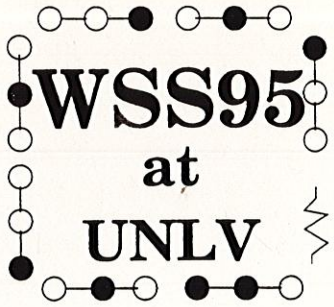

PROCEEDINGS OF THE SECOND WORKSHOP ON SELF-STABLIZING SYSTEMS

May 28-29, 1995

Technical Report
Department of Computer Science
University of Nevada, Las Vegas
Box 454019
Las Vegas, NV 89154-4019

PREFACE

The topic of self-stabilizing systems has received growing attention in recent years. The interest in self-stabilization reflects its importance in existing distributed systems. It allows automatic recovery following transient faults and the possibility of starting a distributed system without a global starting signal. In 1974, E. W. Dijkstra published the pioneering paper in this field, presenting self-stabilizing mutual-exclusion protocols. The progress since 1974 proves that the self-stabilization paradigm is not tied solely to the mutual-exclusion task. A (short) partial list of the tasks that are currently discussed includes: topology update, clock synchronization, leader election, graph algorithms, and flow control.

Self-stabilization is applicable to many tasks in communication networks and multiprocessor computers. In fact, some protocols have been designed to be self-stabilizing, without identifying them as such, simply to meet operational requirements. We believe that the growing knowledge of self-stabilization should become widespread among both practitioners and theoreticians, so that we can all benefit from robust distributed systems in the future.

The papers in this volume were contributed for presentation at the Second Workshop on Self-Stabilizing Systems, held May 28-29, 1995, in Las Vegas, Nevada, USA. The workshop is sponsored by the Department of Computer Science, University of Nevada, Las Vegas and the Information Science Research Institute, University of Nevada, Las Vegas. The papers were selected by the program committee according to their relevance to the workshop and quality. Submissions were not formally refereed and authors are expected to submit their papers to fully refereed journals. In particular, authors are encouraged to submit the full version of the paper to the Chicago Journal of Theoretical Computer Science that will dedicate a special issue to selected papers presented at the workshop.

The program committee would like to thank all the authors who submitted extended abstracts for consideration. We would also like to thank William W. Wells (Dean, College of Engineering, UNLV), Lawrence L. Larmore (Chairman, Department of Computer Science, UNLV), and Thomas Nartker (Director, Information Science Research Institute, UNLV) for their support in sponsoring the workshop. Last, but not least, we thank the following colleagues for assistance in evaluating the submissions: Eyal Adi, Nian-Shing Chen, Jorge A. Cobb, Michael Franz, Haim Gurin, Furman Haddix, Gene Itkis, Alexander Keizelman, Galia Kopelman, Yaacov Kuperstein, Claudia Lerner, Chenadan Lin, Tomer Lifshitz, Leon Meister, Boris Meltser, Marco Schneider, Aharon Tubman, and Lih-Chyau Wuu.

Shlomi Dolev (Program Chair)
Ajoy K. Datta (Local Arrangement Chair)
Las Vegas, May 1995

Electronic information may be obtained through:
URL: http://www.cs.bgu.ac.il/~dolev/WSS95
URL: http://www.unlv.edu/~datta/wss.html

Program Committee

James E. Burns (Bellcore)

Shlomi Dolev (Ben-Gurion Univ.)

Mohamed Gouda (Univ. of Texas)

Ted Herman (Univ. of Iowa)

Shing-Tsaan Huang (Tsing Hua Univ. Taiwan)

Janos Simon (Univ. of Chicago)

John Spinelli (Union College)

Participants

Amisaki, Takashi (*ami@cis.shimane-u.ac.jp*)	Shimane University
Arora, Anish (*anish@cis.ohio-state.edu*)	The Ohio State University
Beauquier, Joffroy (*jb@lri.fr*)	L.R.I. University of Paris-Sud
Bourgon, Brian (*bourgon@nevada.edu*)	University of Nevada, Las Vegas
Burns, James E. (*burns@nova.bellcore.com*)	Bellcore
Cherkaoui, Omar (*cherkaoui.omar@uqam.ca*)	Universite du Quebec In Montreal
Datta, Ajoy Kumar (*datta@cs.unlv.edu*)	University of Nevada, Las Vegas
Debas, Oliver (*debas@lri.fr*)	LRI Paris SUD
Deffaix, Chris (*deffaix@lif.cicrp.jussieu.fr*)	LAFORIA Paris 6
Delaet, Sylvie (*delaet@lri.fr*)	L.R.I. University of Paris-Sud
Derby, Jerry (*jderby@nevada.edu*)	University of Nevada, Las Vegas
Dolev, Shlomi (*dolev@cs.bgu.ac.il*)	Ben-Gurion University
Gacs, Peter (*gacs@cs.bu.edu*)	Boston University
Gewali, Laxmi P. (*laxmi@cs.unlv.edu*)	University of Nevada, Las Vegas
Ghosh, Sukumar (*ghosh@cs.uiowa.edu*)	The University of Iowa
Gouda, Mohamed G. (*gouda@cs.utexas.edu*)	University of Texas at Austin
Gupta, Arobinda (*agupta@cs.uiowa.edu*)	University of Iowa
Herman, Ted (*herman@cs.uiowa.edu*)	University of Iowa
Hoover, Jim (*hoover@cs.ualberta.ca*)	University of Alberta
Huang, Shing-Tsaan (*sthuang@nthu.edu.tw*)	National Tsing Hua University
Johnen, Colette (*colette@lri.fr*)	L.R.I. University of Paris-Sud
Kakugawa, Hirotsugu (*kakugawa@se.hiroshima-u.ac.jp*)	Hiroshima University
Kranakis, Evangelos (*kranakis@scs.carleton.ca*)	Carleton University
Kutten, Shay (*kutten@watson.ibm.com*)	IBM
Lawrence, James (*jamesel@cs.unlv.edu*)	University of Nevada, Las Vegas
Lin, Chengdian (*lin@cs.uchicago.edu*)	University of Chicago
Masuzawa, Toshimitsu (*masuzawa@is.aist-nara.ac.jp*)	Nara Inst. of Sci. and Tech. (NAIST)
Schneider, Marco (*marco@cs.utexas.edu*)	University of Texas at Austin
Shukla, Sandeep Kumar (*sandeep@fcs.albany.edu*)	SUNY - ALBANY
Simon, Janos (*simon@cs.uchicago.edu*)	U of Chicago
Tokura, Nobuki (*tokura@ics.es.osaka-u.ac.jp*)	Osaka University
Tsujino, Yoshihiro (*tsujino@ics.es.osaka-u.ac.jp*)	Osaka University
Varghese, George (*varghese@askew.wustl.edu*)	Washington University in St. Louis
Yen, I-Ling (*yen@cps.msu.edu*)	Michigan State University

CONTENTS

Paper Number 1

A Fault-Tolerant and Self-Stabilizing Protocol for the Topology Problem

Toshimitsu Masuzawa

A Fault-Tolerant and Self-Stabilizing Protocol for the Topology Problem[†]

Toshimitsu Masuzawa
Graduate School of Information Science
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara, 630-01, Japan
e-mail: masuzawa@is.aist-nara.ac.jp

Abstract We investigate the possibility of designing protocols that are both self-stabilizing and fault-tolerant, in asynchronous model of distributed systems. It has been known that no such protocols exist even for fundamental problems, including the counting problem (i.e., the problem to determine the number of processors) in a ring network. In this paper, we define a more generalized problem called the topology problem in faulty networks (i.e., the problem to find a network topology that may contain false information about connections between faulty processors), and show that there exists a self-stabilizing and fault-tolerant protocol for the topology problem if the neighbors' IDs are initially available at each processor. This verifies that the local information about neighbors' IDs is significant for designing fault-tolerant and self-stabilizing protocols.

Keywords Distributed Algorithms, Self-stabilization, Fault-tolerance, Topology problem, Asynchrony

1 Introduction

A *self-stabilizing* protocol is a protocol that achieves its intended behavior regardless of the initial network configuration (i.e., global state). Thus, a self-stabilizing protocol is resilient to *any number* and *any type* of *transient* faults: after the last fault occurs, the protocol starts to converge to its intended behavior. The transient faults model the faults such that the memory of processors or the messages in transit are corrupted by the fault, but the programs of the processors are left unchanged. The concept of self-stabilization was introduced by Dijkstra [11], and has attracted a great deal of attention in recent years (e.g., [1, 2, 3, 5, 6, 7, 9, 12, 13, 17, 18, 20]).

A *k-fault-tolerant* protocol (for a *specific* fault model) is a protocol that always achieves its intended behavior regardless of k faulty processors deviating from their protocols, when started from a *designated* initial configuration. A variety of fault models have been studied: they ranges from relatively benign crash faults to completely arbitrary faults (cf. [16]).

The two concepts of reliability, self-stabilization and fault-tolerance, have been traditionally investigated separately. Gopal and Perry [15], however, combine the two concepts for the first time. They consider the *general omission fault* (i.e., a faulty processor is subject to send and/or receive omission, and/or crashing), and show a compiler that transforms a fault-tolerant protocol for a *synchronous* system into a fault-tolerant and self-stabilizing protocol. They also present a fault-tolerant and self-stabilizing consensus protocol in an *asynchronous* system using unreliable failure detectors introduced in [10].

Anagnostou and Hadzilacos [4] consider the *crash fault* (i.e., a faulty processor stops prematurely and does nothing from that point on) and show that no 1-fault-tolerant and self-stabilizing protocols exist for a wide range of problems, including the *counting problem* (i.e., the problem to determine the size of the distributed system) in ring networks.

[†]This work was supported in part by the Scientific Research Grant-in-Aid from Ministry of Education, Science and Culture of Japan, and the Telecommunications Advancement Foundation's Research Grant.

They also present randomized 1-fault-tolerant and self-stabilizing protocols for the unique naming problem in ring networks.

In this paper, we investigate the possibility of protocols that are fault-tolerant (as to the *crash faults*) and self-stabilizing under some additional assumption: each processor initially knows the IDs of its neighbors.

We consider the *topology problem* (i.e., the problem to find the topology of the network). The problem is one of the fundamental problems in distributed systems, and several algorithms have been designed for the problem (cf. [8, 14, 19]). But no fault-tolerant and self-stabilizing algorithms have been proposed.

In the context of fault-tolerant and self-stabilizing algorithms, it may be impossible to determine the exact topology of the network, since it may impossible to determine connections between faulty processors. Thus, we first define the *topology problem* in a faulty network as a problem to find a network topology that may contain false information about connections between faulty processors. The problem is regarded as a generalized problem of the counting problem, since the solution for the counting problem can be easily obtained from that for the topology problem.

For the topology problem, we present a fault-tolerant and self-stabilizing protocol that solves the problem in spite of k faulty processors in $(k+1)$-connected networks. This verifies that the local information about neighbors' IDs plays an important role in design of fault-tolerant and self-stabilizing protocols.

The protocol in this paper uses *both* the neighbors' IDs and the information about connectivity. From this, we can think of a natural question: Is it possible to design a fault-tolerant and self-stabilizing protocol using *only* the neighbors' IDs or using *only* the connectivity information?

In this paper, we deny the possibility, that is, we show the information used in the protocol is minimal information to solve the problem. The result in [4] implies that there exists no 1-fault-tolerant and self-stabilizing protocol for the counting problem, if only the connectivity information is available and the neighbors' IDs are not available. This is because Anagnostou and Hadzilacos consider only a ring network and can use, in design of protocols, the knowledge that the network is 2-connected. In this paper, we also prove that the information about the neighbors' IDs is not sufficient information for designing a fault-tolerant and self-stabilizing protocol: we prove that any k-fault-tolerant and self-stabilizing protocol that solves the counting problem in $(k+1)$-connected networks cannot solve the problem in some k-connected network even if there exists *no* faulty processor.

The rest of this paper is organized as follows. Section 2 presents the computation model of distributed systems and several definitions. Section 3 shows a k-fault-tolerant and self-stabilizing topology protocol. The impossibility result is mentioned in Section 4.

2 Model

2.1 Distributed systems

A *distributed system* $D = (N, A)$ consists of a *processor network* (simply called a network) N and a *protocol* A. The network N is represented by an undirected graph $N = (P, L)$ where the vertex set P is the set of the processors and the edge set L is the set of the bidirectional communication links (simply called links). We consider a network of arbitrary topology. If $(p, q) \in L$ holds for processors p and q, then p and q are called *neighbors*. The protocol A is a collection of algorithms, one for each processor in the system.

Table 1: Glossary of notations

$N = (P, L)$	the network with processor set P and link set L.
p, q	processors.
id_p	the unique ID of processor p.
NID_p	the ID set of p's neighbors.
$\mathcal{N}^k$	the set of all k-connected networks.
$R_{p,q}$	the register used for communication from p to q.
$\mathcal{F}(E)$	the set of faulty processors in execution E.
$\mathcal{C}(E)$	the set of correct processors in execution E (i.e., $\mathcal{C}(E) = P - \mathcal{F}(E)$).
Π	a problem.
$\mathcal{L}_\Pi(N, F)$	the set of legal execution of problem Π on N with faulty processors F.
TI_p	the variable of processor p that stores the topology information.
AP	an application protocol. Each processor p has an application process AP_p.
TP	the topology computation protocol. Each processor p has the topology computation process TP_p.
RP	the reset protocol. Each processor p has the reset process RP_p.

Let $N = (P, L)$ be a network and p and q be processors in P. A *p-q path* is a sequence $(p_0(= p), p_1, p_2, \ldots, p_\ell(= q))$ of distinct processors such that $(p_i, p_{i+1}) \in L$ for each i $(0 \leq i \leq \ell - 1)$. Two p-q paths are *internally disjoint*, if the paths share no processor except for p and q. For a positive integer k, a *k-connected network* is a network where there exist at least k internally disjoint p-q paths for any distinct processors p and q. The set of all k-connected networks is denoted by $\mathcal{N}^k$.

We adopt the *link-register* model introduced in [13]. Two neighbors p and q communicate by the use of two *shared communication register* $R_{p,q}$ and $R_{q,p}$. The register $R_{p,q}$ (resp. $R_{q,p}$) can be written only by p (resp. q) and read only by q (resp. p); we say that p (resp. q) owns $R_{p,q}$ (resp. $R_{q,p}$). Every communication register is atomic with respect to read and write operations: all read and write actions to the same register can be serialized in time. In this paper, we simply use the term *register* for the shared communication register.

A processor is a (possibly infinite) state machine. A *configuration* of a system is a vector of processor states and register contents. Let S_i be the state set of the i^{th} processor and Σ_j be the set of symbols that can be stored in the j^{th} register.[1] The set C of all possible configurations is

$$C = (S_1 \times S_2 \times \cdots \times S_n \times \Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_m)$$

where n (resp. m) is number of the processors (resp. registers) in the system.

2.2 Protocols and processor faults

Each processor p has a unique ID (denoted by id_p), and executes its program composed of *atomic steps*. We assume that processor activity is managed by the *read/write demon* introduced in [13]: in any configuration, a single processor is activated to execute a single atomic step. In an atomic step, a processor can change its own state (i.e., execute internal computation) and execute either a single read or a single write operation, but not both.

