then we give some previous research results for each variant.

2.2.1 Variants of Dining Philosophers problem

The Dining Philosophers problem was first presented by Dijkstra [9] in 1971. His classic problem consists of 5 philosophers sitting around a table, with a fork placed between each neighboring philosopher. The problem has been solved for several extensions that specify how many forks each philosopher requires and how the forks are shared. There are three natural generalizations:

Restricted - Sharing Dining Philosophers System is a Dining Philosophers System, where $\forall f_i \in \mathcal{F}$ there exist exactly two distinct philosophers p_j and p_l,

such that $f_i \in F(j)$ and $f_i \in F(l)$. This corresponds to a system that permits each philosopher to use several forks, but each fork is shared by exactly two philosophers. It can also be modeled by an arbitrary connected graph with vertices as philosophers and edges as forks.

Restricted - Forks Dining Philosophers System is a Dining Philosophers System where $\forall p_i \in \mathcal{P}$, $|F(i)| = 2$. This system permits each fork to be shared by several philosophers, but each philosopher use exactly two forks. It can also be modeled by an arbitrary connected graph with edges as philosophers and vertices as forks.

General Dining Philosophers System is the most general version of the Dining Philosophers System, where no restriction is applied to the number of forks used by each philosopher and number of philosophers that can share a single fork. This can be modeled by an arbitrary bipartite graph with philosophers and forks represented by nodes, and edges as sharing relations.

As shown in the previous section, no matter how the forks are shared, the problem can always be modeled by a connected general graph with edges as virtual forks, where each virtual fork is shared by only two philosophers. Therefore both the Restricted - Forks Dining Philosophers problem and the General Dining Philosophers problem can be reduced to Restricted - Sharing Dining Philosophers problem.

2.2.2 Previous Work on the Dining Philosophers Problem

Lehmann and Rabin's Work [21]

The specification assumed by Lehmann and Rabin is a ring, where vertices are considered as philosophers and edges as forks or vice versa. Lehmann and Rabin generalized Dijkstra's model [9] to a ring with n nodes. The implementation required is in shared memory systems, where adjacent philosophers have access to shared objects representing forks. The authors assume that neighboring philosophers never access a shared object exactly at the same time. Furthermore, they also assume that a philosopher may check that a fork is free and pick it up in one atomic step without being interrupted by a neighbor. Philosophers also have a consistent local orientation on their forks (left or right). As in all variants, no philosopher eats forever. Lehmann and Rabin implemented the fork objects with read/write variables in fully distributed and completely symmetric (see page 4) systems. They presented two Las Vegas randomized algorithms. One is deadlock-free and the other is lockout-free.

Algorithm Description: In the deadlock-free solution, they implemented $\langle \text{grab}(\cdot) \rangle$ by letting a hungry philosopher choose a fork (left or right) uniformly and randomly, and then wait until she manages to pick it up. After that she checks the other fork, if it is free, she acquires it and starts $\langle \text{eat} \rangle$. Otherwise she drops the fork that she is holding, and starts again by choosing a random fork. After eating, a philosopher executes $\langle \text{release}(\cdot) \rangle$ by dropping both forks one at a time in an arbitrary order.

The scheduler can arrange moves among philosophers, so that some philosopher will keep trying to eat but never succeed, because a neighbor grabs her second fork just before she is about to pick it up. For example, let philosopher r and s be the

neighbors of philosopher p and f_{rp} and f_{sp} be the forks that p shares with r and s respectively. Suppose p is hungry, randomly chooses f_{rp} as her first choice, and picks it up successfully. Before p is about to get f_{sp}, s becomes hungry, and picks up f_{sp}. Since philosopher p failed to acquire her second fork, then she drops f_{rp} and tries again. The scheduler can repeat this strategy indefinitely. Therefore p will never get both forks and eat.

To avoid such starvation, the authors introduced a *Courteous Philosopher's Algorithm* that provides lockout-freedom. The algorithm only modifies how a philosopher acquires her first fork. A trying philosopher randomly chooses a fork. When the fork is free she will not pick it up until she has higher *priority* than her neighbor sharing the fork. A philosopher has higher priority on a fork if one of the following conditions is true, otherwise she has lower priority on it.

1. The neighbor sharing the fork with her is not trying.
2. Nobody has used the fork before.
3. The last use of the fork was by her neighbor.

Thus while a philosopher p is continuously trying, each of her neighbors can get the fork shared with her once. After that when either becomes hungry again, with probability a half it will choose the fork she shares with p as her first choice. In that case none of the above conditions is true, thus she cannot preempt p by holding the fork shared with p again. Therefore with probability 1 every trying philosopher will eat and nobody starves.

Complexity: There is no complexity analysis in this paper.

In shared memory settings, the most interesting criterion to measure the per-

formance of solutions to the Dining Philosophers problem is *waiting time*, which indicates how long it takes a hungry philosopher to eat. Since philosophers in the first algorithm may starve, there is no upper bound on the waiting time of this algorithm. In the Courteous Philosopher's Algorithm, it is possible that all philosophers except one have higher priority on their left forks and lower priority on their right forks and everyone chooses her right fork as the first choice. Thus all philosophers except one line up in the waiting relations, allowing only one philosopher to eat at a time. Therefore the waiting time of the worst case in this algorithm is $\Theta(n)$, where n is the total number of philosophers on the ring.

Further Discussion: Lehmann and Rabin pointed out and proved that there is no deterministic, deadlock-free solution to the Dining Philosophers problem in fully distributed and completely symmetric systems.

Their algorithm does not explicitly indicate that any object stronger than atomic read/write variables are used. However, a philosopher must be able to check that a fork is free and pick it up in one atomic step. Otherwise some neighbor may pick up the fork in between, and eventually both the neighboring philosophers hold the shared fork at the same time. This required composite action can be achieved by implementing each fork as a test-and-set object. The test-and-set operation can simulate the check and pick up operations provided by the atomic variables. The reset operation implements the release operation of the atomic variables.

Chandy and Misra's Work [6]

The specification assumed by Chandy and Misra is the Restricted - Sharing Dining Philosophers problem, in which a philosopher needs several forks but each fork is

shared by exactly two philosophers. They assumed the underlying communication graph is a static, finite, simple, undirected and connected graph. The implementation required is in message passing systems, where philosophers communicate with each other by exchanging messages. Chandy and Misra implemented the fork objects with tokens in fully distributed systems. They devised a deterministic algorithm that provides lockout-freedom.

Algorithm Description: For each pair of neighbors, there exists two tokens, *fork* and *request-token*, which normally reside at different philosophers. They implemented $\langle \text{grab}(\cdot) \rangle$ by letting a hungry philosopher send the request-tokens corresponding to all her missing forks. To ensure that every hungry philosopher eats eventually, priority on a fork is maintained among all philosophers who use it. After sending a request-token, either 1) the philosopher receives the fork, or 2) her neighbor is hungry and has higher priority on the fork, or 3) her neighbor is eating. After eating, a philosopher becomes lower priority on all the forks shared with her neighbors. Thus a hungry philosopher will either get all her forks or her priority on some fork rises after the neighbor using that fork ate. The hungry philosopher with higher priority on all forks will acquire all of them and start eating. As will be discussed shortly, given an appropriate initial configuration, this situation always happens.

To implement the priority idea, each fork is associated with a state, *clean* or *dirty*. If a philosopher holds a dirty fork, then she has low priority on that fork. Otherwise she has high priority on it. An eating philosopher makes all her forks dirty. Forks only get clean when they are transferred from one philosopher to another. Upon receiving a request-token, a philosopher will send the fork to her neighbor if the

fork is dirty and she is not eating. A hungry philosopher eats if she holds all her shared forks, and each fork is either clean, or she does not hold the corresponding request-token for it.

Since each fork is represented by a token, and it has to reside in one of the two philosophers that use it, the way to implement the $\langle \text{release}(\cdot) \rangle$ operation is different from Lehmann and Rabin's. Instead of setting the fork free, the philosopher still keeps the fork token, but makes it dirty. Therefore upon receiving a request-token from a neighbor, she will release the fork to the neighbor if she is not eating.

The whole system works properly, if initially the high priority relation does not form a cycle. Therefore a special initial configuration needs to be set up as follows. All philosophers are thinking, and all forks are dirty. Philosophers are labeled with indices. For each pair of neighbors, the one with lower order holds the shared fork, and the other one holds the corresponding request-token. Any way to label philosophers such that the high priority relation does not forms a cycle is sufficient in the initial setting. For convenience, the authors suggested a total order among the philosophers.

Complexity: There is no complexity analysis in this paper.

Since the solution is developed for message passing systems, we are interested primarily in the *message complexity*, which characterizes the number of messages sent and received by a hungry philosopher before she can eat. Because a hungry philosopher exchanges a constant number of messages between each of her neighbor before eating, the message complexity is bounded above by $O(\delta)$, where δ is maximum number of forks that a philosopher uses. For time complexity, the worst case,

similar to the one in Lehmann and Rabin's second algorithm, can also happen here. Therefore the waiting time of the worst case is $\Theta(n)$, where n is the total number of philosophers.

Further Discussion: Fairness is provided by maintaining priorities on forks among all philosophers. This is similar to the technique used in the second algorithm of Lehmann and Rabin.

It is important to prevent the high priority relation from forming a cycle, which can cause deadlock in the system. For example, consider a ring, where initially each philosopher has one dirty fork. If all of them are hungry, then each can get the other fork from her neighbor because that fork is dirty, implying that her neighbor's priority on it is low. However she has to give up the fork she holds for the same reason. Eventually every philosopher holds a clean fork and will not give it up. Therefore, nobody can acquire both forks and eat.

For convenience the authors proposed a total order among all philosophers in the initial state, , which is equivalent to every philosopher having a distinct identifier. In this case the system is not completely symmetric. In fact, assigning locally distinct numbers to philosophers is sufficient. Consider any existing cycle in the communication graph. Because the numbers of philosophers in a cycle cannot keep increasing, there must exist a philosopher that has an identifier less than both neighbors on the cycle. That philosopher will have both dirty forks shared with neighbors on the cycle, which implies this philosopher does not have high priority on either forks. Thus high priority cannot form cycles.

To prove that progress is achieved, construct the directed graph $G = (\hat{P}, A)$,

where $\hat{P}$ is the set of all philosophers who are hungry, and directed edge $\langle p, q \rangle \in A$ if and only if q has higher priority on the forks shared with p. G is acyclic, hence contains at least one sink (that is node with no outgoing edges), say r. Philosopher r has high priority on all forks shared with her hungry neighbors. Thus r will either acquire all her forks or another neighbor of r becomes hungry, joins G, and stops r because she has higher priority than r. Since there are a finite number of philosophers in the system, eventually there are no more philosophers that become hungry without exiting the graph. But any philosopher that exits the graph has just executed $\langle \text{eat} \rangle$.

Herescu and Palamidessi's Work [16, page 82]

The specification assumed by Herescu and Palamidessi is the Restricted - Forks Dining Philosophers problem, in which every philosopher uses exactly two forks but a fork can be shared by several philosophers. They assumed the underlying communication graph is a static, finite, undirected and connected graph. The graph allows multiple edges. The implementation required is in shared memory systems. Philosophers have a consistent local orientation on their forks (left or right). Herescu and Palamidessi implemented the fork objects with test-and-set objects in fully distributed and completely symmetric systems. They presented two Las Vegas randomized algorithms. One is deadlock-free and the other intends to provide lockout-freedom but fails to do so.

Algorithm Description: The first algorithm provides deadlock-freedom. The implementation of $\langle \text{grab}(\cdot) \rangle$ and $\langle \text{release}(\cdot) \rangle$ is similar with Lehmann and Rabin's first one. However acquiring a fork is achieved by a test-and-set operation, and releasing a fork is achieved by a reset operation. Another difference is that every

fork has a variable, which contains an integer in range $[0, m]$, where m is no less than the total number of forks in the system. Instead of randomly choosing a fork from left or right as the first choice, a philosophers always tries to pick up her fork with bigger value first. If both forks have the same value, the philosopher will choose the right one. To ensure the values of both forks are eventually different, a philosopher changes the value of the fork she is holding, if it happens to have the same value as her other fork.

The authors also gave another randomized algorithm that was supposed to provide lockout-freedom, but it failed to do so. We discuss the problem and fix the algorithm and present it in the "Further Discussion" paragraph.

Complexity: The authors did not give complexity analysis in their paper. We will give the upper bound on waiting time of the repaired lockout-free solution.

Further Discussion: Herescu and Palamidessi pointed out that both algorithms of Lehmann and Rabin failed in a more general case, where each fork is shared by an arbitrary number of philosophers. Both of their algorithms are variants of Lehmann and Rabin's. They required a bound on the size of the network (number of nodes), in order to bound the size of a integer set from which every fork is assigned a label. Since the values of forks are used to break symmetry, it is enough for every fork to have a locally distinct value. Therefore only local information, such as maximum degree, needs to be known, and the bound could be made independent of the size of the whole system.

Since in Herescu and Palamidessi's model, forks are shared by an arbitrary number of philosophers (possibly more than two), the priority on a fork is among several

philosophers, and there may also be several philosophers that try to get the same fork at a time. Now 'low' and 'high' are not enough to denote philosophers' priorities on forks, and philosophers need to monitor the actions of more neighbors. Therefore in their second algorithm, each fork maintains two more data structures, one is a set of philosophers who are trying to get the fork, the other one is a list, which keeps track, in time order, of the philosophers who have used the fork. In the execution of $\langle \text{grab}(\cdot) \rangle$, every hungry philosopher chooses the fork with bigger value as her first choice. She waits until the fork is free and one of the priority checks is true:

1. The philosopher has not eaten before.

2. All the trying neighbors used the fork after her last meal.

This algorithm [16, 25] fails to provide fairness even for the simple topology of a line. For example, consider the system $G = (F, P)$, where vertex set, $F = \{f_1, f_2, f_3\}$, represents set of forks. Edge set, $P = \{p_1, p_2\}$, represents set of philosophers. And $F(1) = \{f_1, f_2\}$, $F(2) = \{f_2, f_3\}$. That is, philosopher p_1 uses fork f_1 and f_2, philosopher p_2 uses fork f_2 and f_3. Let fork f_2 have a smaller value than both f_1 and f_3, so it will always be the second choice of each philosopher. Thus either of them holding her first choice can pick up f_2 as long as it is free. Then p_1 can stop p_2 from eating by grabbing f_2 whenever p_2 is about to pick it up. For example, suppose p_1 and p_2 are both hungry and holding their first choices. Let p_1 get f_2 and eat later on. After eating, p_1 releases both f_1 and f_2, becomes hungry right away, and acquires f_1. This configuration is the same with the initial state. The scheduler can always let p_1 picks up f_2, and make the above situation happens forever. Philosopher p_2 will never get f_2, and therefore never eat.

To fix this problem, every philosopher has to do the priority checks on the second

fork as well. Each philosopher still only waits for her first choice, which means if the second choice is not free or both priority checks fail, then the philosopher will drop her first fork and start from scratch. Now in the above example, if p_1 ate and became hungry again,while p_2 is continuously hungry, both p_1's priority checks on f_2 are false. Thus p_1 cannot pick it up again before p_2 eats.

The worst case that could happen in this fixed algorithm is similar to that in Lehmann and Rabin's second algorithm, where the whole system is a ring and all philosophers except one line up in a waiting relation. Therefore the waiting time of the worst case in this algorithm is $\Theta(n)$, where n is the size of the network.

Nancy Lynch's Work [23]

The specification assumed by Lynch is a Static Resource Allocation problem. She first gave a formal definition [23, page 256] for the problem as follows:

> A *resource problem* P is a quadruple $(R(P), U(P), \mathcal{R}(P), \mathcal{U}(P))$, where $R(P)$ and $U(P)$ are disjoint, possibly infinite sets (of resources and users, respectively), where $\mathcal{R}(P)$ is a mapping from $U(P)$ to the set of finite nonempty subsets of $R(P)$ (indicating the resources required by each user), where $\mathcal{U}(P)$ is a mapping from $R(P)$ to finite nonempty subsets of $U(P)$ (indicating the users for each resource), and where $r \in \mathcal{R}(P)(u)$ if and only if $u \in \mathcal{U}(P)(r)$.

Then she modeled the problem as a graph, in which the vertices represent resources. Put an edge between two nodes, if the corresponding resources have a common user. Both users and resources are processes. The implementation required

is in systems with a reliable communication setting (shared registers or messages exchanges system) through which processes exchange information.

As indicated at the beginning of this chapter, the Dining Philosophers problem is a fundamental version of the Static Resource Allocation problem. Lynch's model can be directly mapped onto a General Dining Philosophers problem[3]. In this section, we present Lynch's algorithm in the form of the Dining Philosophers setting by considering users as philosophers and resources as forks.

Lynch assumed a coloring algorithm was applied to the system initially, so that if two forks are used by a common philosopher, then they have distinct colors. There is a total order among all colors. Lynch presented a deterministic algorithm that provided lockout-freedom in a fully distributed system with the above assumption.

Algorithm Description: Lynch implemented the $\langle \text{grab}(\cdot) \rangle$ operation as follows. A hungry philosopher puts herself in the queue of her fork that has the smallest color. She remains in the queue until she becomes the head of it, which implies she can get the corresponding fork. That fork will put the philosopher into the queue of the next fork that she uses in the coloring order if it exists. Eventually when the philosopher has become the head of the queues of all her forks, she has obtained all her forks and starts $\langle \text{eat} \rangle$.

In the execution of $\langle \text{release}(\cdot) \rangle$, a philosopher who has eaten leaves the front position of the queues of each of her forks. This permits some hungry neighbors to

[3]Lynch assumes that the resources also execute programs. This is slightly different from our object oriented model of the Dining Philosophers problem, in which only philosophers are active processes. However we will show in the 'Algorithm Description' that this does not violate our model by demonstrating that the operations by users together with the operations by resources implement $\langle \text{grab}(\cdot) \rangle$ and $\langle \text{release}(\cdot) \rangle$ on the fork object.

grab the fork shared with her.

Complexity: Assume that σ is an upper bound on process step time, ν is an upper bound on the time for a user to return a granted resource, γ is an upper bound on message collection time, and δ is an upper bound on message delivery time. Let contention(P) denote the maximum number of users of any resource, and let $|c|$ denote the number of colors. Then the time from when a user starts to require its resources until the user gets all of them is bounded above by

$$(\text{contention}(P)^{|c|} - 1)\nu + O(|c|\text{contention}(P)^{|c|}(\sigma + \gamma + \delta))$$

Further Discussion: The performance of the algorithm depends on a good coloring algorithm. The upper bound of waiting time does not depend on the size of the whole system, but a function of local parameters only (such as the maximum number of users of each resource and maximum number of resources for each user).

Choy and Singh's Work [8]

The specification assumed by Choy and Singh is a special Static Resource Allocation problem, where each resources is shared by only two processes. They claimed that their solution to this special case can easily be extended to the general case where a resource is shared by several processes. They also pointed out that the Dining Philosophers problem is a graph-theoretic formalization of the Resource Allocation problem. They only present lockout-free solutions to the Dining Philosophers problem.

They used the arbitrary graph model in [22], where vertices represent philosophers and each edge represents a fork shared by the end vertices. This setting is the same

as the Restricted - Sharing Dining Philosophers problem. They assumed that a node coloring mechanism had been applied to the system initially, so that each node has a locally distinct label in the range 0 to δ, where δ is the maximum number of forks used by a philosopher[4].

Choy and Singh implemented the fork objects with tokens in fully distributed message passing systems. Their deterministic algorithms provide robustness when stop failures happen.

Algorithm Description: Similarly to other solutions in the message passing systems, every pair of neighboring philosophers carries a fork token and a request-token. To collect missing forks, a philosopher simply sends the corresponding request-tokens to its neighbors. The policy here to resolve conflicts is to give higher priority to the process with smaller label. To avoid the situation where a process with smaller label always preempts its neighbors by holding forks shared with them, they introduced a mechanism called a *doorway*. Only the processes inside the doorway are allowed to gather forks, and the process will exit from the doorway after eating. They divided the execution of $\langle \text{grab}(\cdot) \rangle$ into two major actions: a hungry process first tries to enter the doorway, then starts to collect missing forks. A philosopher inside the doorway can acquire the fork if either 1) the corresponding neighbor has bigger label and is not eating, or 2) the corresponding neighbor is outside the doorway. Like other solutions in the message passing settings, a philosopher still keeps all the fork tokens after eating, and just looses priority on all of them. Therefore in operation $\langle \text{release}(\cdot) \rangle$ a philosopher exits from the doorway. A process cannot enter the doorway twice if a

[4]There exists a greedy node-coloring algorithm that color nodes with $\Delta + 1$ colors, where Δ is the degree of the graph.

neighbor continuously stays inside the doorway. Thus a process cannot preempt a neighbor with higher label more than once before that neighbor eats.

The authors first described an *asynchronous doorway.* To enter the asynchronous doorway, a process p checks the states of its neighbors one by one. For any neighbor inside the doorway, p waits until the first time that neighbor exits. After observing a neighbor is outside the doorway, p will not check it anymore. When p finishes checking all neighbors, it enters the doorway. Once p is inside the doorway, its neighbors can be in the doorway at most once, because that neighbor will notice that p is in the doorway, and will wait until p exits. Consider the subgraph, G, of the communication graph of the system that is induced by the nodes that are inside the doorway. At some point, G will not grow anymore, and after that, the process with local minimum label will successfully grab all its forks, and leave the subgraph. Eventually p will either become a local minimum or all neighbors of p will leave. Thus p will eat. Therefore every process inside the doorway will eventually eat and come out, implying that every hungry process outside the doorway has a chance to enter. However, in an algorithm using the asynchronous doorway, the following scenario is possible. A process with label δ enters the doorway. While it is collecting forks, all its neighbors with label $\delta - 1$ enter the doorway one at a time to preempt it. This can happen recursively for every such neighbor. Therefore the algorithm has exponential response time.

To improve the response time, the authors introduced another doorway called a *synchronous doorway.* To enter the synchronous doorway, a process waits until all its neighbors are simultaneously outside the doorway. The synchronous doorway separates the processes into two groups, early-arriving and late-arriving. The early-

arriving processes enter the doorway, and cannot be interrupted by the late-arriving processes before using the forks. However a process waiting outside the doorway could be prevented from entering the doorway forever if its neighbors take turns entering the doorway. For example, a process p has two neighbors q and r. Initially p and q are outside the doorway and both are hungry, and r is inside the doorway. Process p and q have to wait until r finishes eating. When r exits from the doorway q enters immediately. Suppose r becomes hungry again. Now p and r have to wait for q to finish, which is symmetric to the initial state. The scheduler can make this happen forever, and process p never enters the synchronous doorway. Therefore process p starves.

To prevent this case, an asynchronous doorway is placed in front of the synchronous doorway. A hungry process will try to enter the asynchronous doorway first and then the synchronous doorway. A process inside the synchronous doorway is considered to be outside the asynchronous doorway. Only processes inside the synchronous doorway can collect forks. The process exiting from the synchronous doorway will be blocked outside the asynchronous doorway until all its neighbors exit the asynchronous doorway, meaning that they enter the synchronous doorway. In the above example, when r comes out of the synchronous doorway, it will stay outside the asynchronous doorway until p enters the synchronous doorway. Therefore r will not stop p from entering the synchronous doorway again.

The combination of the asynchronous doorway and synchronous doorway is called a *double doorway*. The double doorway attains the advantages from both the asynchronous doorway and synchronous doorway. It prevents starvation and provides good response time. The authors gave two optimizations to remove unnecessary

waiting at the doorways.

1) Since the synchronous doorway is to prevent a process from preempting a neighbor with higher label, a process needs only to wait for all the neighbors with higher labels to exit.

