

1-1-2008

Self-stabilizing protocol for anonymous oriented bi-directional rings under unfair distributed schedulers with a leader

Chitwan Kumar Gupta
University of Nevada, Las Vegas

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

Repository Citation

Gupta, Chitwan Kumar, "Self-stabilizing protocol for anonymous oriented bi-directional rings under unfair distributed schedulers with a leader" (2008). *UNLV Retrospective Theses & Dissertations*. 2406.
<http://dx.doi.org/10.25669/8gs6-rekc>

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

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

SELF-STABILIZING PROTOCOL FOR ANONYMOUS ORIENTED
BI-DIRECTIONAL RINGS UNDER UNFAIR DISTRIBUTED
SCHEDULERS WITH A LEADER

By

Chitwan Kumar Gupta

Bachelor of Engineering
University Of Rajasthan, Jaipur
May 2005

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University Of Nevada, Las Vegas
December 2008

UMI Number: 1463509

INFORMATION TO USERS

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

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

UMI[®]

UMI Microform 1463509

Copyright 2009 by ProQuest LLC.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 E. Eisenhower Parkway
PO Box 1346
Ann Arbor, MI 48106-1346



Thesis Approval
The Graduate College
University of Nevada, Las Vegas

November, 17, 2008

The Thesis prepared by

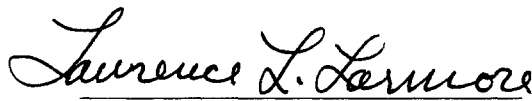
CHITWAN KUMAR GUPTA

Entitled


SELF-STABILIZING PROTOCOL FOR ANONYMOUS ORIENTED BI-DIRECTIONAL RINGS
UNDER UNFAIR DISTRIBUTED SCHEDULERS WITH A LEADER.

is approved in partial fulfillment of the requirements for the degree of

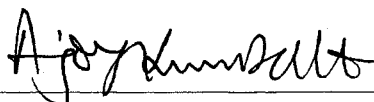
MASTER OF SCIENCE IN COMPUTER SCIENCE



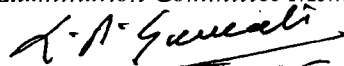
Examination Committee Chair



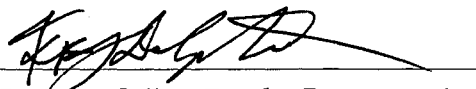
Dean of the Graduate College



Examination Committee Member



Examination Committee Member



Graduate College Faculty Representative

ABSTRACT

Self-Stabilizing Protocol For Anonymous Oriented Bi-directional Rings Under Unfair Distributed Schedulers With A Leader

By

Chitwan Kumar Gupta

Dr. Lawrence L. Larmore, Examination Committee Chair
School of Computer Science
University of Nevada, Las Vegas

We propose a self-stabilizing protocol for anonymous oriented bi-directional rings of any size under unfair distributed schedulers with a leader. The protocol is a randomized self-stabilizing, meaning that starting from an arbitrary configuration it converges (with probability 1) in finite time to a legitimate configuration (i.e. global system state) without the need for explicit exception handler or backward recovery. A fault may throw the system into an illegitimate configuration, but the system will autonomously resume a legitimate configuration, by regarding the current illegitimate configuration as an initial configuration, if the fault is transient. A self-stabilizing system thus tolerates any kind and any finite number of transient faults. The protocol can be used to implement an unfair distributed mutual exclusion in any ring topology network.

Keywords: self-stabilizing protocol, anonymous oriented bi-directional ring, unfair distributed schedulers. Ring topology network, non-uniform and anonymous network, self-stabilization, fault tolerance, legitimate configuration.

TABLE OF CONTENTS

ABSTRACT.....	iii
ACKNOWLEDGEMENT	v
CHAPTER 1 INTRODUCTION	1
1.1 Distributed Systems	1
1.2 Self-Stabilization.....	2
1.3 Related Work	2
1.4 Contributions.....	3
1.5 Outline of the Thesis.....	4
CHAPTER 2 ORIGIN OF DINING PHILOSOPHERS PROBLEM.....	5
2.1 Definitions.....	5
2.2 Dining Philosophers Problem (DPP)	7
2.3 Survey of Non Self-Stabilizing DPP.....	8
2.4 Survey of Self-Stabilizing DPP	10
CHAPTER 3 DINING PHILOSOPHERS.....	12
3.1 The Dining Philosophers Problem	12
3.1.1 Livelock and Deadlock	14
3.2 The Chain Version of the Dining Philosophers Problem.....	14
3.3 Proof of Correctness of DPCHAIN	19
3.4 The Algorithm DPRING.....	21
3.4.1 Formal Definition of DPRING	23
3.5 Proof of Correctness of DPRING	27
CHAPTER 4 CONCLUSION AND FUTURE RESEARCH	31
BIBLIOGRAPHY.....	32
VITA.....	37

ACKNOWLEDGMENT

I would like to thank all people who have helped and inspired me during my thesis. I especially want to thank my advisor, Dr. Lawrence L. Larmore, for his guidance during my thesis and study at University of Nevada- Las Vegas. His perpetual energy and enthusiasm in research had motivated me. In addition, he was always accessible and willing to help me with my research. As a result, research life became smooth and rewarding for me. I warmly thank Dr. Ajoy K. Datta, for his valuable advice and friendly help. Throughout my thesis-writing period, he and Dr. Larmore provided encouragement, sound advice, good teaching, good company, and lots of good ideas. I would have been lost without them. Dr. Laxmi P. Gewali, Dr. Rohan Dalpatadu, and Dr. Ajoy K. Datta deserve special thanks as my thesis committee members and advisors. My deepest gratitude goes to my family for their unflagging love and support throughout my life; this thesis is simply impossible without them.

CHAPTER 1

INTRODUCTION

In this thesis, we present a self-stabilizing protocol for anonymous oriented bi-directional rings of any size under unfair distributed schedulers. Self-stabilization is a well-known paradigm of non-masking fault tolerant distributed algorithms [21, 10, 9]. Self-stabilization introduced by Dijkstra, [1], provides an uniform approach to fault-tolerance, [8]. We are particularly interested in non-uniform (i.e. all processors don't perform the same algorithm) and anonymous network (i.e. no processor has a distinct identifier). This protocol guarantees that, regardless of the initial state, the system will eventually converge to the intended behavior without the need for explicit exception handler or backward recovery.

1.1 Distributed Systems

A distributed system in its simplest form can be presented as a set of processors connected over a communication medium. The processors make local computations and exchange messages using the communication medium. Distributed systems can be classified as synchronous or asynchronous. Processors can be synchronous or asynchronous depending on how the local computations are made. The communication medium can be synchronous or asynchronous depending on how the communication between the processors is accomplished.

1.2 Self-Stabilization

Self-Stabilization is an important concept for distributed computing and communication networks. It describes a system's ability to recover automatically from unexpected failure. It is also an important issue for multiagent systems, as they are distributed and communicative systems. Self-stabilization is a framework for dealing with channel or memory failures. After a failure the system is allowed to temporarily exhibit an incorrect behavior, but after a period of time as short as possible, it must behave correctly, without external intervention, [22]. The practical appeal of stabilizing protocols is that they are *simpler* (i.e., they avoid a slew of mechanisms to deal with a catalog of anticipated faults), and they are *more robust* (e.g., they can recover from transient faults such as memory corruption as well as common faults such as link and node crashes), [12].