[1] For convenience, we assume total orders on the processors and on the registers. These orders are used only to describe the configuration, and cannot be used in designing protocols.

An *execution* E of a protocol A is an infinite alternating sequence of configurations and atomic steps $E = c_0, a_0, c_1, a_1, \ldots$ where c_{i+1} $(i \geq 0)$ is reachable from c_i by a single atomic step a_i according to A. Notice that the protocol A defines a transition function that describes what are the possible legal steps that can be executed from a configuration. Configuration c_0 is called an *initial configuration* of E.

A processor is *faulty* if its behavior deviates from that prescribed by the protocol. We consider only *crash faults* of processors: a faulty processor stops prematurely and does nothing from that point on, however, it behaves correctly before stopping. In the model of the state machine, occurrence of the crash faults is modeled as execution of a special step called a *crash step*. The crash step changes the processor state into a special state, *crash state*, and has no effect on registers. In the crash state, no further step can be executed. The crash step can be executed at any state except for the crash state.

Given an execution E of a protocol A, it is possible to identify the set $\mathcal{F}(E)$ of faulty processors (i.e. those in the crash state after some point) and the set $\mathcal{C}(E)(= P - \mathcal{F}(E))$ of correct processors. If every processor in $\mathcal{C}(E)$ makes infinitely many steps in E, then E is called a *fair execution* of A. We consider only fair executions in this paper, and simply use the term an *execution* for a fair execution.

We assume a model of an *asynchronous system*: there is no assumption on the number of steps each processor executes in any prefix of an execution. Note that processor faults cannot be detected in such an asynchronous system because it is impossible to determine whether a processor has actually crashed or is only "very slow". This makes it difficult (sometimes impossible) to design fault-tolerant protocols in asynchronous systems.

Following [4], we introduce a *round* of an execution to measure the efficiency of protocols. For convenience, we assume that computation of each processor p proceeds in *cycles* as follows. In each cycle, p takes the following steps in a specified order: it reads its neighbors' registers, updates its state, and writes into the registers it owns. Such a cycle could be interrupted at any point by a crash step.

Let $E = c_0, a_0, c_1, a_1, \ldots$ be an execution of a protocol. We define the first *round* of E to be a minimal prefix of E such that each processor that is not in the crash state at c_0 has completed at least one cycle or executes the crash step in the prefix. Letting $c_{f(1)}$ be the last configuration of the first round, the second round of E is defined to be a minimal subsequence $c_{f(1)}, a_{f(1)}, c_{f(1)+1}, a_{f(1)+1}, \ldots c_{f(2)}$ such that each processor that is not in the crash state at $c_{f(1)}$ has completed at least one cycle or executes the crash step in the subsequence. The following rounds are defined in the similar way.

2.3 Fault-tolerance and self-stabilization

A *problem* specifies the required behavior of processors. Formally we define a problem to be a set of *legal executions* (i.e. those satisfying the problem requirement). For a network N and a set F of faulty processors, a problem Π on N with faulty processors F is defined by specifying a set of legal executions of Π on N with faulty processors F. The set of legal executions is denoted by $\mathcal{L}_\Pi(N, F)$. Letting $\mathcal{N}$ be a network set and k be a non-negative integer, a problem Π is called to be defined for $\mathcal{N}$ up to k faults, if $\mathcal{L}_\Pi(N, F)$ is defined for every network N in $\mathcal{N}$ and every processor set F satisfying $|F| \leq k$.

In what follows, we say that a configuration is a *designated initial configuration*, if each processor is in a prescribed initial state and each register contains a prescribed symbol as its initial value.

Definition 1 **(k-fault-tolerant protocol)** Let $\mathcal{N}$ be a network set, k be a non-negative integer, and Π be a problem defined for $\mathcal{N}$ up to k faults. A protocol A is a *k-fault-tolerant* (*k-ft*) protocol of Π in $\mathcal{N}$, if the following holds:

for any execution E of A starting from the *designated* initial configuration in any network $N(\in \mathcal{N})$, if it satisfies $|\mathcal{F}(E)| \leq k$, then E is an execution in $\mathcal{L}_\Pi(N, \mathcal{F}(E))$. ■

Definition 2 (k-fault-tolerant and self-stabilizing protocol) Let $\mathcal{N}$ be a network set, k be a non-negative integer, and Π be a problem defined for $\mathcal{N}$ up to k faults. A protocol A is a *k-fault-tolerant and self-stabilizing* (*k-ftss*) protocol of Π in $\mathcal{N}$, if the following holds:
for any execution E of A starting from *any* configuration in any network $N(\in \mathcal{N})$, if it satisfies $|\mathcal{F}(E)| \leq k$, then there exists a *suffix* E' of E such that E' is also a *suffix* of an execution in $\mathcal{L}_\Pi(N, \mathcal{F}(E))$. ■

A 0-*ftss* protocol is simply called a *self-stabilizing* (*ss*) protocol.

2.4 Counting and topology problems

Anagnostou and Hadzilacos ([4]) show that there exists no 1-*ftss* protocol for solving the *counting problem* in ring networks. The counting problem is defined as follows.

Definition 3 (counting problem) The solution of the *counting problem* Π^c in a network $N = (P, L)$ is $|P|$ (the number of processors of N). An execution $E = c_0, a_0, c_1, a_1, \ldots$ is legal if and only if there exists a non-negative integer i such that every correct processor in $\mathcal{C}(E)$ knows the solution in every configuration c_j $(j \geq i)$. ■

The impossibility result for the counting problem in [4] is strong: it holds even if a protocol is randomized, the ring is oriented, processors have unique IDs, or a distinguished processor exists.

In this paper, we consider a more generalized problem, the *topology problem* (i.e., a problem to determine the network topology), and show that, for every non-negative integer k, there exists a *k-ftss* protocol for the problem in $\mathcal{N}^{k+1}$ (i.e., for the set of $(k+1)$-connected networks) if every processor initially knows the IDs of its neighbors as well as its own ID. We assume that its own ID and the neighbors' IDs cannot be corrupted by transient errors: at any configuration, every processor correctly knows the neighbors' IDs as well as its own ID.

Note that for any pair of faulty processors it may be impossible to determine whether there is a link connecting them (the processors may crash before reporting what their links are). In other words, we can only determine the topology of the subnetwork that consists of links connecting to correct processors. Thus, the topology problem in a faulty network is defined as follows.

Definition 4 (topology problem) Let $N = (P, L)$ be a network of a system and F be a subset of its processors. A network $N' = (P', L')$ is a solution of the *topology problem* Π^t on N with faulty processors F, if and only if

$$(P' = P) \bigwedge ((L - \{e | e \in F \times F\}) \subseteq L' \subseteq (L \cup \{e | e \in F \times F\}))$$

holds. An execution $E = c_0, a_0, c_1, a_1, \ldots$ is legal, if and only if there exists a non-negative integer i such that all correct processors in $\mathcal{C}(E)$ know N' in every configuration c_j $(j \geq i)$ where N' is a solution of Π^t on N with faulty processors $\mathcal{F}(E)$. The configurations c_j $(j \geq i)$ are called *final legal configurations.* ■

From the solution of the topology problem, we can obtain the solution of the counting problem. Thus, we can regard the topology problem as a generalized problem of the counting problem.

Even if we solve the topology problem, we cannot obtain the exact topology, that is, the solution may contain false information about connections between faulty processors.[2] However, its information concerning the links incident to

[2] In this sense, the *topology approximation problem* is a better name for the problem.

correct processors contains no false connections, and it is highly expected that the solution of the topology problem is useful in many situations. For example, consider a routing problem for transferring a message from a processor, say p, to a processor, say q. In the situation where there may exist k undetectable faulty processors, one of its solutions is to send $k+1$ copies of the message along $k+1$ internally-disjoint p-q paths. We can determine the paths from the solution of the topology problem. These determined paths may contain some non-existent paths because the solution of the topology problem may not be an exact topology. Nevertheless, we can transfer a message from p to q (if both p and q are non-faulty), since the paths include at least a path that consists of only non-faulty processors and actually existing links. Moreover, since all processors obtain the same topology information (as the solution of the topology problem), all non-faulty processors can determine the same $k+1$ p-q paths by executing the same sequential algorithm on the topology information. Thus, the internal processors on these paths can determine (from the source p and the destination q) which neighbor it should relay the message to, even if the message does not contain the information about the whole path.

3 Fault-tolerant and self-stabilizing topology protocol

3.1 Overview of the *k-ftss* topology protocol

In the *k-ftss* topology protocol, each processor p maintains a local variable, TI_p, to store *topology information* it knows (Fig. 2). Topology information is a set of *local connections* and the local connection is a set of links incident to a single processor. In the protocol, the local connection is represented by a pair of a processor ID and the set of IDs of its neighbors.

The *k-ftss* topology protocol proposed in this paper is based on the following simple (non-stabilizing) *k-ft* topology protocol in $\mathcal{N}^{k+1}$. At the designated initial configuration, $TI_p = \{(id_p, NID_p)\}$ holds at each processor p, where NID_p stands for the ID set of p's neighbors, and every register is empty. Every processor p repeatedly exchanges its topology information with all neighbors, and updates TI_p by augmenting it with the newly obtained local connections. Since we consider only the crash faults, there is a some point after which the topology information does not change. The final topology information is same at all correct processors, since there exists a non-faulty path between any pair of correct processors. Moreover, the final topology information is a solution of the topology problem, since it contains the local connections of all correct processors.

In a *k-ftss* topology protocol, however, we can make no assumption on the initial value of TI_p: the initial topology information may contain false local connections, that is, it may contain non-existent processors and/or links. Thus, the protocol described above is not a self-stabilizing one. The idea for making the protocol self-stabilizing is to ensure that if the topology information contains a false local connection then some correct processor "eventually" detects the inconsistency. In the *k-ftss* topology protocol in this paper, therefore, when the topology information is updated, its consistency is checked. Whenever some processor detects inconsistency on its topology information, it issues a *reset request* to clear false topology information from the whole network.

We give a framework for designing a *k-ftss* protocol. In the framework, we consider a general problem apart from the topology problem, and, hence, we consider an *application protocol* instead of a topology protocol. The *k-ftss* protocol is designed in a layered manner, and consists of the following two protocols.

(A) *k-ftss reset protocol*: It provides reset facility and reliable communication between neighbors. Each processor p has a *reset process* RP_p to execute the reset protocol.

(B) *k-ft application protocol with inconsistency detector*: It is a *k-ft* protocol for a problem, say Π. Each processor p has an *application process* AP_p. It exchanges information with other application processes on its neighbors (using reliable communication the reset protocol provides) and solves Π. It also checks consistency of the information, and issues a reset request whenever it detects inconsistency.

These protocols communicate with each other by the following events: each protocol executes *output events* that are treated as *input events* by the other protocol.

Definition 5 (interface between AP_p and RP_p) The following events are executed at each processor p.

1. $AP_p \rightarrow RP_p$: The following events are generated by an application process AP_p and are treated as input events by a reset process RP_p.
   - $Send_{p,q}(m)$ (for every neighbor q of p): AP_p sends a message m to AP_q.
   - $Request_p$: AP_p issues a reset request to reset the system configuration.
2. $RP_p \rightarrow AP_p$: The following events are generated by RP_p and are treated as input events by AP_p.
   - $Free_{p,q}$ (for every neighbor q of p): RP_p is ready to accept another message sent from AP_p to AP_q.
   - $Receive_{q,p}(m)$ (for every neighbor q of p): RP_p delivers AP_p a message m that RP_p receives from q.
   - $Signal_p$: AP_p should be initialized (i.e. AP_p should restart the application process from the initial state that the protocol prescribes). ■

Using the events described above, an application process AP_p sends a message m to a neighboring application process AP_q as follows. First, AP_p executes $Send_{p,q}(m)$. The event is also an input event of a reset process RP_p, and make RP_p write the message m (with additional information for the reset protocol) into register $R_{p,q}$. When RP_q reads m from $R_{p,q}$, it executes $Receive_{p,q}(m)$ to deliver m to AP_q, and sends acknowledgment to RP_p through a register $R_{q,p}$. On reading the acknowledgment from $R_{q,p}$, RP_p executes $Free_{p,q}$ to inform AP_p that RP_p is ready to accept another message to AP_q.

For simplicity, we make the following assumptions without loss of generality.

(A1) After AP_p executes $Send_{p,q}(m)$ for some message m, it does not execute $Send_{p,q}(m')$ for another message m' until $Free_{p,q}$ occurs.

(A2) After $Free_{p,q}$ occurs, AP_p eventually executes $Send_{p,q}(m)$ for some message m, or p crashes.

Assumption (A1) may cause an "internal" deadlock within a processor p such that AP_p is waiting for $Free_{p,q}$ while RP_p is waiting for $Send_{p,q}$. In what follows, however, we assume that such an "internal" deadlock cannot occur, that is, every processor is "internally" consistent. This assumption is made only for simplicity, because it is easy to avoid such an "internally" inconsistent state.

3.2 *k-ft* topology protocol with inconsistency detector

This subsection shows the *k-ft topology protocol with inconsistency detector* TP, that is, an application protocol for the topology problem.

The protocol is based on the *k-ft* topology protocol described in the previous subsection. In the *k-ftss* topology protocol, however, we can make no assumption on the initial topology information each processor p stores in variable TI_p. Therefore, each topology computation process TP_p includes a process called a *inconsistency detector* to detect inconsistency on TI_p. When TI_p is updated, the inconsistency detector checks the new value of TI_p. If it detects inconsistency on TI_p, TP_p executes $Request_p$ to reset the configuration.

Now we only describe the way to detect inconsistency on the topology information. The topology information TI_p is clearly inconsistent, if at least one of the following holds. (In the sequel, NID_p stands for the ID set of p's *real* neighbors.)

(I1) TI_p does not contain (id_p, NID_p).

(I2) TI_p contains (id, NID) and (id', NID') such that $id = id'$ and $NID \neq NID'$.

(I3) TI_p contains (id, NID) and (id', NID') such that $id' \in NID$ and $id \notin NID'$.

If topology information satisfies at least one of the above conditions, it is called *locally inconsistent*; otherwise, it is called *locally consistent.*

Some inconsistency on the topology information, however, cannot be detected only by the above conditions. Consider the following configuration of a network $N = (P, L)$. Let p be a processor in P and id' be a non-existent processor ID (i.e., P contains no processor whose ID is id'). Assume that the topology information TI_q of every processor q except for p satisfies

$$TI_q = \bigcup_{r \in P - \{p\}} \{(id_r, NID_r)\} \cup \{(id_p, NID_p \cup \{id'\})\},$$

and p is in the crash state. In the configuration, the topology information of every correct processor is locally consistent. But it is not a solution of the topology problem, since it includes the non-existent ID id'. To avoid such a situation, we use information about the connectivity of the network.