2) The asynchronous doorway is to prevent a process from blocking a neighbor from entering the synchronous doorway by continuously entering it. And a process only waits for neighbors with higher labels before entering the synchronous doorway. Therefore a process that tries to enter the asynchronous doorway only waits for all the neighbors with lower label to exit.

At the end of the paper, the author introduced *failure locality* to measure the robustness of an algorithm in the presence of stop failures. First define a *waiting chain* to be a directed path $\langle p_1, ...p_k \rangle$, where p_i and p_{i+1} are neighboring philosophers, and p_i is waiting for a fork held by p_{i+1} for $i = \{1, ..., k-1\}$. The length of a waiting chain is the length of the corresponding directed path. If a process p stops executing its code, all the processes along any waiting chain and behind p will not make progress anymore. Therefore the failure locality is measured by the length of waiting chains. To improve the failure locality, the authors limited the length of any possible waiting chains by using an improved policy, called *fault-tolerant fork collection*, to solve the conflict. Define a fork to be a *high fork* to a process, if the process has higher priority on it than its neighbor, otherwise it is a *low fork*. A process always tries to get its missing low forks first, and it starts requesting its missing high forks only when it holds all its low forks. While a process is waiting for a low fork, it releases any high fork that it has upon request. Thus a process waiting for low forks will not stop any neighbors from collecting forks. A process that is holding all its forks is ready to eat,

and will not give up any forks before eating. It is possible to show that the length of any waiting chain is at most 2 by using the 'fault-tolerant fork collection' scheme. Consider a process p who sent a request to its neighbor q for fork f_{pq}, and q does not release the fork. There are three possible cases:

1) process q has failed, or

2) process q is eating or ready to eat, or

3) process q has higher priority on fork f_{pq}, has collected all its low forks, and is waiting for a high fork from another neighbor r.

In case 1) and 2), q is not waiting for anyone, so one end of the waiting chain is q. If fork f_{pq} is a low fork for process p, then the waiting chain ends at process p and has length 1. Otherwise p must have all its low forks and is collecting high forks, which implies that p might stop a neighbor s from eating by holding one of s's low forks. In this case the waiting chain ends at s and has length 1. In case 3) process p is waiting for a low fork, therefore the waiting chain stops growing from p. Process r can hold the low fork shared with q, only if it is failed or eating or ready to eat, which implies the other end of the waiting chain is s. Thus the waiting chain has length 2.

Clearly, by exploiting the 'fault-tolerant fork collection' scheme, the length of any possible waiting chain among the processes in the collecting forks stage is limited to 2. Each doorway added to the algorithm increases the waiting chain by 1. Thus, in the algorithm using double doorway, the length of any waiting chain among all hungry processes is no more than 4.

Complexity: Let Δ be maximum number of forks used by a philosopher. The response time in the worst case of the algorithm using the asynchronous doorway and the double doorway is $\Theta(\Delta^{\Delta+2})$ and $\Theta(\Delta^2)$, respectively. The failure locality in the worst case of the algorithm using the original fork collection policy is $\Theta(\Delta)$. In the algorithms using the improved policy, the failure locality reduces to a constant. The one using the asynchronous doorway is 3, and the one using the double doorway is 4.

Further Discussion: The coloring algorithm the authors used is not a distributed algorithm, so we assume the coloring is part of the initial state of the system. Therefore the algorithm requires locally distinct identifier rather than working for a completely symmetric system.

2.3 Drinking Philosophers Problem

The Drinking Philosophers Problem is a generalization of the Dining Philosophers Problem. An informal and imprecise but common description of the Drinking Philosophers problem in the literature is given as follows. The Drinking Philosophers problem consists of a set of philosophers sharing a set of beverages. Each philosopher has an unchanged neighborhood. Philosophers indefinitely cycle through three states: tranquil, thirsty, and drinking. When a philosopher is tranquil, she does not interact with others. From time to time, she may get thirsty and want to drink. To drink she acquires a set of dynamically determined beverages. A philosopher may need a different set of beverages each time she becomes thirsty. Every drinking philosopher will eventually finish and release all the beverages. The problem is to devise a sys-

tem, in which no two neighboring philosophers drink simultaneously if they need a common beverage, and every thirsty philosopher will drink eventually.

In the above description, the conflicts between philosophers are not clearly characterized. Also, it allows two non-neighboring philosophers to drink a common beverage simultaneously, but it does not mention how they do so. To clarify these ambiguities, we model the possible conflicts by the neighboring relations among philosophers, and we make multiple copies of a beverage if, in the original description, it is shared by several non-neighboring philosophers.

In this section, we first use the object oriented model and the graph model to more precisely specify the Drinking Philosophers problem. Then we briefly compare the Drinking Philosophers problem with the Dining Philosophers problem. After that we present Chandy and Misra's result, which uses the solution to the Dining Philosophers problem as a subroutine, and Welch and Lynch's modular interpretation of Chandy and Misra's work. At the end some solutions that do not use the algorithm of the Dining Philosophers problem are presented.

2.3.1 Object Oriented Specification

A Drinking Philosophers system $(\mathcal{P}, \mathcal{B})$ consists of a set of philosophers (processes) $\mathcal{P} = \{p_1, p_2, ..., p_n\}$, and a set of beverages (objects) $\mathcal{B} = \{b_1, b_2, ..., b_k\}$. Associated with each $p_i \in \mathcal{P}$ is an nonempty and static set of beverages $B(i) \subseteq \mathcal{B}$.

The set of beverages $\mathcal{B}$ supports two operations: $\langle \text{grab}(B) \rangle$ and $\langle \text{release}(B) \rangle$, where $B \subseteq \mathcal{B}$.

Let $S = o_1, o_2, ...$ be any sequence of operations on $\mathcal{B}$. To give the validity condition on S, we define another variable $\mathcal{CB}_i$ inductively by:

1) $\mathcal{CB}_0 = \mathcal{B}$

2) $\mathcal{CB}_i = \begin{cases} \mathcal{CB}_{i-1} \cup B_i & \text{if } o_i = \langle \text{release}(B_i) \rangle \\ \mathcal{CB}_{i-1} \backslash B_i & \text{if } o_i = \langle \text{grab}(B_i) \rangle \end{cases}$

Then S is valid provided

a) if $o_i = \langle \text{release}(B) \rangle$ for $B \subseteq \mathcal{B}$, then $B \cap \mathcal{CB}_{i-1} = \emptyset$.

b) if $o_i = \langle \text{grab}(B) \rangle$ for $B \subseteq \mathcal{B}$, then $B \subseteq \mathcal{CB}_{i-1}$.

The program for any philosopher $p_i \in \mathcal{P}$ is:

Do Forever:

$\langle$thirsty$\rangle$:B // where $B \subseteq B(i)$ is the output of an arbitrary local choice

$\langle \text{grab}(B) \rangle$

$\langle$drink$\rangle$

$\langle \text{release}(B) \rangle$

End Do

The $\langle$thirsty$\rangle$:B and $\langle$drink$\rangle$ are arbitrary operations that do not act on $\mathcal{B}$.

The validity condition only captures the exclusion property, which requires that no two philosophers can drink simultaneously if they need a common beverage. Normally lockout-freedom is also required under the assumption that for any philosopher's $\langle$drink$\rangle$, there exists a subsequent $\langle \text{release}(\cdot) \rangle$ by the same philosopher. Lockout-freedom is a property that for any $\langle \text{grab}(B) \rangle$ operation by philosopher $p_i \in \mathcal{P}$, there exists a subsequent operation $\langle$drink$\rangle$ by the same philosopher.

2.3.2 Graph Model

The Drinking Philosophers problem can be modeled by a labeled bipartite graph $G = (\mathcal{P}, \mathcal{B}, E)$, where vertices sets, $\mathcal{P}$ and $\mathcal{B}$, represent a collection of philosophers and beverages, respectively. An edge (p, b) is in E, if and only if $p \in \mathcal{P}$, $b \in \mathcal{B}$ and philosopher p may drink beverage b.

Every philosopher has a dynamic label called state. The state of a philosopher $p \in \mathcal{P}$, denoted as *state*(p), is in {tranquil, thirsty, drinking}. The only state transitions for philosophers are tranquil $\rightarrow$ thirsty $\rightarrow$ drinking $\rightarrow$ tranquil. Transitions from tranquil to thirsty and drinking to tranquil are spontaneous. Every edge is also associated with a state, *red* or *green*. The only state transitions for edges are red $\rightarrow$ green $\rightarrow$ red. An edge $(p, b) \in E$ is green implies that either *state*(p)= thirsty and p wants to drink b or *state*(p)= drinking and p is drinking from b. Otherwise it is red.

The global configuration always satisfies the following property:

- Local Consistency: In any configuration, if *state*(p)=tranquil, then $\forall e \in E$, such that $e = (p, b)$ where $b \in \mathcal{B}$, e is red. Let c be a configuration where *state*(p)=thirsty and $e = (p, b) \in E$ is red (or green). Let $\hat{c}$ be the first subsequent configuration where *state*(p)=tranquil. In all configurations from c to $\hat{c}$, e remains in red (or green, respectively).

- Exclusion: In any configuration, if *state*(p)=drinking, then $\nexists q \neq p \in \mathcal{P}$ such that *state*(q)=drinking and edges (p, b) and (q, b) are both green for a beverage $b \in \mathcal{B}$.

Under the assumption that any philosopher with state drinking will change her state to tranquil later on, the global configuration of the graph is required to satisfy

the strong progress property:

- Lockout-Freedom: In any configuration, if *state*(p)=thirsty, then there exists a subsequent configuration, in which *state*(p)=drinking.

2.3.3 Drinking Philosophers Problem vs. Dining Philosophers Problem

From the object oriented specification, we can see that the Drinking Philosophers problem is a parameterized Dining Philosophers problem. In the Dining Philosophers problem, a hungry philosopher acquires all the forks adjacent to her, while in the Drinking Philosophers problem, a thirsty philosopher acquires a subset of the beverages adjacent to her. Thus in the Drinking Philosophers problem, all actions except ⟨drink⟩ are associated with a variable B, which indicates the set of beverages the philosopher currently needs.

In the graph model $G = (\mathcal{P}, \mathcal{B}, E)$ of the Drinking Philosophers problem, for any configuration c build a general graph $G_c = (P_c, E_c)$ where $\mathcal{P}_c = \mathcal{P}$. If *state*($p$)=thirsty, *state*($q$)=thirsty in c, and there exists a beverage $b \in \mathcal{B}$ such that both (p, b) and (q, b) are in E and both are green, then place an edge (p, q) in E_c. Now the Drinking Philosophers problem reduces to the Dining Philosophers problem, since philosophers always compete with all their neighbors. However a philosopher in the Drinking Philosophers problem may choose a different set of beverages each time she becomes thirsty, which implies the exclusion relationship between neighboring philosophers changes from time to time. The edges of G_c are not static, but the vertices (philosophers) never disappear or emerge. Therefore the graph model of the Drinking Philosophers problem is the graph model of the Dining Philosophers problem with dynamic edges.

2.3.4 Previous Work on the Drinking Philosophers Problem

Chandy and Misra's Work [6]

In Chandy and Misra's specification, philosophers and their neighboring relations are modeled as a static, finite, undirected and connected graph, in which the vertices represent philosophers and edges represent the neighboring relation between philosophers. They presented a deterministic algorithm in fully distributed systems, where beverage objects are implemented by tokens.

Algorithm Description: The authors pointed out that applying the strategy in the solution to the Dining Philosophers problem directly might result in deadlock. For example, let neighboring philosophers p and q share two beverages b_1 and b_2, where p is drinking b_1 and q is drinking b_2. Suppose they both become tranquil and then become thirsty for both b_1 and b_2. Therefore p will yield on b_1 and q will yield on b_2. Chandy and Misra provided a solution that resolves conflicts in the above symmetric situation by using their Dining Philosophers' solution as a subroutine. Every philosopher runs both the Dining Philosophers' and the Drinking Philosophers' algorithm. The state of every philosopher is a combination of the dining region (thinking, hungry, and eating) and drinking region (tranquil, thirsty, and drinking). However state ⟨hungry, tranquil⟩ and ⟨eating, tranquil⟩ are never reached.

For each pair of neighbors, there exist a token called *bottle* for each shared beverage and a corresponding *request-token*. When a philosopher, say p, gets thirsty, she decides to drink a set of beverages and becomes hungry in the dining region. In the execution of ⟨grab(·)⟩, a philosopher tries to acquire the bottles for all beverages that

she chose. To collect a missing bottle, she sends the corresponding request-token to her neighbor. Upon receiving the request-token, the neighbor, say q, will satisfy the request, if either 1) q does not need the bottle or 2) q is not drinking and p is eating in the dining region. Their Dining Philosophers solution guarantees that every hungry philosopher eats. Therefore p will get all missing beverages from her neighbors and start $\langle$drinking$\rangle$ eventually.

After drinking a philosopher does not need the priority on all beverages given by the eating region of the Dining Philosophers problem. Therefore in $\langle$release$(\cdot)\rangle$ a philosopher exits from the $\langle$eating$\rangle$ operation of the Dining Philosophers solution.

Complexity: The author did not give the complexity analysis. The waiting time of the Drinking Philosophers' solution is bounded by the waiting time of the Dining Philosophers' solution being used, because the additional overhead is small. Chandy and Misra's Drinking Philosophers' solution has waiting time $\Theta(n)$, since they used their own solution to the Dining Philosophers' problem, which has the same complexity.

Further Discussion: Chandy and Misra's algorithm does not work under completely symmetric systems. This is because the Dining Philosophers subroutine used in their algorithm starts from a special initial configuration, where every philosopher has a locally distinct identifier.

In Chandy and Misra's graph model, the vertices represent philosophers and edges represent the neighboring relations rather than the possible conflicts among philosophers. They described an extra set to denote the set of beverages accessible to philosophers. Their exclusion property requires that neighboring philosophers

cannot drink the same beverage, while philosophers not adjacent to each other can do so. This description can be modified to an equivalent one by making k copies for each beverage, if there are k pairs of neighboring philosophers that potentially need it. Then place an edge between these pairs of philosophers for each copy. Now two philosophers are neighbors only if they might acquire the same beverage. This modified model is equivalent to our graph model introduced in Section 2.3.2.

Welch and Lynch's Work [26]

Welch and Lynch exploited the same idea as Chandy and Misra's. One contribution of their paper is to give a modular description of the Drinking Philosophers problem by modeling it within an I/O automaton model. This allows one to plug in an arbitrary lockout-free solution to the Dining Philosophers problem. Their problem description can be converted to our object oriented description and is presented in Appendix A.

Algorithm Description: The authors implemented the $\langle \text{grab}(\cdot) \rangle$ operation as follows. As soon as a philosopher p enters her drinking thirsty region, she enters the dining hungry region in the subroutine. She tries to collect the beverages that she needs but lacks. If her neighbors do not need those beverages, they satisfy p's requests. Otherwise they defer the requests. A philosopher in the dining eating region has higher priority on all her shared beverages than the corresponding neighbors. When p enters the dining eating region, she sends demand messages for the missing beverages. Upon receiving a demand, a philosopher will always give the beverage to the sender (if she is using the beverage, she will wait until she finishes using it.) Eventually p will get all the beverages, and will enter the drinking region by executing

$\langle$drink($\cdot$)$\rangle$. Just as in Chandy and Misra's solution, in the $\langle$release($\cdot$)$\rangle$ operation, a philosopher exits from the dining eating region.

Complexity: The authors showed that in a system of n philosophers the maximum waiting time for a Drinking Philosopher to enter her critical region is dominated by the maximum waiting time for a Dining Philosopher to enter her critical region in the subroutine. By replacing the $O(n)$ time subroutine in Chandy and Misra's solution with an $O(1)$ time subroutine, they claimed that their algorithms have $O(1)$ worse case waiting time.

Further Discussion: By claiming that the Dining Philosophers algorithm of [23] has waiting time $O(1)$, they mean the time complexity is independent of the size of the network. But it does depend on some local parameters, such as the number of forks needed by one philosopher and the number of philosophers sharing one fork. Since they present the problem with an I/O automaton model, the validity conditions of the system are given by making constraints on the transitions of the automaton. This idea is essentially the same as our object oriented description, which gives validity conditions on the computations of the system.

Ginat, Shankar, and Agrawala's Work [13]

The authors present a basic problem description and two generalizations. They gave deterministic solutions to the basic problem and the first generalization in fully distributed message passing systems, where beverage objects are implemented with tokens. In their algorithms, each philosopher is assumed to have a locally distinct label, so the solutions are not completely symmetric.

Basic Drinking Philosophers problem: The authors modeled the problem as an undirected simple graph, where nodes represent philosophers, and neighboring philosophers share only one bottle. A bottle is associated with every edge, implying that every bottle is shared by exactly two philosophers. When a philosopher becomes thirsty, she needs a nonempty subset of the bottles associated with her incident edges. Two philosophers are called neighbors if there is an edge between them.

Algorithm Description: Each philosopher p maintains two nondecreasing integers: s_num_p and max_rec_p. Integer s_num_p refers to p's session number, which is p's last drinking session number if p is tranquil, p's upcoming drinking session if p is thirsty, and p's current drinking session if p is drinking. Integer max_rec_p indicates the biggest session number received by p from her neighbors so far. An extended session number is a combination of a philosopher's session number and her identifier, $(\text{s\_num}_p,\text{id}_p)$. A philosopher p has higher priority on the beverage shared with q, if p has a smaller extended session number than q, where $(\text{s\_num}_p,\text{id}_p) < (\text{s\_num}_q,\text{id}_q)$, if and only if $(\text{s\_num}_p < \text{s\_num}_q)$ or $(\text{s\_num}_p = \text{s\_num}_q$ and $\text{id}_p < \text{id}_q)$. For every beverage shared between two philosophers, there exist a token called bottle and a corresponding request token.

In the execution of $\langle \text{grab}(\cdot) \rangle$, a thirsty philosopher p sets s_num_p to a value bigger than max_rec_p, and sends the corresponding request token, including her session number and her label, to the neighbor who is holding a bottle that she needs. Upon receiving a request token from a neighbor, a philosopher will update her max_rec value and will release the bottle if she does not need it or her neighbor has higher priority on it. If a philosopher is drinking when she receives a request token, she will

release the corresponding bottle after drinking. Once a philosopher releases a bottle, she cannot get it back again before her neighbor uses it. The reason is that when p received the request token from q, p update her max_rec$_p$ to a value at least as big as q's current session number. When p becomes thirsty again, she set her session number bigger than max_rec$_p$, which is bigger than q's current session number. Also a philosopher will not change her session number when she is thirsty. Therefore p does not have higher priority on the bottle shared with q. This ensures a philosopher looses the priority on all beverages after drinking. Thus in $\langle \text{release}(\cdot) \rangle$ a philosopher can still keep the beverage tokens as in some other algorithms for message passing systems.

The algorithm works correctly if the initial configuration satisfies the following properties: 1) Every philosopher has a locally distinct label. 2) Every philosopher p is tranquil and s_num$_p$=max_rec$_p$= 0. 3) For every beverage shared between a pair of neighbors, one has the bottle, the other has the corresponding request token.

The Multiple-instance Extension of the Drinking Philosophers Problem: The authors extended the basic problem to a more general one, where two neighbors share n instances of the bottle, and a philosopher may need any number up to n of these bottles each time.

Algorithm Description: This algorithm is similar to the one solving the basic case. To save messages, a philosopher sends one request token including the total number of instances she needs, her session number and her identifier. Upon receiving a request token, a philosopher updates her max_rec value, and releases the additional number of instances her neighbor needs in a single message if the sum of the numbers

that they each need is less than the total number shared by them or her neighbor has higher priority.

The Multiple-type Extension of the Drinking Philosophers Problem: The authors also mentioned a still more general extension, where two neighboring philosophers may share several types of bottles, and every bottle type has several instances. They did not give the algorithm for this case, but they pointed out that the algorithm is based on the one solving the multiple-instance extension. The difference is that the bottle type is now included in the request token. A philosopher will notify her neighbor of the total number of instances she needs for each type of bottle.

Complexity: They only discussed the message complexity. If a philosopher uses k bottles, then the number of messages exchanged with her neighbors before she starts drinking is at most $2k$.

Further Discussion: The authors solved the Drinking Philosophers problem without using a Dining Philosophers' solution as a subroutine. Therefore, the message complexity is reduced by saving the communications in the subroutine. The drawbacks are that the algorithms use unbounded session numbers, and work only under some initial configurations, where every philosopher has a locally distinct identifier and every fork resides in a special place. In all algorithms, an atomic step contains several single actions. In fact the algorithms still work, if the atomic step is reduced to a single action.

CHAPTER 3

Process Synchronization Problems

In the previous chapter, we introduced the Resource Allocation problem, in which the resources (or forks, beverages) are passively collected by their users (or philosophers). In contrast, the concurrent entities in the Process Synchronization problems are active or partially active, and they can decide whether to interact with others or not. While trying to synchronize with others, they either coordinate among themselves or they are controlled by some distributed or centralized coordinators. Different problems may require a different number of synchronization points.

In the following sections, we present three typical Process Synchronization problems: Committee Coordination, Multiway Rendezvous, and Multiparty Interaction.

3.1 Committee Coordination Problem

The Committee Coordination problem is informally defined to consist of a set of professors organized into committees. Each committee has a static membership. A professor can be available or unavailable. An available professor can attend a meeting

held by any committee of which she is a member. An unavailable professor will not attend any meeting. The restrictions on meetings are as follows:

Synchronization: a committee meeting may be started only if all members of that committee are available.

Exclusion: no two committees may meet simultaneously if they have a common member.

All meetings are assumed to terminate in finite time. The problem is to devise a protocol that satisfies the above restrictions, and also guarantees that if all members of a committee are available, then at least one of them will attend some meeting.

3.1.1 Object Oriented Specification

A Committee Coordination system $(\mathcal{C} \cup \mathcal{R}, \mathcal{P})$ consists of two sets of processes, a collection of committee coordinators (abbreviated as committees) $\mathcal{C} = \{c_1, c_2, ..., c_n\}$ and a collection of professor controllers $\mathcal{R} = \{r_1, r_2, ..., r_k\}$. The set of objects $\mathcal{P} = \{p_1, p_2, ..., p_k\}$ represents a collection of professors. Associated with each $c_i \in \mathcal{C}$ is a nonempty and static set of professors $P(i) \subseteq \mathcal{P}$.

The set of professors $\mathcal{P}$ supports four operations: $\langle \text{grab}(P) \rangle$, $\langle \text{release}(P) \rangle$, for some $P \subseteq \mathcal{P}$, and $\langle \text{become\_available}(p) \rangle$, $\langle \text{go\_on\_holiday}(p) \rangle$, for some $p \in \mathcal{P}$.

Let $S = o_1, o_2, ...$ be any sequence of operations on $\mathcal{P}$. To give the validity condition on S we define another variable $\mathcal{CP}_i$ inductively by:

1) $\mathcal{CP}_0 = P$, where $P \subseteq \mathcal{P}$ is arbitrary.

2) $$\mathcal{CP}_i = \begin{cases} \mathcal{CP}_{i-1} \cup P_i & \text{if } o_i = \langle \text{release}(P_i) \rangle \\ \mathcal{CP}_{i-1} \backslash P_i & \text{if } o_i = \langle \text{grab}(P_i) \rangle \\ \mathcal{CP}_{i-1} \cup \{p\} & \text{if } o_i = \langle \text{become\_available}(p) \rangle \\ \mathcal{CP}_{i-1} \backslash \{p\} & \text{if } o_i = \langle \text{go\_on\_holiday}(p) \rangle \end{cases}$$

Then S is valid provided

1) if $o_i = \langle \text{release}(P) \rangle$ for $P \subseteq \mathcal{P}$, then $P \cap \mathcal{CP}_{i-1} = \emptyset$.