1.3 Related Work

The first self-stabilizing algorithms was introduced by Dijkstra[1]. Schneider[21] presented a survey on early research on self-stabilization.Katz and Perry [3] showed how to compile an arbitrary asynchronous protocol into a stabilizing equivalent. Their general transformation is expensive; hence more efficient (and possibly less general) techniques are needed. Techniques that transform any locally checkable protocol into a stabilizing equivalent are given in [12, 13].

In [4] Burns and Pachl presented a deterministic algorithm for uniform unidirectional rings of prime size and proved that no deterministic solution exists for rings of composite size. Itkis, Lin, and Simon [5] present a deterministic constant-space self-stabilizing protocol for leader election on uniform bidirectional asynchronous rings

of prime size. In their model, there is a central daemon that picks an enabled processor each time to make an atomic move. The chosen processor can read the states of its two neighbors at the same time to determine its next state.

Dolev, Israeli, and Moran [14] presented a randomized self-stabilizing leader election protocol that tolerates addition or deletion of processors and links. Their protocol uses $O(\log n)$ bits per node. Ghosh and Gupta [15] introduced a self-stabilizing leader-election algorithm that recovers quickly from small-scale transient faults. Higham and Myers [20] gave a randomized self-stabilizing algorithm that solves token circulation and leader election on anonymous, uniform, synchronous, and unidirectional rings of arbitrary but known size, in which each processor state and message has size in $O(\log n)$. Kakugawa and Yamashita [16] presented a probabilistic uniform self-stabilizing algorithm on uniform rings that does guarantee an upper bound between two critical section entries.

1.4 Contribution

Many of the previous works on the self-stabilizing mutual exclusion problem either assume a central daemon or assume unfair daemon for uniform unidirectional rings or assume unfair daemon for non-uniform bi-directional rings but use message passing model . We present an self-stabilizing protocol under unfair daemon for oriented bi-directional non-uniform ring without using message passing model. In [16], Kakugawa and Yamashita claimed that “there is no such system when the number n of processes (i.e., ring size) is composite, even if a fair central-daemon (c-daemon) is assumed” and there was an open question to design a self stabilizing algorithm that solves the mutual

exclusion problem under an unfair distributed scheduler. We answer the open question of [16] and present a self-stabilizing algorithm for anonymous oriented bi-directional rings of any size under unfair distributed schedulers with a leader.

1.5 Outline of the Thesis

We give definitions of some topics involved in this research and an overview of dining philosopher problem including survey of self-stabilizing and non self-stabilizing dining philosopher problem in Chapter 2.

In Chapter 3, first we give the solution to the simplified version of dining philosophers problem. We consider chain topology instead of ring topology and present a solution by DPCHAIN algorithm. Then we consider ring topology instead of chain topology and present a solution by DPRING algorithm. It also includes the proof of correctness of both DPCHAIN and DPRING algorithm.

We finish with concluding remarks in Chapter 4.

CHAPTER 2

ORIGIN OF DINING PHILOSOPHERS PROBLEM

2.1 Definitions

Mutual Exclusion: Mutual Exclusion is a fundamental problem in the area of distributed computing. Concurrent processes come into conflict with each other when they are competing for the use of the same resource. They are not necessarily aware of each other, but the execution of one process may affect the behavior of competing processes. Mutual Exclusion is a collection of techniques for sharing resources so that different processes do not conflict and cause unwanted interactions. Examples of such resources are fine-grained flags, counters or queues, used to communicate between code that runs concurrently, such as an application and its interrupt handlers.

Consider a system of n processors. Every processor, from time to time, may need to execute a critical section in which exactly one processor is allowed to use some shared resource. A distributed system solving the mutual exclusion problem must guarantee the following two properties [18]:

- (i) **Mutual Exclusion:** Exactly one processor is allowed to execute its critical section at any time.
- (ii) **Fairness:** Every processor must be able to execute its critical section infinitely often.

One of the most commonly used techniques for mutual exclusion is the semaphore.

Starvation: Starvation is a control problem due to the enforcement of mutual exclusion. Consider we have three processes, P1, P2, and P3, competing for a resource R. Suppose each of them require periodic access to R, which is not sharable, and P1 is first granted access to R. Then when P1 exits its critical section, either P2 or P3 may be allowed access to R. Assume that R is allocated to P3 and P1 requires access to R again. If the operating system alternately allocates R to P1 and P3, then P2 has to wait indefinitely and thus experience starvation, [24].

Deadlock: Deadlocks form one of the important error categories of concurrent computer systems, [32]. A set of processes, or threads, is resource deadlocked if each process in the set requests a resource, a lock, held by another process in the set, forming a cycle of lock requests. In communication deadlocks, messages are the resources for which processes wait.

Four conditions must hold for deadlock to occur:

1. Exclusive use – when a process accesses a resource, it is granted exclusive use of that resource.
2. Hold and wait – a process is allowed to hold onto some resources while it is waiting for other resources.
3. No preemption – a process cannot preempt or take away the resources held by another process.
4. Cyclical wait – there is a circular chain of waiting processes, each waiting for a resource held by the next process in the chain.

Deadlock can occur whenever two or more processes are competing for limited resources and the processes are allowed to acquire and hold a resource (obtain a lock)

thus preventing others from using the resource while the process waits for other resources.

Scheduler: All Components of (Processors and communication links) of distributed systems may not share the same speed assumptions (i.e. one processor may execute its code speedily, while many others are very slow.). The scheduler is a way to model such different behaviors. A scheduler chooses processors to execute their code at a given time. The scheduler (also known as daemon) is said to be *fair* if it selects every process infinitely many times; otherwise, it is *unfair*, [18].

2.2 Dining Philosophers Problem (DPP)

The problem of the dining philosophers, proposed by Dijkstra in [17], is a very popular example of control problem in distributed systems, and has become a typical benchmark for testing the expressiveness of concurrent languages and of resource allocation strategies. The dining philosophers problem is a simple case of general resource-allocation problem. The situation is modeled by a graph on the set of processors with an edge between two nodes if they share some resource (Each resource is thus represented by the edges of a complete graph connecting the processors that have access to it). Each processor handles a sequence of jobs; each job in the sequence of a processor has a resource requirement that is a subset of the resources accessible to that processor, [23]. For a job to be executed, all of the required resources must be available for exclusive use by its processor. This can be interpreted as saying that the processor must control the edges incident to it corresponding to the needed resources.

Traditionally, the problem is described in terms of the following informal scenario. There are n philosophers (users) seated around a table, usually thinking. Between each pair of philosophers is a single fork (resource). From time to time, any philosopher might become hungry and attempt to eat. In order to eat, the philosopher needs exclusive use of the two adjacent forks. After eating, the philosopher needs exclusive use of the two adjacent forks. After eating, the philosopher relinquishes the two forks (i.e., perform an exit protocol) and resume thinking.

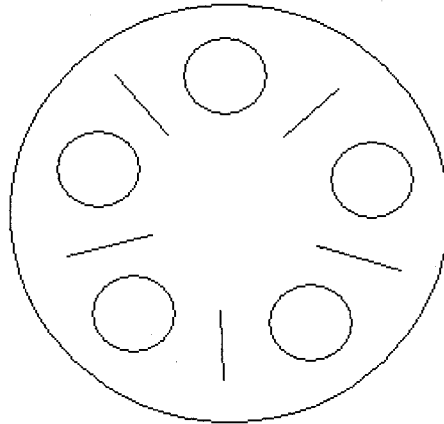


Figure 1.1: Dining Philosophers Problem (for $n=5$)

2.3 Survey of Non Self-Stabilizing DPP

The dining philosophers problem was first introduced in a specialized setting of a ring of five philosophers by Dijkstra in [17]. The problem was later generalized to the current setting of arbitrary graphs by Lynch in [19]. In this generalization, processes and resources are modeled by a graph with each vertex representing a process, and each edge representing a resource shared by the end vertices. The first work to consider the response time explicitly was the seminal work by Lynch [33] who considered the

problem in the context of resource allocation. Lynch's algorithm provides an upper bound on the response time of a job. The solution of dining philosophers problem proposed by M. Rabin and D. Lehmann [25] is fully distributed and does not involve any central memory or any process with which every philosopher can communicate. They exhibit a probabilistic solution for dining philosophers problem which guarantees, with probability one, that every hungry philosopher eventually gets to eat.