We introduce an *extended network* of the topology information. Consider locally consistent topology information $TI = \{(id_i, NID^i) \mid 1 \leq i \leq n'\}$[3]. If a processor ID id appears in some NID^j $(1 \leq j \leq n')$ but there is no h $(1 \leq h \leq n')$ such that $id_h = id$, then we say that id is a *pending ID* of TI, and the set of the pending ID of TI is denoted by $pen(TI)$. We can induce a network from the topology information TI, by considering the topology information as a set of links. An extended network is defined to be a network obtained by augmenting the induced network with the links between all pairs of the pending IDs. Formally, the extended network $exN(TI)$ of TI is a network $exN(TI) = (exP(TI), exL(TI))$ where

- $exP(TI) = \bigcup_{1 \leq i \leq n'} (\{id_i\} \cup NID^i)$, and
- $exL(TI) = \bigcup_{1 \leq i \leq n'} \{(id_i, v) \mid v \in NID^i\} \cup \{(v, w) \mid v, w \in pen(TI)\}$.

Example 1 Let TI be locally consistent topology information such that

$$TI = \{(1, \{2, 4, 8, 9\}), (4, \{1, 3, 5\}), (5, \{4, 8, 9\}), (8, \{1, 5, 6, 9\}), (9, \{1, 5, 6, 8\})\}.$$

Then, $pen(TI) = \{2, 3, 6\}$, and the extended network of TI is shown in Fig. 1. In the figure, white circles denote the pending IDs of TI. ■

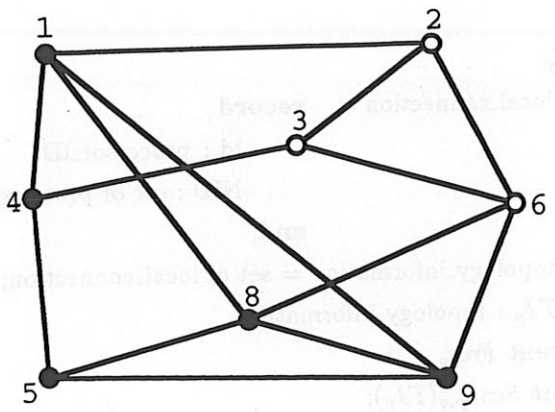

Figure 1: An extended network

Concerning the extended network, the following lemma can be proved.

Lemma 1 Let k be any non-negative integer, and $N = (P, L)$ be any $(k+1)$-connected network. For any nonempty $P'(\subseteq P)$, the extended network $exN(TI^{P'})$ is $(k+1)$-connected, where $TI^{P'} = \{(id_p, NID_p) \,|\, p \in P'\}$. ■

From the lemma, we can think of the following inconsistency condition on TI_p.

(I4) The extended network $exN(TI_p)$ is not $(k+1)$-connected.

Figure 2 shows the fault-tolerant topology computation process TP_p with the inconsistency detector. The function *inconsistent*(TI_p) becomes *true*, if and only if at least one of the conditions of (I1), (I2), (I3), and (I4) holds.

Lemma 2 The topology computation protocol TP has the following properties, when it is executed on any $(k+1)$-connected network with at most k faulty processors.

(T1) In any execution starting from the designated initial configuration, no processor p executes $Request_p$ (i.e., at any configuration reachable from the designated initial configuration, function *inconsistent*(TI_p) never becomes *true* at any processor p).

(T2) When started from any configuration,

(a) a correct processor p executes $Request_p$ in $O(n)$ rounds (n denotes the number of processors in the network),

(b) at a correct processor p, $Signal_p$ occurs in $O(n)$ rounds, or

(c) it reaches a final legal configuration in $O(n)$ rounds.

■

[3] Since it is possible that NID^i is different from the ID set of the *real* neighbors of id_i, we use notation NID^i to distinguish it from NID_i.

(a) Variable TI_p

```
type local_connection =  record
                           id : processor_ID;
                           NID : set of processor_ID;
                         end;
    topology_information = set of local_connection;
var  TI_p : topology_information;
```

(b) On input event $Free_{p,q}$

output $Send_{p,q}(TI_p)$;

(c) On input event $Receive_{q,p}(m)$

$TI_p := TI_p \cup m$;

if $inconsistent(TI_p)$ **then output** $Request_p$;

(d) On input event $Signal_p$

$TI_p := \{(id_p, NID_p)\}$;

Figure 2: Topology computation protocol TP: processor p

3.3 *k-ftss* reset protocol

Several papers [5, 6, 9, 18] present *ss* reset protocols that resume execution of an application protocol AP from a configuration that is reachable from the designated initial configuration. A *k-ftss* reset protocol RP satisfies the following requirements.

(R1) When started from any configuration, a *k-ftss* reset protocol RP eventually resets an application protocol AP, in spite of at most k faulty processors, to a configuration that is reachable from its designated initial configuration, if sufficiently many *Requests* occur.

Awerbuch et al. [9] give a formal specification of an *ss* reset protocol. It specifies the requirements of a reset protocol using properties called *timeliness*, *causality*, and *consistency*. We can naturally extend the specification to that for a *k-ftss* reset protocol. However, we omit the extended specification in this paper, since the extension is straightforward.

While several *ss* reset protocols are proposed, there exists no fault-tolerant and self-stabilizing reset protocol. Now, we show the idea of the *k-ftss* reset protocol proposed in this paper.

We realize the *k-ftss* reset protocol using *ss link protocols* and *version numbers*, which are common technique. The details of the *k-ftss* reset protocol is shown in Fig. 3.

The link protocol is a protocol to provide reliable communication between neighbors. A processor p sends information to a neighbor q by writing some message into register $R_{p,q}$. Because of asynchrony, it is possible that q reads $R_{p,q}$ many times before p writes some new message (i.e., message duplicate), or that p writes some new message into $R_{p,q}$ before q reads the previous message (i.e., message loss). To avoid such message duplicate and loss, many *ss* protocols use *ss* link protocols as communication primitives [2, 3, 4, 17]. The link protocol has the following property:

(a) On input event $Send_{p,q}(m)$

$s\_msg_p[q] := m$; $s\_vn_p[q] := vn_p$;
repeat
  with $R_{p,q}$ **do** $msg := s\_mes_p[q]; vn := s\_vn_p[q]; sf :=$"S";
until $R_{q,p}.rf =$"R";
repeat
  $R_{p,q}.sf :=$"I";
until $R_{q,p}.rf =$"W";
output $Free_{p,q}$;

(b) On input event $Request_p$

$vn_p := vn_p + 1$;
output $Signal_p$;

(c) To receive messages from q

repeat forever
  repeat
    $R_{p,q}.rf :=$"W";
    with $R_{q,p}$ **do** $r\_msg_p[q] := msg; r\_vn_p[q] := vn; Z := sf$;
  until $Z =$"S";
  if $r\_vn_p[q] = vn_p$ **then**
    output $Receive_{q,p}(r\_msg_p[q])$
  else if $r\_vn_p[q] > vn_p$ **then**
    $vn_p := r\_vn_p[q]$;
    output $Signal_p$;
    output $Receive_{q,p}(r\_msg_p[q])$;
  repeat
    $R_{p,q}.rf :=$"R";
  until $R_{q,p}.sf =$"I";

Figure 3: *k-ftss* reset protocol RP: processor p

- if both p and q are correct, then the sequence of messages received by $Receive_{p,q}$ is equal to the sequence of messages sent by $Send_{p,q}$, and
- otherwise, the sequence of messages received by $Receive_{p,q}$ is equal to a prefix of the sequence of messages sent by $Send_{p,q}$.

In the *k-ftss* reset protocol, we use the *ss* link protocol presented in [2]. For every pair $(p,q) \in L$, we use the link protocols $LP(p,q)$ and $LP(q,p)$. The link protocol $LP(p,q)$ (resp. $LP(q,p)$) is used to transfer messages from processor p (resp. q) to its neighbor q (resp. p).

The protocol $LP(p,q)$ is a simple 4-state protocol. In Fig. 3, the link register $R_{p,q}$ (resp. $R_{q,p}$) contains a flag sf for the sender p (resp. a flag rf for the receiver q). The flag sf is either **I** ("Idle") or **S** ("Sending"), and the flag rf is either **W** ("Waiting") or **R** ("Receiving").

Whenever an application process AP_p detects inconsistency on the information it has, it executes $Request_p$, and initializes its state on receipt of $Signal_p$[4]. We can consider that AP_p initiates a new version of the application process. To distinguish messages between old and new versions, a *version number* is used in RP. Each reset process RP_p has a local counter vn_p to store the current version number, and attaches the version number to messages it sends. The similar idea is used in many protocols (e.g., [5, 14, 20]). When RP_p receives a message $\langle m, vn' \rangle$ from a neighboring reset process RP_q, it compares the attached version number vn' with its current version number vn_p. If $vn' = vn_p$, then RP_p executes $Receive_{q,p}(m)$ to deliver the message m to AP_p. If $vn' > vn_p$, then RP_p sets vn_p to vn', executes $Signal_p$ to initialize AP_p, and executes $Receive_{q,p}(m)$. If $vn' < vn_p$, RP_p simply ignores the message m.

When AP_p detects inconsistency on its information, it executes $Request_p$ in order to reset the system using the reset protocol. Then, RP_p increments vn_p by one and executes $Signal_p$ in order to initialize AP_p. In the protocol, no special message is used to broadcast the reset request: since the application process repeatedly sends messages to its neighbors (assumption (A2)), the reset request is broadcast over the whole network by messages with the new version number.

In Fig. 3, each register $R_{p,q}$ has a field vn to store a version number. Each processor p has variables $s\_msg_p[q]$ and $s\_vn_p[q]$ for each neighbor q to store the message and the version number that p is sending. The variables $r\_msg_p[q]$ and $r\_vn_p[q]$ is similarly used to store the received message and version number.

3.4 *k-ftss* topology protocol

Theorem 1 Let TP be the topology computation protocol shown in Fig. 2, and RP be the reset protocol shown in Fig. 3. Consider the protocol TP^* such that TP and RP run concurrently in TP^*. The protocol TP^* is a *k-ftss* protocol for the topology problem in $\mathcal{N}^{k+1}$, and it reaches a final legal configuration in $O(n)$ rounds where n is the number of processors in the network.

(Outline of proof) Consider an arbitrary execution of TP^*. Let VN be a set of version numbers that are stored (at processors or link registers) at the initial configuration and are received by some correct processors in the execution, and let vn^* be the maximum value of VN. (Notice that there may exist some version number at the initial configuration that is not received by any correct processor because of crash faults.)

When a processor p increments its version number, it sets correctly its local connections to TI_p (i.e., it executes $TI_p := \{(id_p, NID_p)\}$). Thus, the topology information with version number $vn^* + 1$ contains no false connections

[4] For simplicity of the protocol description, even in the case that AP_p executes $Request_p$, AP_p initializes its state on receipt of $Signal_p$ from reset process RP_p.

at any configuration. Therefore, once a correct processor increments its version number to $vn^* + 1$, it is clear from Lemma 2 that TP^* reaches a final legal configuration in $O(n)$ rounds.

On the other hand, from the property of the reset protocol RP, TP^* reaches a configuration, say c^*, in $O(n)$ rounds where every correct processor has a version number vn^* or larger.

Consider the execution after the configuration c^*. If some correct processor p executes $Request_p$ or $Signal_p$ in $O(n)$ rounds, then its version number becomes $vn^* + 1$ and TP^* reaches a final legal configuration in $O(n)$ rounds. If there is no correct processor that executes $Request_p$ or $Signal_p$ in $O(n)$ rounds, it follows from Lemma 2 that TP^* reaches a final legal configuration in $O(n)$ rounds. ■

4 Impossibility result

The previous section shows a *k-ftss* protocol for the topology problem in $\mathcal{N}^{k+1}$. In the protocol, every processor uses additional information as to the neighbors' IDs and the connectivity. In this section, we consider the following question: Is it possible to design a *k-ftss* topology protocol using *only* the neighbors' IDs or using *only* the connectivity information? In what follows, we deny the possibility even for the counting problem. Notice that the topology problem is at least as hard as the counting problem.

We consider the case that *only* connectivity information is available at each processor, first. Anagnostou and Hadzilacos [4] show that no 1-*ftss* protocol exists for the counting problem even in ring networks. Since a ring network is 2-connected, their result implies that no 1-*ftss* protocol exists for the counting problem, even if each processor knows that the network is 2-connected (and each processor does not know its neighbors' IDs). Thus, it is impossible to design a *k-ftss* counting protocol using *only* the connectivity information.

Second, we consider the case that *only* the neighbors' IDs are available at each processor. If network partition occurs because of processor faults, then the counting problem clearly becomes unsolvable. Thus, there exists no *k-ftss* counting protocol in a k-connected network, even if the neighbors' IDs are available at every processor. However, it may be possible to design an *ss* counting protocol that tolerates up to k faulty processors if these k faults does not cause network partition (i.e. if the surviving network is connected). The following theorem denies the possibility.

Theorem 2 Let CP be any *k-ftss* protocol that solves the counting problem in $\mathcal{N}^{k+1}$. The protocol CP is not an *ss* protocol in $\mathcal{N}^k$: there is some k-connected network such that CP cannot solve the problem in the network even if no faulty processor exists.

(Outline of proof) To prove the theorem by contradiction, we assume that CP can solve the counting problem in any k-connected network with no faulty processor (when started from any configuration).

We show the contradiction using the fact that a faulty processor cannot be distinguished from a "very slow" processor. Thus, if some "very slow" processors execute no steps during a sufficiently long period, the protocol runs as if the processors were faulty.

In what follows, we show the scenario: if k processors are slowed down for a sufficiently long period in some k-connected network N_2, the protocol CP runs as if the k processors were faulty in some other $(k+1)$-connected network N_1. We construct N_1 and N_2 so that they are different in size.

Let $N_1 = (P_1, L_1)$ be a $(k+1)$-connected network and let $n = |P_1|$. Letting p be a processor such that $p \notin P_1$, we define a network $N_2 = (P_2, L_2)$ as follows:

- $P_2 = P_1 \cup \{p\}$.

- $L_2 = L_1 \cup \{(p, p') \mid p' \in P'\}$, where P' is a subset of P_1 such that $|P'| = k$.

Notice that $|P_2| = n + 1$ holds. It can be also proved that the network N_2 is k-connected.

Since CP is a *k-ftss* counting protocol in $\mathcal{N}^{k+1}$, CP solves the problem in N_1 even if all processors in P' are faulty, that is, it eventually reaches a configuration where every correct processor in $P_1 - P'$ considers n as the solution of the counting problem in N_1.

Now consider the execution of CP on N_2. Let C_a be a set of configurations of N_2 where every processor in P_2 considers $n + 1$ as the solution of the counting problem, and let C_b be a set of configurations of N_2 where every processor in $P_1 - P'(= P_2 - (P' \cup \{p\}))$ considers n as the solution of the counting problem. We call a configuration in C_a (resp. in C_b) a legal configuration (resp. an illegal configuration).