2) if $o_i = \langle \text{grab}(P) \rangle$ for $P \subseteq \mathcal{P}$, then $P \subseteq \mathcal{CP}_{i-1}$.

3) if $o_i = \langle \text{become\_available}(p) \rangle$ for $p \in \mathcal{P}$, then $p \notin \mathcal{CP}_{i-1}$.

4) if $o_i = \langle \text{go\_on\_holiday}(p) \rangle$ for $p \in \mathcal{P}$, then $p \in \mathcal{CP}_{i-1}$.

The program for any committee $c_i \in \mathcal{P}$ is:

Do Forever:

$\langle \text{grab}(P(i)) \rangle$

$\langle \text{meet} \rangle$

$\langle \text{release}(P(i)) \rangle$

$\langle \text{adjourn} \rangle$

End Do

The $\langle \text{adjourn} \rangle$ and $\langle \text{meet} \rangle$ are arbitrary operations that do not act on $\mathcal{P}$.

The program for any professor controller $r_j \in \mathcal{R}$ is:

Do Forever:

$\langle$become_available(p_j)$\rangle$

$\langle$go_on_holiday(p_j)$\rangle$

End Do

The validity condition captures: 1) the exclusion property, which guarantees that no two committees can meet simultaneously if they have a common member, and 2) the synchronization property, which guarantees that a committee may meet only when all its members are available. Normally deadlock-freedom is also required under the assumption that every meeting terminates. Deadlock-freedom is the property that after any operation o_i, such that there exists a j such that $P(j) \subseteq \mathcal{CP}_i$, either there is a subsequent operation $\langle$meet$\rangle$ by a committee $c_k \in \mathcal{C}$ and $P(k) \cap P(j) \neq \emptyset$, or there is a subsequent operation $\langle$go_on_holiday(p_l)$\rangle$ for $p_l \in P(j)$.

3.1.2 Graph Model

In a bipartite graph $G = (C, P, E)$, vertex sets C and P represent a collection of committees and professors respectively. An edge (c, p) is in E, if and only if $c \in C$, $p \in P$ and professor p is a member of committee c.

Every professor has a dynamic label called state. The state of a professor $p \in P$, denoted as *state*(p), is in {holiday, available, meeting(c)}. The label transitions are holiday $\rightarrow$ available, available $\rightarrow$ meeting(c), available $\rightarrow$ holiday, and meeting(c) $\rightarrow$ holiday . The transitions from holiday to available and meeting(c) to holiday are spontaneous. The global configuration of the graph is required to satisfy the following properties:

- Exclusion: In any configuration, if *state*(p)=meeting(c_i), then $\nexists q$ such that

$(q, c_i) \in E$ and *state*(q)= meeting(c_j), where $j \neq i$.

- Synchronization: In any configuration, if $\forall p \in \mathcal{P}$ such that $(p, c_i) \in E$, *state*(p)=meeting(c_i), then there exists a previous configuration, in which $\forall p \in \mathcal{P}$ such that $(p, c_i) \in E$, *state*(p)=available. In all the intermediate configurations, for all such p, *state*(p)=available or meeting(c_i).

Under the assumption that any professor with label meeting($\cdot$) will change his label to holiday later on, the global configuration of the graph is required to satisfy the following property:

- Deadlock-Freedom: In any configuration, if $\forall p \in \mathcal{P}$ such that $(p, c_i) \in E$, *state*(p)=available, then there exists a subsequent configuration, in which $\exists p \in \mathcal{P}$ such that $(p, c_i) \in E$ and *state*(p)=meeting(c_j), where $c_j \in C$.

3.1.3 Previous Work on the Committee Coordination Problem

Chandy and Misra's Work [6, page 334]

The specification of Chandy and Misra assumes the following:

1) A committee c_i starts executing $\langle$grab(P_i)$\rangle$ if and only if the last operation applied to any professor $p \in P_i$ is $\langle$become_available(p)$\rangle$, which implies whenever every member is available a committee tries to convene a meeting.

2) After an operation $\langle$release(P)$\rangle$, the next operation applied to any professor $p \in P$ is $\langle$go_on_holiday(p)$\rangle$, which implies that a professor becomes unavailable right after she leaves a meeting.

3) An available professor will remain available unless some committee, of which she is a member, convenes a meeting.

The implementation required is in message passing systems.

Algorithm Description: The authors first gave a trivial solution, where a committee starts a meeting if all members are available and no neighboring committee[1] is meeting. The solution is correct if there exists an extra process that collects the state information from all professors and centralizes all decisions about committee meetings.

They then introduced a solution in distributed settings, where both committees and professors are processes. The exclusion requirement is solved by mapping the Committee Coordination problem onto the Dining Philosophers problem by considering professors as forks and committees as philosophers. Thus neighboring committees corresponds to neighboring philosophers.

In the operation $\langle \text{grab}(P(i)) \rangle$ by a committee c_i, it first executes $\langle \text{grab}(F(i)) \rangle$ of a lockout-free solution to the Dining Philosophers problem, which guarantees operation $\langle \text{eat} \rangle$ will eventually happen. A committee can convene a meeting if and only if it is executing $\langle \text{eat} \rangle$ and all its members are available.

It is important to check whether every member is still available before an eating committee executes $\langle \text{meet} \rangle$. It is possible that the neighboring committees, say C and D, become hungry, and C eats and subsequently meets. After C's eating, the committee D can eat. However some of its members that are also in C may no longer be available. So D cannot meet without checking the state of all its members. If an eating committee notices that not all of its members are available, then it becomes thinking right away.

[1]Two committees are called neighbors if they have a common member.

The states of the professors are communicated asynchronously. Therefore care is required to ensure that every available member is still available when the collection is done. The authors claimed that it was safe for a committee c to collect states of its members after it started eating. At this point, none of its neighbors can be eating, implying none of its neighbors will convene a meeting. No member of c will participate in a meeting during the collection of c. Thus every available member remains available. The authors gave two alternative ways for a committee to collect information from its members. In the first one, a committee polls every member to determine whether they are available. In the second one, a professor will actively report its state to all the committees of which she is a member.

Checking the states of all members is also required in the thinking to hungry transition of a committee. However no extra work needs to be done, because even if a committee becomes hungry based on a wrong decision, and subsequently eats, it will check the states of all its members again before meeting. The above argument ensures that the committee will not start a meeting if some members are unavailable.

An available professor stays available until she attends a meeting and becomes unavailable right after she leaves a meeting. To implement this, in the execution of $\langle \text{release}(P) \rangle$ the committee calls the operation $\langle \text{go\_on\_holiday}(p) \rangle$ for every $p \in P$. The program of a professor p only contains the single operation $\langle \text{become\_available}(p) \rangle$.

Complexity: The author did not give the complexity analysis. But it is easy to see that the response time is bounded by the response time of the Dining Philosophers' solution used, because there is only a constant amount of overhead before the call to the Dining Philosophers subroutine.

Further Discussion: The original problem description is quite general. It does not specify when a professor changes state from available to unavailable. So it is possible that a professor remains available even after a committee of which she is a member convened and then adjourned a meeting. Also no assumption is made about when a professor joins and leaves a meeting. A professor may attend a meeting, leave the meeting, and then rejoin the same meeting again. The authors claimed the reason for giving a very general specification is to include a variety of situations. However in the implementation, where the synchronization is solved by the second alternative, the professors become unavailable only after they attend a meeting and the meeting ends. Therefore there is no reentry allowed in this case.

The author also mentioned a generalization of the Committee Coordination problem. An available professor waits for a subset of the committees of which she is a member; the subset may be different each time she becomes available. This generalization is comparable to the extension of the Dining Philosophers problem to the Drinking Philosophers problem. The author pointed out that the solution to this generalization can be obtained by minor modifications to the original solution.

3.2 Multiway Rendezvous Problem

The Multiway Rendezvous problem is essentially the same as the Committee Coordination problem as pointed out by many people, such as Choy and Singh [8], or Bagrodia [2]. It has the same object oriented and graph theoretical specifications as the Committee Coordination. Instead of having professors come together to attend a meeting held by a committee, in the Multiway Rendezvous problem, processes get

together to execute some event. In this section we will introduce Bagrodia's results by first paraphrasing his informal problem description and solutions, then fixing the flaws in his problem description.

3.2.1 Previous Work on the Multiway Rendezvous Problem

Bagrodia's Work [2]

In a message passing system, let $P = \{p_1, p_2, ..., p_n\}$ be a set of processes, and $E = \{e_1, e_2, ..., e_m\}$ be a set of events. A process p_i participates in a set of events $E_i \subseteq E$. An event e_k involves a set of processes $P_k \subseteq P$. Sets E and P are both static sets.

A process is either *idle* or *active*. Every process satisfies the following conditions:

- An idle process remains idle until it commits to some event.
- A process commits to an event e_k only when it determines that all other processes in P_k will also do so.
- An idle process can commit to at most one event at any time.
- An idle process becomes active if it commits to some event.
- An active process autonomously makes the transition to become idle.

An event is either *enabled* or *disabled*. An event e_k is enabled if and only if all processes in P_k are idle. Otherwise it is disabled. An event e_k is executed if and only if each process that is in P_k has committed to e_k.

The problem is to devise an algorithm that allows an idle process to commit to an enabled event such that the following properties are satisfied [2, page 1054]:

1) Safety:

a) Exclusion: If a process p_i commits to an event e_k, then $\forall p_j \in P_k$, p_j cannot commit to another event. In other words, conflicting events cannot be executed simultaneously.

b) An active process cannot commit to any event.

2) Liveness:

a) Synchronization: If process p_i commits to event e_k, then all processes that belong to P_k will eventually commit to e_k.

b) Progress: If all processes that belong to the process-set P_k of some event e_k are idle, then eventually some p_j that belongs to P_k must become active. This property ensures that if an event is enabled, it is eventually disabled.

The author designed centralized, partial distributed and fully distributed deterministic algorithms in message passing systems.

Algorithm Description: Bagrodia's algorithms used message counts to solve synchronization. The total number of times a process has become idle or active is called *idle-count* or *active-count*, respectively. There are some special processes called event managers. An event manager M may control a set of events $E(M) \subseteq E$. Event manager M maintains the idle-counts and active-counts for all the processes in P_k if $e_k \in E(M)$. All counters are initialized to zero. When a process p_i in the system becomes idle, it sends a ready message to all the managers that control an event in E_i. On receiving a ready message from a process, the manager increments the

idle-count of the process by 1. If the manager find that there exists an event e_k, such that $\forall p_i \in P_k$, p_i's idle-count is greater than its active-count by 1, then the manager may inform each p_i to commit to e_k, and increment the active-count for p_i by 1.

The event manager plays the role of committee coordinator, and the above actions correspond to the $\langle \text{grab}(\cdot) \rangle$ operation in the Committee Coordination problem.

The author first built a centralized algorithm where there is only one event manager that manages all the events in E. The exclusion is easily satisfied in the algorithm.

Then he designed a partially decentralized algorithm, where there exist several managers. In order to ensure exclusion, they used a token circulating among the managers. Only the one holding the token can schedule its events.

Finally he presented a modified algorithm that is decentralized. This algorithm used the message counts to solve the synchronization problem as above, and the selection technique of the Chandy and Misra's committee coordination algorithm to solve exclusion problem. Specifically, the exclusion is solved by mapping the Multiway Rendezvous problem onto the Dining (or Drinking) Philosophers problem. The event manager corresponds to the philosopher in the Dining Philosophers (or Drinking Philosophers) problem. Event managers M_i and M_j are neighbors if there exists event $e_k \in E(M_i)$ and $e_l \in E(M_j)$, such that $P_k \cap P_l \neq \emptyset$. An event manager may schedule an event only when the corresponding philosopher is eating.

Complexity: The author did a simulation study to compare the performance of the three algorithms. The response time for each algorithm is measured from the instant that an event becomes enabled to the instant that it is selected for execution. Two

components determine the total response time for multiway rendezvous algorithms:

1) Synchronization Time: Time taken by the algorithm to ascertain that a given event is enabled.

2) Selection Time: Time taken by an algorithm to select an event for execution.

They showed how variations in model parameters affects one or the other component and consequently the response time. The parameters include the network topology, the average time to transmit a message between processes, and the synchronization pattern among processes in the system.

Further Discussion: In the algorithms, processes can start and end an event at different times. There may not exist a synchronous point where everyone is executing the event, but there is a synchronous point where every process is idle and ready to commit to an event.

The synchronization property described in the problem description is very vague. There is no precise condition for a process to commit to an event. A process will commit to an event if it somehow knows all the other processes that belongs to the same event will also do so. Also the synchronization requirement does not avoid multi-entry. The algorithms do not allow this by using the message counts.

Their definition of enable is not correct. Once a process commits to an enabled event e_k, e_k becomes disabled. This prevents all the other processes in P_k from committing to e_k. We correct this definition as follows: an event is enabled if all the processes that belong to it either are idle or committed to it.

3.3 Multiparty Interaction Problem

In both the Committee Coordination problem and the Multiway Rendezvous problem, there is a coordinator or manager associated with each committee or event respectively. These coordinators or managers make decisions on when the corresponding committees start meeting or the corresponding events can be executed. They are responsible for guaranteeing the computation of the whole system satisfies the exclusion, synchronization, and progress properties. If they do not exist, professors or the processes need to be totally active, and coordinate among themselves. This lack of the coordinators and managers distinguishes the Multiparty Interaction problem from the Committee Coordination problem and the Multiway Rendezvous problem. Another difference is that the Multiparty Interaction problem requires two synchronization points, while the Committee Coordination problem and the Multiway Rendezvous problem require only one.

In this section we first give the object oriented specification and graph theoretical description of the Multiparty Interaction problem. Then we present Yuh-Jzer Joung's solution to this problem.

3.3.1 Object Oriented Specification

A Multiparty Interaction system $(\mathcal{P}, \mathcal{I})$ consists of a set of processes $\mathcal{P} = \{p_1, p_2, ..., p_n\}$, and a set of objects $\mathcal{I} = \{i_1, i_2, ..., i_k\}$ called interactions. Each process $p_k \in \mathcal{P}$ participates in a fixed set of interactions $I(k) \subseteq \mathcal{I}$. Each interaction i_l is associated with a fixed set $P(l) = \{p_k \in P | i_l \in I(k)\}$.

The set of interactions $\mathcal{I}$ supports three operations: $\langle$become_ready(I)$\rangle$ for

$I \subseteq \mathcal{I}$, and $\langle$start_interaction$(i)\rangle$, $\langle$end_interaction$(i)\rangle$ for $i \in \mathcal{I}$.

Let $S = o_1, o_2, ...$ be any sequence of operations on $\mathcal{I}$. To give the validity conditions on S, first recall that $\langle$operation_name$\rangle_p$ denotes the operation executed by process p. Then S is valid provided:

1) if in the interval between o_i $=\langle$start_interaction$(u)\rangle_{p_r}$ and the following o_j $=\langle$end_interaction$(u)\rangle_{p_r}$, there exists an operation $\langle$start_interaction$(v)\rangle_{p_s}$, where $u \neq v$ and $p_s \in I(u)$, then there exists a previous operation $\langle$end_interaction$(u)\rangle_{p_s}$ after operation o_i. and

2) if o_i $=\langle$start_interaction$(u)\rangle_{p_r}$, then $\forall p_s \in P(u)$ $\exists j < i$, such that o_j $=\langle$become_ready$(I(s))\rangle_{p_s}$, and there exists at most one operation executed by p_s between o_j and o_i, it can only be $\langle$start_interaction$(u)\rangle_{p_s}$. and

3) for every i, such that o_i $=\langle$end_interaction$(u)\rangle_{p_r}$, there exists distinct previous operations $\langle$start_interaction$(u)\rangle_{p_s}$ by all $p_s \in I(u)$.

The program for any process $p_k \in \mathcal{P}$ is:

Do Forever:

$\langle$become_ready$(I(k))\rangle$

$\langle$start_interaction$(i)\rangle$ for $i \in I(k)$

$\langle$end_interaction$(i)\rangle$

$\langle$local computation$\rangle$

End Do

The $\langle$local computation$\rangle$ is an arbitrary operation that does not act on $\mathcal{I}$.

The validity conditions only capture: 1) the exclusion property, which guarantees that no philosopher can participate in two interactions at the same time, and 2) the synchronization property, which guarantees that an interaction may only start when all processes that participate in it becomes ready, and a process cannot finish an interaction until all the other processes that participate in the same interaction have started. A strong progress property, lockout-freedom, which delivers fairness, is also required under the assumption that every interaction terminates. An interaction i_t is *enabled* if and only if for every $p_j \in P(t)$, p_j's most recent action is $\langle \text{become\_ready}(I(j)) \rangle$. Lockout-freedom is the property that for any interaction i_t that is enabled infinitely often, there exist infinite occurrences of $\langle \text{start\_interaction}(t) \rangle_{p_j}$ $\forall p_j \in P(t)$.

3.3.2 Graph Model

In a bipartite graph $G = (P, I, E)$, vertex sets P and I represent a collection of processes and interactions respectively. An edge $(p, i) \in E$ if and only if process p participates in interaction i.

Every process has a dynamic label called state. The state of a process $p \in P$, denoted as *state*(p), is in {idle, ready, execute(i)}. The state transitions are idle $\rightarrow$ ready $\rightarrow$ execute(i) $\rightarrow$ idle. The transitions from idle to ready and execute(i) to idle are spontaneous.

Let $C = c_1, c_2, ...$ be a sequence of global configurations. Any $c_i \in C$ satisfies the following properties:

- Exclusion: If *state*(p)=execute(r) in c_i and *state*(q)=execute(s), such that $r \neq s$ and $(q, r) \in E$, then there exists a previous configuration c_j, where

state(p)=execute(r) and *state*(q)=execute(r), and in all the intermediate configurations *state*(p)=execute(r).

- Synchronization: Suppose *state*(p)=execute(r) in c_i. Let c_j be a previous configuration of c_i, such that *state*(p)=execute(r) in c_j, *state*(p)=ready in c_{j-1}, and *state*(p)=execute(r) in all intermediate configuration between c_j and c_i. Let c_k be a subsequent configuration of c_i, such that *state*(p)=execute(r) in c_k, *state*(p)=idle in c_{k+1}, and *state*(p)=execute(r) in all intermediate configuration between c_i and c_k.

  1. There exists a configuration c_m before c_j, in which $\forall q \in P$ such that $(q, r) \in E$, *state*(q)=ready, and in all the configurations between c_m and c_j, the state of q is either ready or execute(r).
  2. There exists a configuration c_n between c_j and c_k, in which $\forall q \in P$ such that $(q, r) \in E$, *state*(q)=execute(r).

An interaction r is enabled in a configuration if $\forall p \in P$ such that $(p, r) \in E$, *state*(p)=ready. This definition is equivalent the one on page 71. An interaction r is *executed* in a configuration if $\forall p \in P$ such that $(p, r) \in E$, *state*(p)=execute(r). Under the assumption that any process with state execute($\cdot$) will change its state to idle later on, any sequence of global configurations C is required to satisfy the strong progress property:

- Lockout-Freedom: If a configuration where r is enabled appears infinitely often, then a configuration where r is executed will occur infinitely often.

3.3.3 Previous Work on the Multiparty Interaction Problem

Joung's Work [19]

Joung gave a problem specification as follows: [19, page 311]

> We assume a fixed set of sequential processes $p_1, ..., p_n$ which interact by engaging in multiparty interactions $X_1, ..., X_m$. Each multiparty interaction X_i involves a fixed set of processes $P(X_i)$. Initially, each process in the system is in its *local computing phase* which does not involve any interaction with other processes. From time to time, a process becomes ready for a set of potential interactions of which it is a member. After executing any one of the potential interactions the process returns to its local computing phase.
>
> Assume that a process starting an interaction will not complete the interaction until all other participants have started the interaction. Assume further that a process will eventually complete an interaction if all other participants have started the interaction.

The problem is to devise an algorithm to schedule interactions satisfying the following requirements:

1. Exclusion: No two interactions can be in execution simultaneously if they have a common member. (An interaction is in execution if all its members have stared it.)

2. Synchronization: If a process p starts X, then all other processes in $P(X)$ will eventually start X.

3. Strong Interaction Fairness (SIF): If an interaction is enabled infinitely often, then it will be executed infinitely often. (An interaction is enabled if its participants are all ready, and becomes disabled when some of them starts an interaction.)

Algorithm Description: They provided two randomized algorithms, one for message passing systems and the other for shared memory systems. For the shared memory algorithm, only single-writer multiple-reader variables are used. Their algorithms are completely decentralized, meaning that there is no coordinating process, and also symmetric in the sense that all processes are anonymous and execute the same code. Both algorithms guarantee SIF with probability 1 under the following two assumptions: (A1) processes do not stop executing their programs, and (A2) a process's transition to a state ready for interactions does not depend on the random choices performed by other processes.

When a process p becomes ready for interaction, it randomly chooses one interaction X from the set of interactions it is willing to execute. It informs (by sending messages or by writing information to shared variables) other processes in $P(X)$ of its interest in executing X, and waits for Δ time, where Δ is a parameter of the algorithm. When Δ time elapses, p collects the information from all the other processes in $P(X)$. If all of them are ready to execute X, then p will set a flag indicating the successful establishment of X, and start executing X. If p notices the flag was set by another process, it also starts X. If neither of the above cases is true, p will give up X, and start from the beginning again.

A process p is *monitoring* an interaction X, if p has chosen X and is waiting

for its Δ-interval to expire. Interval Δ is called the monitoring time. If p does not monitor X long enough, then it may not see others' choices when it completes its monitoring phase. Thus possibly no interaction is established even if the random choices of all processes in $P(X)$ coincide. If the monitoring time of all the process in $P(X)$ overlap, then some process will notice the agreement, and establish X. In the algorithms Δ is chosen to be the sum of the previous non-monitoring time of all processes in $P(X)$. The appropriate choice of Δ gives the fairness property.

Complexity: Joung claimed the following [19, page 329]:

> Assume that a process may be ready for k potential interactions at a time, and each interaction involves m participants.
>
> Suppose that the time to execute a local action is negligible compared to the communication time for delivering a message.
>
> If the message transmission time is c, then the time complexity is dominated by
>
> $$4c \cdot m \cdot k^m$$
>
> In the above, since m messages are sent in parallel in each interval c, the expected number of messages needed to establish an interaction per process is no greater than
>
> $$4m^2 \cdot k^m$$

Further Discussion: In the problem description, the author failed to point out that a process cannot re-start an interaction X while other processes are still executing X. Both algorithms avoid this case.

In Joung's problem description, two synchronization points are required: all processes are ready and all of them are executing an interaction. However this does not make the Multiparty Interaction problem more difficult than other Process Synchronization problems, because by repeatedly using a technique to achieve synchronization, one can achieve any number of synchronization points desired.

CHAPTER 4

Comparison and Analysis of Different Process Coordination Problems

In Chapter 2 and 3, we studied the six different kinds of Process Coordination problems listed in Figure 4.1. Every problem consists of two different components. A synchronization property is required on the components in the third column. As a result of synchronization, a coordination activity executed by the components in the second column is triggered. For example, in the Dining Philosophers problem when all forks needed by a philosopher are gathered together, that philosopher starts eating. Also an exclusion property is required on the behaviors of the components in the second column. For example, in the Dining Philosophers problem no two philosophers can eat simultaneously if they use a common fork. Different Process Coordination problems have different assumptions on which component is active and which is passive. As shown in Figure 4.1, all the red components are active and all the blue ones are passive. In the first three problems the forks, beverages, and resources are passively collected by their users to achieve the synchronization

point. Whereas in the latter three problems the professors and the processes are actively participating in the synchronization procedure. Therefore we call the first three problems the General Resource Allocation problem and the last three ones the Process Synchronization problem.