Styer and Peterson [38] extended and augmented Lynch's idea [33] to give an algorithm that guarantees a bound on the waiting time of a job that is polynomial in the number of processors at some maximum distance from the processor to which job is assigned. B. Awerbuch and M. Saks,[23] presented a new deterministic algorithm for a general job scheduling problem (generalizing the drinking (and dining) philosophers problem) that guarantees a response time that is not much more than the square of the lower bound. The unique feature of their algorithm is that resources are not explicitly collected; rather a job at the front of the queue simply executes its job, and the properties of the queue ensure that no conflicting job will execute at the same time.

A few non-stabilizing solution to the diners problem with optimal failure locality are also known [34, 42,49]. Choy and Singh [42] investigated the fault-tolerance of distributed algorithms in asynchronous message passing systems with undetectable process failures. They considered two specific synchronization problems the dining philosophers problem and the binary committee coordination problem. The abstraction of a bounded doorway is introduced as a general mechanism for achieving individual progress and good failure locality. Using it as a building block, optimal fault-tolerant algorithms are constructed for the two problems. Sivilotti, Pike and Sridhar [34]

presented a new algorithm for the dining philosophers problem that has optimal failure locality. As a refinement, the algorithm can be easily parameterized by a simple failure model to achieve super-optimal failure locality in the average case. Tsay and Bargodia [49] presented an algorithm that combines the idea of a dynamic priority scheme with the use of a preemptive fork collecting strategy. Its response time is $O(n)$, where n is the total number of processes, if no failures actually occur or $O(n^2)$ in the presence of failures.

2.4 Survey of Self-Stabilizing DPP

Besides the non-stabilizing solution to the diners problem, a number of stabilizing solutions are published as well [35, 36, 45,48]. Antonoiu and Srimani [35] proposed a new protocol that is id-based and does not use any shared variable as opposed to the self-stabilizing traditional mutual exclusion algorithm, which is anonymous and does use shared link registers. It is also based on read/write atomicity [26] of operations and operates under a distributed demon.

Beauquier, Datta, Gradinariu and Magniette [36] presented a self-stabilizing solution to the local mutual exclusion problem that is the extension of dining philosophers problem to any arbitrary network. They proposed a transformation technique that to transform self-stabilizing algorithms under weaker daemons into algorithms, which maintain the self-stabilization property, and also work under any arbitrary distributed daemon. Arora and Nesterenko [51] combined the stabilization and crash fault tolerance to present an efficient and inexpensive solution to the dining philosophers problem for a rich class of faults-malicious crashes.

Hoover and Poole [46] presented self-stabilizing dining philosophers algorithm that was inspired by the self-stabilizing dining philosophers algorithm presented by Gouda [52]. In Gouda's solution, one of the philosophers is required to behave differently than the others in order to introduce asymmetry. Datta, Gradinariu and Raynal [27] presented a self-stabilizing solution to the mobile philosophers problem (for asynchronous model) that is a new version of the dining philosophers problem. They assume that the resources form a logical ring (as in dining philosophers problem) and the philosophers can move around a logical ring formed out of a dynamic network.

CHAPTER 3

DINING PHILOSOPHERS

3.1 The Dining Philosophers Problem

In this section, we give a self-stabilizing asynchronous distributed algorithm for the Dining Philosophers Problem, in the composite model of computation. We first describe the problem formally, guided by the presentation given by Lynch [16].

Each philosopher P_i is represented by two processes, the user U_i , and the agent, which we also call P_i . The user decides when to request and return the resources, and the agent actually executes the algorithm. We are also given resources f_1, \dots, f_n . In order for a request by U_i to be satisfied, the P_i must have use of both f_{i-1} and f_i (except that P_1 uses f_n and f_1), and no two philosophers may simultaneously have use of the same resource. We refer to f_{i-1} and f_i as the left and right resources of P_i , and to P_{i-1} and P_{i+1} as the left and right neighbors of P_i .

Figure 3.1, which is similar to Figure 11.2 of [16], shows the network of processes and resources in the case $n = 5$.

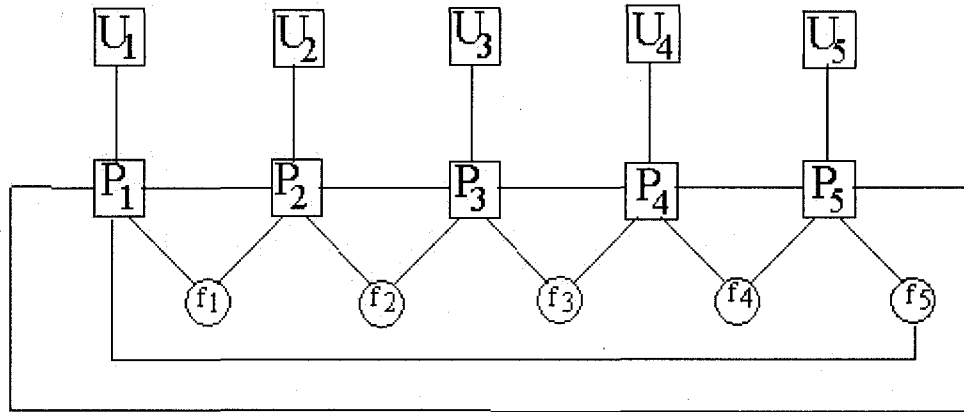


Figure 3.1: Network of Processors and Resources for the Dining Philosophers Problem

- U_i has only two states, request and sat .
- P_i can either lock or release either of its resources.
- P_i can use f_j if P_i is holding f_j and no other process is holding f_j .
- If U_i is in state request and P_i can use both neighboring resources, P_i will begin executing. Eventually the request will be satisfied, after which U_i will change its state to sat .

A solution to the Dining Philosophers Problem consists of a protocol (program) for each agent process, such that every request by any user process is eventually satisfied. A configuration is said to be illegitimate if two processes are simultaneously holding the same resource, or if a processor is attempting to execute without holding both resources. We call the first situation contention, and we call the second situation premature execution. A configuration is legitimate otherwise.

3.1.1 Livelock and Deadlock

Note that, if two processes are holding the same resource, neither can be executing. It could happen that two processes simultaneously lock the same resource. One of the processes must release the resource if this occurs. But if both release the resource simultaneously, they could both lock it again simultaneously. This cycle could continue indefinitely, a situation known as livelock.

On the other hand, a configuration could occur where each user U_i is in state request, and each agent process P_i holds f_i , and refuses to release it until it can lock f_{i+1} also. In this situation, called deadlock, nothing can happen, and the requests are never satisfied.

3.2 The Chain Version of the Dining Philosophers Problem

We first give a solution to a simplified version of the Dining Philosophers Problem, where the topology is that of a chain rather than a ring. We still have philosophers P_1, \dots, P_n , with user processes $U_1 \dots U_n$, but we have $n + 1$ resources, namely $f_0, f_1 \dots f_n$, as shown in Figure 3.2. Our solution is a distributed algorithm in the composite model of computation. Each process has shared variables that can be read by its neighbor processes.