We consider the following strategy followed by the adversary for scheduling steps of processors in N_2 with no faulty processors. The adversary runs CP until it reaches a legal configuration. Since CP is an *ss* protocol in N_2 with no faulty processors, it eventually reaches a legal configuration. Once such a configuration is reached, the adversary slows down the processors in P'. Let E be execution of CP during the processors in P' make no step. Let E' be a projection of E to P_1, that is, E' is obtained from E by removing the states and atomic steps of p and the registers incident to p. We can prove that E' is also an execution of CP in N_1 where all of the processors in P' are faulty. Thus, if E' is sufficiently long, it contains a configuration where every processor in $P_1 - P'$ considers n as the solution. From this, if the adversary slows down the processors in P' for sufficiently long in N_2, CP reaches an illegal configuration. After CP reaches the illegal configuration, the adversary activates the processors in P'. Then, CP eventually reaches a legal configuration, since it is an *ss* counting protocol in N_2 with no fault.

By following this strategy repeatedly, the adversary can schedule processor steps so that illegal configurations appear infinitely often. This contradicts the assumption that CP is an *ss* counting protocol in N_2 with no faulty processor. ■

5 Conclusions

We proposed a k-fault-tolerant and self-stabilizing protocol for the topology problem in asynchronous model of distributed systems. In the protocol, every processor uses accurate information about the neighbors' IDs and the network connectivity. We also showed that the information is minimal information to solve the problem, that is, the problem is unsolvable if processors cannot use both the neighbors' IDs and the connectivity information.

We designed the protocol in a layered manner, and proposed a k-fault-tolerant and self-stabilizing reset protocol as an underlying protocol. In the reset protocol, we use a version number. During execution of the reset protocol, the version number becomes at most one greater than the maximum value stored (at processors or link registers) in its initial configuration. Since we can make no assumption on the initial value, however, the version number is unbounded. It is one of our future works to investigate the possibility of a k-fault-tolerant and self-stabilizing reset protocol with bounded space.

Acknowledgments: The main part of this work was done during my stay at Computer Science Department, Cornell University as a visiting researcher. I am most grateful to Prof. Sam Toueg for his helpful discussions and many important comments that improved this work very much. I also thank Prof. Nobuki Tokura (Osaka University) and Prof. Hideo Fujiwara (NAIST) for their suggestions to my research, and thank anonymous reviewers for their helpful comments.

References

[1] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self stabilizing protocols for general netoworks. In *Proc. 4th WDAG (LNCS 486)*, pages 15–28, 1990.

[2] E. Anagnostou and R. El-Yaniv. More on the power of random walk: uniform, bounded self-stabilizing protocols. In *Proc. 5th WDAG (LNCS 579)*, pages 31–51, 1991.

[3] E. Anagnostou, R. El-Yaniv, and V. Hadzilacos. Memory adaptive self-stabilizing protocols. In *Proc. 6th WDAG (LNCS 647)*, pages 203–220, 1992.

[4] E. Anagnostou and V. Hadzilacos. Tolerating transient and permanent failures. In *Proc. 7th WDAG (LNCS 725)*, pages 175–188, 1993.

[5] A. Arora and M. Gouda. Distributed reset. In *Proc. 10th Conference on Foundations of Software Technology and Theoretical Computer Science (LNCS 472)*, pages 316–331, 1990.

[6] B. Awerbuch and R. Ostrovsky. Memory-efficient self-stabilizing network RESET. In *Proc. 13th ACM PODC*, pages 254–263, 1994.

[7] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proc. IEEE 32nd FOCS*, pages 268–277, 1991.

[8] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Bounding the unbounded. In *Proc. 13th IEEE INFOCOM*, pages 776–783, 1994.

[9] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev. Self-stabilization by local checking and global reset. In *Proc. 8th WDAG (LNCS 852)*, pages 326–339, 1994.

[10] T. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proc. 10th ACM PODC*, pages 325–340, 1990.

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

[12] S. Dolev. Optimal time self stabilization in dynamic systems. In *Proc. 7th WDAG (LNCS 725)*, pages 160–173, 1993.

[13] S. Dolev, A. Israeli, and S. Moran. Self stabilization of dynamic systems assuming only read/write atomicity. In *Proc. 9th ACM PODC*, pages 103–117, 1990.

[14] S. G. Finn. Resynch procedures and a fail-safe network protocol. *IEEE Trans.Comm.*, COM-27(6):840–845, June 1979.

[15] A. Gopal and K. J. Perry. Unifying self-stabilization and fault-tolerance. In *Proc. 12th ACM PODC*, pages 195–206, 1993.

[16] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems (2nd edition)*, chapter 5, pages 97–145. Addison-Wesley, 1993.

[17] A. Israeli and M. Jalfon. Token management schemes and random walks yields self stabilizing mutual exclusion. In *Proc. 9th ACM PODC*, pages 119–131, 1990.

[18] S. Katz and K. J. Perry. Self-stabilizing extensions for message-passing systems. In *Proc. 9th ACM PODC*, pages 91–101, 1990.

[19] J. M. Spinelli and R. G. Gallager. Event driven topology broadcast without sequence numbers. *IEEE Trans. Comm.*, 37(5):468–474, May 1989.

[20] G. Varghese. Self-stabilization by counter flushing. In *Proc. 13th ACM PODC*, pages 244–253, 1994.

Paper Number 2

Maximum Flow Routing

Mohamed G. Gouda and Marco Schneider

MAXIMUM FLOW ROUTING

Mohamed G. Gouda and Marco Schneider

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

Abstract

We describe a protocol for routing and allocating virtual circuits from any vertex to a designated destination in a computer network. In this algorithm, the network vertices maintain a maximum flow spanning tree whose root is the designated destination. Every requested virtual circuit is allocated along the maintained tree. However, as virtual circuits are allocated along the tree, the tree may lose its property of being a maximum flow tree, and so need to be updated. We describe a novel protocol for the network vertices to periodically update the maintained tree to keep it a maximum flow tree. This protocol is stabilizing, and has the nice property that while the tree is being updated, it always remains a tree whose root is the designated destination.

1. Introduction

Computer networks can be represented as connected, undirected graphs, where vertices represent computers and edges with positive capacities represent channels between computers. The flow of a path in such a network is the minimum capacity for an edge in that path.

Identifying maximum flow paths in networks is useful for establishing and maintaining virtual circuits in computer networks. To establish a virtual circuit with some required capacity between two vertices in a network, a maximum flow path between the two vertices is first identified in the network. If the flow of the identified path is greater than or equal the required capacity of the circuit, then the circuit is established along the identified path. Otherwise, the circuit is rejected.

Also, if an edge in an established circuit happens to fail, a maximum flow path between the two ends of the failed edge is identified. Then the two disjointed parts of the circuit are re-connected by a connection along the identified path [4].

In this paper, we describe a protocol for allocating virtual circuits from any vertex to a designated destination in a computer network. In our protocol, the network vertices maintain a maximum flow spanning tree whose root is the designated destination. Every requested virtual circuit is allocated

along the maintained tree. However, as virtual circuits are allocated along the tree, the tree may lose its property of being a maximum flow tree, and need to be updated. We describe a protocol for periodically updating the maintained tree to keep it a maximum flow tree. This protocol is stabilizing [8], and has the nice property that while the tree is being updated, it always remains a tree.

2. Maximum Flow Trees

A network N is a connected and undirected graph whose set of vertices is V and whose set of edges is E. One vertex r in V is called the root of N. Associated with each edge {v, w} in E is a non-negative integer c.{v, w}, called the capacity of edge {v, w}.

A path in N is a non-empty sequence of distinct vertices in V such that each pair of consecutive vertices in the sequence is an edge in E. A path is called rooted iff its last vertex is the root r of N. For example, the path that consists of the single vertex r is rooted.

The flow of a rooted path in N is the minimum capacity of an edge in the path. Thus, if a rooted path p in N is $\langle v_0 ; \ldots ; v_k \rangle$, then

the flow of p =

$$\min \{ c.\{v_i, v_{i+1}\} \mid 0 \leq i < k \}$$

A rooted path p in N is called a maximum flow path iff for every rooted path q that has the same first vertex as p in N,

the flow of p $\geq$ the flow of q

The flow of a vertex v, other than the root r, in N is the flow of a maximum flow path, whose initial vertex is v, in N. The flow of r is the maximum edge capacity in N.

A spanning tree T of N is called a maximum flow tree iff every rooted path in T is a maximum flow path in N.

The following theorem, taken from [7], establishes the relation between the maximum weight spanning trees of a network [5] and the maximum flow trees of the same network.

Theorem 1: Each maximum weight spanning tree of a network N is a maximum flow tree of N. The converse is not necessarily true.

A corollary of this theorem is that each network has a maximum flow tree.

3. Maximum Flow Tree Protocol

In this section, we present a distributed protocol, taken from [7], for maintaining a maximum flow tree T in an arbitrary network N = (V, E). The protocol is distributed because it consists of several programs, one for each vertex in V, and the program of each vertex v has few constants that relate to vertex v only. In particular, the program of a vertex v has the following four constants:

i. D is an upper bound on the number of vertices in the longest path in N.

ii. F is the maximum edge capacity in network N.

iii. H is the set H of neighbors of vertex v in network N.

iv. c[w] is the capacity for each edge {v, w} incident at vertex v in N.

The program of each vertex v also has three variables: p.v, d.v, and f.v. When the protocol terminates, p.v is the parent of vertex v in the maintained maximum flow tree T, d.v is the number of edges on the path from vertex v to the root r in tree T, and f.v is the flow of v.

The program of each vertex v has two actions as follows.

begin <action> [] <action> **end**

Each of the two actions is of the following form.

<guard.w> --> <statement.w>

where w is a parameter that denotes a neighbor of v, <guard.w> is a boolean expression over the variables of v and the variables of w, and <statement.w> is a sequence of assignment statements that assign values to variables of v based on the current values of the variables of v and the variables w.

An action "<guard.w> --> <statement.w>" is enabled (for execution) when there is a neighbor w of v that makes the boolean expression <guard.w> true. An enabled action is executed by executing the sequence of assignment statements in its <statement.w>.

The program of any vertex v, other than the root r, in network N is as follows.

constant

D : **integer**,
F : **integer**,
H : **set** {w|w is a neighbor of v in N },
c : **array** [H] **of** 0..F /* c[w] = capacity of {v, w} */

variable

```
p.v   :   H,          /* parent of v in T */
d.v   :   0..D,       /* distance of v from r in T*/
f.v   :   0..F        /* flow of v  in T*/
```

parameter

```
w     :   H           /* a neighbor of v*/
```

begin

```
p.v = w          ^
( d.v ≠ min {d.w + 1, D}  or  f.v ≠ min {f.w, c[w]} )
  -->
d.v := min {d.w + 1, D};
f.v  := min {f.w, c[w]}
```

```
[]    p.v ≠ w          ^
      d.w < D-1        ^ ( d.v = D  or  (d.w + 1 - D*min {f.w, c[w]}) < (d.v - D*f.v) )
        -->
      p.v := w;
      d.v := min {d.w + 1, D};
      f.v  := min {f.w, c[w]}
```

end

This program has two actions. In the first action, vertex v detects that its distance d.v from the root or its flow f.v is not consistent with that of its parent. In this case, vertex v updates both d.v and f.v to be consistent, respectively, with d.w and f.w of its parent w.

In the second action, vertex v detects that it has a neighbor w such that the following three conditions hold. First, w is not the current parent of v. Second, the distance from the root to w is less than D-1. Third, if w is to become the parent of v, then v will either have more flow or will have the same flow and a smaller distance from the root. In this case, vertex v makes w its parent and updates its variables d.v and f.v accordingly.

The program for the root vertex r has only two constants, D and F, and two variables, d.r and f.r. This program is as follows.

```
constant
    D    :    integer,
    F    :    integer

variable
    d.r  :    0..D,                    /* distance of r from r */
    f.r  :    0..F                     /* flow of r in  T      */

begin
    d.r ≠ 0  or  f.r ≠ F  -->  d.r := 0 ;  f.r := F
end
```

The one action in this program ensures that variables d.r and f.r are assigned the two values 0 and F, respectively.

A proof of the correctness and stabilization of this protocol is presented in [7]. This proof is similar to those given in [1], [2], and [3].

4. Routing Using the Maximum Flow Tree Protocol

From the above protocol, each vertex v in network N knows its parent vertex p.v in the maintained maximum flow tree T. When a virtual circuit is to be established through a vertex v (to the root vertex r), vertex v routes the circuit through its parent vertex p.v in the maximum flow tree. In other words, the circuit is established over the edge {v, p.v}.

Each established virtual circuit is of some capacity. Thus, establishing a virtual circuit with a capacity cp decreases the remaining capacity in each edge along the circuit by the value cp. Similarly, when an established circuit with capacity cp is removed, the remaining capacity in each edge along the circuit is increased by the value cp.

Each vertex v keeps track of the remaining capacity cr[w] for each edge {v, w} incident at v in the network. Vertex v follows the next two rules to keep its remaining capacity array cr current.

i. When a virtual circuit with capacity cp is established over an edge {v, w}, then vertex v updates its cr array as follows.

    cr[w] := cr[w] - cp

    Also, vertex w updates its cr array as follows.

    cr[v] := cr[v] - cp

(This can happen only when both cr[w] in v and cr[v] in w are greater than cp.)

ii. When a virtual circuit with capacity cp that was established over an edge {v, w} is removed, then vertex v updates its cr array as follows.

cr[w] := cr[w] + cp

Also, vertex w updates its cr array as follows.

cr[v] := cr[v] + cp

As the remaining capacity cr arrays become very different from the capacity c arrays used in maintaining the maximum flow tree, the maintained tree is no longer a maximum flow tree. Therefore, the maintained tree needs to be updated periodically.

One protocol for updating the maintained maximum flow tree is as follows. Each node in the network periodically uses its cr array to update its c array by executing the statement c := cr. Unfortunately, whenever a node executes such a statement, the original maximum flow tree protocol may start executing and cause the maintained tree to lose its property of being a tree for some time, until the protocol finally converges to a new tree.

In this paper, we are interested in protocols for updating the maintained maximum flow tree such that the maintained tree is always a tree, even when it is being updated. One such protocol is discussed in the next section.

5. Adaptive Maximum Flow Tree Protocol

In this section, we present a protocol for adapting the maximum flow tree to changes in the remaining channel capacities. This protocol proceeds in successive rounds: round.0, round.1, round.2, ...

In each odd round (i. e. round.1, round.3, ...), every vertex uses its cr array to update its c array, and enables its first action in the original maximum flow tree protocol. Note that these first actions, when executed, do not change the maintained tree. They merely compute the correct flow along every path in the maintained tree. The round terminates when the values of variables d.r and f.r are 0 and F, respectively, and the values of variables d.v and f.v of each vertex v, other than the root r, are consistent with those of its parent w in the maintained tree as follows.

$$d.v = \min\{d.w + 1, D\}$$

$$f.v = \min\{f.v, c[w]\}$$

Recall that at the beginning of an odd round, each c array is assigned the value of the corresponding cr array. Thus at the end of the round, the maintained tree may not be a maximum flow tree with respect to the new c arrays.

In each even round (i. e. round.0, round.2, ...), every vertex enables its two actions in the original maximum flow tree protocol. Although the maintained tree may be updated in this round, it can be shown that each action execution in this round keeps the maintained structure a tree. The round terminates when the values of variables p.v, d.v, and f.v of every vertex v define a maximum flow tree.