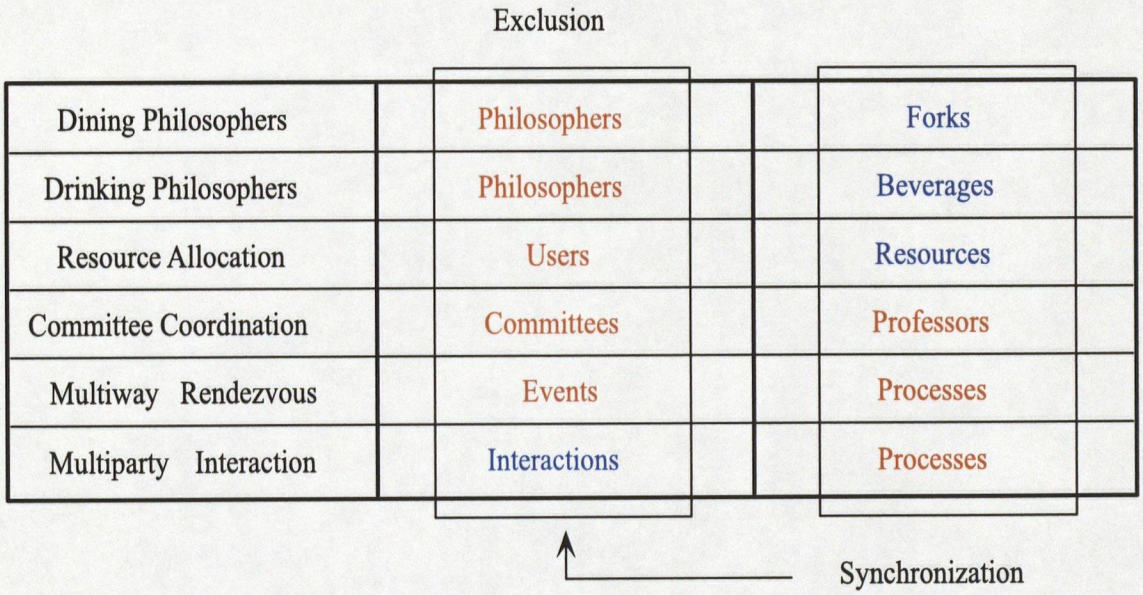


Figure 4.1: Process Coordination Problems

In the set of General Resource Allocation problems, the Dining Philosophers problem corresponds to a graph-theoretic formalization of the set of Static Resource Allocation problems, and can also be used to solve the Dynamic Resource Allocation problem. In the set of Process Synchronization problems, the Committee Coordination problem has the most general problem description and the loosest requirements, which makes it representative of this class. Therefore the Dining Philosophers problem and the Committee Coordination problem are considered as representatives for the Resource Allocation problem and the Process Synchronization problem, respec-

the Resource Allocation problem and the Process Synchronization problem, respectively.

In this chapter we first compare the similarity and difference between the two classes of the Process Coordination problem by comparing the Dining Philosophers problem and the Committee Coordination problem. Using the uniform problem specifications in the previous two chapters makes the comparison much easier and clearer. Then we further discuss the relations between the Dining Philosophers problem and the Multiparty Interaction problem, which are dual problems with respect to the role of the active and passive components of the system. At the end, we give some comments on the atomicity requirements in the Resource Allocation problem.

4.1 Dining Philosophers vs. Committee Coordination

Assume professors in the Committee Coordination problem play the role of forks in the Dining Philosophers problem, and committees play the role of philosophers. These two problems differ in two aspects:

Difference-I : In the Committee Coordination problem, professors could be ready to attend meetings or be idle. A committee cannot meet if one of its members is idle. In the Dining Philosophers problem, forks that are not being used are always available for philosophers.

Difference-II : Every committee is always trying to meet as long as all its members are ready, whereas if a philosopher is not hungry, she does not try to eat even if all forks are available.

To eliminate the differences, we can modify either the Dining Philosophers problem or the Committee Coordination problem.

4.1.1 Modified Dining Philosophers Problem

To address Difference-I, assign a state *clean* or *dirty* to each fork. A dirty fork is not available (i.e. in dishwasher). A philosopher can only pick up a clean fork. Therefore a philosopher cannot eat if one of its forks is dirty.

To address Difference-II, let philosophers become hungry whenever all their forks are clean. With this modification, the exclusion, synchronization, and progress requirements in the modified Dining Philosophers problem are similar to the requirements in the Committee Coordination problem. The only difference is that the latter one requires only deadlock-freedom as opposed to lockout-freedom. Table 4.1 compares these two problems in the object oriented framework. The object oriented specification for the Committee Coordination problem described in Table 4.1 is not the same as the one given in Section 3.1.1. It is easy to see however that these two descriptions both capture the behaviors and requirements of the Committee Coordination problem and are equivalent. We use this description because it highlights the similarity between the modified Dining Philosophers problem and the Committee Coordination problem.

4.1.2 Modified Committee Coordination Problem

Based on the graph model $G = (P, C, E)$ of the Committee Coordination problem described in Section 3.1.2, we model the modified problem with a graph $G' = (P', C', E')$ built from G, such that $P' = P$ and $C' = C \cup \{\hat{c}\}$. We create a special

	Modified Dining Philosophers	**Committee Coordination**
Objects	set of forks $\mathcal{F}$	set of professors $\mathcal{P}$
Local Fields	$C = \emptyset$,$U = \mathcal{F}$, initially	$A = \emptyset$, $I = \mathcal{P}$, initially
Public Methods	$\langle$get_forks(F)$\rangle$ $F \subseteq \mathcal{F}$ precondition: $F \subseteq C$ effect: $U \leftarrow U \backslash F$ $C \leftarrow C \backslash F$	$\langle$get_profs(P)$\rangle$ $P \subseteq \mathcal{P}$ precondition: $P \subseteq A$ effect: $I \leftarrow I \backslash P$ $A \leftarrow A \backslash P$
	$\langle$release_forks(F)$\rangle$ effect: $U \leftarrow U \cup F$	$\langle$release_profs(P)$\rangle$ effect: $I \leftarrow I \cup P$
Local Methods	$\langle$become_clean(f)$\rangle$ $\forall f \in \mathcal{F}$ precondition: $f \in U$ effect: $C \leftarrow C \cup \{f\}$	$\langle$become_available(p)$\rangle$ $\forall p \in \mathcal{P}$ precondition: $p \in I$ effect: $A \leftarrow A \cup \{p\}$
Processes	set of philosophers $\{p_1 ... p_n\}$	set of committees $\{c_1 ... c_n\}$
Program	$\forall p_i$ $\exists F_i \in \mathcal{F}$ $\langle$get_forks(F_i)$\rangle$ $\langle$eat$\rangle$ $\langle$release_forks(F_i)$\rangle$ $\langle$think$\rangle$	$\forall c_i$ $\exists P_i \in \mathcal{P}$ $\langle$get_profs(P_i)$\rangle$ $\langle$meet$\rangle$ $\langle$release_profs(P_i)$\rangle$ $\langle$adjourn$\rangle$
Restriction	No philosopher eats forever, no fork remains dirty.	No committee meets forever, no professor remains unavailable.
Synchro-nization	A philosopher may eat only if all her forks are clean.	A committee may start meeting only if all its members are available.
Exclusion	No two philosophers can eat simultaneously, if they use a common fork.	No two committees can meet simultaneously, if they have a common member.
deadlock-freedom	If some philosophers want to eat, some philosopher will eat.	If some committees want to meet, some committee will meet.
lockout-freedom	If a philosopher wants to eat, she will eat eventually.	Not applicable

Table 4.1: Modified Dining Philosophers vs. Committee Coordination

committee $\hat{c}$, which has dynamic membership and meets all the time. To eliminate Difference-I, we simulate the occasional idle time of a professor p by generating an edge $(\hat{c}, p)$. With this adjustment, professors that are not attending any meeting (including $\hat{c}$) are always available.

To deal with Difference-II, the decision about when a committee tries to meet is made by the committee instead of depending on the states of all members of the committee. Also a trying committee can actively gather all its members in order to meet. This modified Committee Coordination problem is no different from the original Dining Philosophers problem.

4.2 Further Discussion on the Dining Philosophers problem and the Multiparty Interactions problem

In the Multiparty Interactions problem only processes are active, and in the Dining Philosophers problem only philosophers are active. It is easy to see that these two problems are dual to each other by exchanging the active and passive roles of the concurrent entities, which are the processes and forks. In this section we will further discuss the relations between these two problems.

There are two natural ways to map the Dining Philosophers problem onto the Multiparty Interactions problem.

First, consider the processes as forks and interactions as philosophers. Then forks are active and try to get together for a certain philosopher. This does not quite fit the meaning of the Dining Philosophers problem.

We can also think of both forks and philosophers as processes. An interaction

consists of a philosopher and all the forks she uses. Each interaction involves two kind of processes: an active process which is the philosopher who tries to execute the interaction and several passive processes which are the forks that only give responses. Every active process participates in only one interaction.

4.3 Atomicity in the Resource Allocation problem

In the set of Resource Allocation problems, the exclusion property requires that for every resource, only one user can have it at a time. Effort needs to be made in order to guarantee this. There are different ways to do that under different system settings.

In message passing systems, a resource can be denoted as a single token circulating among its users. Because the token cannot be duplicated, only one user at a time can have the resource by holding the corresponding token. The exclusion requirement is easy to achieve in this setting.

In the shared memory model, using only atomic read/write variables is not sufficient. Normally, acquiring a resource involves two operations: 1) read the current state of the resource to check whether it is free; 2) update the state of the resource to complete its collection. An atomic step of the operations provided by read/write variables is either read or write (also called update). Thus if two neighboring users both finished the first operation and found that a shared resource is free, then they could proceed to the second step and both get the resource. To avoid this, one way is to make operation 1 and 2 a composite atomic step so that nobody can be interrupted before it picks up a resource. Another way is to use stronger objects instead

of read/write variables to protect the resources. For example the test-and-set operation provided by test-and-set objects can finish acquiring a resource in one atomic step.

In both settings, if the resources are active, which means they can respond to requests from users, then the exclusion property can be accomplished by allowing a resource to respond to only one request at a time.

CHAPTER 5

Self-Stabilization

In Chapter 2 we demonstrated the representative role of the Dining Philosophers problem in the set of Resource Allocation problems. And as described in Chapter 4, with slight modifications, solutions to the Dining Philosophers problem can also be used to solve the Committee Coordination problem, which is a fundamental version of Process Synchronization. Every Process Coordination problem can be solved by either using solutions to the Dining Philosophers problem directly, or calling solutions to the Dining Philosophers problem as a subroutine, or using modified solutions to the Dining Philosophers problem. This is because the Dining Philosopher's solution can always be used to build the exclusion and progress requirements. Therefore an efficient and robust solution to the Dining Philosophers problem is very useful. Since self-stabilization is a strong fault tolerant model in distributed systems, the goal of the remainder of this thesis is to design a self-stabilizing solution to the Dining Philosophers problem.

In this chapter, we introduce formal definitions of self-stabilization, and review a

useful design and proof technique, *fair composition.* Then we enrich the fair composition for more general use. This enhanced version is only applicable to randomized algorithms and provides one of the techniques needed in Chapter 6.

5.1 Self-Stabilization Preliminaries

In a distributed system, the global configuration of the system is a combination of the local states of all system components. Let S be a distributed system and C be all possible global configurations of S. Let L be a subset of C. System S *converges to* L if it is guaranteed to arrive at a configuration in L in a finite number of steps, regardless of its initial configuration. This behavior is also called *Convergence.* System S is *closed under* L if starting from any configuration in L, all subsequent configurations of the system are in L. This behavior is also called *Closure.* System S is *self-stabilizing for* L if it converges for L and is closed under L. A self-stabilizing system does not need to be initialized, because it can start from an arbitrary configuration and, by convergence, eventually reach a legitimate configuration. Also a self-stabilizing system can recover automatically after system failures, because one can always assume that the configuration after any failure is the arbitrary initial configuration. Normally L is called a set of legitimate configurations and is defined by giving a predicate P over C, such that a configuration is legitimate if it satisfies P. Figure 5.1 shows the intuition of self-stabilization.

In an asynchronous distributed system, the activities of the components are assumed to be arranged by a scheduler. An execution is produced by a scheduler that determines, for each partial execution, what subset of components will take a step

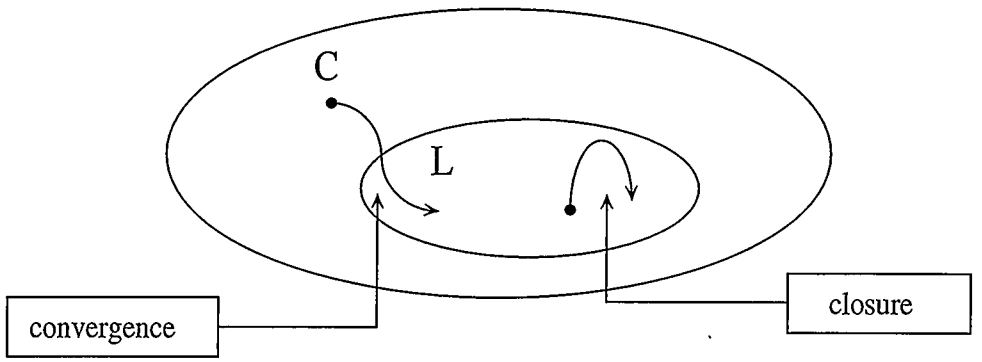


Figure 5.1: Self-Stabilization

of their program to extend the execution to the next configuration. The following definition formalizes Dolev's description in his book [11, page 9,23].

Definition 5.1.1. *Let P and Q be predicates over configurations. A system running algorithm A is* self-stabilizing for Q given P *under a set S of schedulers, if the following conditions are true:*

Convergence: *Starting from any configuration satisfying P and for any scheduler in S, in a finite number of steps algorithm A, the system will converge to a configuration satisfying Q.*

Closure: *For any configuration satisfying Q, all subsequent configurations of the system satisfy Q.*

Definition 5.1.1 is only applicable to deterministic algorithms. The following definition is the natural extension to randomized algorithms. The certainty conditions are replaced by probabilistic conditions.

Definition 5.1.2. *Let P and Q be predicates over configurations. A system running algorithm A is* randomized self-stabilizing for Q given P *under a set S of schedulers, if the following conditions are true:*

Convergence: *Starting from any configuration satisfying P and for any scheduler in S, with probability 1 in a finite number of steps algorithm A, the system will converge to a configuration satisfying Q.*

Closure: *For any configuration satisfying Q, all subsequent configurations of the system satisfy Q.*

In this thesis we consider S to be the set of weakly fair distributed schedulers (see page 9) and omit explicit reference to S when there is no ambiguity.

5.2 Original Fair Composition

Fair composition is a technique introduced by Dolev, Israeli and Moran[11, 12] for designing, analyzing, and proving the correctness of complex self-stabilizing algorithms. To tolerate transient faults, self-stabilizing algorithms never terminate. In our application, non-terminating algorithms are expressed as a loop in the form:

Do Forever:

⟨block of operations⟩

End Do

Define an *iteration* of an execution of such a non-terminating algorithm to be the execution of one pass through the loop. The term *round* of an execution is usually

used to denote enough steps, so that every process has done at least one step of its algorithm. More formally, it can be defined inductively by:

1) The first round of an execution is the shortest prefix of the execution that contains at least one step of every process.

2) The ith round of an execution is the shortest prefix of the suffix of the execution after round $i-1$ that contains at least one step of every process.

Correspondingly, define a *super round* inductively by:

1) The first super round of an execution is the shortest prefix of the execution that contains at least one iteration of every process.

2) The ith super round of an execution is the shortest prefix of the suffix of the execution after round $i-1$ that contains at least one iteration of every process.

We now give the definition of fair composition.

Definition 5.2.1. *The algorithm constructed from algorithms A and B by alternating steps of A and B in any way that guarantees that in any execution, steps of A and steps of B are both executed infinitely often, is a* fair composition *of algorithms A and B and is denoted as $A \circ B$.*

Based on Definition 5.2.1, notation $B \circ A$ is equivalent to $A \circ B$.

A super round of a fair composition $A \circ B$ is a partial execution that contains enough steps so that every process finishes an iteration of both A and B.

The idea of fair composition is to compose two algorithms together to obtain stronger results. Let P, Q and R be predicates over configurations, algorithm A be (randomized) self-stabilizing for Q given P, and algorithm B be (randomized) self-stabilizing for R given Q. We would like algorithm $A \circ B$ to be (randomized)

self-stabilizing for R given P (see Figure 5.2). To achieve this, algorithms A and B need to ensure some conditions that allow the composition to behave like A until P becomes true and like B after that.

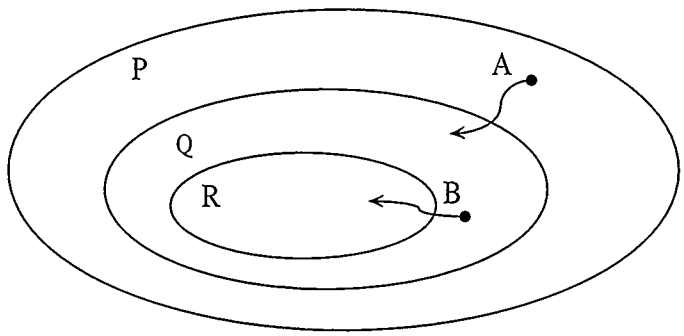


Figure 5.2: Fair Composition of algorithms A and B

Dolev [11, page 22-24] gave the precise restrictions on algorithms A and B to guarantee the behavior of the composition. Informally the conditions are[1]:

1. Algorithm B does not modify any variables used by A. This ensures that algorithm A will behave in the composition in the same way as it does alone. Thus algorithm $A \circ B$ will achieve a configuration satisfying Q given P.

2. In any configuration satisfying P, algorithm A does not obstruct B. This ensures that in any configuration satisfying P, algorithm B acts in the same way as if A did not exist, which implies that $A \circ B$ will converge to and then remain in a configuration satisfying R.

[1]We omit repeating the precise conditions because in the next section we will describe more general ones.

Dolev's conditions guarantee the convergence of $A \circ B$ by preventing the system from moving "backwards" once the goal of algorithm A is achieved.

5.3 Enriched Fair Composition

If the second algorithm modifies the variables used by the first one, Dolev's conditions are not satisfied. It is still possible to establish convergence of the composed algorithms, provided the algorithms meet some different required properties. In the situation considered here the second algorithm can destroy the progress achieved by the first one (see the dashed line in Figure 5.3). If the algorithm is randomized and this happens only with low probability, and once it does happen, the resulting configuration is one that permits the first one to "try again", then eventually, the composition will achieve the combined goal and remain there.

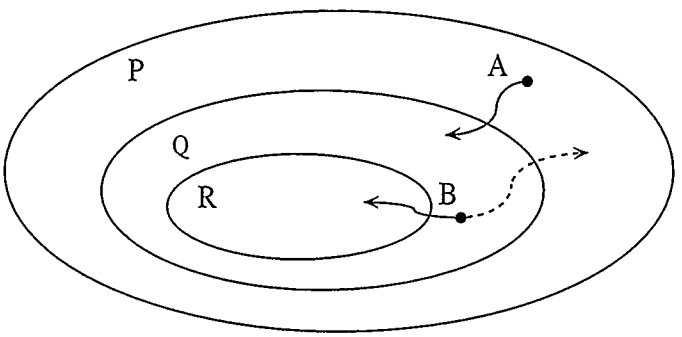


Figure 5.3: Enriched Fair Composition

In this section, we relax the requirements on the two algorithms that are being composed so that both can randomly modify a set of common variables. All algorithms discussed in this section are assumed to be randomized. Since the issues in

the randomized setting are very subtle, we will introduce a lot of notation.

Definition 5.3.1. *Let P, Q and R be predicates over configurations. An algorithm A is* random(p, k) self-stabilizing for predicate R else P given Q, *if the following conditions are true:*

Probabilistic Convergence: *For any configuration satisfying Q and for any distributed scheduler, after at most k super rounds of algorithm A a configuration c is reached, and*

i) c satisfies either R or P.

ii) with probability at least p, c satisfies R.

Closure: *For any configuration satisfying R, all subsequent configurations of the algorithm A satisfy R.*

In the rest of this thesis, we use Figure 5.4 to illustrate that algorithm A is random(p, k) self-stabilizing for predicate R else P given Q.

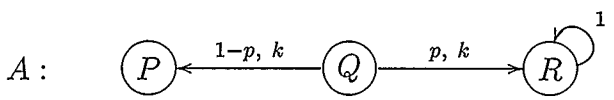


Figure 5.4: random(p, k) self-stabilizing for predicate R else P given Q

If $P = Q$, then algorithm A is as shown in Figure 5.5.

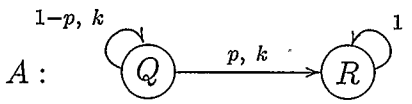


Figure 5.5: random(p, k) self-stabilizing for predicate R else Q given Q

Lemma 5.3.1. *Let Q and R be predicates over configurations. If an algorithm A is random(p,k) self-stabilizing for predicate R else Q given Q, and $p > 0$, then A is randomized self-stabilizing for R given Q.*

Proof. For any configuration satisfying predicate Q and for any distributed scheduler, after k super rounds of algorithm A, a configuration c is reached. With probability at least p, c satisfies R, otherwise it satisfies Q. The probability that after lk super rounds of algorithm A the system remains in a configuration satisfying Q is at most $(1-p)^l$. Because $p > 0$, $\lim_{l\to\infty}(1-p)^l = 0$. Thus starting from any configuration satisfying Q, with probability 1 algorithm A will converge to a configuration satisfying R. Based on the closure property of A, any subsequent configuration will also satisfy R. Therefore algorithm A is random self-stabilizing for predicate R given Q. □

Corollary 5.3.2. *Let Q and R be predicates over configurations and let A be random(p,k) self-stabilizing for R else Q given Q. Then starting from any configuration satisfying Q, the expected number of super rounds for algorithm A to converge to a configuration satisfying R is at most $\frac{k}{p}$.*

Proof. From any configuration satisfying Q, after k super rounds of algorithm A, with probability at least p the new configuration satisfies R, otherwise the system remains in a configuration satisfying Q. Hence the expected number of super rounds for A to converge to a configuration satisfying R from an arbitrary configuration satisfying R, denoted as $E[Q,R]$, is:

$$E[Q,R] \leq k + (1-p)E[Q,R]$$

implying

$$E[Q,R] - (1-p)E[Q,R] \leq k$$

$$pE[Q,R] \leq k$$

$$E[Q,R] \leq \frac{k}{p}.$$

□

When two algorithms share some variables, different ways to compose algorithms may cause different behaviors of the composition. In a super round of a composition, the ratio of the number of steps taken from the two algorithms may affect the convergence property of $A \circ B$. For example, if one does not have a chance to interfere with the other very often, then the composition may converge faster. However, if one algorithm takes too many steps in each super round, and keeps interfering with the other, then the composition may never converge. Therefore, when we describe the property of fair composition of algorithm A and B in a randomized setting, we assume a particular fixed composition. In the rest of this paper, we use $A \oplus B$ to represent a fixed composition of A and B, and $A \circ B$ to denote any possible fair composition of A and B. Note that $A \oplus B$ and $B \oplus A$ are different fair compositions. For example, let $A \oplus B$ be a fair composition in which steps of A are executed more frequently than steps of B. In composition $B \oplus A$, since algorithm B is in the position of A, it will be executed more often.