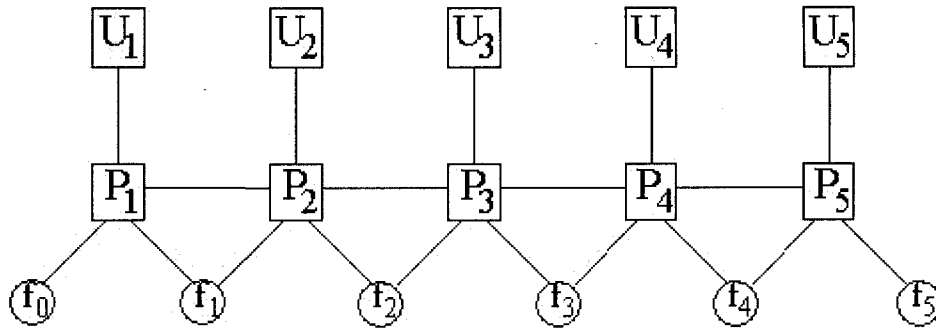


Figure 3.2: Network of Processors and Resources for the Chain Dining Philosophers Problem

We write:

User(P_i) = U_i

Left(P_i) = P_{i-1}

Right(P_i) = P_{i+1}

Variables of DPCHAIN:

$P.\text{flag} \in \{A, B\}$. This variable can be read by P 's neighbors.

$P.\text{state} \in \{\text{waiting}, \text{executing}, \text{idle}\}$. This variable can be read by User (P), but not by P 's neighbor agents.

Functions of DPCHAIN:

Define $\text{reverse}(A) = B$ and $\text{reverse}(B) = A$.

$\text{Holds_Left}(P) \equiv P$ is holding its left resource.

$\text{Holds_Right}(P) \equiv P$ is holding its right resource.

$\text{Left_Free}(P) \equiv$ no process is holding P 's left resource.

$\text{Right_Free}(P) \equiv$ no process is holding P 's right resource.

$$\text{Left_Nbr_Flag}(P) = \begin{cases} \text{reverse}(P.\text{flag}) & \text{if } P = P_1 \\ \text{Left}(P).\text{flag} & \text{otherwise} \end{cases}$$

$$\text{Right_Nbr_Flag}(P) = \begin{cases} P.\text{flag} & \text{if } P = P_n \\ \text{Right}(P).\text{flag} & \text{otherwise} \end{cases}$$

$\text{Left_Enabld}(P) \equiv \text{Left_Nbr_Flag}(P) \neq P.\text{flag}$

$\text{Right_Enabld}(P) \equiv \text{Right_Nbr_Flag}(P) = P.\text{flag}$

$\text{Has_Tokens}(P) \equiv \text{Left_Enabld}(P) \wedge \text{Right_Enabld}(P)$

$\text{Error}(P) \equiv (\text{Holds_Left}(P) \wedge \neg \text{Left_Enabld}(P)) \vee$

$(\text{Holds_Right}(P) \wedge \neg \text{Right_Enabld}(P)) \vee$

$((P.\text{state} = \text{executing}) \wedge (\neg \text{Holds_Right}(P) \vee \neg \text{Holds_Left}(P))) \vee$

$((P.\text{state} = \text{idle}) \wedge (\text{Holds_Right}(P) \vee \text{Holds_Left}(P)))$

Macros of DPCHAIN:

$\text{Lock_Right}(P)$: P locks its right resource.

$\text{Lock_Left}(P)$: P locks its left resource.

$\text{Release_Right}(P)$: P releases its right resource.

$\text{Release_Left}(P)$: P releases its left resource.

$\text{Release_Tokens}(P)$: $P.\text{flag} \leftarrow \text{reverse}(P.\text{flag})$.

The Flags A and B, and virtual tokens. Since livelock is caused by different processes simultaneously locking the same resource, we can avoid that problem by a scheme which enables only one process to lock a resource at any given time. We use the concept of a token, where possession of a token enables a process to lock resources.

In DPCHAIN tokens are virtual. There is no variable called “token” in our code. Instead, we implement tokens by the use of a shared variable $P.\text{flag}$ for each process P . The value of $P.\text{flag}$ is always either A or B, and a process P “has its right token” if P ’s flag is the same as that of its right neighbor (if it has a right neighbor) and P “has its left

token” if P’s flag is different from that of its left neighbor (if it has a left neighbor). By default, P1 always “has its left token” and Pn always “has its right token.” By this simple method, using only one bit per process, we guarantee that no two adjacent processes have both both of its tokens, while simultaneously guaranteeing that at least one process in the chain has both of its tokens.

Clauses and Priorities: The third column of each action given in Table 3.1 consists of a list of clauses, each of which is a Boolean expression over the variables and functions which are computable by a process P. All of those clauses must be true for the action to be enabled. Priorities are also assigned in Table 3.1. The guard of each action contains the unwritten clause that no action whose priority number is lower is enabled. For example, if Error (P) holds, then no action other than Action A1 is enabled.

The Program: The algorithm DPCHAIN is almost anonymous, i.e., all processes have the same program, except for the two end processes of the chain, whose programs are very slightly different, due to the slightly different definition of Left_Enabl_d for P1 and Right_Enabl_d for Pn.

Table 3.1: Actions of DPCHAIN

A1 priority 1	Detect Error	Error (P)	→ Release_Left(P) Release_Right(P) P.state ← idle
A2 priority 2	Read Request	P.state = idle User(P).state = request	→ P.state ← waiting
A3 priority 2	Read Satisfaction	P.state ≠ idle User(P).state = sat	→ P.state ← idle
A4 priority 3	Release Tokens	P.state = idle Has_Tokens(P)	→ Release_Tokens(P)
A5 priority 3	Release Left	¬Has_Tokens(P) P.state ≠ executing Holds_Left(P)	→ Release_Left(P)
A6 priority 3	Release Right	¬Has_Tokens(P) P.state ≠ executing Holds_Right(P)	→ Release_Right(P)
A7 priority 3	Lock Left	P.state = waiting Has_Tokens(P) Left_Free(P)	→ Lock_Left(P)
A8 priority 3	Lock Right	P.state = waiting Has_Tokens(P) Right_Free(P)	→ Lock_Right(P)
A9 priority 3	Start Execution	P.state = waiting Has_Tokens(P) Holds_Left(P) Holds_Right(P)	→ P.state ← executing Release_Tokens(P)

3.3 Proof of Correctness of DPCHAIN

Lemma 3.1:

(a) From an arbitrary configuration, the network will reach a legitimate configuration within one round.

(b) From a legitimate configuration, the network will never reach an illegitimate configuration.

Proof. If the configuration is illegitimate by contention, i.e., two processes simultaneously hold a resource, within one round, at least one of these processes will notice the contention and release the process by executing Action A1. Thus, there will be no more contention after one round has elapsed. If the configuration is illegitimate by premature execution of some P, then P will execute Action A1, returning to the state idle. No action of DPCHAIN can cause a new contention or premature execution to occur, so the system will never enter an illegitimate configuration from a legitimate configuration. Henceforth, we will assume that the network is always in a legitimate configuration.

Pseudo-Time. We define an integral function $\tau(P)$ for all processors P as follows:

$$\tau(P_i) = \begin{cases} 0 & \text{if } i = 1 \\ \tau(P_{i-1}) - 1 & \text{if } i > 1 \text{ and Left_Enabld}(P_i) \\ \tau(P_{i-1}) + 1 & \text{otherwise} \end{cases}$$

Remark 3.1 For any $1 \leq i, j \leq n$, $|\tau(P_i) - \tau(P_j)| \leq |i - j|$

Let $\text{Num}(P)$ be the number of times that P has executed `Release_Tokens` since the network was initialized. Let $\Delta_i = \text{Num}(P_i) - \text{Num}(P_1) - \frac{1}{2} \tau(P_i)$.