To keep track of the current round, each vertex v has a variable s.v whose value is the number of rounds executed so far. To determine whether the current round is odd or even, vertex v needs only to check whether or not s.v **mod** 2 equals 0.

Before initiating the next round, the root vertex r waits long enough until all activities in the current round have ceased. Then the root r increments its variable s.r to start the next round. Each other vertex v starts to participate in the next round, when it observes that the value of s.w of a neighbor w is larger than the value of its own s.v.

The program of any vertex v, other than the root r, in network N is as follows.

constant

D	:	**integer**,
F	:	**integer**,
H	:	**set** {w|w is a neighbor of v in N }

input

cr	:	**array** [H] **of** 0..F

variable

c	:	**array** [H] **of** 0..F,	
p.v	:	H,	/* parent of v in T */
d.v	:	0..D,	/* distance of v from r in T*/
f.v	:	0..F,	/* flow of v in T*/
s.v	:	**integer**	/*seq. number of round*/

parameter

w : H /*a neighbor of v*/

begin

/*start an odd round*/

s.v < s.w ^ s.w **mod** 2 ≠ 0

-->

s.v := s.w;

c := cr

[] /*execute in each round*/

s.v = s.w ^

p.v = w ^

(d.v ≠ min {d.w + 1, D} or f.v ≠ min {f.w, c[w]})

-->

d.v := min {d.w + 1, D};

f.v := min {f.w, c[w]}

[] /*start an even round*/

s.v < s.w ^ s.w **mod** 2 = 0

-->

s.v := s.w

[] /*execute in each even round*/

s.v = s.w ^ s.v **mod** 2 = 0 ^

p.v ≠ w ^

d.w < D-1 ^ (d.v = D or (d.w + 1 - D*min {f.w, c[w]}) < (d.v - D*f.v))

-->

p.v := w;

d.v := min {d.w + 1, D};

f.v := min {f.w, c[w]}

end

The program of vertex v has four actions. In the first and third actions, vertex v detects that the value of variable s.w of a neighbor w is larger than its own s.v, indicating that the next round has started. In this case, v assigns its s.v the value of s.w, and if s.w **mod** 2 ≠ 0 indicating that the new round is odd, v uses its cr array to update its c array.

The second action in this program is the same as the first action in the original maximum flow tree protocol except for adding the conjunct (s.v = s.w) to the guard. The fourth action in this program is the same as the second action in the original maximum flow tree protocol except for adding the conjunct (s.v = s.w ^ s.v **mod** 2 = 0) to the guard.

The program of the root vertex r is as follows.

constant

D : **integer**,
F : **integer**,
H : **set** {w|w is a neighbor of r in N }

variable

d.r : 0..D, /* distance of r from r */
f.r : 0..F /* flow of r in T*/
s.r : **integer**

parameter

w : H /*a neighbor of r*/

begin

timeout no other action in the network is enabled --> s.r := s.r + 1

[] s.r < s.w --> s.r := s.w

[] d.r ≠ 0 or f.r ≠ F --> d.r := 0 ; f.r := F

end

The program of the root r has three actions. In the first action, when r is certain that the activities of the current round have terminated, it starts a new round by incrementing the value of s.r by one. This action is called a timeout action because it can be implemented using time-outs as discussed in Section 7. In the second action, r observes that its s.r has a smaller value than s.w of some neighbor w. This observation indicates that an error has occurred, as the value of s.r should always be equal or larger than the value of s.v for any vertex v in the network. Thus, r corrects the observed error by assigning s.v the value of s.w. The third action of r is identical to the third action of r in the original maximum flow tree protocol.

6. Correctness of the Adaptive Tree Protocol

In this section, we discuss some interesting properties of the adaptive maximum flow tree protocol. Our presentation starts with some definitions concerning the states, transitions, and computations of this protocol.

A state of the adaptive maximum flow tree protocol is defined by a value for each variable in each vertex in the protocol. The value of a variable is from the domain of values for that variable.

A state of the protocol is called odd, or even, iff in that state, variable s.r in the root r has an odd value, or even value, respectively.

A state of the protocol is called balanced iff in that state, the following two conditions are satisfied.

i. For every vertex v, s.v = s.r, where r is the root.

ii. Each action of each vertex, other than the first action of the root, is disabled (and so cannot be executed).

It is straightforward to show that the following two conditions are satisfied in any balanced state.

i. For the root r,

$$d.r = 0$$
$$f.r = F$$

ii. For every vertex v, other than the root r, there is a neighbor w such that

$$s.v = s.r$$
$$p.v = w$$
$$d.v = \min \{d.w + 1, D\}$$
$$f.v = \min \{f.w, c[w]\}$$

A balanced state is called a milestone iff in that state, the values of the p.v variables, where v is any vertex other than the root r, define a tree whose root is r.

A milestone state is called a target iff in that state, the values of the p.v variables, where v is any vertex other than the root r, define a maximum flow tree whose root is r.

A triple (s, c, s') is called a transition of the protocol iff the following three conditions are satisfied.

i. s and s' are two states of the protocol.

ii. c is an action of some vertex in the protocol.

iii. Executing action c when the protocol is in state s yields the protocol in state s'.

States s and s' are called the pre-state and post-state of the transition, respectively.

An infinite sequence (s.1, c.1, s.2, c.2, ...) is called a computation of the protocol iff the following three conditions are satisfied.

i. s.1, s.2, ... are states of the protocol.

ii. c.1, c.2, ... are actions in some processes in the protocol.

iii. Every triple (s.i, c.i, s.(i+1)) in the sequence is a transition of the protocol.

Next, we present five interesting properties of an arbitrary computation of the protocol. Consider an arbitrary computation C of the protocol, and let for i = 0, 1, ... , T.i denote the i-th occurrence of a transition, if any, whose action is the timeout action of the root r, in computation C. Also, let S.i be the pre-state of transition T.i, for i = 0, 1, ...

Lemma 1: For every k, k = 0, 1, ..., computation C has a transition T.k.

Lemma 2: S.0 is a balanced state.

Lemma 3: If S.0 is an odd balanced state, then for every k, k = 0, 1, ... ,

S.(2*k + 1) is an even target state, and

S.(2*k + 2) is an odd milestone state.

Lemma 4: If S.0 is an even balanced state, then S.1 is an odd balanced state, and for every k, k = 0, 1, ... ,

S.(2*k + 2) is an even target state, and

S.(2*k + 3) is an odd milestone state.

Lemma 5: In each state, that occurs after the first target state in computation C, the values of the p.v variables define a tree whose root is r.

7. Refinements of the Adaptive Tree Protocol

The adaptive maximum flow tree protocol in Section 5 admits a number of refinements that can make it attractive in some of applications. In this section, we briefly discuss three of these refinements.

First, this protocol can be extended to allow a network to have a multiple maximum flow trees such that each vertex in the network is the root of some tree. With this extension, virtual circuits can be

established from any vertex to any other vertex in the network. This extension can be achieved by making variables p.v, d.v, f.v, and s.v for every vertex v arrays, rather than single variables, as follows.

variables

p.v : **array** [V] **of** H,
d.v : **array** [V] **of** 0..D,
f.v : **array** [V] **of** 0..F,
s.v : **array** [V] **of integer**

In these declarations, V is the set of all vertices in the network. Thus, p.v[w] is the parent of vertex v in the tree whose root is w, d.v[w] is the distance from vertex v to vertex w in the tree whose root is w, and so on.

Second, there are two methods to implement the timeout action of the root r in the adaptive maximum flow tree protocol. In one method, time-outs are used as follows. After the root r increments its variable s.r, it waits long enough until it is certain that all activities in the new round have terminated, before it times out and increment s.r one more time to initiate the next round. In the other method, the protocol is augmented with an algorithm for detecting that all actions, other than the timeout action of the root, have terminated. Several stabilizing termination detection algorithms have been proposed earlier, for example [6] and [10], and any of them can be augmented with our protocol to implement the timeout action.

Third, instead of using an integer sequence number s.v for every vertex v, binary sequence numbers can be used in the adaptive maximum flow tree protocol. However, to keep the protocol stabilizing, the protocol needs to be augmented with another protocol for maintaining a shortest path tree whose root is r. (One such protocol is discussed in [2].) Let q.v be the parent of any vertex v, other than the root r, in the shortest path tree. Then, the conjunct ($s.v < s.w$) in the guards of the first and third actions of vertex v is replaced by the guard ($s.v \neq s.w \wedge q.v = w$). It is straightforward to show that the resulting protocol is stabilizing.

References

[1] A. Arora and M. G. Gouda, "Distributed Reset", IEEE Transactions on Computers, Vol. 43, No. 9, September 1994, pp 1026-1039.

[2] A. Arora, M. G. Gouda, and T. Herman, "Composite Routing Protocols", Proc. of the Second IEEE Symposium on Parallel and Distributed Processing, 1990.

[3] N. S. Chen, F. P. Yu, and S.T. Huang, "A Self-Stabilizing Algorithm for Constructing Spanning Trees", Information Processing Letters, Vol. 39, pp. 147 - 151, 1991.

[4] C. E. Chow, J. D. Bickell, and S. Syed, "Performance Analysis of Fast Distributed Link Restoration Algorithms", Accepted in the International Journal of Digital and Analog Communications Systems, 1994.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, MIT Press and McGraw-Hill, 1990.

[6] M. G. Gouda and M. Evangelist, "Convergence/Response Tradeoffs in Concurrent Systems", Proc. of the Second IEEE Symposium on Parallel and Distributed Processing, 1990.

[7] M. G. Gouda and M. Schneider, "Stabilization of Maximum Flow Trees", Invited Talk, Proceedings of the third Annual Joint Conference on Information Sciences, 1994, pp. 178-181. A full version was submitted to the journal of Information Sciences.

[8] M. Schneider, "Self-Stabilization", ACM Computing Surveys, Vol. 25, No. 1, March 1993.

[9] M. Schneider, Ph. D. Dissertation, The University of Texas at Austin, in preparation, 1995.

[10] G. Varghese, "Self-Stabilization by Counter Flushing", Proceedings of the 1994 ACM Symposium on Principles of Distributed Computing.

Paper Number 3

SuperStabilizing Protocols for Dynamic Distributed Systems

Shlomi Dolev and Ted Herman

SuperStabilizing Protocols for Dynamic Distributed Systems

(Extended Abstract)

Shlomi Dolev*
Ben-Gurion University
dolev@cs.bgu.ac.il

Ted Herman†
University of Iowa
herman@cs.uiowa.edu

Abstract

Two aspects of reliability of distributed protocols are a protocol's ability to recover from transient faults and a protocol's ability to function in a dynamic environment. Approaches for both of these aspects have been separately developed, but have drawbacks when applied to an environment that has both transient faults and dynamic changes. This paper introduces definitions and methods for addressing both concerns in the design of systems.

A protocol is *superstabilizing* if it is (i) self-stabilizing, meaning that it is guaranteed to respond to an arbitrary transient fault by eventually satisfying and maintaining a *legitimacy* predicate, and (ii) it is guaranteed to satisfy a *passage* predicate at all times when the system undergoes topology changes starting from a legitimate state. The passage predicate is typically a safety property that should hold while the protocol makes progress towards re-establishing legitimacy following a topology change.

Specific contributions of the paper include: the definition of the class of superstabilizing protocols; metrics for evaluating superstabilizing protocols; superstabilizing protocols for colouring and spanning tree construction; a general method for converting self-stabilizing protocols into superstabilizing ones; and a generalized form of a self-stabilizing topology update protocol which may have useful applications for other research.

1 Introduction

The most general technique enabling a system to tolerate arbitrary transient faults is *self-stabilization*: a protocol is self-stabilizing if, in response to any transient fault, it converges to a legitimate state in finite time. The characterization of legitimate states, given by a legitimacy predicate, specifies the protocol's function. Such protocols are generally evaluated by studying the efficiency of convergence, which entails bounding the time of convergence to a legitimate state following a transient fault. Other aspects of convergence, for instance safety properties, are of less interest since arbitrary transient faults can falsify any non-trivial safety property.

The model of a *dynamic* system is one where communication links and processors may fail and recover during normal operation. Protocols for dynamic systems are designed to cope with such failures and recovery without global reinitialization. These protocols consider only global states that are reachable from a predefined initial state under a *constrained sequence of failures*; under such an assumption, the protocols attempt to cope with failures with as few adjustments as possible. Thus, whereas self-stabilization research largely ignores the behaviour of protocols between the time of a transient fault and restoration to a legitimate state, dynamic protocols make guarantees about behaviour at all times (during the period between a failure event and the completion of necessary adjustments).

Superstabilization

Superstabilizing protocols combine benefits of both self-stabilizing and dynamic protocols. We retain the idea of a legitimate state, but partitition illegitimate states into two classes, depending on whether or not they satisfy a *passage* predicate. Roughly speaking, a protocol is superstabilizing if it is (i) self-stabilizing, and (ii) when started

*Part of this research was supported by TAMU Engineering Excellence funds and by NSF Presidential Young Investigator Award CCR-9396098.
†This research was supported in part by the Netherlands Organization for Scientific Research (NWO) under contract NF 62-376 (NFI project ALADDIN: Algorithmic Aspects of Parallel and Distributed Systems).

in a legitimate state and a topology change occurs, the passage predicate holds and continues to hold until the protocol reaches a legitimate state.

The passage predicate is specified along with a class of topology changes. Since a legitimacy predicate is dependent on system topology, a topology change will typically falsify legitimacy. The passage predicate must therefore be weaker than legitimacy, but strong enough to be useful; ideally, the passage predicate should be the strongest predicate that holds when a legitimate state undergoes any of a class of topology change events. One example for a passage predicate is the existence of at most one token in a mutual exclusion task; whereas in a legitimate state exactly one token exists, a processor crash could eliminate the token yet not falsify the passage predicate. Similarly, for the leader election task, the passage predicate could specify that at most one leader exists.

Superstabilizing protocols are evaluated in several ways. Of interest are the worst-case convergence time, i.e., the time required to estabilish a legitimate state following either a transient fault or a topology change, and the scope of the convergence in terms of how much of the network's data must be changed as a result of convergence. We classify superstabilizing protocols by the following complexity measures: *stabilization time* is the maximum amount of time it takes for the protocol to reach a legitimate state; *superstabilization time* is the maximum amount of time it takes for a protocol starting from a legitimate state, followed by a single topology change, to reach a legitimate state; *adjustment measure* is the maximum number of processors that must change their local states, upon a topology change from a legitimate state, so that the protocol is in a legitimate state.

Background and Motivation

Many distributed protocols have been designed to cope with continuous dynamic changes [AAG87, AGH90, AM92, AGR92]. These protocols make certain assumptions about the behavior of processors and links during failure and recovery; for instance, most of them do not consider the possibility of processor crashes and they assume that every corrupted message is identified and discarded. If failures are frequent, these restrictive assumptions can be too optimistic.