Definition 5.3.2. *Let algorithm A be random(p,k) self-stabilizing for predicate R else P given Q. Algorithm B is* (Q,r) right non-interfering with A via $A \oplus B$, *where* $0 \leq r \leq 1$, *if for any configuration satisfying Q and for any distributed scheduler,*

after at most k super rounds of algorithm $A \oplus B$, a configuration c is reached, such that

1. *c satisfies either R or P.*
2. *with probability at least $p \cdot r$, c satisfies R.*

Similarly, we introduce a dual definition:

Definition 5.3.3. *Let algorithm A be random(p, k) self-stabilizing for predicate R else P given Q. Algorithm B is* (Q, r) left non-interfering with A via $B \oplus A$, *where $0 \leq r \leq 1$, if for any configuration satisfying Q and for any distributed scheduler, after at most k super rounds of algorithm $B \oplus A$, a configuration c is reached, such that*

1. *c satisfies either R or P.*
2. *with probability at least $p \cdot r$, c satisfies R.*

Note that algorithm $A \oplus B$ (or $B \oplus A$) is not necessarily random$(p \cdot r, k)$ self-stabilizing for predicate R else P given Q, because the closure property is not guaranteed.

Definition 5.3.4. *Let $C_1, ..., C_k$ be a sequence of sets of configurations, such that $C_i \subseteq C_{i-1}$ for $i = 2, ..., k$. Define the predicate S_i over configurations by $S_i(c)$ if and only if $c \in C_i$. The sets C_1 to C_k are called* nested sets, *and the predicates S_1 to S_k are called* nested predicates. *We use the same notation to denote nested predicates, $S_k \subseteq S_{k-1} \subseteq \cdots \subseteq S_1$.*

Clearly if $R \subset Q$ in Definitions 5.3.2 and 5.3.3, then algorithm B is also (R, r) right (left) non-interfering with A.

Now we are ready to introduce the main theorem for enriched fair composition.

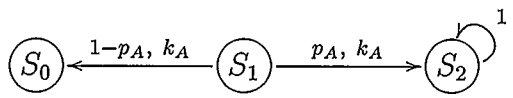
Theorem 5.3.3. *Let S_0, S_1, S_2, and S_3 be nested predicates over configurations, and $A \oplus B$ be a fixed fair composition of algorithms A and B. Given the following four conditions:*

1. *Algorithm A is random(p_A, k_A) self-stabilizing for predicate S_2 else S_0 given S_1.*
2. *Algorithm B is random(p_B, k_B) self-stabilizing for predicate S_3 else S_0 given S_2.*
3. *Algorithm A is $(S_2, 1)$ left non-interfering with B via $A \oplus B$.*
4. *Algorithm B is (S_1, r) right non-interfering with A via $A \oplus B$.*

Then algorithm $A \oplus B$ is random$(p_A p_B r, k_A + k_B)$ self-stabilizing for predicate S_3 else S_0 given S_1.

Proof. We have algorithm A is random(p_A, k_A) self-stabilizing for predicate S_2 else S_0 given S_1.

A :



Algorithm B is (S_1, r) right non-interfering with A via $A \oplus B$. Thus, by Definition 5.3.2, for any configuration satisfying S_1 and for any distributed scheduler, after at most k_A super rounds, algorithm $A \oplus B$ converges to a configuration satisfying S_2 with probability at least $p_A \cdot r$, otherwise the system goes to a configuration satisfying S_0.

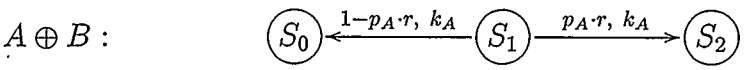


Figure 5.6: $A \oplus B$'s behavior in configuration satisfying S_1

Algorithm A is $(S_2, 1)$ non-interfering with B via $A \oplus B$ and $S_3 \subset S_2$. Therefore algorithm $A \oplus B$ has the same behavior as B from any configuration satisfying S_2 and S_3. And algorithm $A \oplus B$ is random(p_B, k_B) self-stabilizing for predicate S_3 else S_0 given S_2.

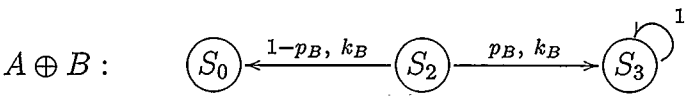


Figure 5.7: $A \oplus B$'s behavior in configuration satisfying S_2

Combining the behaviors given in Figures 5.6 and 5.7, algorithm $A \oplus B$ yields the behavior shown in Figure 5.8:

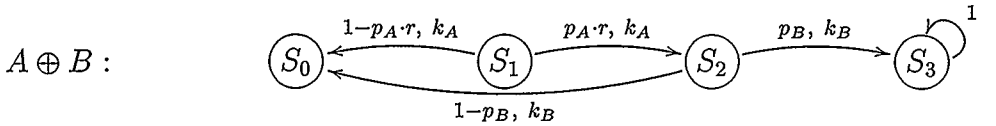


Figure 5.8: Algorithm $A \oplus B$

Thus, algorithm $A \oplus B$ is random$(p_A p_B r, k_A + k_B)$ self-stabilizing for predicate S_3 else S_0 given S_1. □

The next corollary says that the composition of algorithms that satisfy the conditions of Theorem 5.3.3 produce a random self-stabilizing algorithm for predicate S_3 given S_1 if $S_0 = S_1$. The proof is similar to Lemma 5.3.1.

Corollary 5.3.4. *Let S_1, S_2, and S_3 be nested predicates over configurations, and $A \oplus B$ be a fixed fair composition of algorithms A and B. Given the following four conditions:*

1. *Algorithm A is random(p_A, k_A) self-stabilizing for predicate S_2 else S_1 given S_1.*
2. *Algorithm B is random(p_B, k_B) self-stabilizing for predicate S_3 else S_1 given S_2.*
3. *Algorithm A is $(S_2, 1)$ left non-interfering with B via $A \oplus B$.*
4. *Algorithm B is (S_1, r) right non-interfering with A via $A \oplus B$.*

Then algorithm $A \oplus B$ is randomized self-stabilizing for predicate S_3 given S_1.

The following corollary gives the expected number of super rounds for $A \oplus B$ to converge.

Corollary 5.3.5. *Let S_1, S_2, and S_3 be nested predicates over configurations, and $A \oplus B$ be a fixed fair composition of algorithms A and B. Given the following four conditions:*

1. *Algorithm A is random(p_A, k_A) self-stabilizing for predicate S_2 else S_1 given S_1.*
2. *Algorithm B is random(p_B, k_B) self-stabilizing for predicate S_3 else S_1 given S_2.*
3. *Algorithm A is $(S_2, 1)$ left non-interfering with B via $A \oplus B$.*
4. *Algorithm B is (S_1, r) right non-interfering with A via $A \oplus B$.*

Then starting from any configuration satisfying S_1 and for any distributed schedulers, the expected number of super rounds for algorithm $A \oplus B$ to converge to a configuration satisfying S_3 is at most $\frac{k_B p_A r + k_A}{p_A r p_B}$.

Proof. Let $E[X, Y]$ denote the expected number of super rounds for $A \oplus B$ to converge to a configuration satisfying Y from an arbitrary configuration satisfying X. By linearity of expectation, $E[S_1, S_3]$ is the sum of $E[S_1, S_2]$ and $E[S_2, S_3]$.

From any configuration satisfying S_1, after k_A super rounds of algorithm $A \oplus B$, with probability at least $p_A r$ the new configuration satisfies S_2, and with probability no more than $1 - p_A \cdot r$ the system remains in a configuration satisfying S_1. Hence $E[S_1, S_2]$ is:

$$E[S_1, S_2] \leq k_A + (1 - p_A r) E[S_1, S_2]$$

therefore:

$$E[S_1, S_2] - (1 - p_A r) E[S_1, S_2] \leq k_A$$

implying

$$p_A r E[S_1, S_2] \leq k_A$$

$$E[S_1, S_2] \leq \frac{k_A}{p_A r} \qquad (1)$$

From any configuration satisfying S_2, after k_B super rounds of algorithm $A \oplus B$, with probability at least p_B the new configuration satisfies S_3, and with probability no more than $1 - p_B$ the system goes back to a configuration satisfying S_1. Hence $E[S_2, S_3]$ is:

$$\begin{aligned} E[S_2, S_3] &\leq k_B + (1 - p_B)(E[S_1, S_2] + E[S_2, S_3]) \\ &\leq k_B + (1 - p_B)(\frac{k_A}{p_A r} + E[S_2, S_3]) \ (by\ (1)). \end{aligned}$$

Therefore:

$$E[S_2, S_3] - (1 - p_B) E[S_2, S_3] \leq k_B + (1 - p_B) \frac{k_A}{p_A r},$$

implying:

$$p_B E[S_2, S_3] \leq k_B + (1 - p_B)\frac{k_A}{p_A r}$$

$$E[S_2, S_3] \leq \frac{k_B + (1 - p_B)\frac{k_A}{p_A r}}{p_B}. \qquad (2)$$

Thus the expected number of super rounds for algorithm $A \oplus B$ to converge to a configuration satisfying S_3 from an arbitrary configuration that satisfies S_1 is:

$$\begin{aligned} E[S_1, S_3] &= E[S_1, S_2] + E[S_2, S_3] \\ &\leq \frac{k_A}{p_A r} + \frac{k_B + (1 - p_B)\frac{k_A}{p_A r}}{p_B} \ (by\ (1)\ and\ (2)) \\ &= \frac{k_A p_B}{p_A r p_B} + \frac{k_B p_A r + (1 - p_B)k_A}{p_A r p_B} \\ &= \frac{k_A p_B + k_B p_A r + k_A - p_B k_A}{p_A r p_B} \\ &= \frac{k_B p_A r + k_A}{p_A r p_B} \end{aligned}$$

□

Generally, we can repeatedly compose a set of algorithms, which have desired properties, to build a composition that achieves the strongest of a nested sequence of predicates.

First we define a fair composition $\bigoplus_{i=1}^{k} A_i$ of a sequence of algorithms $\{A_1, A_2, ...\}$ inductively by:

1) $\bigoplus_{i=1}^{2} A_i = A_1 \oplus A_2$

2) $\bigoplus_{i=1}^{k} A_i = \bigoplus_{i=1}^{k-1} A_i \oplus A_k$

Because $\oplus$ denotes an arbitrary but fixed fair composition of two algorithms, $\bigoplus$ also represents an arbitrary but fixed fair composition of a sequence of algorithms.

Theorem 5.3.6. *Let $S_0, ..., S_m$ be a set of nested predicates over configurations. Let algorithms $A_1, ..., A_m$ have the following properties:*

1. *Algorithm A_i is random(p_i, k_i) self-stabilizing for predicate S_i else S_0 given S_{i-1}, for $i = 1, ..., m$.*
2. *Algorithm A_i is $(S_j, 1)$ left non-interfering with A_j via any fair composition $A_i \circ A_j$, for $i = 1, ..., m-1$ and $j = i+1, ..., m$.*
3. *Algorithm A_i is (S_1, r_i) right non-interfering with $\mathcal{A}_{i-1} = \bigoplus_{l=1}^{i-1} A_l$ via $\mathcal{A}_{i-1} \oplus A_i$, for $i = 2, ..., m$.*

Then the particular fair composition of algorithm A_1 to A_m, $\mathcal{A} = \bigoplus_{i=1}^{m} A_i$, is randomized self-stabilizing for predicate S_m given S_0.

Proof. By induction on the total number of algorithms m:

Basis: $m = 2$, the result is true by Corollary 5.3.4.

Induction steps: Suppose the theorem is true for $m = 1, ..., n-1$. And we show that the theorem still holds for $m = n$. Let algorithms $A_1, ..., A_n$ have properties 1, 2, and 3. By the induction hypothesis, algorithm $\bigoplus_{i=1}^{n-1} A_i$ is random $\left(\prod_{i=1,j=2}^{n-1,n-1} p_i r_j, \sum_{l=1}^{n-1} k_l \right)$ self-stabilizing for predicate S_{n-1} given (S_0, S_0). Because for $i = 1, ..., n-1$, algorithm A_i is $(S_n, 1)$ left non-interfering with A_n via any fair composition $A_i \circ A_n$, the fair composition of A_1 to A_{n-1}, $\bigoplus_{i=1}^{n-1} A_i$, is $(S_n, 1)$ left non-interfering with A_n via any fair composition $\bigoplus_{i=1}^{n-1} A_i \circ A_n$. Therefore by Corollary 5.3.4, a fair composition of algorithm $\bigoplus_{i=1}^{n-1} A_i$ and A_n, denoted as $\bigoplus_{i=1}^{n} A_i$, is random self-stabilizing for predicate S_m given S_0. □

Corollary 5.3.7. *Starting from any configuration satisfying predicate S_0 and for any distributed schedulers, algorithm $\mathcal{A}$ converges to a configuration satisfying S_m after an expected number $\frac{\sum_{l=1}^{m} k_l}{\prod_{i=1,j=2}^{m,m} p_i r_j}$ of super rounds.*

CHAPTER 6

Self-stabilizing Dining Philosophers

6.1 Motivation

In Chapter 2 we described several previous papers on the Dining Philosophers problem with different assumptions on system models. In these papers, only Choy and Singh's solutions are fault tolerant. Their algorithms limit the damage caused by a process's stop failure into a fixed range around that process. However Choy and Singh's algorithms are not self-stabilizing. Their algorithms must start from a special initial configuration, where every philosopher has a locally distinct label. Also the algorithms do not provide the mechanism to recover automatically from deadlocks caused by arbitrary initial configuration or failures of communication channels.

Gouda [14] presented a self-stabilizing solution to the Dining Philosophers problem in a ring model. In his algorithm, symmetry is broken by letting one of the philosophers behave differently from the others. Therefore the system is not completely symmetric.

Inspired by Gouda's work, Hoover and Poole [18] designed a self-stabilizing solu-

tion in the same topology. In their algorithm, every philosopher executes the same program. Symmetry is broken by using a token circling on the ring of philosophers. Only the philosopher holding the token is enabled and only enabled philosophers can execute the next operations of their programs. Thus their algorithm depends on a self-stabilizing token system, and it unnecessarily prevents concurrency between neighboring philosophers.

Both Gouda's algorithm and Hoover and Poole's algorithm solve a restricted version of the Dining Philosophers problem in systems with undesirable constraints. In this chapter, we solve a related problem and show how this problem when generalized can solve the general self-stabilizing Dining Philosophers problem in fully distributed and completely symmetric systems. The generalization of the related problem is not provided in details in this thesis. But the primary idea will be discussed in Section 7.2.

As indicated on page 25, any general Dining Philosophers problem can be reduced to the Restricted - Sharing Dining Philosophers problem. Beauquier, Datta, Gradinariu, and Magniette [3] presented a self-stabilizing solution for the Local Mutual Exclusion problem, which is similar to the Restricted - Sharing Dining Philosophers problem. Their algorithm, designed for fully distributed and completely symmetric systems, requires unbounded registers. In the paper they also gave an algorithm with bounded registers, but it apparently has flaws [5]. Even though the Local Mutual Exclusion problem has similar exclusion and progress requirements to the Dining Philosophers problem's, as shown later, algorithms for the Local Mutual Exclusion problem cannot efficiently solve the Dining Philosophers problem.

In the following sections, we first give the graph model of the Local Mutual

Exclusion problem, then compare it with the Dining Philosophers problem under the same model.

6.1.1 Local Mutual Exclusion

In a simple graph $G = (P, E)$, vertex set P represents a collection of processes. An edge (p, q) is in E if and only if $p, q \in P$ and processes p and q can communicate with each other.

Every process has a dynamic label called state. The state of a process $p \in P$, denoted as *state*(p), is in {entry, critical section, exit} and the only state transitions are entry $\rightarrow$ critical section $\rightarrow$ exit $\rightarrow$ entry. Transition from critical section to exit is spontaneous.

The global configuration is required to satisfy exclusion, which is a property that for any configuration, no two neighboring processes can be in the state "critical section" simultaneously.

Under the assumption that any process in the state "critical section" will change its state to "exit" later on, the global configuration is required to satisfy lockout-freedom, which is a property that for any configuration where *state*(p)=entry, there exists a subsequent configuration where *state*(p)=critical section.

6.1.2 Local Mutual Exclusion vs. Dining Philosophers

Comparing the general descriptions of these two problems, their similarities are revealed by both the exclusion and the lockout-freedom properties. In the Local Mutual Exclusion problem, no neighboring processes can be in the critical section at the same time, which is equivalent to no neighboring philosophers eating simulta-

neously. Also, the requirement that every process in the Local Mutual Exclusion problem gets a chance to enter the critical section is equivalent to the requirement that every hungry philosopher in the Dining Philosophers problem eats eventually.

In the Dining Philosophers problem, the transition from state thinking to hungry is spontaneous. A philosopher only interacts with her neighboring philosophers if she is hungry. She does not participate in the shared protocol while she is thinking. In contrast, the Local Mutual Exclusion problem requires that every process interacts with its neighbors whether or not it wants to access to the critical section. Therefore the Local Mutual Exclusion is essentially the same as a Dining Philosophers problem without the thinking state.

One intuition is to use a solution to the Local Mutual Exclusion problem as a subroutine to solve the Dining Philosophers problem. Every philosopher executes the solution to the Local Mutual Exclusion problem, which guarantees that the philosopher will enter the critical section. Whenever she does, if she is hungry, then she grabs all her forks and eats, otherwise she is thinking, so she exits from the critical section immediately. Thus using Beauquier and Datta's self-stabilizing algorithm, we can design a self-stabilizing solution to the Dining Philosophers problem. However algorithms constructed this way are not efficient, because a hungry philosopher may have to wait until all her neighbors enter their critical sections even if they are all thinking before she herself can eat.

In the following section, we present the techniques to build a self-stabilizing Dining Philosophers' solution in fully distributed and completely symmetric systems.

6.2 Self-Stabilizing Solution to the Dining Philosophers Problem

To build a self-stabilizing solution to the Dining Philosophers problem, we must have self-stabilizing techniques to accomplish both exclusion and lockout-freedom.

In shared memory settings, exclusion is usually ensured by using test-and-set objects or read/write variables with composite atomicity. In message passing settings, exclusion is ensured by assigning a single token to every fork so that only one philosopher can hold the token at a time. Because objects and variables themselves are correct and self-stabilizing token circulating is a solved problem, exclusion is easy to achieve in a self-stabilizing system.

In most of the papers we studied, lockout-freedom is ensured by applying priorities to philosophers that share one common fork. A philosopher p has higher priority than q on the fork shared between them is usually captured by a directed edge from q to p. The solutions provide a technique to ensure lockout-freedom as long as priorities do not form cycles. Therefore a cycle-free initial configuration is required. In the self-stabilizing setting, no assumption can be made on the initial configuration, priorities may form cycles at the beginning or after transient faults during the execution of the algorithms. As a result deadlock happens.

Suppose a cycle detection mechanism can detect and eliminate any cycles formed by priorities. Then combined with this extra mechanism, most solutions provide lockout-freedom in a self-stabilizing setting. In such solutions, the length of chains formed by priorities depends on either the size of the whole network, such as in Chandy and Misra's solution, or some local parameters, such as in Lynch's solution.

Therefore the self-stabilizing cycle detection protocol should be able to find cycles of any length up to the size of the network.

Choy and Singh [8] solved lockout-freedom in a different way. In their solutions, a synchronous doorway, a asynchronous doorway, and a fault-tolerant fork collection scheme are used. To make their solutions self-stabilizing, one needs to implement both kinds of doorways in a self-stabilizing way.

In the rest of this thesis, we choose the first course and design a self-stabilizing cycle detection algorithm that can find cycles of any length in a network.

6.2.1 Cycle Detection

Finding cycles of any length in a network becomes easy if every process has a globally distinct identifier. For simplicity, we assume that identifiers are just integers. Suppose each process p carries a local set W, which contains the identifiers of all the processes that can be reached through a directed path from p. Also p has a local variable called *id*, which is the identifier of p. A self-stabilizing cycle detection algorithm for a process p is shown in Figure 6.1.

Because in a completely symmetric system, all processes are identical and do not have distinct identifiers, our original goal to design a self-stabilizing solution to the Dining Philosophers problem reduces to assigning a globally distinct identifier to every process in a self-stabilizing manner.. Figure 6.2 sketches the idea to construct a self-stabilizing Dining Philosophers system. In this figure, nodes represent goals and a node's children are the subgoals required to achieve the parent goal. Note that all leaves are solved problems except the self-stabilizing distinct identifiers generation.

```
1: Do Forever:
2:     for every neighbor q such that there is a directed edge from p to q do
3:         send the set W ∪ {id} to q
4:     end for
5:     for every set S received from a neighbor do
6:         if id ∈ S then
7:             FOUND A CYCLE
8:         else
9:             W ← W ∪ S
10:        end if
11:    end for
12: End Do
```

Figure 6.1: Cycle Detection Based on Globally Distinct IDs

6.2.2 Assigning Distinct Labels up to Distance k

Self-stabilizing distinct identifier generation can be achieved by designing a self-stabilizing algorithm that assigns distinct labels up to distance k to processes in a network. When k equals the diameter of the network, every process has a different label from all the others.

Currently, there exist several such algorithms for $k = 1$ or 2. The idea of these algorithms is simple. When $k = 1$, every process keeps checking whether it has a distinct label from all its neighbors. If not, it randomly chooses a new label from a big range. For $k = 2$, every process keeps checking whether two of its neighbors have the same label. Because a process can distinguish one neighbor from another, two different neighbors must be at distance 1 or 2. If they have the same label, one of them should be notified to randomly choose a new label.

Assigning distinct labels up to distance 3 is much more difficult. Suppose a process x finds out that one of its neighbors, say y, has a neighbor labeled z, which

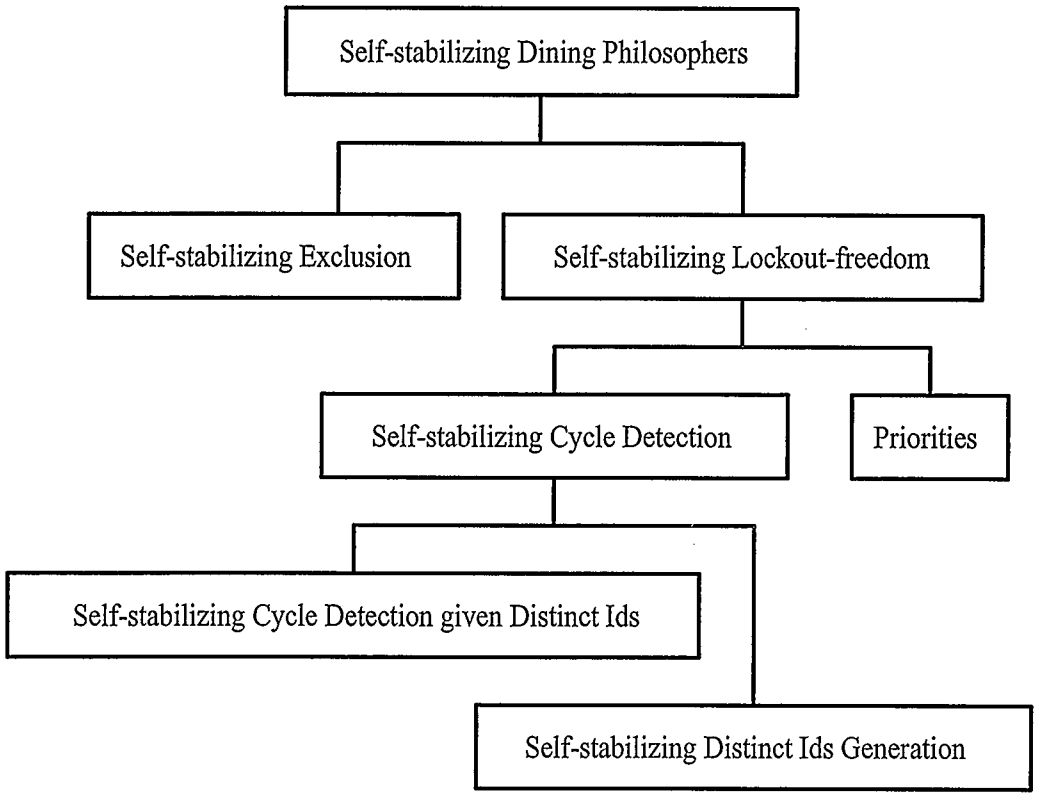


Figure 6.2: Self-stabilizing Dining Philosophers for Anonymous Networks

is the same label as another of x's neighbors (see Figure 6.3). Then there exist two possible cases:

1) Process x and y shares a common neighbor labeled z as shown in Figure 6.4.