Lemma 3.2 For any $1 \leq i \leq n$, Δ_i is constant.

Proof. Whenever P_i executes `Release_Tokens`, $\text{Num}(P_i)$ increases by 1 and $\tau(P_i)$ increases by 2. Whenever P_1 executes `Release_Tokens`, $\text{Num}(P_1)$ increases by 1 and $\tau(P_i)$ decreases by 2.

$$\text{Let } T = \frac{1}{4} n(n-1) + n \text{Num}(P_1) + \frac{1}{2} \sum_{i=2}^n \tau(P_i) \quad (3.1)$$

Remark 3.2 T is an integer, and $n \text{Num}(P_1) \leq T \leq n \text{Num}(P_1) + \frac{1}{2} n(n-1)$

Lemma 3.3 During any given step, T increases by the number of processes that execute `Release_Tokens` during that step.

Proof. Execution of `Release_Tokens(P_i)` causes $\tau(P_i)$ to increase by 2 if $i > 1$, and hence causes T to increase by 1. Execution of `Release_Tokens(P_1)` causes $\text{Num}(P_1)$ to increase by 1 and causes $\tau(P_i)$ to decrease by 2 for all $i > 1$, and hence causes T to increase by 1.

Lemma 3.4 Starting from any configuration, T eventually increases.

Proof. Pick i such that $\tau(P_i)$ is minimum. Then `Has_Tokens(P_i)`, which implies that P_i will eventually execute `Release_Tokens`, by executing Action A4 or A9. By Lemma 3.3 we are done.

Lemma 3.5 Starting from any given configuration, for any $1 \leq i \leq n$, P_i eventually executes `Release_Tokens`.

Proof. By Lemma 3.4, T increases without bound. By (3.1), $\text{Num}(P_1)$ increases without bound, since the other two terms of the right side are bounded.

$\text{Num}(P_i) = \text{Num}(P_1) + \frac{1}{2} \tau(P_i) + \Delta_i$ by the definition of Δ_i , $\tau(P_i) \geq -i$, and Δ_i is constant.

Thus $\text{Num}(P_i)$ increases without bound.

Lemma 3.6 If $\text{User}(P).\text{state} = \text{request}$, $P.\text{state} \neq \text{executing}$, and `Has_Tokens(P)`, then P will eventually execute Action A9.

Proof. By Lemma 3.5, P will execute Release_Tokens. Since P cannot execute Action A4, it must execute Action A9.

Theorem 3.1 The algorithm DPCHAIN is correct.

Proof. If a process receives a request from its user, then, by Lemma 3.6, it must eventually execute Action A9, after which it must eventually complete that execution.

3.4 The Algorithm DPRING

We now adapt the algorithm DPRING to the ring topology, as described at the beginning of this chapter. If we use the same code as DPCHAIN, deadlock can occur.

Note that in DPCHAIN, both end processes, P1 and Pn, have programs that are slightly different from the middle processes, P2, . . . , Pn-1. We will do the same for DPRING. The difference is that in DPRING the end processes are neighbors, and so we must ensure that they do not execute simultaneously.

As in DPCHAIN, we let each process has two virtual tokens, one that it shares with its left neighbor, the other with its right. Each middle process has just one flag, and it uses the same rules as DPCHAIN to decide whether it holds none, one, or both of its tokens. Each end process has two flags, P.left_flag and P.right_flag. P1.right_flag and Pn.left_flag are their “normal” flags, which are used to determine whether they hold the resources they share with their middle neighbors. The other flag, P1.left_flag or Pn.right_flag, is used by the end process to decide whether it holds the “end token,” i.e., permission to use the end resource. If both end processes’ end flags are equal, Pn has the end token. If they are different, P1 has the token.

As long as neither end resource has a request, the end token shuttles endlessly back and forth between the end processes. Each time an end process has the end token, it checks to see whether it has its other token and also whether its status is "waiting." If both are true, it keeps the end token and waits, if necessary, until its middle neighbor (P_2 or P_{n-1}) has finished executing, and then locks both resources and starts executing and releases both tokens by reversing both flags. In all other cases, it immediately releases the end token by releasing just its end flag.

An end process has more variables than a middle process, but it allow their neighbors to see its variables selectively in such a way that, to its middle neighbor, the end process appears to be just another process. Thus, P_2 sees $P_1.right$ but not $P_1.left$, while P_{n-1} sees $P_n.left$ but not $P_n.right$.

Each middle process runs exactly the same code as a middle process of DPCHAIN. In fact, there is no need for the process to even know that it is running DPRING instead of DPCHAIN. For that reason, we simply use Table 3.1 for the actions of a middle process.

The Deadlock Problem. There is a deep mathematical reason that it is difficult for an asynchronous algorithm on the ring to avoid deadlock. This has to do with the fact that the topology of the ring is not simply connected, i.e., it has a non-contractible cycle. (The same kind of problem arises some certain other distributed problems on any non-simply connected topology, such as construction of a synchronizer.)

If we attempt to use a strict analog of DPCHAIN on the ring, deadlock may result. The key to resolving this problem is to break the cycle in some way. We do this by

designating P_1 and P_n to be end (or leader) processes, with codes that differ from that of the normal processes, P_2, \dots, P_{n-1} .

We do this by assigning colors to the tokens. The normal tokens that shuttle back and forth between the normal processes we assign the color 0. the one end token that shuttles back and forth between P_1 and P_n , we assign the color 1.

The tokens have different priorities. If a process has a token of color 0, it holds it until it has the other token. But if a process (always an end process) needs tokens of both colors, and it has a token of color 1 but not the token of color 0, it releases the token of color 1 even if its state is waiting. This scheme prevents deadlock.

The scheme can be extended to other topologies by having more colors, although that is beyond the scope of this thesis.

3.4.1 Formal Definition of DPRING

We let P_1, \dots, P_n be the agent processors, and $U_i = \text{User}(P_i)$ the corresponding user processors. We assume a ring topology, i.e., P_i and P_{i+1} are adjacent for $i < n$, and P_n and P_1 are adjacent. Each P_i has the same code, except for P_1 , which is the leader .

The code for P_1 is given in Table 3.2. We have carefully designed DPRING so that, from the viewpoint of any process other than the leader, it is identical to DPCHAIN.

Thus, the code for P_i , for $i > 0$ is given in Table 3.1

We write:

$$\text{User}(P_i) = U_i$$

$$\text{Left}(P_i) = \begin{cases} P_n & \text{if } i = 1 \\ P_{i-1} & \text{otherwise} \end{cases}$$

$$\text{Right}(P_i) = \begin{cases} P_1 & \text{if } i = n \\ P_{i+1} & \text{otherwise} \end{cases}$$

Variables of DPRING:

$P.\text{flag} \in \{A,B\}$ if $P \in \{P_2, \dots, P_{n-1}\}$. This variable can be read by P 's neighbors.

$P.\text{left_flag} \in \{A,B\}$ if $P \in \{P_1, P_n\}$. This variable can be read by $\text{Left}(P)$.

$P.\text{right_flag} \in \{A,B\}$ if $P \in \{P_1, P_n\}$. This variable can be read by $\text{Right}(P)$.

$P.\text{state} \in \{\text{waiting, executing, idle}\}$ This variable can be read by $\text{User}(P)$, but not by its neighbor agents.