A number of researchers [DIM93, KP93, APV91] suggest a self-stabilizing approach to deal with dynamic systems. In these approaches, a state following a topology change is seen as an inconsistent state from which the system will converge to a state consistent with the new topology. Although self-stabilization can deal with dynamic systems, the primary goal of self-stabilizing protocols is to recover from transient faults, and this view has influenced the design and analysis of self-stabilizing protocols. For instance, for a correct self-stabilizing protocol, there are no restrictions on the behavior of the system *during* the convergence period: the only guarantee is convergence to a legitimate state.

Self-stabilization's treatment of a dynamic system differs from that of the dynamic protocols cited above in the way that topology changes are modelled. The dynamic protocols assume that topology changes are *events* signaling changes on incident processors. Self-stabilizing protocols take a necessarily more conservative approach that is entirely state-based: a topology change results in a new state from which convergence to a legitimacy is guaranteed, with no dependence on a signal. Yet when the system is in a legitimate state and a fault happens to be a detected event, can the behavior during the convergence be constrained to satisfy some desired safety property? For instance, is it possible in these situations for the protocol to maintain a "nearly legitimate" state during convergence?

The issue can be motivated by considering the problem of maintaining a spanning tree in a network. Suppose the spanning tree is used for virtual circuits between processors in the network. When a tree link fails, the spanning tree becomes disconnected; yet virtual circuits entirely within a connected component can continue to operate. We would like to restore the system to have a spanning tree so that existing virtual circuits in the connected components remain operating; thus a least-impact legitimate state would be realized by simply choosing a link to connect the components.

One thesis of this paper is that self-stabilizing protocols can be designed with dynamic change in mind to improve response. Self-stabilizing protocols proposed for dynamic systems do not use the fact that processor can detect that it is recovering following a crash; consequently there is no possibility of executing an "initialization" procedure during this recovery.[1] A key observation for this paper is that a topology change is *usually* a detectable event; and in cases where a topology change is not detected, we use self-stabilization as a fall-back mechanism to deal with the change. The remainder of the paper illustrates aspects of superstabilization with selected protocols and a general method that converts self-stabilizing protocols into protocols that are superstabilizing.

2 Dynamic System

A system is represented by a graph where processors are nodes and links are (undirected) edges. An edge between two processors exists iff the two processors are *neighbours*; processors may only communicate if they are neighbours. Each processor has a unique identifier taken from a totally ordered domain. We use p, q, and r to denote processor identifiers. Processors communicate using registers, however application of the model to a message-passing system is intended; we outline an implementation of the register model in terms of message-based constructions in [DH95].

Associated with each processor p are code, local variables, a program counter, a shared register, and an input variable N_p, which is a list of processors q that are neighbours of p. Invariantly, neighbourhoods satisfy $p \notin N_p$ and $q \in N_p \Leftrightarrow p \in N_q$. A processor can write to its own shared register, but may only read shared registers belonging to neighbouring processors. The code of a processor is a sequential program; a program counter is associated with each processor. An *atomic step* of a processor (in the sequel referred to as steps) consists of the execution of one statement in a program. In one atomic step, a processor performs some internal computation and at most one register operation. A processor has two possible register operations, read and write.

Our model specifies that a step consists of a statement execution; we have in mind a conventional instruction-execution architecture, where each statement corresponds to some low-level code. However, to make presentation of protocols concise, we give descriptions at a higher level in terms of programs with assignment statements and control structures (**forall**, **do**, etc.); it should be understood that these descriptions can be resolved into lower-level programs where statements translate into atomic steps. In the protocol presentations, we also make the convention that advancing the program counter beyond the last statement of a program returns the program counter to the program's first statement; thus each program takes the form of an infinite loop.

Local variables of processors are of two types: variables used for computations and *field image* variables. The former are denoted using unsubscripted variable names such as x, y, and A. The field image variables refer to fields of registers; these variables are subscripted to refer to the register location, for instance e_p refers to a field of processor p's register and y_q refers to a field of processor q's register. Program statements that assign to field images or use field images in calculations are not register operations: the field image is essentially a cache of an actual register field. A processor p's read(q) operation, defined for $q \in N_p$, atomically reads the register of processor q and assigns all corresponding field images (e.g. e_q, y_q, etc.) at processor p. A write operation atomically sets all fields of p's register to current image values. For convenience, we also permit a local calculation to specify field image(s) with a write statement, for instance write($e_p := 1$) sets field image e_p and writes to p's register.

The state of a processor p fully describes the values of its local variables, program counter, shared register, and its neighbourhood N_p (although a processor cannot change its neighbourhood, it is convenient to include N_p in the state of p for subsequent definitions). In the sequel we occasionally refer to the state of a processor as a *local state.*

[1] Moreover, some systems do provide a "start" signal. Self-stabilizing protocols do not specify any special starting state or action upon receiving such a signal. Note that a computation invoked by a start signal resembles the case of dynamic system starting with all processors simultaneously recovering from a crash.

The state of the system is a vector of states of all processors; a system state is called a *global state.* For a global state σ and a processor q, let $\sigma[q]$ denote the local state of q in state σ. A *computation* is a sequence of global states $\Theta = (\theta_1, \theta_2, \cdots)$ such that for $i = 1, 2, \cdots$ the global state θ_{i+1} is reached from θ_i by a single step of some processor. A *fair* computation is a computation that is either finite or contains infinitely many steps of each (non-crashed) processor.

We write $\sigma \vdash \mathcal{P}$ to denote that global state σ satisfies predicate $\mathcal{P}$. A legitimacy predicate $\mathcal{L}$ is typically specified with respect to some property of interest encoded by $\mathcal{P}$; for instance, $\mathcal{P}$ can specify that exactly one processor has a token for mutual exclusion. The predicate $\mathcal{L}$ specifies permissible values of all register fields, program counters, and local variables so that $\mathcal{P}$ remains invariantly true in a computation.

A system *topology* is a specific system configuration of links and processors. Each processor can determine the current status of its neighbourhood from its local state (via N_p); thus the system topology can be extracted by a program from a global state of the system. Let $\mathcal{T}.\alpha$ denote the topology for a given global state α. Dynamic changes transform the system from one topology $\mathcal{T}.\alpha$ to another topology $\mathcal{T}.\beta$ by changing neighbourhoods and possibly removing or adding processors.

A topology change *event* is the removal or addition of a single neighbourhood component (link, processor, or processor and subset of its incident links), together with the execution of certain atomic steps specified in the sequel. Topology changes involving numerous links and processors can be modelled by a sequence of single change events. The crash of processor p is denoted crash_p; the recovery of processor p is denoted recov_p; crash_{pq} and recov_{pq} denote link failure and recovery events. In our model, a processor crash and a link crash are indistinguishable to a neighbour of the event: if p and q are neighbours and crash_p occurs, then we model this event by crash_{pq} with respect to reasoning about processor q. Similarly, a recov_p event is indistinguishable from a recov_{pq} event with respect to reasoning about a neighbour q of p. We say that a topology change event $\mathcal{E}$ is *incident* on p if $\mathcal{E}$ is recov_p, crash_{pq}, or recov_{pq}. We extend this definition to be symmetric: $\mathcal{E}$ is incident on p iff p is incident on $\mathcal{E}$.

For most of the protocols presented in this paper, each processor is equipped with an *interrupt statement,* which is a statement concerned with adjusting to topology change. A topology change $\mathcal{E}$ incident on p causes the following to atomically occur at p: the input variable N_p is changed to reflect $\mathcal{E}$, the interrupt statement of the protocol is atomically executed, and p's program counter is set to the first statement of its program. Note that if $\mathcal{E}$ is incident on numerous processors, then all incident neighbourhoods change to reflect $\mathcal{E}$ and all processors execute the interrupt statement atomically with event $\mathcal{E}$. Thus the transition by $\mathcal{E}$ from $\mathcal{T}.\alpha$ to $\mathcal{T}.\beta$ changes more than neighbourhoods; states α and β also differ in the local states of processors incident on $\mathcal{E}$ due to execution of interrupt statements at these processors.

A *trajectory* is a sequence of global states in which each segment is either a fair computation or a sequence of topology change events. For purposes of reasoning about self-stabilization, we follow the standard method of proving properties of computations, not trajectories. Dynamic change is handled indirectly in this approach: following an event $\mathcal{E}$, if there are no further changes for a sufficiently long period, the protocol self-stabilizes in the computation following $\mathcal{E}$ in the trajectory.

3 Superstabilization

The definition of superstabilization takes the idea of a "typical" change into account by specifying a class Λ of topology change events. A self-stabilizing procotol is superstabilizing with respect to events of type Λ, if starting from a legitimate state followed by a Λ-event, the passage predicate holds continuously:

Definition 3.1 A protocol P is *superstabilizing* with respect to Λ iff P is self-stabilizing and for every trajectory Φ beginning at a legitimate state and containing a single topology change event of type Λ, the passage predicate holds for every $\sigma \in \Phi$. □

Although Definition 3.1 considers trajectories with a single change, we emphasize that the intent is to handle trajectories with multiple changes (each change is completely accomodated before the next change occurs). Our definition could be modified to state this explicitly, however we have chosen this simpler form to streamline presentations.

A primary motivation for superstabilization is the notion of a "low–impact" reaction by a protocol to dynamic change. Intuitively, this means that changes necessary in response to dynamic change should affect relatively few processors and links. To formalize this notion, we introduce an *adjustment measure*. To define an adjustment, we return to the notion of legitimacy and a predicate $\mathcal{P}$ that effectively characterizes the legitimacy predicate $\mathcal{L}$. Let $var(\mathcal{P})$ be the minimal collection of variables and fields upon which $\mathcal{P}$ depends. Call $\mathcal{O}$ the state-space ranging only over the $var(\mathcal{P})$ data. The expression $\delta[\mathcal{O}]$ denotes a system state projected onto the $\mathcal{O}$ state-space. Now we consider a function $\mathcal{F}: \mathcal{O} \longrightarrow \mathcal{O}$. Function $\mathcal{F}$ maps states $\delta[\mathcal{O}]$ to states $\sigma[\mathcal{O}]$ that satisfy $\sigma \vdash \mathcal{L}$, where δ is any state that can be obtained from a legitimate state followed by a Λ-topology change $\mathcal{E}$. The idea is that $\mathcal{F}$ represents the strategy of a superstabilizing protocol in reacting to an event $\mathcal{E}$, choosing a new legitimate state following dynamic change. We rank a function $\mathcal{F}$ by means of an *adjustment measure* $\mathcal{R}$. The adjustment measure $\mathcal{R}$ is the maximum number of processors having different $\mathcal{O}$-states between $\sigma[\mathcal{O}]$ and $\mathcal{F}(\sigma[\mathcal{O}])$, taken over all states σ derived from some state $\delta \vdash \mathcal{L}$ followed by some change event $\mathcal{E} \in \Lambda$. A definition of $\mathcal{F}$ with a small adjustment measure $\mathcal{R}$ implies that few adjustments are necessary in response to a topology change.

A superstabilizing protocol *respects* $\mathcal{F}$ if the protocol implements $\mathcal{F}$, meaning that it responds to a dynamic change by some computation Θ taking the system from a state $\sigma[\mathcal{O}]$ to a state $\mathcal{F}(\sigma[\mathcal{O}])$ and changes $\mathcal{O}$-states at the minimum number of processors necessary in order to establish the new legitimate state given by $\mathcal{F}$. If a protocol respects $\mathcal{F}$, then we say that the protocol has adjustment measure $\mathcal{R}$.

To describe the time complexity of a protocol, the notion of a cycle is introduced. A *cycle for a processor* p is a minimal sequence of steps in a computation so that an iteration of the protocol at processor p executes the program for p from first to last statement. All the programs of this paper are constructed so that a processor p's cycle consists of reading all of p's neighbour registers, some local computation, and writing into p's register. The time-complexity of a computation is measured by *rounds*, defined inductively as follows. Given a computation Φ, the first round of Φ terminates at the first state at which every processor has completed at least one cycle; round $i+1$ terminates after each processor has executed at least one cycle following the termination of round i.

The *stabilization time* of a protocol is the maximum number of rounds it takes for the protocol to reach a legitimate state starting from an arbitrary state. The *superstabilization time* is the maximum number of rounds it takes for a protocol starting from an arbitrary legitimate state σ, followed by an arbitrary Λ-change event $\mathcal{E}$, to again reach a legitimate state.

4 Superstabilizing Colouring

This section exercises the definitions and notation developed in Sections 2 and 3 for a simple allocation problem. Let $\mathcal{C}$ be a totally ordered set of colours satisfying $|\mathcal{C}| \geq 1 + \Delta$, where Δ is a bound on the number of neighbours a processor has in any trajectory. Each processor p has a register field $colour_p$. The predicate $\mathcal{P}$ of interest is: $colour_p \in \mathcal{C}$ for every processor p and no two neighbouring processors have equal colours. A legitimate state for the colouring protocol is any state such that (i) predicate $\mathcal{P}$ is satisfied, and (ii) for each computation that starts in such a state, no processor changes colour in the computation. To define the passage predicate, we extend the domain of

self-stabilizing section :
S1 A, B := $\emptyset$, $\emptyset$
S2 **forall** $q \in N_p$
S3 **do** read(q); A := A $\cup$ $colour_q$; **if** $q > p$ **then** B := B $\cup$ $colour_q$ **od**
S4 **if** (($colour_p = \perp \wedge \perp \notin B$) $\vee$ ($colour_p \neq \perp \wedge colour_p \in B$))
then $colour_p$:= $choose(C \setminus A)$
S5 write
interrupt section :
E1 write (**if** ($\mathcal{E} = recov_p \vee (\mathcal{E} = recov_{pq} \wedge p > q)$) **then** $colour_p := \perp$)

Figure 1: Superstabilizing Colouring Protocol for Processor p.

a colour field to include $\perp \notin C$. The passage predicate $\mathcal{Q}$ is: $colour_p \in C \cup \{\perp\}$ for every processor p and, for any neighbouring processors p and q, $colour_p = colour_q$ iff $colour_p = \perp$.

Figure 1 presents a protocol for the colouring problem. The function $choose(S)$ selects the minimum colour from a set S (and is undefined if S is empty). The protocol of Figure 1 has two parts: one part is a self-stabilizing protocol, modified to deal with the $\perp$ element; the other part lists the interrupt that deals with topology change events. The self-stabilizing section perpetually scans for a colour conflict with the set of neighbouring processors having a larger identifier. The interrupt statement writes to the register, conditionally changing the $colour_p$ field in case the topology change event is a restart of the processor or a link.

Theorem 1 The colouring protocol is self-stabilizing and converges in $O(n)$ rounds.

Proof: We show by induction on the number of processors, that following round i, $0 \leq i \leq n$, the i largest-identifier processors have permanent, non-$\perp$ colour assignments such that no conflict with a neighbour of higher identity exists among these i processors. The basis for the induction is trivial since the empty set of processors satisfies the assertion. Now suppose the claim holds following round k. We examine the effect of round $(k+1)$ with the respect to processor r, where r is the $(k+1)^{th}$ largest processor identifier. In this round, processor r chooses some non-$\perp$ colour differing from any colour of a neighbouring processor with larger identity. The choice is deterministic, based on the colours of the larger identities. By hypothesis, these larger identity colour assignments are permanent, so following round $(k+1)$ and for all subsequent rounds, processor r's colour is fixed and differs from the colours of all neighbours of larger identity. Thus after $(n+1)$ rounds, all processors have permanent colour assignments. □

The class of topology events considered for the protocol is $\Lambda(k)$, which includes any crash event, any link $recov_{pq}$ event, and any $recov_p$ event subject to the restriction that at most k links incident on p recover at the same instant that p recovers; thus $\Lambda(0)$ allows only processor or link recovery events that are not simultaneous, whereas $\Lambda(\Delta)$ includes the possibility of a processor and all its links recovering as a single event.