2) Process x's neighbor labeled z and y's neighbor labeled the same are different processes as shown in Figure 6.5.

It is easy to see that case 1 is legal but case 2 should be eliminated because these two processes labeled z are at distance at most 3 and have the same label. But process x cannot tell which case really exists based on the information it collects. Similar situations may occur when $k \geq 4$. Therefore the major task is to develop a

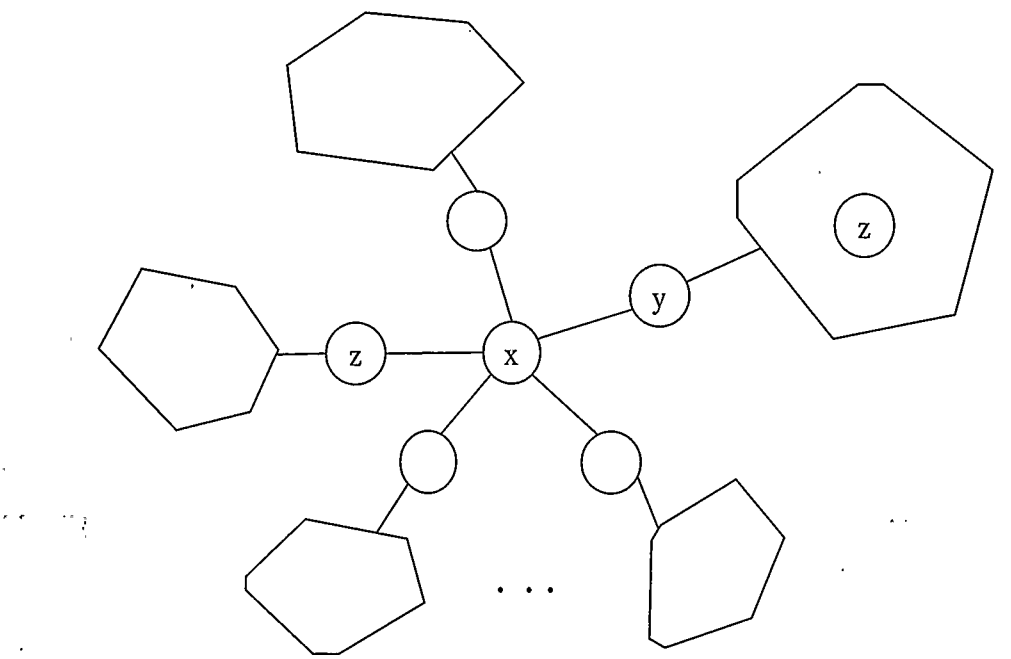


Figure 6.3: Distinct Labels up to Distance 3

technique to distinguish the above two cases. No previous research has been done on this in the self-stabilizing setting.

In the following section, we present a self-stabilizing algorithm for $k = 3$. Similar techniques but more involved can be applied to implement other cases where $k \geq 4$. This is further discussed in Section 7.2.

6.3 Assigning Distinct Labels up to Distance 3

The problem is to assign labels to every process in the system such that every pair of processes within distance 3 have distinct labels.

The system is modeled by an arbitrary graph $G = (P, E)$, where vertices represent

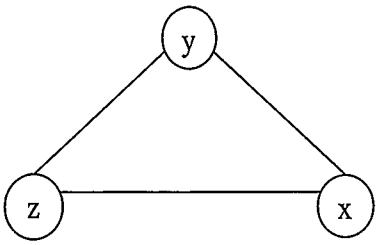


Figure 6.4: Case 1

Figure 6.5: Case 2

processes. Neighboring processes communicate with each other through shared link registers. The system works under read/write atomicity and is fully distributed and completely symmetric.

6.3.1 Solution Strategy

The strategy is to build a sequence of algorithms LD_i for $i = 1, 2, 3$, such that LD_i is randomized self-stabilizing for configurations where labels of processes are distinct up to distance i, given any configuration where labels of processes are distinct up to distance $i - 1$. Using Theorem 5.3.6 introduced in Chapter 5, we fairly compose LD_1, LD_2, and LD_3 together to obtain a randomized algorithm self-stabilizing for configurations where labels of processes are distinct up to distance 3. Algorithm LD_1 is easy to construct. Algorithm LD_2 is also quite straightforward. In fact, algorithms already exist in the literature [7, 15] that are randomized self-stabilizing for

configurations where each pair of processes within distance 2 has distinct labels given an arbitrary initial configuration. Call the algorithm presented by Chattopadhyay, Higham and Seyffarth [7] *Label_*D_2. Our strategy is to design an algorithm LD_3, which can be composed with Label_D_2 to obtain the objective algorithm *Label_*D_3. Using Corollary 5.3.4, we then prove given an arbitrary configuration, algorithm Label_D_3 is self-stabilizing for configurations where every pair of processes within distance 3 have distinct labels.

One possibility to build LD_3 would be to have every process compares its own label with the labels of others at distance 3. To do so a process collects information of all neighbors up to distance 3. To reduce the amount of information, another possibility is to let every process use knowledge of neighbors within distance 2. It compares the labels of its immediate neighbors and neighbors at distance 2. If a process x finds that its information is consistent with the case in Figure 6.3, it has to distinguish between cases in Figure 6.4 and 6.5. If the real situation is the case in Figure 6.5, then x alarms its neighbor labeled z. When a process receives an alarm, it randomly chooses a new label from a large range. The following sections give the data structure, algorithm, and proof of algorithm LD_3.

6.3.2 Data Structures

Each process has access to two kinds of variables, local variables and shared link registers. A local variable is only accessible to its owner. As described in Section 1.2.1, a link register is read and written by its owner and read by a neighbor. Since an anonymous system is usually assumed to be locally oriented (see page 8), every process in the system has a local name for each of the link registers shared with its

neighbors. Suppose a process x has δ_x neighbors and names every link from 1 to δ_x. It refers to the two link registers shared with a neighbor through link i as OUT_x^i and IN_x^i, where OUT_x^i is the register written by x and IN_x^i is the one written by the neighbor of x across link i. Notice that the neighbor of x through link i likely has different local names for these two registers. Tables 6.1 and 6.2 show the variables accessed by process x.

x.label	label of process x
$x.N$	a set containing labels of all x's neighbors
$x.M$	a set containing messages
x.alarm	a dangerous label of one of x's neighbor
x.counter_yz[1]	counter for cycle ($\langle x, y, z \rangle$) consistency check
$x.f$	a random coin flip

Table 6.1: Local Variable of Process x

OUT_x^i.label	label of process x
$\text{OUT}_x^i.N$	a set of labels of all x's neighbors
$\text{OUT}_x^i.M$	a set of messages for cycle consistency check
OUT_x^i.alarm	the neighbor's dangerous label

Table 6.2: Shared Link Register OUT_x^i

Correspondingly, register IN_x^i has the same fields as listed for OUT_x^i in Table 6.2. Assume x has a function, *link(y)*, which returns its index of its link to a neighbor labeled y. If the link does not exist, then the operation, in which this function is invoked, will be skipped.

[1] x.counter_yz is an atomic read/write variable associated with different pair of labels y and z. It is dynamically created and destroyed during the execution of the algorithm. There might exist any number of this type of variables in the system at a time.

6.3.3 . Algorithm LD_3

The algorithm LD_3 is a fair composition of the *main algorithm* and the *validity check*, where each step of the main algorithm and the validity check is executed alternatively.

Algorithm LD_3: $\langle$main algorithm$\rangle$	A
$\oplus$ $\langle$validity check$\rangle$	B

In the main algorithm, every process's responsibility is to detect and report dangerous labels of its immediate neighbors. In the validity check, every process needs to respond to the alarms it received and makes sure its information about its neighborhood is up-to-date and the information it wrote to every neighbor agrees with its neighborhood.

Even though the algorithm is designed in a link register model, sometimes it is more intuitive to present the algorithm using a message passing terminology. In the following sections, we describe the communications as processes sending information to their neighbors.

A: Main Algorithm

The main algorithm, itself, is a fair composition of $\langle$Check – 3 Cycle Consistent$\rangle$ and $\langle$Participate – 3 Cycle Consistent$\rangle$, where the steps of both parts are executed equivalently often.

Main Algorithm: $\langle$Check – 3 Cycle Consistent$\rangle$	$A.1$
$\oplus$ $\langle$Participate – 3 Cycle Consistent$\rangle$	$A.2$

The purpose of ⟨Check – 3 Cycle Consistent⟩ is to detect the situation shown in Figure 6.3, and when it occurs to distinguish case 1 and case 2 shown in Figure 6.4 and 6.5, respectively. The purpose of ⟨Participate – 3 Cycle Consistent⟩ is to assist ⟨Check – 3 Cycle Consistent⟩ by forwarding information initiated by a different process during the execution of ⟨Check – 3 Cycle Consistent⟩. Whenever an interruption is received from the validity check, the ⟨Check – 3 Cycle Consistent⟩ is interrupted and starts from scratch.

A.1: ⟨Check – 3 Cycle Consistent⟩

In the procedure ⟨Check – 3 Cycle Consistent⟩ (shown in Figure 6.6), a process x collects information from its direct neighbors and neighbors at distance 2. When it finds that a label z appears in both its direct neighborhood and one of its neighbors', say y's, neighborhood (see line 2), it tries to identify whether its neighbor labeled z and y's neighbor labeled the same are different processes or not. First x invokes a function ⟨symmetry⟩(*A*.1.1) (see line 3), which returns false if it found evidence showing that these two processes are different. Therefore, when the function returns false, x informs its neighbor labeled z to randomly choose a new label (see line 4). If ⟨symmetry⟩ returns true, then it cannot immediately distinguish the two processes labeled z. In this case another function ⟨mini_cycle_consistency⟩(*A*.1.2) is initiated by the process with the smallest label in $\{x, y, z\}$ (see line 5 and 6). Function ⟨mini_cycle_consistency⟩ uses randomization to distinguish x's neighbor z and y's neighbor labeled z. When ⟨mini_cycle_consistency⟩ returns false, x alarms one of its neighbors (see line 7).

```
1: for i ← 1, 2, ... δ_x, do
2:   for all z ∈ x.N ∩ IN_x^i.N do
3:     if ¬ symmetry(z, IN_x^i.label) then
4:       OUT_x^link(z).alarm ← z
5:     else if x.label< min{z, IN_x^i.label} then
6:       if ¬ mini_cycle_consistency(min{z, IN_x^i.label}, max{z, IN_x^i.label})
         then
7:         OUT_x^link(x.alarm).alarm ← x.alarm
8:       end if
9:     end if
10:  end for
11: end for
```

Figure 6.6: Procedure ⟨Check – 3 Cycle Consistent⟩

A.1.1: ⟨symmetry⟩

The function ⟨symmetry⟩ (shown in Figure 6.7) checks whether x's neighbor labeled z has a neighbor labeled y. If it does not, then the function returns false, which indicates x's and y's neighbor labeled z must be different processes, because they have different neighborhoods.

```
1: symmetry(z, y) ≡ y ∈ IN_x^link(z).N
```

Figure 6.7: Function ⟨symmetry(z, y)⟩

When ⟨symmetry⟩ returns true, either x and y have a common neighbor z (as shown in Figure 6.4), or one of the following cases is true.

Case 1: there exists a long path with repeated occurrence of labels x, y, and z (shown in Figure 6.8).

Case 2: there exists a multiple cycle with repeated occurrence of labels x, y, and z (shown in Figure 6.9).

If the real situation is case 1, then ⟨symmetry⟩ when invoked by the processes

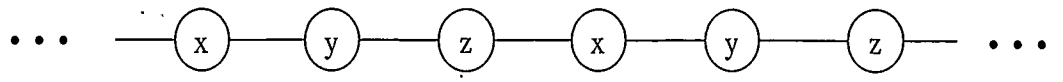


Figure 6.8: long path with x, y, z

next to both ends of the path will return false. As a result, these two processes will send alarms to their neighbors, which are the two ends of the path. With high probability, they will choose a label other than x, y, or z, and therefore leave the path. Similar things will happen on the shortened path, and if every process chooses well, the path eventually will disappear.

If the real situation is case 2, then no process on the multiple cycle can deterministicly distinguish this from the minimum cycle shown in Figure 6.4. In this case randomization is employed in the function ⟨mini_cycle_consistency⟩. To avoid every process on the multiple cycle initiate the ⟨mini_cycle_consistency⟩, only the one with minimum label does so.

A.1.2: ⟨**mini_cycle_consistency**⟩

This function (shown in Figure 6.10) returns a boolean variable but also modifies a local variable x.alarm.

The major task of ⟨mini_cycle_consistency⟩ is to distinguish Figure 6.9 and 6.4. When process x invokes function ⟨min_cycle_consistency(y, z)⟩, it increases its counter by one and randomly flips a coin (see line 1 and 2), then it sends a message containing its updated counter and coin flip to its neighbor y.

As will be discussed soon, every process executing ⟨Participate – 3 Cycle Consistent⟩

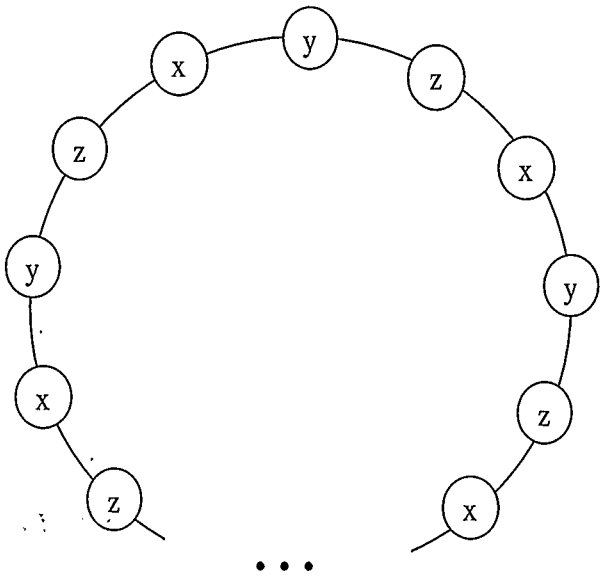


Figure 6.9: multiple cycle with x, y, z

forwards such messages initiated by a different process. If the minimum cycle shown in Figure 6.4 exists, then the message will be eventually delivered back to x. If the multiple cycle shown in Figure 6.9 exists, then x sends its message to the next x along the cycle. With probability a half, x generated a different coin flip from the proceeding process labeled x on this cycle.

Because the system is asynchronous, the counter helps x distinguish the messages of the current invocation of ⟨mini_cycle_consistency⟩ from the messages of the previous invocation. If the counter in the incoming message is 1 less than x's local counter, x will consider it as an old message from the last consistency check and will wait for it to be updated.

If x receives a message from its neighbor z, which contains the same counter and

1: $\langle$increment_counter$(y, z)\rangle$ *A.1.2.1*
2: $x.f \leftarrow$ random coin flip
3: Let cond1$(M) \equiv (\exists\ \langle c, A, B, C, f\rangle \in M) \wedge (A = z) \wedge (B = x.\text{label}) \wedge (C = y)$
$\wedge\ (c \neq x.\text{counter\_yz}-1)$
4: cond2 $\equiv$ symmetry_check (z, y)
5: cond3 $\equiv$ symmetry_check (y, z)
6: **repeat**
7: $\text{OUT}_x^{\text{link}(y)}.M \leftarrow \text{OUT}_x^{\text{link}(y)}.M \uplus \langle x.\text{counter\_yz}, x.\text{label}, y, z, x.f\rangle$
8: $x.M \leftarrow \text{IN}_x^{\text{link}(z)}.M$
9: **until** $(\neg\text{cond1}(x.M)) \vee (\neg\text{cond2}) \vee (\neg\text{cond3})$
10: **if** cond1$(x.M)$ **then**
11: **for all** $\langle c, A, B, C, f\rangle \in M \wedge A = \text{IN}_x^{\text{link}(z)}.\text{label} \wedge B = x.\text{label} \wedge C = \text{IN}_x^{\text{link}(y)}.\text{label}$ **do**
12: $\text{OUT}_x^{\text{link}(y)}.M \leftarrow \text{OUT}_x^{\text{link}(y)}.M \backslash \langle c, B, C, A, f\rangle$
13: **if** $c = x.\text{counter\_yz} \wedge f = x.f$ **then**
14: return true
15: **else**
16: $x.\text{alarm} \leftarrow z$
17: return false
18: **end if**
19: **end for**
20: **else if** cond2 **then**
21: $x.\text{alarm} \leftarrow z$
22: free x.counter_yz
23: return false
24: **else**
25: $x.\text{alarm} \leftarrow y$
26: free x.counter_yz
27: return false
28: **end if**

Figure 6.10: Function $\langle$mini_cycle_consistency$(y, z)\rangle$

the same coin flip as its own local variables (see line 13), x considers this message to be the one initiated by itself. Process x's information is consistent with the case in Figure 6.4. Therefore function ⟨mini_cycle_consistency⟩ returns true (see line 14).

If the counter of the incoming message from z coincides with x's local counter but the coin flip does not, or the counter does not equal to x's local variable and is not one smaller than x's local variable, then x knows this is a message from another process with the same label. Therefore x has the evidence of the existence of a multiple cycle (or it is embedded in a long path). Function ⟨mini_cycle_consistency⟩ sets x's local alarm (x.alarm) to z and returns false (see line 16 and 17).

While x is waiting for the message from z, it keeps invoking ⟨symmetry(z, y)⟩ and ⟨symmetry(y, z)⟩. If one of them returns false, then x knows the symmetry case has disappeared. It sets its local alarm (x.alarm) to the dangerous label (see line 21 and 25) and destroy the local variable x.counter_yz (see line 22 and 26). Function ⟨mini_cycle_consistency⟩ also returns false (see line 23 and 27).

All messages traveling among processes have the following format: $\langle c, A, B, C, f \rangle$, where A, B, and C are the labels of the sender, the receiver, and the process to which this message will be forwarded, respectively. Variables c and f are the local counter and coin flip of the sender. The message is initiated by the process with label $\min\{A, B, C\}$.

We introduce a send-overwrite operation denoted as $\uplus$, which is used whenever a process x writes a message to some neighbor. Operation $\text{OUT}_x^i.M \leftarrow \text{OUT}_x^i.M \uplus$ ⟨counter,r,s,t,flip⟩ is defined in Figure 6.11.

This operation will overwrite any message in the destination set with the same middle three fields but different counter or coin flip.

1: **for all** $\langle c, A, B, C, f\rangle \in \text{OUT}_x^i.M$ **do**
2: **if** $(A = r) \wedge (B = s) \wedge (C = t) \wedge (c \neq \text{counter} \vee f \neq \text{flip})$ **then**
3: $\text{OUT}_x^i.M \leftarrow \text{OUT}_x^i.M \setminus \langle c, r, s, t, f\rangle$
4: **end if**
5: **end for**
6: $\text{OUT}_x^i.M \leftarrow \text{OUT}_x^i.M \cup \langle \text{counter}, r, s, t, \text{flip}\rangle$

Figure 6.11: Operation $\text{OUT}_x^i.M \leftarrow \text{OUT}_x^i.M \uplus \langle \text{counter}, r, s, t, \text{flip}\rangle$

A.1.2.1: $\langle$**increment_counter**$(y, z)\rangle$

Process x has a variable x.counter_yz that counts how many times x invokes the function $\langle$mini_cycle_consistency$\rangle$ to detect a particular cycle $\langle x, y, z\rangle$. This counter helps x to distinguish between an old message from a previous invocation and an updated message from the current invocation.

When x invokes $\langle$mini_cycle_consistency$\rangle$ for cycle $\langle x, y, z\rangle$, if the corresponding counter does not exist, then x creates one (see line 2). Otherwise x increases the counter by one then modulo by n, which is the size of the network (see line 4).

As will be discussed later, this local counter will be destroyed in the validity check when x has an evidence showing that cycle $\langle x, y, z\rangle$ does not exist.

1: **if** x.counter_yz does not exist **then**
2: new x.counter_yz
3: **else**
4: x.counter_yz $\leftarrow (x.\text{counter\_yz} + 1) \bmod (n)$
5: **end if**

Figure 6.12: Function $\langle$increment_counter$(y, z)\rangle$

A.2: $\langle$**Participate – 3 Cycle Consistent**$\rangle$

In $\langle$Participate – 3 Cycle Consistent$\rangle$ (shown in Figure 6.13), a process forwards all the messages initiated by $\langle$Check – 3 Cycle Consistent$\rangle$ by a different process.

For every incoming message, if the middle three fields are consistent with x's local information and x is not the minimum label (see line 3), then x treats it as a valid message from another process and forwards it to the targeted neighbor (see line 4).

1: **for** $i \leftarrow 1, 2, \ldots, \delta_x$ **do**
2: **for all** $\langle c, A, B, C, f \rangle \in \text{IN}_x^i.M$ **do**
3: **if** $A = \text{IN}_x^i.\text{label} \wedge B = x.\text{label} \wedge C \in x.N \wedge B > \min\{A, C\}$ **then**
4: $\text{OUT}_x^{\text{link}(y)}.M \leftarrow \text{OUT}_x^{\text{link}(y)}.M \uplus \langle c, B, C, A, f \rangle$
5: **end if**
6: **end for**
7: **end for**

Figure 6.13: Procedure ⟨Participate – 3 Cycle Consistent⟩

B: Validity Check

In validity check, a process x invokes four functions: ⟨Check for Alarm⟩, ⟨Collect and Update Neighbors' Labels⟩, ⟨Message Consistency Check⟩, and ⟨Counter Consistency Check⟩. The program of validity check is as follows:

1: **Do Forever:**
2: ⟨Check for Alarm⟩ $B.1$
3: ⟨Collect and Update Neighbors' Labels⟩ $B.2$
4: ⟨Message Consistency Check⟩ $B.3$
5: ⟨Counter Consistency Check⟩ $B.4$
6: **End Do**

$B.1$ ⟨Check for Alarm⟩

In function ⟨Check for Alarm⟩ (shown in Figure 6.14), a process x checks whether it received any alarm that contains the same value as x's own label (see line 2). If so, the alarm is called a *valid alarm* and x randomly chooses a new label from a

large range $\{1, ..., R\}$ (line 3), erases all the messages it wrote to its neighbors (line 5 and 6), and send an interruption to the main algorithm (line 7). We assume R is at least as big as $2\Delta^6(3\Delta^2 + 1)$ where Δ is the maximum degree of any process in the network.

```
1: for i ← 1, 2, ..., δ_x do
2:    if IN_x^i.alarm = x.label then
3:       x.label ← random number from 1 to R ≥ 2Δ^6(3Δ^2 + 1)
4:       for i ← 1, 2, ..., δ_x do
5:          OUT_x^i.label ← x.label
6:          OUT_x^i.M ← ∅
7:          send interruption to main algorithm
8:       end for
9:    end if
10: end for
```

Figure 6.14: Function ⟨Check for Alarm⟩

B.2 ⟨Collect and Update Neighbors' Labels⟩

In ⟨Collect and Update Neighbors' Labels⟩ (shown in Figure 6.15), a process x collects labels from all its neighbors (see line 1), and update the information it sends to every neighbor (see line 3 and 4).

```
1: x.N ← ⋃_{1≤i≤δ_x} IN_x^i.label
2: for i ← 1, 2, ..., δ_x do
3:    OUT_x^i.label ← x.label
4:    OUT_x^i.N ← x.N
5: end for
```

Figure 6.15: Function ⟨Collect and Update Neighbors' Labels⟩

$B.3$ ⟨Message Consistency Check⟩

In this function (shown in Figure 6.16), a process x checks every message $\langle c, A, B, C, f\rangle$ it wrote to its neighbor. If one of the following conditions is true (see line 3), then x erases the message (see line 4) and sends an interruption to the main algorithm (see line 5):

- The neighbor to which x wrote this message has a label different from B
- x does not have a neighbor with label C
- x's label is the minimum among the three labels, and c or f does not coincide with x's local counter and coin flip.
- x's label is not the minimum among the three labels, and there is no corresponding source from neighbor labeled with C

1: **for** $i \leftarrow 1, 2, \ldots, \delta_x$ **do**
2: **for all** $\langle c, A, B, C, f\rangle \in \mathrm{OUT}_x^i.M$ **do**
3: **if** $(\mathrm{IN}_x^i.\mathrm{label} \neq B) \vee (C \notin x.N) \vee (A < \min\{B, C\} \wedge (c \neq x.\mathrm{counter} \vee f \neq x.f)) \vee (A > \min\{B, C\} \wedge \langle c, C, A, B, f\rangle \notin \mathrm{IN}_x^{\mathrm{link}(C)}.M)$ **then**
4: $\mathrm{OUT}_x^i.M \leftarrow \mathrm{OUT}_x^i.M \backslash \langle r, A, B, C, f\rangle$
5: send interruption to main algorithm
6: **end if**
7: **end for**
8: **end for**

Figure 6.16: Function ⟨Message Consistency Check⟩

$B.4$ ⟨Counter Consistency Check⟩

The purpose of a local variable x.counter_yz is to count how many times that x invoked function ⟨mini_cycle_consistency⟩ to detect the existence of a particular

cycle $\langle x, y, z \rangle$. When x does not has a neighbor y or z (see line 2) anymore, it is straightforward that such a cycle does not exist. In this case x will not invoke function $\langle$mini_cycle_consistency$(y, z)\rangle$. Therefore the local variable x.counter_yz should be destroyed (see line 3).

```
1: for all x.counter_yz do
2:    if y ∉ x.N ∨ z ∉ x.N then
3:       free x.counter_yz
4:    end if
5: end for
```

Figure 6.17: Function ⟨Counter Consistency Check⟩

6.3.4 Proof Outline

First define *i-local-secure* and *i-local-insecure* for some positive integer i.