Functions of DPRING:

$\text{Holds_Left}(P) \equiv P$ is holding its left resource.

$\text{Holds_Right}(P) \equiv P$ is holding its right resource.

$\text{Left_Free}(P) \equiv$ no process is holding P 's left resource.

$\text{Right_Free}(P) \equiv$ no process is holding P 's right resource.

$$\text{Left_Nbr_Flag}(P) = \begin{cases} P_n.\text{right_flag} & \text{if } P = P_1 \\ P_1.\text{right_flag} & \text{if } P = P_2 \\ \text{Left}(P).\text{flag} & \text{otherwise} \end{cases}$$

$$\text{Right_Nbr_Flag}(P) = \begin{cases} P_1.\text{left_flag} & \text{if } P = P_n \\ P_n.\text{left_flag} & \text{if } P = P_{n-1} \\ \text{Right}(P).\text{flag} & \text{otherwise} \end{cases}$$

$\text{Left_Enabl}(P) \equiv \text{Left_Nbr_Flag}(P) \neq P.\text{flag}$

$\text{Right_Enabl}(P) \equiv \text{Right_Nbr_Flag}(P) = P.\text{flag}$

$\text{Has_Tokens}(P) \equiv \text{Left_Enabl}(P) \wedge \text{Right_Enabl}(P)$

$$\text{Rest}(P) \equiv \begin{cases} \neg\text{Holds_Right}(P) \wedge \neg\text{Holds_Left}(P) \wedge (P.\text{state} = \text{idle}) \\ \wedge \neg\text{Left_Enabl}(P) \wedge \neg\text{Right_Enabl}(P) & \text{if } P \in \{P_1, P_n\} \\ \neg\text{Holds_Right}(P) \wedge \neg\text{Holds_Left}(P) \wedge (P.\text{state} = \text{idle}) & \text{otherwise} \end{cases}$$

$$\begin{aligned} \text{Error}(P) \equiv & (\text{Holds_Left}(P) \wedge \neg\text{Left_Enabl}(P)) \vee \\ & (\text{Holds_Right}(P) \wedge \neg\text{Right_Enabl}(P)) \vee \\ & ((P.\text{state} = \text{executing}) \wedge (\neg\text{Holds_Right}(P) \vee \neg\text{Holds_Left}(P))) \vee \\ & ((P.\text{state} = \text{idle}) \wedge (\text{Holds_Right}(P) \vee \text{Holds_Left}(P))) \end{aligned}$$

Macros of DPRING:

Lock_Right(P): P locks its right resource.

Lock_Left(P): P locks its left resource.

Release_Right(P): P releases its right resource.

Release_Left(P): P releases its left resource.

Release_Tokens(P):

if P = P1 then

P.left_flag ← (Pn.right_flag)

P.right_flag ← reverse(P2.flag)

else if P = Pn then

P.left_flag ← Pn-1.flag

P.right_flag ← reverse(P1.left_flag)

else

P.flag ← reverse(P.flag)

endif

Table 3.2: Actions of DPRING for $P \in \{P1, Pn\}$

B1 priority 1	Correct Error	Error (P)	→ Release_Left(P) Release_Right(P) P.state ← idle
B2 priority 2	Read Request	P.state = idle User(P).state = request	→ P.state ← waiting
B3 priority 2	Read Satisfaction	P.state ≠ idle User(P).state = sat	→ P.state ← idle
B4 priority 3	Release Right Token	P = P1 P.state = idle Right_EnablD(P)	→ Release_Tokens(P)
B5 priority 3	Release Left Token	P = Pn P.state = idle Left_EnablD(P)	→ Release_Tokens(P)
B6 priority 3	Lock Left	P.state = waiting Has_Tokens(P) Left_Free(P)	→ Lock_Left(P)
B7 priority 3	Lock Right	P.state = waiting Has_Tokens(P) Right_Free(P)	→ Lock_Right(P)
B8 priority 3	Start Execution	P.state = waiting Has_Tokens(P) Holds_Left(P) Holds_Right(P)	→ P.state ← executing Release_Tokens(P)
B9 priority 4	Shuttle Left	P = P1 Left_EnablD(P)	→ P.left_flag ← Pn.right_flag
B10 priority 4	Shuttle Right	P = Pn Right_EnablD(P)	→ P.right_flag ← reverse(P1.left_flag)

3.5 Proof of Correctness of DPRING

Lemma 3.7

- (a) From an arbitrary configuration, the network will reach a legitimate configuration within one round.
- (b) From a legitimate configuration, the network will never reach an illegitimate configuration.

The proof is the same as that of Lemma 3.1. Henceforth, we will assume that the network is always in a legitimate configuration.

Pseudo-Time. We define an integral potential $\tau(P)$ for all processors P as follows:

$$\tau(P_i) = \begin{cases} 1 & \text{if } i = 1 \\ \tau(P_{i-1}) - 1 & \text{if } i > 1 \text{ and Left_Enabl}(P_i) \\ \tau(P_{i-1}) + 1 & \text{otherwise} \end{cases}$$

Remark 3.3 For any $1 \leq i, j \leq n$, $|\tau(P_i) - \tau(P_j)| \leq |i - j|$

Let $\text{Num}(P)$ be the number of times that P has executed `Release_Tokens` since the network was initialized. Let $\Delta_i = \text{Num}(P_i) - \text{Num}(P_1) - \frac{1}{2} \tau(P_i)$.

Lemma 3.8 For any $1 \leq i \leq n$, Δ_i is constant.

The proof is the same as that of Lemma 3.2.

Let
$$T = \frac{1}{4} n(n-1) + n \text{Num}(P_1) + \frac{1}{2} \sum_{i=2}^n \tau(P_i) \tag{3.2}$$

Remark 3.4 T is an integer, and $n \text{Num}(P_1) \leq T \leq n \text{Num}(P_1) + \frac{1}{2} n(n-1)$

Lemma 3.9 During any given step, T increases by the number of processes that execute `Release_Tokens` during that step.

The proof is the same as that of Lemma 3.3.

Lemma 3.10 If the configuration is legitimate, then within four rounds, either $P_1.\text{left_flag}$

or $P_n.right_flag$ will change.

Proof.

Case I: $Left_Enabl(P_1)$. We have two subcases.

Subcase I.a: $P_1.state = waiting$ and $Right_Enabl(P_1)$. Within three rounds, P_1 will execute Action B8. (P_1 may have to execute one or both locking actions, B6 and B7, first)

Subcase I.b: $P_1.state \neq waiting$ or $\neg Right_Enabl(P_1)$. Then, P_1 is enabled to execute Action B9. Within one round, either P_1 will execute Action B9, or that action will be neutralized by Action B8 being enabled, reducing to subcase I.a.

Case II: $Right_Enabl(P_n)$. Similar to Case I.

Lemma 3.11 If P is a minimum of the function τ , then P eventually executes $Release_Tokens$.

Proof. $P = P_i$ for some $1 \leq i \leq n$.

Case I: $1 < i < n$.

Subcase I.a: $P.state = waiting$. Then P will execute Action A9 within three rounds.

Subcase I.b: $P.state = idle$. Then within one round, either P will execute Action A4, and we are done, or P will execute Action A2, reducing to Subcase I.a.

Subcase I.c: $P.state = executing$. Eventually $User(P).state = sat$, after which P executes A3, reducing to Subcase I.b.

Case II: $i = 1$. Then $Right_Enabl(P)$.

Subcase II.a: $P_1.state = waiting$. By Lemma 3.10, $Left_Enabl(P)$ will hold within four rounds, and within three more rounds, P_1 will execute Action B8.