Theorem 2 The colouring protocol is superstabilizing with a superstabilizing time of $O(k)$ and adjustment measure $\mathcal{R} = (k+1)$, where $\Lambda(k)$ is the class of topology change events.

Proof: In the case of any crash event, the protocol remains in a legitimate state. In the case of a recov event, for any new link introduced by the event, one or both of the incident processors has colour $\perp$ as a result; thus the passage predicate holds. Moreover, at most $(k+1)$ processors can have colour $\perp$ as a result of the recov event; by an argument similar to that given in the proof of Theorem 1, a legitimate state is obtained in $O(k)$ rounds, and the passage predicate holds invariantly in the computation; the adjustment measure $\mathcal{R} = (k+1)$ follows from the fact that only those processors incident on the change event adjust colour during the period of superstabilization. □

```
self-stabilizing section :
S1   x, y := ∞, ⊥
S2   forall q ∈ N_p
S3      do read(q); if x > (d_q + w_pq) then x, y := (d_q + w_pq), q od
S4   if p = r then d_p, t_p := 0, r else d_p, t_p := x, y
S5   write
interrupt section :   skip
```

Figure 2: Superstabilizing Tree Protocol for Processor p.

The colouring protocol illustrates qualitative and quantitative aspects of superstabilization. The qualitative aspect is illustrated by the fact that the convergence following a topology change does not violate a passage predicate. This ensures better service to the user when no catastrophe takes place (i.e. in the absence of a severe transient fault). Quantitative aspects can be seen by the $O(k)$ convergence time and adjustment measure. The same protocol, when started in an arbitrary initial state induced by a transient fault, might take $O(n)$ rounds to converge and a processor could change colours $O(n)$ times during this convergence. Indeed if the superstabilizing component of the protocol is removed, namely the interrupt statement, then $O(n)$ rounds can be required for convergence following even a single topology change event starting from a legitimate state.

5 Superstabilizing Tree

Constructing a spanning tree in a network is a basic task for many protocols. Several distributed reset procedures, including self-stabilizing ones, rely on the construction of a rooted spanning tree to control synchronization. All existing deterministic self-stabilizing algorithms to construct spanning trees rely on processor or link identifiers to select, for example, a shortest-path tree or a breadth-first search tree. In a dynamic network, a change event can invalidate an existing spanning tree and require that a new tree be computed. Although computation is required when a change event crash_{pq} removes one of the links in the current spanning tree, one would hope that a change event recover_{pq} would require no adjustment to an existing spanning tree. Yet all the self-stabilizing spanning tree algorithms we know (e.g., [DIM93, AG90, AK93]) construct a BFS or DFS tree and thus require, in some cases, recomputation of the tree when a link recovers, regardless of whether the network currently has a spanning tree or not. The reason is that a processor cannot locally "know" that the system has stabilized and must make a deterministic choice of edges to be included in the tree. We propose a superstabilizing approach to tree construction. The protocol given in this section successfully "ignores" all dynamic changes that add links to an existing spanning tree or crash links not contained in the tree.

All trajectories considered in this section are free of crash_p or recover_p events; the number of processors remains fixed at n and we give every processor access to the constant n. We also suppose that the network remains, at all states in a trajectory, connected.

The basic idea of the protocol is the construction of a least-cost path tree to a processor r designated as the root of the tree. The key innovation of the protocol lies in the definition of link costs. Each link is assigned a cost in such a way that links that are part of the tree have low cost whereas links outside the tree have high cost. Each processor p has two register fields t_p and d_p. The field t_p ranges over identifiers of processors. The register d_p contains a non-negative integer. The function w maps a pair of processor identifiers to an integer: $w_{pq} = 1$ if $t_p = q$ and $w_{pq} = n$ otherwise.

Figure 2 shows the code of the superstabilizing spanning tree protocol. The predicate $\mathcal{P}$ of interest for the tree

protocol is that $(\forall p, q : p \neq r \wedge t_p = q : q \in N_p)$ and that the collection of t_p variables $\{t_p \mid p \neq r\}$ represents a spanning, directed tree rooted at r. A legitimate state for the tree protocol is any state such that (i) predicate $\mathcal{P}$ is satisfied, and (ii) for each computation that starts in such a state, no processor changes a t_p variable in the computation.

Theorem 3 The spanning tree protocol self-stabilizes in $O(n)$ rounds. (See Appendix for proof.)

We define the class of change events Λ for purposes of superstabilization to be any recov_{pq} event or any crash_{pq} event such that neither $t_p = q$ nor $t_q = p$ holds at the moment of the crash_{pq} event. The passage predicate $\mathcal{Q}$ for the superstabilization property is identical to $\mathcal{P}$.

Theorem 4 The spanning tree protocol is superstabilizing for the class Λ with superstabilization time $O(1)$ and adjustment measure $\mathcal{R} = 1$. (See Appendix for proof.)

6 General Superstabilization

This section introduces a general method for achieving superstabilization with respect to the class Λ of single topology changes. Our general method can be seen as a compiler that takes self-stabilizing protocol P and outputs a new protocol P^s that is both self-stabilizing and superstabilizing. This is done by modifying protocol P and superimposing a new component called the *superstabilizer*. The superstabilizer uses, as a tool, a self-stabilizing update protocol. The following subsection describes our update protocol, after which we give an overview of the superstabilizer.

Update Protocol

To simplify the presentation of our general method for superstabilizing protocols, we employ a self-stabilizing update protocol. We view the update protocol as the simplest and clearest self-stabilizing protocol for large class of tasks including: leader-election, topology update and diameter estimation. To describe the task of the update protocol, suppose every processor p has some field image x_p; for the moment, we consider x_p to be a constant. The *update* problem is to broadcast each x_p to all processors. This problem is called *topology update* when the field x_p contains all the local information about p's links and network characteristics. Many dynamic system are already equipped with a topology update protocol that notifies processors of the current topology; in such instances our general method acts as an extension to this existing topology update. An optimal time ($\Theta(d)$ round) self-stabilizing solution to the topology update is given in [SG89, Do93]. To ensure a desired deterministic property of the protocol, we assume that the neighbourhood of a processor N_p is represented as an ordered list.

Let each processor p have, in addition to x_p, a field e_p, where e_p contains three-tuples of the form $\langle q, u, k\rangle$, in which q is a processor identifier, u is of the same type as x_p, and k is a non-negative integer. Let $\mathsf{dist}_{\mathcal{T}}(p, q)$ be the minimum number of links contained in a path between processors p and q in topology $\mathcal{T}$; the third component of a tuple is intended to represent the dist-value for the processor named in the tuple's first component. We make some notational conventions in dealing with tuples: with respect to a given (global) state, $\langle q, x_q, k\rangle$ is a tuple whose second component contains the current value of field x_q. In proofs and assertions, we specify tuples partially: $\langle q, , \rangle \in e_p$ is the assertion that processor p's e-field contains a tuple with q as its first component. Each processor uses local variables A and B that range over the same set of tuples that e_p does. For field image e_p and set variables A and B, we assume that set operations are implemented so that computations on these objects are deterministic.

The update protocol's code uses the following definitions. Let $\mathsf{processors}(A)$ be the list of processor identifiers obtained from the first components of tuples in A. Let $\mathsf{mindist}(q, A)$ be the first tuple in A having a minimal third

```
C1   A, B := ∅, ∅
C2   forall q ∈ N_p do read(q); A := A ∪ e_q od
C3   A := A \ \⟨p, *, *⟩; A := A++⟨*, *, 1⟩
C4   forall q ∈ processors(A) do B := B ∪ {mindist(q, A)} od
C5   B := B ∪ {⟨p, x_p, 0⟩}; e_p := initseq(B)
C6   write
```

Figure 3: Update Protocol for Processor p.

component of any tuple whose first component is q (in case no matching tuple exists, then mindist is undefined.) Define $A \setminus\setminus \langle q,*,*\rangle$ to be the list of tuples obtained from A by removing every tuple whose first component is q. Define $A{+}{+}\langle *,*,1\rangle$ to be the list of tuples obtained from A by incrementing the third component of every tuple in A. Define initseq(A) by the following procedure: (1) sort the tuples of A in ascending order of the third element of a tuple; (2) from this ordered sequence of tuples, compute the maximum initial prefix of tuples with the property: if $\langle q,u,k\rangle$ and $\langle q',u',k'\rangle$ are successive tuples in the prefix, then $k' \leq k+1$. Then initseq(A) is the set of tuples in this initial prefix.

For the update protocol, we define a *distance-stable* state to be any state for which (1) each processor p has exactly one tuple $\langle q,y,\mathrm{dist}(p,q)\rangle$ in its e_p field for every processor q in the network reachable by some path from p in the current topology; (2) e_p contains no other tuples; and (3) each computation that starts in such a state preserves (1) and (2). A *legitimate* state for the update protocol is a distance-stable state in which requirement (1) is strengthened to: each processor p has exactly one tuple $\langle q,x_q,\mathrm{dist}(p,q)\rangle$ in its e_p field for every processor q — in other words, the x-field images are accurate. Figure 3 presents the protocol.

Theorem 5 The update protocol of Figure 3 self-stabilizes in O(d) rounds. (see Appendix for proof).

A corollary of self-stabilization is that, if one of the x_p fields is dynamically changed, the protocol will effectively broadcast the new x_p value to other processors. Of particular interest are some properties that relate a sequence of changes to an x_p field to the sequence x_p values observed at another processor q. More specifically, if processor p writes, over the course of a computation, the values $c_1, c_2, \ldots$ into x_p, and no processor q reads (via update images of x_p) a value c_k and then later reads a value c_j for $j < k$, then we call the update protocol *monotonic*. A monotonic update protocol guarantees that the sequence of values in any field image is a subsequence of the values written to the corresponding register field. It can be shown that the protocol Figure 3 is monotonic in any computation starting from a legitimate state (by induction on a lexicographic measure composed of path length and the ordering of links by a processor's neighbourhood; essentially the deterministic ordering of links defines a broadcast tree). In the context of a dynamic system, we could also require that monotonicity hold in any trajectory that begins with a legitimate state. Unfortunately, the update protocol of Figure 3 does not have this stronger property; however, a limited form called *impulse* monotonicity is satisfied.

Impulse Monotonicity. Let σ be a legitimate state for the update protocol in a topology $\mathcal{T}$ where $x_p = c_0$ at σ. Let λ be the state obtained by making a single topology change $\mathcal{E}$ to $\mathcal{T}$ and the assignment $x_p := c_1$. Let Φ be a topology-constant computation originating with state λ. Impulse monotonicity is satisfied if, for any states ρ and γ in Φ such that ρ occurs before γ: if processor q sees c_1 as the value of x_p at state ρ, then q sees c_1 as the value of x_p at state γ.

Impulse monotonicity is useful in the following way: if x_p is changed "slowly enough", meaning that the protocol successfully stabilizes between changes to x_p, then a FIFO broadcast of x_p-values is obtained. In the sequel, we

introduce an acknowledgement mechanism so that a processor does not change the broadcast value of interest until all other processors within a connected component have received the current value.

Theorem 6 The update protocol of Figure 3 enjoys impulse monotonicity. (Proof appears in [DH95].)

The Superstabilizer

The superstabilizer makes use of function $\mathcal{F}$, described in Section 3, to determine a new legitimate state for protocol P following a topology change $\mathcal{E}$. It is the responsibility of the superstabilizer to "hide" $\mathcal{E}$ from any processor in such a way that no user of protocol P can observe a state inconsistent with the current topology; this is done by making the global transition between legitimate states for different topologies effectively atomic, thus sparing procotol P from any stabilization effort.

The superstabilizer consists of two components, a modified version of the update protocol and an interrupt statement. Atomic steps of P and the update protocol are then interleaved. We modify P, as follows: each action of P at processor p is guarded by a boolean variable $freeze_p$ so that when $freeze_p$ holds, no action of P is enabled at processor p and its program counter remains static. Our superstabilizer ensures that, starting from any initial state, all *freeze* fields eventually become *false* in the absence of topology changes.

The combination of the superstabilizer and modified protocol P results in a superstabilizing protocol P^s. A legitimate state for P^s is any state in which: (1) the variables, fields and program counter with respect to P satisfy $\mathcal{LP}$ (where $\mathcal{LP}$ is the legitimacy predicate for the base protocol P); (2) the update protocol component of the superstabilizer is in a legitimate state (all e-fields have accurate tuples); (3) every *freeze* variable is *false*; and (4) each computation that starts in such a state preserves (1)–(3).

The passage predicate for our general method is defined in terms of the *freeze* fields. For any state σ, let $warm(\sigma)$ denote the set of processors having *freeze* fields that are *false*. Let $\sigma[warm]$ be the vector of local states of processors in $warm(\sigma)$. We call σ *warm-legitimate* if there exists a state γ and a topology $\mathcal{T}.\gamma$, where $\gamma \vdash \mathcal{LP}$, such that $\sigma[warm] = \gamma[warm]$. In words, σ is warm-legitimate if it appears to be a legitimate state (with respect to some topology) when we disregard any processor p with $freeze_p = true$. The passage predicate $\mathcal{Q}$ for the general method is that the protocol is in a warm-legitimate state.

The interface between the superstabilizer and P at processor p consists not only of the $freeze_p$ variable, but a pseudo-variable snap_p, which is defined to be the collection, with respect to protocol P, of all local variables, shared fields, and the program counter of P for processor p. The superstabilizer can read and write snap_p. We denote by *snap* a set of snap_p variables, one for each processor. Our general method is, in brief, the following: after a topology change, P is frozen at all processors and a *snap* value is recorded; subsequently a *snap* value appropriate for the new topology is computed and each frozen processor is assigned its portion of the new *snap* value; and finally all processors are thawed.

The programming notation given in Section 2 makes local images of register fields available to program operations: such images can be of a processor's own register or that of its neighbouring processors; for example, the code of Figure 3 permits processor p to refer to e_q for $q \in N_p$. The update protocol makes an image of each processor's x-field available to every other processor within a connected component. For the superstabilizer, we extend the programming notation to allow any processor to refer to fields of any other processor. Thus processor p can refer to x_q for any $q \in \mathrm{processors}(e_p)$ by using images provided in the e-field's tuples. Of course, these images may be out-of-date, which necessitates synchronization measures in the superstabilizer; such synchronization is achieved in *phases* to coordinate freezing and snapshots.