Definition 6.3.1. *A process is* i-local-secure *if and only if its label is distinct from all others within distance* i *and is also distinct from all labels in shared link registers or local variables of processes within distance* i. *Otherwise the process is called* i-local-insecure.

Then define a predicate i-secure for some integer i over configurations as follows:

Definition 6.3.2. *Let c be a configuration.* i-secure$(c) \equiv$ *every process in the system is* i-local-secure *in configuration c*

Predicate 0-secure means there is no restriction on the label of each process. It is easy to see that a sequence of such predicates 1-secure, 2-secure,... are nested.

The self-stabilizing algorithm Label_D_2 [7] assigns to every process a label such that processes within distance 2 have distinct labels. This algorithm is random-

ized self-stabilizing for 2-secure given 0-secure. By using the technique presented in Chapter 5, we will show that starting from any arbitrary configuration, given any configuration satisfying 0-secure, a fair composition of Label_D_2 and LD_3 is random self-stabilizing for 3-secure, which implies that every pair of processes within distance 3 have distinct labels. Based on Corollary 5.3.4, we need to establish that there exists a particular fair composition Label_$D_2 \oplus \mathrm{LD}_3$, also called Label_D_3, that satisfies the following three requirements.

1. Algorithm LD_3 is random(p, k) self-stabilizing for predicate 3-secure else 0-secure given 2-secure.

2. Algorithm LD_3 is (0-secure, r) right non-interfering with Label_D_2 via Label_D_3.

3. Algorithm Label_D_2 is (2-secure, 1) left non-interfering with algorithm LD_3 via Label_D_3.

Algorithm Label_D_2 does not change labels of any processes at all in any configuration satisfying 2-secure. Thus the requirement 3 is satisfied for any fair composition of Label_D_2 and LD_3. The next two sections establish the first two requirments respectively.

6.3.5 Self-stabilization of LD_3

To demonstrate that algorithm LD_3 is random(p, k) self-stabilizing for predicate 3-secure else 0-secure given 2-secure, for $0 < p \leq 1$ and a positive integer k, we need to prove the probabilistic convergence property and closure property. Because the algorithm has read/write atomicity, a process A may delay arbitrarily between reading a shared register from one neighbor B and writing to a shared link register

of another neighbor C, even when these are consecutive steps in A's algorithm. Thus process A may convey out-of-date information to C. This could, for example, cause process C to send an alarm unnecessarily. One of the subtleties of the proof is to show that such phenomena do not cause serious problems. The notion of 3-local-secure and 3-local-insecure processes is used for the proofs of both convergence and closure.

Closure Property

Closure can be proved by establishing that once predicate 3-secure holds, no process will receive an alarm.

Theorem 6.3.1. *Starting from any configuration satisfying* 3*-secure, for any distributed scheduler, all subsequent configurations of algorithm* LD_3 *satisfies* 3*-secure.*

Proof. In any configuration satisfying 3-secure, if a process x found that its neighbor y has a neighbor labeled the same as one of x's other neighbor z, then process x, y, and z form a cycle. Without loss of generality, assume x.label $<$ y.label $<$ z.label, then x will invoke $\langle$ mini_cycle_consistency(y.label, z.label)$\rangle$. Both y and z will execute $\langle$Participate – 3 Cycle Consistent$\rangle$. The message initiated by x will be copied back to it eventually with the same counter and coin flip. Therefore based on line 13 of Figure 6.10, $\langle$ mini_cycle_consistency(y.label, z.label)$\rangle$ will return true. Process x will not set an alarm. □

Probabilistic Convergence Property

Convergence is more difficult to prove. We prove it by showing that starting from a configuration satisfying 2-secure but not 3-secure, after a bounded number of super rounds of algorithm LD_3, either 3-secure becomes true, or some 3-local-insecure

process chooses a new label. If every 3-local-insecure process is lucky enough to become 3-local-secure after choosing a new label, then a configuration satisfying 3-secure is reached. Otherwise a 3-local-insecure process may choose a label of some neighbor within distance 3. It may even choose a label within distance 2, thus falsifying 2-secure. However 0-secure is always maintained. Our proof contains the following two steps:

1) *Choices Happen*, in which we show that if the system stays in configurations satisfying 2-secure but not 3-secure for a bounded time, then some process chooses a new label.

2) *Convergence Happens*, in which we model algorithm LD_3 with a Markov process, and show that with probability 1 the system converges to 3-secure.

Choices Happen We will establish the following: if the current configuration c satisfies 2-secure but not 3-secure, then let algorithm LD_3 run for a finite number of super rounds. Either some process chooses a new label within these super rounds, or predicate 3-secure becomes true, or starting from the new configuration some process will choose a new label shortly.

First we introduce some useful definitions.

Definition 6.3.3. *All registers accessed by process x* match with reality, *if:*

1. $x.N = \bigcup_{y \in N(x)} y.label.$
2. $OUT_x^i.label = x.label$ *for* $i = 1, ..., \delta_x$
3. $OUT_x^i.N = x.N$
4. $\forall i \in [1, \delta_x]$ *and* $\forall \langle c, A, B, C, f \rangle \in OUT_x^i.M$, *the following three conditions hold:*

(a) $A = x.label$, $B = IN_x^i.label$, *and* $C \in x.N$

(b) if $A < min\{B, C\}$ *then* $c = x.counter$ *and* $f = x.f$

(c) if $A > min\{B, C\}$ *then* $\langle r, C, A, B, f\rangle \in IN_x^{link(C)}.M$

Conditions 1, 2, and 3 ensure that x has up-to-date knowledge of all neighbors' labels and broadcasts up-to-date information to all its neighbors. Condition 4 ensures all messages written by x have correct information of the real system. Condition $4(a)$ ensures that the middle three components are consistent with the sender, the receiver and one of the sender's neighbor, respectively. Condition $4(b)$ ensures that if x initiated the message, then the counter and coin flip are consistent with x's local values. Condition $4(c)$ ensures that if x is only passing the message, then there exists a source from which x received the information.

Define a predicate LC over configurations as follows:

Definition 6.3.4. *Predicate* $LC \equiv$ *the values in all shared link registers or local variables match with reality.*

Definition 6.3.5. A multiple cycle with repeated pattern $\langle x, y, z\rangle$ *is a cycle* $(p_1, ..., p_n)$, *such that* $n \geq 6$ *and is a multiple of* 3 *and the labels of* p_1 *to* p_n *can be represented by the following regular expression:*

$$(xyz)^+$$

Definition 6.3.6. A long path with repeated pattern $\langle x, y, z\rangle$ *is a sequence of processes* $(p_1, ..., p_n)$, *such that* $n \geq 4$ *and the labels of* p_1 *to* p_n *can be represented by the following regular expression:*

$$(\epsilon \cup z \cup yz) \cdot (xyz)^+ \cdot (x \cup xy \cup \epsilon)$$

Lemma 6.3.2. *Within 3 super rounds of algorithm LD_3, if no process chooses new label, then predicate LC becomes true.*

Proof. In the first super round, every process will broadcast its own label to its direct neighbors, which implies $\forall x \in P$, $\text{OUT}_x^i = x.\text{label}$ for $i = \{1, ..., \delta_x\}$. In the second super round, every process collects the updated label of its direct neighbors and broadcast this updated information, which implies $\forall x \in P$, $x.N$ and $\text{OUT}_x^i.N$ contain the real labels of x's direct neighbors for $i = \{1, ..., \delta_x\}$. Based on these correct registers, in the third super round, every process x will clear up old messages in $\text{OUT}_x^i.M$ for $i = \{1, ..., \delta_x\}$. Therefore after three super rounds of algorithm LD_3, if no process chooses a new label, then the values in all shared link register and local variables of every process in the system match with reality, which implies predicate LC is true. □

In any configuration where 2-secure $\wedge$ LC $\wedge$ ($\neg$ 3-secure) holds, if a process with label x notices that its neighbors labeled y and z each has a neighbor labeled z and y respectively, and $x < y < z$, it will initiate a $\langle \text{mini\_cycle\_consistency}(y, z) \rangle$. There are three possible situations:

1. The process and its neighbor labeled y and z form a cycle of length 3.

2. There exists a long path with repeated pattern $\langle x, y, z \rangle$

3. There exists a multiple cycle with repeated pattern $\langle x, y, z \rangle$

Case 1 is legal, and it is easy to check from the algorithm that nobody changes its label in this case. Case 2 and 3 need to be eliminated. Lemmas 6.3.3 and 6.3.4 show how the elimination is achieved.

Lemma 6.3.3. *In any configuration satisfying* 2*-secure* $\wedge$ *LC, if there exists a long path with repeated pattern* $\langle x, y, z \rangle$*, then within two super rounds, the processes at each end of the path will choose a new label.*

Proof. Let $(p_1, ..., p_m)$ be a maximal length path containing the repeated pattern $\langle x, y, z \rangle$. Process p_2 will notice that its neighbor p_3 has a neighbor p_4 labeled the same as p_1, while p_1 does not have a neighbor labeled the same as p_3. Therefore $\langle$symmetry$\rangle$ of process p_2 returns false, and p_2 informs p_1 to choose a new label. Process p_{m-1} is in the similar situation, and will also alarm its neighbor p_m. Processes p_1 and p_m will receive the alarm in the execution of $\langle$Check for Alarm$\rangle$ in the next super round, and will randomly choose a new label. □

Lemma 6.3.4. *In any configuration satisfying* 2*-secure* $\wedge$ *LC, if there exists a multiple cycle with repeated pattern* $\langle x, y, z \rangle$*, then the expected number of super rounds of algorithm* LD_3 *before at least one process on the cycle chooses a new label is at most* 3*.*

Proof. Without loss of generality, suppose $x < y < z$. If a process with label x has the same counter as its successor, then with probability a half, they will generate different coin flips. The $\langle$mini_cycle_consistency$\rangle$ of its successor will return false (see line 17 of Figure 6.10) in 2 expected number of super rounds.

If a process with label x has a counter that is not the same as or that is not one smaller than its successor's counter, then within one super round, the $\langle$mini_cycle_consistency$\rangle$ of its successor will return false.

The only remaining case is that a process with label x has a counter that is one smaller than its successor. In this case its successor will consider its incoming

message as an out-of-date message from itself, and will wait for an updated message. Because the counter is incremented modulo a number, which is bigger than one third the net work size, it is impossible for all processes with label x on the cycle to have a counter one bigger modulo the number than their predecessor. Therefore at least one process will receive a message with a counter that is not equal to or one smaller than its local value. One of the previous two cases apply to this process.

As a result, within expected 2 super rounds of LD_3, at least one process in with label x sets an alarm to one of its neighbor. That neighbor will receive the alarm in the next execution of validity check, and will randomly choose a new label. □

Corollary 6.3.5 follows immediately from Lemmas 6.3.3 and 6.3.4 and correctness of ⟨symmetry⟩.

Corollary 6.3.5. *In any configuration satisfying* 2*-secure* $\wedge$ LC $\wedge$ ($\neg$ 3*-secure), within an expected* 3 *super rounds of algorithm* LD_3 *at least one process will choose a new label.*

Theorem 6.3.6. *In any configuration satisfying* 2*-secure, if there exists some* 3*-local-insecure processes, within expected* 6 *super rounds of algorithm* LD_3*, either all processes become* 3*-local-secure or at least one process chooses a new label.*

Proof. From any configuration where some processes are 3-local-insecure, after 3 super rounds of algorithm LD_3, either some process chooses a new label or none of them do. If nobody chooses a new label, by Lemma 6.3.2 predicate LC is true. Now either all processes are secure or there still exists a 3-local-insecure process. If the latter case is true, then by Corollary 6.3.5, at least one process chooses a new label. □

Convergence Happens

We first introduce some useful definitions.

Define the neighborhood of a process up to distance i as follows:

Definition 6.3.7. *For a process x define:* $N_i(x) \equiv \{y \in P | 1 \leq d(x,y) \leq i\}$.

The set $N_i(x)$ contains all x's neighbors within distance i. Clearly, $N_1(x)$ (normally abbreviated as $N(x)$) represents the set of immediate neighbors of process x.

Definition 6.3.8. *For a process x define: δ_x^i = maximum degree of any process $y \in N_i(x)$*

Clearly, δ_x^0 (normally abbreviated as δ_x) is the degree of process x. Let Δ_x be δ_x^2, which is the maximum degree among x and its neighbors within distance 2.

Definition 6.3.9. *For a process x define:*

$$L_i(x) \equiv \bigcup_{y \in N_i(x)} y.label \cup \bigcup_{y \in N_i(x)} y.N \cup \bigcup_{y \in N(x)} y.alarm \cup$$

$$\bigcup_{j=1}^{\delta_x} IN_x^j.alarm \cup \bigcup_{k,l \in N_i(x) \text{ and } k \in N(l)} IN_l^k.label$$

The set $L_i(x)$ consists of all the possible labels existing in the shared link registers or local variables of x's neighbors within distance i. By assuming there exists an edge between every pair of processes in $N_i(x)$, we can give an upper bound on the size of $L_i(x)$. In this algorithm we are only concerned with $L_2(x)$ and $L_3(x)$, whose sizes are bounded by $2(\delta_x^1)^4$ and $2(\delta_x^2)^6$, respectively when $\delta_x^1, \delta_x^2 > 2$, which is overestimated again by $2(\Delta_x)^4$ and $2(\Delta_x)^6$, respectively.

When a process x chooses a new label from the integer set from 1 to M, with probability at least $1 - \frac{|L_2(x)|}{M}$, x's new label is distinct from all neighbors within distance 2, which implies predicate 2-secure is maintained, otherwise 2-secure becomes false.

In this section we prove the convergence behavior of algorithm LD_3 and make a conditional analysis under the following assumption.

Assumption 6.3.7. *Whenever a process $x \in P$ chooses a new label, it never chooses a label in $L_2(x)$.*

Assumption 6.3.7 indicates that whenever a process chooses a new label, it always stays 2-local-secure. Although this assumption is not always true, it happens with high probability. We will remove the conditioning on this assumption later.

If an 3-local-insecure process chooses a new label, then with probability $1 - \frac{|L_3(x)|}{M}$, x becomes 3-local-secure. Otherwise under Assumption 6.3.7 it may stay 3-local-insecure and possibly make some 3-local-secure neighbors 3-local-insecure by choosing their labels. Because a 3-local-secure process has a label different from all existing information of its neighbors within distance 3, a process will not choose a new label while it is 3-local-secure.

Lemma 6.3.8. *If an 3-local-insecure process x chooses a label and is still 3-local-insecure, it causes at most $(\Delta_x)^2$ 3-local-secure processes to become 3-local-insecure.*

Proof. Let $N(x) = \{y_1, \ldots, y_m\}$ for some process x. If $z \in N_2(y_i)$ is a 3-local-secure process, then there can be at most δ_y^0 3-local-secure processes with the same label as z in $N_2(y_i)$. Hence there can be at most $\delta_x^0 \cdot \delta_x^1 \leq (\Delta_x)^2$ 3-local-secure processes in $N_3(x)$

all having the same label. If process x is 3-local-insecure and chooses the label z, then it remains 3-local-insecure and makes at most $(\Delta_x)^2$ processes 3-local-insecure. □

Lemma 6.3.9. *The probability that a 3-local-insecure process x chooses a label and it is still 3-local-insecure is at most $\frac{1}{3(\Delta_x)^2+1}$.*

Proof. A process x chooses a new label uniformly from the set $\{1, ..., R\}$. Since $R \geq 2(\Delta_x)^6(3(\Delta_x)^2+1)$, the probability that process x chooses a label in $L_3(x)$ and stays insecure is less than

$$\begin{aligned} \frac{|L_3(x)|}{M} &\leq \frac{2(\Delta_x)^6}{2(\Delta_x)^6(3(\Delta_x)^2+1)} \\ &= \frac{1}{3(\Delta_x)^2+1} \end{aligned}$$

□

Therefore the probability for any 3-local-insecure process to stay 3-local-insecure after choosing a new label is at most $\frac{1}{3\Delta^2+1}$.

To prove the convergence of algorithm LD_3, we model it as a Markov Process. The expected number of steps is shown to be overestimated by a biased random walk, which, in turn, is shown to use only $2n$ expected steps.

Define the following Markov process:

d-Penalty(p_N) Walk: The states are $\{s_0, \ldots, s_n\}$ such that the probability $\Pr[s_i, s_j]$ of moving from state s_i to s_j at any step is given by:

$$\forall 0 \leq i < n, \Pr[s_i, s_{i+1}] = p_N$$

$$\forall 0 \leq i \leq d, \Pr[s_i, s_0] = q_N$$

$$\forall d < i < n, \Pr[s_i, s_{i-d}] = q_N$$

$$\Pr[s_n, s_n] = 1$$

where $p_N + q_N = 1$.

Observation 6.3.10. *Let s_i denote a configuration where i processes in the system are 3-local-secure. Algorithm LD_3's convergence behavior can be simulated by a d-Penalty(p_N) Walk, where $d = \Delta^2$ and $p_N = 1 - \frac{1}{3\Delta^2+1}$. This Markov process underestimated the convergence speed of algorithm LD_3 by assuming that every time a 3-local-insecure process chooses a label and is still 3-local-insecure, it drags Δ^2 neighbors from 3-local-secure to 3-local-insecure.*

Define a second Markov process as follows:

Biased(p_R) Random Walk: The states are $\{s_0, \ldots, s_n\}$ such that the probability $\Pr[s_i, s_j]$ of moving from state s_i to s_j at any step is given by:

$$\forall 0 \leq i < n, \Pr[s_i, s_{i+1}] = p_R$$

$$\forall 0 < i < n, \Pr[s_i, s_{i-1}] = q_R$$

$$\Pr[s_0, s_0] = q_R, \Pr[s_n, s_n] = 1$$

where $p_R + q_R = 1$. Notice that a Biased(p_R) Random Walk is just a 1-Penalty(p_R) Walk.

Let $\mathrm{E}_R[s_i, s_j]$ denote the expected number of steps for a Biased Random Walk to go from state s_i to state s_j. The following three lemmas compute the value of $\mathrm{E}_R[s_0, s_n]$.

Lemma 6.3.11. $E_R[s_0, s_1] = \frac{1}{p_R}$ *and* $\forall 1 \leq i \leq n-1$, $E_R[s_i, s_{i+1}] = \frac{1}{p_R} + \frac{q_R}{p_R} E_R[s_{i-1}, s_i]$

Proof. From state s_0, after 1 step, with probability p_R the next state is s_1 and with probability $q_R = 1 - p_R$ the system remains in s_0. Hence, $\mathrm{E}_R[s_0, s_1] = 1 + q_R \mathrm{E}_R[s_0, s_1]$ implying $\mathrm{E}_R[s_0, s_1] = \frac{1}{p_R}$.

For $i \geq 1$, from state s_i, after 1 step, with probability p_R the next state is s_{i+1} and with probability $q_R = 1 - p_R$ the next state is s_{i-1}. Hence:

$$\mathrm{E}_R[s_i, s_{i+1}] = 1 + q_R \cdot \mathrm{E}_R[s_{i-1}, s_{i+1}] = 1 + q_R \cdot (\mathrm{E}_R[s_{i-1}, s_i] + \mathrm{E}_R[s_i, s_{i+1}]).$$

Therefore:

$$p_R \cdot \mathrm{E}_R[s_i, s_{i+1}] = 1 + q_R \cdot \mathrm{E}_R[s_{i-1}, s_i]$$

implying

$$\mathrm{E}_R[s_i, s_{i+1}] = \frac{1}{p_R} + \frac{q_R}{p_R} \mathrm{E}_R[s_{i-1}, s_i].$$

□

Lemma 6.3.12. *For any probability* p_R *and* $\forall 0 < i < n$, $E_R[s_{i-1}, s_i] \leq E_R[s_i, s_{i+1}]$.

Proof. The proof proceeds by induction on the state index i. For the base case let $i = 1$.

From Lemma 6.3.11, $\mathrm{E}_R[s_1, s_2] = \frac{1}{p_R} + \frac{q_R}{p_R} \cdot \mathrm{E}_R[s_0, s_1] \geq \frac{1}{p_R} = \mathrm{E}_R[s_0, s_1]$.

For the inductive step, assume that for all $1 \leq i < k$, $\mathrm{E}_R[s_{i-1}, s_i] \leq \mathrm{E}_R[s_i, s_{i+1}]$.

$$\begin{aligned}
\mathrm{E}_R[s_k, s_{k+1}] &= \frac{1}{p_R} + \frac{q_R}{p_R} \mathrm{E}_R[s_{k-1}, s_k] \quad \text{by Lemma 6.3.11} \\
&\geq \frac{1}{p_R} + \frac{q_R}{p_R} \mathrm{E}_R[s_{k-2}, s_{k-1}] \quad \text{by the induction hypothesis} \\
&= \mathrm{E}_R[s_{k-1}, s_k] \quad \text{by Lemma 6.3.11}
\end{aligned}$$

□

Lemma 6.3.13. *If $p_R > q_R$, then $E_R[s_0, s_n] < \frac{n}{p_R - q_R}$.*

Proof. $\mathrm{E}_R[s_0, s_0] = 0$ and for $n \geq 1$

$$\begin{aligned}
\mathrm{E}_R[s_0, s_n] &= \mathrm{E}_R[s_0, s_1] + \sum_{1 \leq i \leq n-1} \mathrm{E}_R[s_i, s_{i+1}] \\
&= \frac{1}{p_R} + \sum_{1 \leq i \leq n-1} \left(\frac{1}{p_R} + \frac{q_R}{p_R} \mathrm{E}_R[s_{i-1}, s_i] \right) \quad \text{(by Lemma 6.3.11)} \\
&= \frac{n}{p_R} + \frac{q_R}{p_R} \mathrm{E}_R[s_0, s_{n-1}]
\end{aligned}$$

Solving this recurrence yields:

$$\begin{aligned}
\mathrm{E}_R[s_0, s_n] &= \sum_{0 \leq i \leq n-1} \frac{n-i}{p_R} \left(\frac{q_R}{p_R}\right)^i \\
&< \frac{n}{p_R} \sum_{0 \leq i \leq n-1} \left(\frac{q_R}{p_R}\right)^i \\
&< \frac{n}{p_R - q_R} \quad \text{(provided } q_R < p_R\text{)}
\end{aligned}$$

□

Associate to each d-Penalty(p_N) walk the *comparable* Biased(p_R) Random walk by setting $p_R = \frac{p_N}{p_N + dq_N}$ (and hence, $1 - p_R = q_R = \frac{dq_N}{p_N + dq_N}$).