Subcase II.b: $P1.state = idle$. Within one round, either Action B4 will execute, in which case we are done, or $P1.state \leftarrow waiting$, reducing to Subcase II.a.

Subcase II.c: $P1.state = executing$. Eventually $User(P).state = sat$, after which P1 will execute Action B3 followed by Action B4, and we are done.

Case III: $i = n$. Similar to Case II.

Lemma 3.12 Starting from any configuration, T eventually increases.

Proof. Let P be a process such that $\tau(P)$ is minimum. By Lemma 3.11, P eventually executes Release_Token. Then, by Lemma 3.9, we are done.

Lemma 3.13 Starting from any given configuration, for any $1 \leq i \leq n$, P_i eventually executes Release_Tokens.

The proof is the same as that of Lemma 3.5, except that we use Lemma 3.12 instead of Lemma 3.4.

Lemma 3.14 If $User(P).state = request$, $P.state \neq executing$, and $Has_Tokens(P)$, then P will eventually execute Action A9 or B8.

Proof.

Case I: $P = P_i$ for $1 < i < n$. By Lemma 3.5, P will execute Release_Tokens. Since P cannot execute Action A4, it must execute Action A9.

Case II: $P = P1$. Within one round, $P1.state \leftarrow waiting$. After that, P1 executes Release_Tokens eventually. by Lemma 3.13. Since P cannot execute Action B4, it must execute Action B8.

Case III: $P = Pn$. Similar to Case II.

Theorem 3.2 The algorithm DPRING is correct.

Proof. If a process receives a request from its user, then, by Lemma 3.14, it must eventually execute Action A9 or Action B8, after which it must eventually complete that execution.

CHAPTER 4

CONCLUSION AND FUTURE RESEARCH

We propose a self-stabilizing solution for non-uniform bi-directional rings under unfair distributed daemon without using message passing model and token circulation method. We present solution for composite size of ring for which Burns and Pachl claim in [4] that there is no solution possible. First we adapt DPCHAIN algorithm for chain topology and then we adapt DPRING Algorithm for ring topology to make sure that both end processes don't execute simultaneously. By this virtual token scheme we prevent livelock, starvation and deadlock in ring topology without using any real token. The scheme can be extended to other topologies by having more colors, although that is beyond the scope of this thesis.

BIBLIOGRAPHY

- [1] E.W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Comm. ACM*, vol. 17, no. 11, pp. 643–644, Nov. 1974.
- [2] E.W. Dijkstra, "A belated proof of self stabilization," *Distributed Computing*, 1:5-6, 1986
- [3] Shmuel Katz and Kenneth Perry. *Self-stabilizing extensions for message-passing systems*. In Proc. 10th ACM PODC Symp., Quebec City, Aug 1990.
- [4] Burns and Pachl, *Uniform self-stabilizing ring* ACM Transactions on Programming Languages and Systems, Vol. 11, No. 2, April 1989, Pages 330-344.
- [5] Itkis, G., Lin, C., Simon, J.: *Deterministic, constant space, self-stabilizing leader election on uniform rings*. In: Workshop on Distributed Algorithms. (1995) 288–302
- [6] M. J. Fischer, N. A. Lynch, and M. S. Paterson, *Impossibility of Distributed Consensus with One Faulty Process*. *Journal of the ACM*, 32(2):374-382, April 1985.
- [7] E. ALLIOMANDI, M. J. Fischer, and N. A. Lynch, *Efficiency of Synchronous Versus Asynchronous Distributed System*. *Journal of the Association for Computing Machinery*, Vol 30, No 3, July 1983, pp 449-456
- [8] Schneider M.: Self-stabilization. *ACM Computing Surveys*, 25 (1993), 45-67
- [9] F. C. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *Journal of Parallel and Distributed Computing*, 35:43–48, 1996.
- [10] M. G. Gouda. *The triumph and tribulation of system stabilization*. In 9th International Workshop on Distributed Algorithms (WDAG), LNCS: 972, pages 1–18. Springer-Verlag, 1995.
- [11] Jensen and Rivindran 02 Jensen, E. Douglas and Binoy Ravindran, Eds. *IEEE Transactions on Computers, Special Section on Asynchronous Distributed Real-Time Systems, August 2002*.
- [12] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev, "Self-Stabilization by Local Checking and Global Reset (Extended Abstract)," *Proc. Eighth Int'l Workshop Distributed Algorithms (WDAG '94)*, pp. 326-339, 1994.

- [13] George Varghese. *Self-stabilization by local checking and correction*. Ph.D. Thesis MIT/LCS/TR-583, MIT, Ott 1992.
- [14] Dolev, S., Israeli, A., Moran, S.: *Uniform dynamic self-stabilizing leader election*. IEEE Transactions on Parallel and Distributed Systems 8 (1997) 424–440
- [15] Ghosh, S., Gupta, A.: *An exercise in fault-containment: Self-stabilizing leader-election*. Information Processing Letters (59) (1996) 281–288
- [16] H. Kakugawa and M. Yamashita (1997), *Uniform and Self-Stabilizing Token Rings Allowing Unfair Daemon*, IEEE Transactions on Parallel and Distributed Systems 8 (1997) 154–162
- [17] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115-138, Oct. 1971. Reprinted in *Operating Systems Techniques*, C.A.R. Hoare and R.H. Perrot, Eds., Academic Press, 1972, pp. 72-93. This paper introduces the classical synchronization problem of Dining Philosophers.
- [18] A. K. Datta, M. Gradinariu, and S. Tixeuil, "Self-stabilizing mutual exclusion using unfair distributed scheduler," Technical Report 1227, LRI, 1999.
- [19] N. Lynch. Fast allocation of nearby resources in a distributed system. Proceedings of the 12th ACM Symposium on Theory of Computing, 70-81, 1980.
- [20] Higham, L., Myers, S.: Self-stabilizing token circulation on anonymous message passing rings. Technical report, University of Calgary (1999)
- [21] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.
- [22] J. Beauquier, M. Gradinariu, and C. Johnen: Self-stabilizing and space optimal leader election under arbitrary scheduler on rings. Internal Report, LRI, Université de Paris-Sud, France. (1999)
- [23] B. Awerbuch and M. Saks, "A Dining Philosophers algorithm with polynomial response time " in Proc. 31st Annu. IEEE Symp. Foundations Comput. Sci., St. Louis, MO, Oct. 1990, pp. 65-74.
- [24] Jinzhong Niu, "Concurrency: Mutual Exclusion and Synchronization - Part 1" <http://www.sci.brooklyn.cuny.edu/~jniu/teaching/csc33200/files/1007-Concurrency01.pdf>
- [25] M. Rabin and D. Lehmann. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In Proceedings of 8th POPL, pages 133-138, 1981.

- [26] S. Dolev, A. Israeli and S. Moran. “*Self Stabilizing of dynamic systems assuming only read/write atomicity*”. Distributed Computing 7:3-16,1993.
- [27] A. K. Datta, M. Gradinariu, M. Raynal, “*Stabilizing mobile philosophers*” in: Information and Processing Letters, IRISA technical report number 1666, 2005, vol. 95, p. 299-306.
- [28] M. H. Karaata, “*Self-Stabilizing Strong Fairness Under Weak Fairness*”, IEEE Transactions on Parallel and Distributed Systems, v.12 n.4, p.337-345, April 2001
- [29] P. Danturi, M. Nesterenko, S. Tixeuil, “*Self-Stabilizing Philosophers With Generic Conflicts*”, in: Eighth International Symposium on Stabilization, Safety, and Security on Distributed Systems (SSS 2006), Dallas, Texas, A. K. Datta, M. Gradinariu (editors), Lecture Notes in Computer Science, Springer Verlag, November 2006, to appear p.
- [30] J. Beauquier, C. Johnen, S. Messika, “*All k-Bounded Policies Are Equivalent For Self-Stabilization*”, in: Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Dallas, Texas, A. K. Datta, M. Gradinariu (editors), Lecture Notes in Computer Science, November 2006 <http://www.lri.fr/parall/publis/messika/sss06.pdf>.
- [31] J. Beauquier, S. Delaët, S. Haddad. “*A 1-Strong Self-Stabilizing Transformer*”, in: Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Dallas, Texas, A. K. Datta, M. Gradinariu (editors), Lecture Notes in Computer Science, November 2006.
- [32] S. Bensalem, J. C. Fernandez, K. Havelund and L. Mounier, “*Confirmation of deadlock potentials detected by runtime analysis*”, In Parallel and Distributed Systems: Testing and Debugging (PADTAD’06), July 2006. Portland, Maine, USA.
- [33] N. Lynch. Upper bounds for static resource allocation in a distributed system. Journal Of Computation And Systems Sciences, 23(2):254-278, October 1981
- [34] P.A.G. Sivilotti, S. M. Pike, N. Sridhar , “*A New Distributed Resource-Allocation Algorithm with Optimal Failure Locality*” Proceedings of the 12th IASTED International Conference on Parallel and Distributed Computing and Systems, volume 2, pages 524-529. IASTED/ACTA press November 2000.
- [35] G Antonoiu and PK Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph tree that stabilizes using read/write atomicity. In Europar '99 Parallel Processing, Proceedings LNCS:1685, pages 823--830, 1999.
- [36] J. Beauquier, A. K. Datta, M. Gradinariu, and F. Magniette. “*Self-stabilizing local mutual exclusion and daemon refinement.*” In Proceedings of the 14th Conference on Distributed Computing (DISC), Lecture Notes in Computer Science 2180, pages 240--254, 2000.

- [37] S. Dolev, M. Gouda, and M. Schneider, “*Memory Requirements For Silent Stabilization*. In *PODC96 Proceeding Of The Fifteenth Annual ACM Symposium on Principles of Distributed Computing*”. Pages 27-34,1996.
- [38] Eugene Styer and Gary Peterson. Improved algorithms for distributed resource allocation. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 105-116, ACM SIGACT and SIGOPS, ACM, 1988.
- [39]S. Delaet, B. Ducourthial, S. Tixeuil, “*Self-Stabilization With r-Operators Revisited*”, in: *Journal of Aerospace Computing, Information, and Communication*, 2006.
- [40] T. Herault, P. Lemarinier, O. Peres, L. Pilard, J. Beauquier. “*Self-Stabilizing Spanning Tree Algorithm for Large Scale Systems (Brief Announcement)*”, in: *Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, Dallas, Texas, A. K. Datta, M. Gradinariu (editors), *Lecture Notes in Computer Science*, November 2006.
- [41] J. Lundelius and N. Lynch. Synthesis of efficient dining philosophers algorithms. January 1988. unpublished manuscript
- [42] Choy, M.; Singh, A.K.; “*Localizing failures in distributed synchronization*” *Parallel and Distributed Systems*, IEEE Transactions on Volume 7, Issue 7, July 1996 Page(s):705 – 716
- [43] M. Gradinariu, S. Tixeuil, “*Conflict Managers for Self-stabilization without Fairness Assumption*”, In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS 2007)*. IEEE, June 2007.
- [44] A. K. Datta , C. Johnen , F. Petit , V. Villain, “*Self-Stabilizing Depth-First Token Circulation In Arbitrary Rooted Networks*”, *Distributed Computing*, v.13 n.4, p.207-218, November 2000
- [45] A. Arora and M. Nesterenko. “*Stabilization-Preserving Atomicity Refinement.*”, *DISC 99*, *Proceedings of the Thirteenth International Symposium On Distributed Computing*. Pages 254-268,1999.
- [46] D. Hoover and J. Poole, “A Distributed Self-Stabilizing Solution to the Dining Philosophers Problem,” *Information Processing Letters*, vol. 41, no. 4, pp. 209-213, Mar. 1992.
- [47] S. Katz and K.J. Perry, Self-stabilizing extensions for message-passing systems, *Tech. Rept. STP-379-89*, MCC, November 1989.

- [48] M. Mizuno and M. Nesterenko. "A Transformation Of Self Stabilizing Serial Model Programs For Asynchronous Parallel Computing Environment", Information Proc. Letter, 66(6): 285-290, 1998.
- [49] Y. Tsay and R. Bargodia, "An Algorithm with Optimal Failure Locality for The Dining Philosopher Problem." Proc. of WDAG 1994, pp. 296--310.
- [50] K. Chandy and J. Misra. "The Drinking Philosophers Problem". ACM TOPLAS, 6(4):632-646, October 1984.
- [51] A. Arora and M. Nesterenko. "Dining Philosophers That Tolerate Malicious Crashes.", in 22nd Int'l Conference on Distributed Computing Systems, 2002, pp. 172--179.
- [52] M.G. Gouda, "The stabilizing philosopher: Asymmetry By Memory And By Action," Tech. Rept. TR-87-12, University of Texas at Austin, 1987.
- [53] D. Bein, A. K. Datta, L. L. Larmore: "Self-stabilizing Space Optimal Synchronization Algorithms on Trees", SIROCCO, LNCS 4056, Chester, UK 2006: 334-348
- [54] D. Bein and A. K. Datta and L. L. Larmore, "On Self-Stabilizing Search Trees" Disc06, LNCS 4167, Stockholm, Sweden, 2006, pages 76-89.
- [55] C. Boulinier, A. K. Datta, L. L. Larmore, and F. Petit, "An Efficient Self-Stabilizing Distributed Algorithm for BFS Tree Construction", SCS, 1, 2007.
- [56] Doina Bein, Ajoy K Datta, Chitwan K Gupta, and Lawrence L. Larmore, "Local Synchronization On Oriented Rings", 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2008), November 21-23, 2008, Detroit, Michigan, Lecture Notes in Computer Science, Volume 5340, pp. 141-156.

VITA

Graduate College
University of Nevada, Las Vegas

Chitwan Kumar Gupta

Local Address:

965 Cottage Grove Ave Apt 66
Las Vegas, Nevada 89119

Home Address:

201, Scheme No.-3, Basant Vihar
Alwar, Rajasthan, India 301001

Degrees:

Bachelor of Engineering, Computer Science, May 2005
University of Rajasthan, Jaipur, India.

Special Honors and Awards:

Database Administrator & Developer , Graduate Assistantship
Disability Resource Center

Teaching Assistantship
School Of Computer Science

Publications:

Doina Bein, Ajoy K Datta, Chitwan K Gupta, and Lawrence L. Larmore,
"Local Synchronization On Oriented Rings", 10th International
Symposium on Stabilization, Safety, and Security of Distributed Systems
(SSS 2008), November 21-23, 2008, Detroit, Michigan, Lecture Notes in
Computer Science, Volume 5340, pp. 141-156.

Thesis Title: Self-Stabilizing Protocol For Anonymous Oriented Bi-directional Rings Under Unfair Distributed Schedulers With A Leader

Thesis Examination Committee:

Chairperson, Dr. Lawrence L. Larmore, Ph. D.

Committee Member, Dr. Ajoy K. Datta, Ph. D.

Committee Member, Dr. Laxmi P. Gewali, Ph. D.

Graduate Faculty Representative, Dr. Rohan J. Dalpatadu, Ph. D.