Let $\mathrm{E}_N[s_i, s_j]$ denote the expected number of steps for a d-Penalty(p_N) walk to go from state s_i to state s_j. The following lemma shows that $\mathrm{E}_N[s_i, s_j]$ will be overestimated by $\mathrm{E}_R[s_i, s_j]$ for the comparable Biased(p_R) Random walk defined above.

Lemma 6.3.14. *The expected number of steps to go from s_i to s_{i+1} for a d-Penalty(p_N) walk, $E_N[s_i, s_{i+1}]$, and the expected number of steps to go from s_i to s_{i+1} for the comparable Biased(p_R) Random walk, $E_R[s_i, s_{i+1}]$, are related by $\forall 0 \leq i < n$, $E_N[s_i, s_{i+1}] \leq E_R[s_i, s_{i+1}]$.*

Proof. The proof proceeds by induction on the state index i. For the base case let $i = 0$.

$$\begin{aligned} \mathrm{E}_R[s_0, s_1] &= \frac{1}{p_R} \quad \text{(by Lemma 6.3.11)} \\ &= \frac{p_N + q_N \cdot d}{p_N} \quad \text{(by value of } p_R\text{)} \\ &\geq \frac{p_N + q_N}{p_N} \quad \text{(because } d \geq 1\text{)} \\ &= \frac{1}{p_N} \quad = \mathrm{E}_N[s_0, s_1]. \end{aligned}$$

For the inductive step assume that $\forall 0 \leq i \leq l-1$, $\mathrm{E}_N[s_i, s_{i+1}] \leq \mathrm{E}_R[s_i, s_{i+1}]$. If $0 < l \leq d$,

$$\mathrm{E}_N[s_l, s_{l+1}] = 1 + q_N \cdot \mathrm{E}_N[s_0, s_{l+1}] = 1 + q_N \cdot (\mathrm{E}_N[s_0, s_1] + \ldots + \mathrm{E}_N[s_{l-1}, s_l]) + q_N \cdot \mathrm{E}_N[s_l, s_{l+1}]$$

Therefore,

$$\begin{aligned} \mathrm{E}_N[s_l, s_{l+1}] &= \frac{1}{p_N} + \frac{q_N}{p_N}(\mathrm{E}_N[s_0, s_1] + \ldots + \mathrm{E}_N[s_{l-1}, s_l]) \\ &\leq \frac{1}{p_N} + \frac{q_N}{p_N}(\mathrm{E}_R[s_0, s_1] + \ldots + \mathrm{E}_R[s_{l-1}, s_l]) \quad \text{(by the induction hypothesis)} \\ &\leq \frac{1}{p_N} + \frac{q_N \cdot l}{p_N}\mathrm{E}_R(s_{l-1}, s_l) \quad \text{(by Lemma 6.3.12).} \end{aligned}$$

In a similar way, if $d < l < n$,

$$\mathrm{E}_N[s_l, s_{l+1}] = 1 + q_N \cdot (\mathrm{E}_N[s_{l-d}, s_{l-d+1}] + \ldots + \mathrm{E}_N[s_{l-1}, s_l]) + q_N \cdot \mathrm{E}_N[s_l, s_{l+1}].$$

Therefore,

$$\begin{aligned} \mathrm{E}_N[s_l, s_{l+1}] &= \frac{1}{p_N} + \frac{q_N}{p_N}(\mathrm{E}_N[s_{l-d}, s_{l-d+1}] + \ldots + \mathrm{E}_N[s_{l-1}, s_l]) \\ &\leq \frac{1}{p_N} + \frac{q_N \cdot d}{p_N}\mathrm{E}_R[s_{l-1}, s_l]. \end{aligned}$$

So, in either case,

$$\begin{aligned}
\mathrm{E}_N[s_l, s_{l+1}] &\leq \frac{1}{p_N} + \frac{q_N \cdot d}{p_N}\mathrm{E}_R[s_{l-1}, s_l] \\
&\leq \frac{p_N + q_N \cdot d}{p_N} + \frac{q_N \cdot d}{p_N}\mathrm{E}_R[s_{l-1}, s_l] \quad \text{(because } p_N + q_N \cdot d \geq 1\text{)} \\
&= \frac{1}{p_R} + \frac{q_R}{p_R}\,\mathrm{E}_R[s_{l-1}, s_l] \quad \text{(by definition of } p_N\text{)} \\
&= \mathrm{E}_R[s_l, s_{l+1}] \quad \text{(by Lemma 6.3.11)}
\end{aligned}$$

□

Corollary 6.3.15. *The expected number of steps for a d-Penalty(p_N) walk to terminate in state s_n from any initial state is at most $\frac{n(p_N+dq_N)}{p_N-dq_N}$.*

Proof. The expectation is maximized for initial state s_0. Let E_R denote the expected number of steps for the comparable Biased(p_R) Random Walk.

$$\begin{aligned}
\mathrm{E}_N[s_0, s_n] &= \mathrm{E}_N[s_0, s_1] + \mathrm{E}_N[s_1, s_2] + \ldots + \mathrm{E}_N[s_{n-1}, s_n] \\
&\leq \mathrm{E}_R[s_0, s_1] + \mathrm{E}_R[s_1, s_2] + \ldots + \mathrm{E}_R[s_{n-1}, s_n] \quad \text{(by Lemma 6.3.14)} \\
&= \mathrm{E}_R[s_0, s_n] \\
&< \frac{n}{p_R - q_R} \quad \text{(by Lemma 6.3.13)} \\
&= \frac{n}{\frac{p_N}{p_N+dq_N} - \frac{dq_N}{p_N+dq_N}} \quad \text{(by Def. of comparable Biased}(p_R)\text{ Random Walk)} \\
&= \frac{n(p_N + dq_N)}{p_N - dq_N}
\end{aligned}$$

□

Therefore based on Observation 6.3.10, the expected number of super rounds for LD_3 to converge to a configuration satisfying 3-secure is $\frac{n(p+dq)}{p-dq}$, where $p = \frac{3\Delta^2}{3\Delta^2+1}$, $q = 1-p$, and $d = \Delta^2$.

Theorem 6.3.16. *Starting from any configuration satisfying 2-secure, for any distributed scheduler, under Assumption 6.3.7, algorithm LD_3 will converge to a configuration satisfying 3-secure after an expected number of $2n$ choosing operations.*

Proof. Based on Observation 6.3.10, the d-Penalty(p_N) walk where $d = \Delta^2$ and $p_N \leq 1 - \frac{1}{3\Delta^2+1}$, overestimates the expected number of super rounds E for algorithm LD_3 to converge to 3-secure. Therefore

$$\begin{aligned}
E &\leq E_N[s_0, s_n] \\
&\leq \frac{n(p_N + dq_N)}{p_N - dq_N} \quad \text{(by Corollary 6.3.15)} \\
&= \frac{n(\frac{3\Delta^2}{3\Delta^2+1} + \Delta^2\frac{1}{3\Delta^2+1})}{\frac{3\Delta^2}{3\Delta^2+1} - \Delta^2\frac{1}{3\Delta^2+1}} \\
&= \frac{n(3\Delta^2 + \Delta^2)}{3\Delta^2 - \Delta^2} \\
&= \frac{n4\Delta^2}{2\Delta^2} \\
&= 2n
\end{aligned}$$

□

Removing Assumption 6.3.7: If, in the course of choosing new labels, no process ever chooses a label that is the same as any of its neighbors within distance 2, then 3-secure holds after an expected $2n$ choices. The probability for a process x to choose labels not in $L_2(x)$ is:

$$1 - \frac{|L_2(x)|}{M} \geq \frac{3\Delta_x^4 + \Delta_x^2 - 1}{3\Delta_x^4 + \Delta_x^2}$$

Therefore the probability for any process in the system not to make such a choice is bounded from below by $\frac{3\Delta^4+\Delta^2-1}{3\Delta^4+\Delta^2}$.

Theorem 6.3.17. *Starting from any configuration satisfying* 2*-secure, and for any distributed scheduler, after expected* $12n$ *super rounds of algorithm* LD_3*, a configuration* c *is reached. With probability at least* $(\frac{3\Delta^4+\Delta^2-1}{3\Delta^4+\Delta^2})^{2n}$ c *satisfies* 3*-secure, otherwise it satisfies* 0*-secure.*

Proof. By Theorem 6.3.16 if a process never choose a label the same as any neighbors within distance 2, after expected $2n$ choices, algorithm LD_3 will converge to a configuration satisfying 3-secure. By Theorem 6.3.6, $2n$ choosing operations happen within at most $12n$ super rounds of algorithm LD_3. The probability that no process chooses a label within distance 2 in $2n$ choosing operations is at least $(\frac{3\Delta^4+\Delta^2-1}{3\Delta^4+\Delta^2})^{2n}$. Even if some process chooses a label within distance 2, predicate 0-secure is always true. □

Corollary 6.3.18. *Algorithm* LD_3 *is random*$((\frac{3\Delta^4+\Delta^2-1}{3\Delta^4+\Delta^2})^{2n}, 12n)$ *self-stabilizing to* 3*-secure else* 0*-secure given* 2*-secure.*

6.3.6 Non-interference of LD_3

As indicated in Section 6.3.4, we need to show that there exists a particular fair composition of Label_D_2 and LD_3, such that algorithm LD_3 is (0-secure, r) right non-interfering with algorithm Label_D_2 via the composition. It is easy to see that algorithm LD_3 does not obstruct Label_D_2, if every process never chooses a label within distance 2. Starting from any configuration satisfying 0-secure, algorithm Label_D_2 converges to a configuration satisfying 2-secure within a finite number of

steps, in which processes will make a finite number of choices in algorithm LD_3, say c. Since the probability that any process does not choose a label within distance 2 is at most $\frac{3\Delta^4+\Delta^2-1}{3\Delta^4+\Delta^2}$ (see page 143), the value of r is bounded above by $(\frac{3\Delta^4+\Delta^2-1}{3\Delta^4+\Delta^2})^c$. Different ways to construct the composition may result in different values for c.

A standard way to compose Label_D_2 and LD_3 would be to have each process execute the steps of the two algorithms alternatively. The value of c depends on the convergence speed of the Label_D_2.

Another way is to make a conditional composition, where each process's program is sketched as follows:

1: run an iteration of Label_D_2
2: **if** no shared link registers or local variables has been changed **then**
3: run an iteration of algorithm LD_3
4: **end if**

With this composition, algorithm LD_3 interferes much less with the Label_D_2 before a configuration satisfying 2-secure is reached. Therefore the value of r in this composition is much bigger than with the standard composition.

Observation 6.3.19. *There exists a fair composition of Label_D_2 and LD_3 that is random self-stabilizing for 3-secure given 0-secure.*

6.4 Further Discussion

As indicated in Section 6.2.1, assigning distinct labels up to distance 2 is easy, because it is easy to distinguish one neighbor from another. Randomization is used to choose new labels; discovering that labels are not distinct at distance 2 is done

deterministically.

In our algorithm randomization is also used to discover the illegitimate case. For example, consider a multiple cycle with repeated pattern $\langle x, y, z \rangle$, where $x < y < z$. If all processes on the cycle with label x have the same counter, then with probability one half, each such process generates a different coin flip from its successor. Thus the error will be discovered by its successor.

The counter in the message also plays a very important role, because a process needs to determine whether an incoming message is out-of-date or not. For example, in a minimum cycle $\langle x, y, z \rangle$, where $x < y < z$, the process with label x will keep calling the function $\langle \text{mini\_cycle\_consistency}(y, z) \rangle$, because both $\langle \text{symmetry}(z, y) \rangle$ and $\langle \text{symmetry}(y, z) \rangle$ always return true. With probability one half, this process may generate different coin flips within two successive invocations. If there is no counters, then in the second invocation, the process label x may read the old message from the first invocation before its new message has been delivered. Then the process with label x will falsely send an alarm to one of its neighbors.

Also a process has to distinguish an old message from itself and a message from its predecessor. For example, in the above multiple cycle situation, if every process with label x has a counter one smaller than its successor, then every such process will consider the incoming message as an old message from itself, and waits for a updated message, which will never arrive. To avoid this, the counter is incremented modulo k where k is bigger than one third the network size. Then it is impossible for all processes in the cycle to have a counter one bigger modulo k than their predecessor.

Another issue is that the expected number of super rounds for algorithm LD_3 to converge is hugely overestimated in Corollary 6.3.18 by assuming that in every three

super rounds only one 3-local-insecure process chooses a new label. Most of the time, with high probability within two super rounds at least half of the 3-local-insecure processes choose a new label. For example, consider a long path or a multiple cycle with repeated pattern $\langle x, y, z \rangle$, where $x < y < z$, if all counters are the same, then with probability a half, each of the processes with label x generate a coin flip different from its successor's. Therefore we expect half of the processes with label x will send an alarm. If the counters are not all the same and every process with label x does not generate a counter that is one smaller than its successor's, then all the x labeled processes will send an alarm. Otherwise, if processes do have a counter one smaller than their successors', as shown in Lemma 6.3.4, this can only happen in a long path. Based on Lemma 6.3.3 processes at both end will receive an alarm and choose new labels, with high probability they become 3-local-secure and leave the path. At this point, all the intermediate processes have not finished their current iteration of the algorithm yet, because they are all waiting for updated messages. Thus the super round is not ended. And eventually all processes except the middle 2 or 3 processes will choose a new label within one super round.

CHAPTER 7

Conclusions

7.1 Summary of Contributions

This thesis contains two major parts: a literature survey including a unification of different kinds of Process Coordination problems, and a self-stabilizing solution to the Dining Philosophers problem.

We studied six different Process Coordination problems including Dining Philosophers, Drinking Philosophers, Resource Allocation, Committee Coordination, Multiway Rendezvous, and Multiparty Interaction. All of these problems have similar exclusion and synchronization requirements. To reveal the similarities and differences among all these problems, we devise two uniform frameworks to model them, the object oriented model and the graph model. These two models capture the behaviors of distributed systems by characterizing the executions and configurations, respectively. We reviewed some significant papers on different problems by paraphrasing the original problem descriptions and solutions. We also repair flaws in some solutions and clarify some misleading problem descriptions.

In the second part, we begin to investigate the robustness of distributed systems under failures. We focus on one of the strongest models, self-stabilizing distributed systems, which handles transient faults. We review a classic fair composition technique for designing and analyzing self-stabilizing algorithms. Then we enriched the fair composition tool for more general use. At the end, we present a technique that can be used to make most existing solutions to the Dining Philosophers problem self-stabilizing. This technique includes a new mechanism that labels processes in a network such that every pair of processes within distance 3 have distinct labels.

7.2 Comments and Future Work

In this thesis, we present a technique that assigns distinct labels up to distance 3 by using an algorithm (Label_D_2) that achieves distinct labels up to distance 2. We design an algorithm LD_3 that can distinguish cases shown in Figure 6.4 and 6.5 in any configuration where processes within distance 2 have distinct labels. The most general technique should be to assign distinct labels up to distance k based on an algorithm that assigns distinct labels up to distance $k-1$ where $k > 3$. To achieve this, one should design an algorithm LD_k that can distinguish the case, which is a long path of length k with the same label at both ends and distinct labels in between, and the case, which is a multiple cycle of length $k-1$ with distinct labels. Although it appears that no new ideas and techniques are required, this general tool is likely much more complicated and involved because every process may collect and handle more information (for example, labels of neighbors up to distance $k-1$).

Self-stabilizing algorithms may not tolerate other system failures. For example,

Beauquier, Datta, Gradinariu, and Magniette's solution [3] to the Local Mutual Exclusion problem is self-stabilizing but does not tolerate stop failures. A stop process could cause a local deadlock among its neighborhood, which will be spread over the rest of the network. Another open research direction is to find the Dining Philosophers' solution which can tolerate more kinds of system failures. Choy and Singh's solutions tolerate stop failure to some extent. It would be desirable if one can make their solutions also self-stabilizing.

Bibliography

[1] H. Attiya and J.L. Welch. *Distributed computing: fundamentals, simulations and advanced topics.* The Mc-Graw Hill Companies, 1998.

[2] R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9), 1989.

[3] J. Beauquier, A.K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *International Symposium on Distributed Computing*, pages 223–237, 2000.

[4] J.A. Bondy and U.S.R. Murty. *Graph Theory with Applications.* Elsevier North Holland, Inc., 1980.

[5] C. Boulinier, S. Cantarell, F. Petit, and V. Villain. A note on a bounded self-stabilizing local mutual exclusion algorithm. Technical report, Technical Report LaRIA-2003-02, 2003.

[6] K.M. Chandy and J. Misra. *Parallel Program Design, A Foundation.* Addison-Wesley Publishing Company, 1988.

[7] S. Chattopadhyay, L. Higham, and K. Seyffarth. Dynamic and self-stabilizing distributed matching. In *Proceedings of the 21th Annual Symposium on Principles of Distributed Computing*, pages 290–297, 2002.

[8] M. Choy and A.K. Singh. Efficient fault tolerant algorithms for resource allocation in distributed systems. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 593–602, 1992.

[9] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica 1*, 1971.

[10] E.W. Dijkstra. Self stabilizing systems in spite of distributed control. In *Communications of the Association of the Computing Machinery*, volume 17, pages 643–644, 1974.

[11] S. Dolev. *Self-Stabilization.* The MIT Press, 2000.

[12] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems*, MCC Technical Report No. STP-379-89, 1989.

[13] D. Ginat, A.U. Shankar, and A.K. Agrawala. An efficient solution to the drinking philosophers problem and its extensions. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, pages 83–93, 1989.

[14] M.G. Gouda. The stabilizing philosopher: asymmetry by memory and by action. Technical Report TR-87-12, University of Texas at Austin, 1987.

[15] M. Gradinariu and C. Johnen. Self-stabilizing neighborhood unique naming under unfair schedular. In *European Conference on Parallel Processing*, volume 2150, pages 458 – 465, 2001.

[16] O.M. Herescu and C. Palamidessi. On the generalized dining philosophers problem. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing*, pages 81–89, 2001.

[17] M. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[18] D. Hoover and J. Poole. A distributed self-stabilizing solution to the dining philosophers problem. In *Information Processing letters*, volume 41, pages 209–213, 1992.

[19] Y.J. Joung. Two decentralized algorithms for strong interaction fairness for systems with unbounded speed variability. *Theoretical Computer Science*, 243(1-2), 2000.

[20] L. Lamport. Solved problems, unsolved problems and non-problems in concurrency, invited address. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 63–67, 1984.

[21] D. Lehmann and M.O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of 8th Annual ACM Symposium on Principles of Programming Languages*, pages 133–138, 1981.

[22] N.A. Lynch. Fast allocation of nearby resources in a distributed system. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 70–81, 1980.

[23] N.A. Lynch. Upper bounds for static resource allocation in a distributed system. *Journal of Computer and System Sciences*, 23(2), 1981.

[24] N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., 1996.

[25] C. Palamidessi. Re: a question on one of your papers. Private Communication, 2001.

[26] J.L. Welch and N.A. Lynch. A modular drinking philosophers algorithm. *Distributed Computing*, 6(4), 1993.

APPENDIX A

Welch and Lynch's I/O Automaton Model for the Drinkers Problem

Let R be a finite non-empty set of resources, and P be a set of users. For any user $p_i \in P$, p_i uses a non-empty set of resource $R_i \subseteq R$. Each resource is shared by at most two users. A user p_i is in one of the following state at a time:

trying region: User p_i is trying to get all its resources.

critical region: Once all the resources in R are obtained, p_i enters its region.

exit region: After using all the resources, p_i enters exit region to do some cleaning up activities, such as return all the resources in R.

remainder region: If p_i is in none of the regions above, p_i is in its remainder region.

In drinking philosophers problem, resources are considered as beverages, and users as philosophers. The corresponding regions are represented by $T_i(B)$, $C_i(B)$, $E_i(B)$, and $R_i(B)$. Every philosopher cycles through these four regions. Each philosopher may request a different set of beverages, B, at each time it enters its

trying region.

The author described the system using the input-output automaton model as follows [26].

> An automaton is a state machine whose state transitions is labeled with actions. Actions are partitioned into *input* actions, *output* actions, and *internal* actions. The input and output actions model communication with the outside world. An execution of an automaton is an alternating sequence of states and actions. An execution is *fair* if the automaton eventually gets to perform a pending output or internal action.

Each system component is modeled by an automaton, and the whole system is also modeled by an automaton, which resulting from the composition of the components.

An automaton solves a drinking philosophers problem if it satisfies the following requirements:

1. Its input actions are $\{T_i(B),\ E_i(B) : 1 \leq i \leq n, B \subseteq B_i, B \neq \emptyset\}$

2. Its output actions are $\{C_i(B), R_i(B) : 1 \leq i \leq n, B \subseteq B_i, B \neq \emptyset\}$

3. An execution is drinking-well-formed if $\forall p_i$ the subsequence of the execution restricted to the following pattern: $T_i(B)$, $C_i(B)$, $E_i(B)$, $R_i(B)$, $T_i(B')$, $C_i(B')$, $E_i(B')$, $R_i(B')$.... The automaton preserves drinking-well-formedness, which means for all execution e' and e where e is the result of extending e' by one output action, if e' is drinking-well-formed, then e is drinking-well-formed.

4. In any drinking-well-formed execution, $\forall i, j, B, B'$ with $i \neq j$ and $B \cap B' \neq \emptyset$, if an occurrence of $C_i(B)$ precedes an occurrence of $C_j(B')$, then $E_i(B)$ occurs

between the $C_i(B)$ and $C_j(B')$.

5. In any fair drinking-well-formed execution, if $\forall i, B$ every occurrence of $C_i(B)$ is followed by an occurrence of $E_i(B)$, then $\forall i, B$ every occurrence of $T_i(B)$ is followed by an occurrence of $R_i(B)$. In other words, if nobody ever stuck in its critical region, then all the philosophers who are in their trying region will eventually enter their critical region.

6. Given i, B and an occurrence of $T_i(B)$ in a drinking-well-formed execution, the occurrence of $T_i(B)$ is *non-overlapping* if for all $j \neq i$ and all B' that intersect B, every preceding occurrence of $T_j(B')$ is followed by an $E_j(B')$ that also precedes the $T_i(B)$, and every following occurrence of $T_j(B')$ follows a $C_i(B)$ that also follows the $T_i(B)$. We say an occurrence of $T_i(B)$ is *non-overlapping*, if nobody occupies or needs beverages in B before the occurrence of the following $C_i(B)$. In any fair drinking-well-formed execution, for all i, B and all occurrences of $T_i(B)$, if the occurrence of $T_i(B)$ is non-overlapping, then the $T_i(B)$ is followed by an occurrence of $C_i(B)$