For convenience in describing the superstabilizer, we divide the x field into four subfields: $x_p = [\, a_p\ h_p\ t_p\ u_p \,]$. To control the phases of superstabilization, the subfield a_p is used; it is a ternary-valued subfield provided for the

three phases of superstabilization. These phases are: *Phase 0* is the normal state of the superstabilizer, in which protocol P is active and the superstabilizer is idle. When $(\forall p :: a_p = 0)$ holds, we consider the superstabilization to be inactive (terminated). *Phase 1* consists of freezing protocol P and collecting snapshots from the frozen processors; also in this phase an election takes place among all processors incident on a topology change to determine a single coordinator of the following phase. Phase 1 is active if $(\exists p :: a_p = 1)$ and $(\forall p :: a_p \leq 1)$. *Phase 2* is concerned with computing a new global state for protocol P and distributing the new state to all processors. Phase 2 is active if $(\exists p :: a_p = 2)$, remains active until acknowledged by all processors, and thereafter terminates in order to resume execution of Phase 0.

To detect progress of phases, we employ an acknowledgement subfield h_p. This subfield is a vector of ternary values whose elements are images known to p of other processor a-subfields: the protocol sets $h_p[r]$ to contain the image of a_r, as determined from p's image of x_r broadcast via the update protocol. Further, since h_p is broadcast via the update protocol to every processor, it is possible for a processor r to test the status of every other processor's image of a_r.

In addition to the a and h-subfields, we define additional subfields of x_p to contain *snap* values. Subfield t_p contains a value of type snap_p, which is the portion of the state of p that is related to P. We also define the subfield u_p to contain a global *snap* value, i.e., $u_p[r]$ contains a snap_r value. We denote by s_p the collection of all t_r images obtained from x_p subfields.

To make a concise presentation, an additional device is used in the code of the interrupt statement. The function $\mathsf{refresh}(e_p)$ reproduces e_p except that the value of the x_p field is updated, i.e., $\mathsf{refresh}(e_p) = (e_p \setminus \langle p, *, * \rangle) \cup \{\langle p, x_p, 0 \rangle\}$.

The interrupt statement for the superstabilizer is given in Figure 4. In response to a topology change $\mathcal{E}$ incident on processor p, the program counter of the protocol is reset to S1, the neighbourhood N_p is adjusted to reflect $\mathcal{E}$, and the write operation is atomically executed. This operation halts P by setting $freeze_p$ to *true*.

The remaining component of the superstabilizer consists of the combination of Figures 3 and 4, i.e., it is a modified update protocol. Statements U1–U8 should be inserted between statements C3 and C4 to obtain the complete protocol. All quantifications over processors in expressions (such as $(\forall q :: a_q = 0)$) are implicitly quantified over $\mathsf{processors}(A) \cup \{p\}$ in the superstabilizer code.

Theorem 7 Protocol P^s is self-stabilizing with self-stabilization time $O(d + K)$, where K is the self-stabilization time of protocol P.

Sketch of proof: For simplicity we suppose the network to be connected. By the construction of P^s, it suffices to prove that the superstabilizer converges in $O(d)$ rounds to $\mathcal{A} = (\forall q :: \neg freeze_q)$ and to show that $\mathcal{A}$ (or some stronger predicate) is stable. Thereafter, in $O(K)$ additional rounds, by base protocol P's self-stabilization, P^s stabilizes. Only statement U8 of the protocol assigns to $freeze_p$; if we show P^s stabilizes to $(\forall q :: a_q = 0)$, then by Theorem 5, all a-field images are broadcast in $O(d)$ rounds; all *freeze* variables are *false* in the following round. Notice that $a_p = 0$ is stable for any p, since none of U1–U8 assign to a_p if $a_p = 0$ holds. Therefore it suffices to show that *some* state statisfying $a_q = 0$ occurs for each processor q within $O(d)$ rounds of any computation. Heading for a contradiction, let processor r be a processor such that $a_r \neq 0$ holds for more than $O(d)$ rounds of some computation; because none of none of U1–U8 assign $a_r := 1$, yet $a_r = 1$ is the precondition of assigning $a_r := 2$ (see U2), we deduce that $a_r \neq 0$ holds continuously for more than $O(d)$ rounds. Suppose $a_r = 2$ holds continuously; by acknowledgements from U5–U7 and stabilization of the update protocol, U3 is eventually assigns $a_r := 0$, which is a contradiction. It remains to consider that $a_r = 1$. There may be more than one processor for which the a-field is continuously 1-valued; let t be such a processor of maximum identifier. By acknowledgements U5–U7 and the update protocol, any processor $q \neq t$ having $a_q = 1$ assigns $a_q := 0$ at U1 within $O(d)$ rounds (because $q < t$). Thus, after $O(d)$

Superstabilization Section:

U1 **if** $(a_p = 1 \wedge (\exists q :: a_q \neq 0 \wedge q > p \wedge (\forall r :: h_r[q] \neq 0)))$
then $a_p := 0$

U2 **if** $(a_p = 1 \wedge (\forall q : q \neq p : a_q = 0) \wedge (\forall q :: h_q[p] \neq 0))$
then $a_p, u_p := 2, \mathcal{F}(s_p)$

U3 **if** $(a_p = 2 \wedge (\forall q :: h_q[p] \neq 1))$
then $a_p := 0$

U4 **forall** $q \in \text{processors}(A) \cup \{p\}$
do

U5 **if** $a_q = 0$ **then** $h_p[q] := 0$

U6 **if** $a_q = 1 \wedge h_p[q] = 0$ **then** $h_p[q] := 1$

U7 **if** $a_q = 2 \wedge h_p[q] = 1$ **then** $h_p[q], \text{snap}_p := 2, u_q[p]$
od

U8 **if** $(\exists q \in \text{processors}(A) \cup \{p\} :: a_q \neq 0)$
then $\mathit{freeze}_p, t_p := \mathit{true}, \text{snap}_p$
else $\mathit{freeze}_p := \mathit{false}$

Interrupt Section:

E1 write

$$\left(\begin{array}{lcl} a_p & := & 1 \\ \mathit{freeze}_p & := & \mathit{true} \\ t_p & := & \text{snap}_p \\ e_p & := & \begin{cases} \emptyset & \text{if } \mathcal{E} = \text{recov}_p \\ \text{refresh}(e_p) & \text{if } \mathcal{E} \neq \text{recov}_p \end{cases} \end{array} \right)$$

Figure 4: Superstabilizer: Update Extension and Interrupt for p.

rounds, t is the only processor having a 1-valued a-field. But again, by acknowledgements U5–U7 and the update protocol, t subsequently assigns $a_r := 2$ at U2 within O(d) rounds, which is a contradiction. □

Theorem 8 Protocol P^s is superstabilizing with superstabilization time O(d).

Sketch of proof: Consider a computation beginning from a state σ that is the result of a single topology change event $\mathcal{E}$ at a legitimate state. By E1, $\sigma \vdash a_r = 1 \wedge \mathit{freeze}_r$ holds for every r incident on $\mathcal{E}$; thus σ is warm-legitimate. As the update protocol broadcasts the 1-valued a-fields, statement U8 sets freeze_p at all processors p within O(d) rounds. Thanks to impulse monotonicity, the condition $(\exists q :: a_q \neq 0)$ observed by update images is stable so long as at least one processor does not change its 1-valued a-field. Within O(d) rounds, statement U1 assures that only one processor t (per connected component) satisfies $a_t = 1$; in at most O(d) subsequent rounds, by acknowledgements and the update protocol, statement U2 executes $a_t := 2$ at processor t. At this point we assert that all processors are frozen; note also that each global state from σ to this point is warm-legitimate. Upon execution of U2, a new global state is computed from combined snapshots. Although $a_t = 2$ is not broadcast monotonically, statements U6 and U7 are coded in such a way that acknowledgement of $a_t = 2$ is monotonic. Therefore, once t observes $(\forall q :: h_q[p] \neq 1)$, it is the case that every processor has received the new global state and assigned P's fields and variables. Thus after O(d) rounds, t executes U3 and the final phase of the superstabilizer begins. In this final phase, stabilization to $(\forall q :: \neg \mathit{freeze}_q)$ occurs within O(d) rounds; the passage predicate holds because, at each state, the subset of unfrozen processors are locally legitimate for the new topology. □

7 Conclusions

There is increasing recognition that dynamic protocols are necessary for many networks. Studying different approaches to programming for dynamic environments is therefore a motivated research topic. Although self-stabilizing techniques for dynamic systems have been previously suggested, explicit research to show how and where these techniques are useful has be lacking. This paper shows how assumptions about interrupts and dynamic change can be exploited with qualitative and quantitative advantages while retaining the fault-tolerant properties of self-stabilization.

In particular, we suggest that when the system is in an illegitimate configuration, reset to a predefined configuration will not take place; instead, the system will reach a legitimate configuration that is close to the current illegitimate configuration (where "close" means small adjustment measure). The benefits of this approach are twofold: first, such a strategy may keep most of the sites of the system unchanged and in working order (as in the example of connections within an unchanged portion of a spanning tree); second, in some cases the amount of work (superstabilizing time) required to reach a close legitimate state can be small (as in our colouring example).

References

[AAG87] Y. Afek, B. Awerbuch and E. Gafni, "Applying Static Networks Protocols to Dynamic Networks," *Proc. of the 28th IEEE Symp. on Foundation of Computer Science* pp. 358-370, 1987.

[AG90] A. Arora and M. G. Gouda, "Distributed Reset," *Proc. FST 10, Springer LNCS*, 472 pp. 316–331, 1990.

[AGH90] B. Awerbuch, O. Goldreich and A. Herzberg, "A Quantitative Approach to Dynamic Networks," *Proc. of the 9th ACM Symp. on Principles of Distributed Computing*, pp. 189-203, 1990.

[AGR92] Y. Afek, E. Gafni and A. Rosen, "The Slide Mechanism with Applications in Dynamic Networks," *Proc. of the 11th ACM Symp. on Principles of Distributed Computing*, pp. 35–46, 1992.

[AK93] S. Aggarwal and S. Kutten, "Time Optimal Self-Stabilizing Spanning Tree Algorithm," *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, 1993.

[AM92] B. Awerbuch and Y. Mansour, "An Efficient Topology Update Protocol for Dynamic Networks," *Proc. of the 6th International Workshop on Distributed Algorithms*, pp. 185-201, 1992.

[APV91] B. Awerbuch, B. Patt-Shamir and G. Varghese: "Self-Stabilization by Local Checking and Correction," *Proc. of the 32nd IEEE Symp. on Foundation of Computer Science* pp. 268–277, 1991.

[Do93] S. Dolev, "Optimal Time Self Stabilization in Dynamic Systems," *Proc. of the 7th International Workshop on Distributed Algorithms* (Springer-Verlag LNCS 725), pp. 160–173, September 1993.

[DIM93] S. Dolev, A. Israeli and S. Moran, "Self-Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity," *Distributed Computing*, 7 pp. 3–16, 1993.

[DH95] S. Dolev and T. Herman, "SuperStabilizing Protocols for Dynamic Distributed Systems (Preliminary Version)", University of Iowa Department of Computer Science Technical Report 95-02, 1995.

[KP93] S. Katz and K. J. Perry, "Self-Stabilizing Extensions for Message-Passing Systems", *Distributed Computing*, 7 pp. 17–26, 1993.

[SG89] J. Spinelli and R.G. Gallager, "Event Driven Topology Broadcast Without Sequence Numbers", *IEEE Transactions on Communication*, Vol. 37, No. 5, (1989) pp. 468-474.

Appendix

Proof of Theorem 3:

Proof: Proof by induction on an arbitrary computation Φ. The induction is based on a directed tree. Let T_r be the maximum subset of processors satisfying: (1) $d_r = 0$, (2) the set $\{t_p \mid p \in T_r \wedge p \neq r\}$ represents a directed tree rooted at r, (3) for $p \in T_r$ and $p \neq r$, register field d_p satisfies $d_p = 1 + d_q$ where $q = t_p$ and, (4) each processor in T_r has executed at least one cycle in Φ. After one round, $d_r = 0$ holds for the remainder of the computation. Therefore, after the first round, T_r is non-empty, containing at least r. The remainder of the proof concerns rounds two and higher, and is organized into three claims.

Claim 1: (T_r is stable). If $p \in T_r$ holds at the beginning of the round, then t_p and d_p do not change during the round. The claim follows by induction on depth of the tree T_r. First we strengthen the hypothesis to state that any node $p \in T_r$ satisfies $d_r = k$, where k is the depth of p in T_r. The basis for the induction is the root r, which satisfies $d_r = 0$; statement S4 assures that d_r does not change during the round. Suppose all nodes of T_r up to depth k satisfy the hypothesis, i.e., they are stable and have d-field equal to the node depth. Consider some $v \in T_r$ at depth $k+1$; by (2) and (3), $d_v = k+1$ at the beginning of the round. Since every d-field has non-negative value and $w_{vz} = n$ for any $z \neq t_v$, and assuming the inductive hypothesis for the node named by t_v, statement S3 cannot compute any lower value for x than $k+1$ and the values t_v and d_v do not change during the round.

Claim 2: (T_r growth). If there exists a processor that is not contained in T_r and $(\forall p : p \notin T_r : d_p > 2n)$ holds at the beginning of the round, then T_r grows by at least one processor by the end of the round. The claim follows by examining processors outside of T_r that also neighbour T_r. Let p be such a processor, outside T_r and neighbour to $q \in T_r$. By Claim 1, $d_q + w_{pq} < 2n$. Therefore, during the round, p cannot choose t_p to be some processor s satisfying $d_s > 2n$. Thus T_r grows by at least one processor.

Claim 3: (d_p growth). Define M_i to be the minimum d-register value of any processor outside of T_r in round i; then $M_{i+1} > M_i$. The claim is verified by considering, for round i and $p \notin T_r$, assignment to each d_p register in that round. During a round, the value obtained for d_p is strictly larger than that of some neighbouring d_q; if $q \in T_r$, then $p \in T_r$ holds at the end of the round; and if $q \notin T_r$, then the claim holds.

A corollary of Claim 3 is that following rounds $2n+2$ and higher, for every $p \notin T_r$, the field d_p satisfies $d_p > 2n$. Consequently for rounds $2n+2$ and higher, by Claim 2, if T_r does not contain all processors, then T_r grows by at least one processor in each successive round. The lemma follows because there are at most n processors. □

Proof of Theorem 4:

Proof: We show that starting from a state δ, $\delta \vdash \mathcal{L}$, followed by a topology change $\mathcal{E}$, $\mathcal{E} \in \Lambda$, resulting in a state σ, that $\sigma \vdash \mathcal{L}$ holds. In the case of $\mathcal{E} = \text{crash}_{pq}$ removing a non-tree link, for either processor p or q the weight of the p–q link $w_{pq} = n$ at state δ; by assumption of $\delta \vdash \mathcal{L}$, it follows that computation of d and t fields produce identical results in any round following σ since these are necessarily based on unit w-values. In the case of $\mathcal{E} = \text{recov}_{pq}$ the weight of the new p–q link is $w_{pq} = n$ at state σ, hence distances are not reduced by addition of the new link and computation of d and t fields produce identical results in any round following σ. Therefore $\sigma \vdash \mathcal{L}$. □

Proof of Theorem 5:

Proof: The the proof is organized as three claims.

Claim 1: Following round i, $i \geq 1$, the e_p field of every processor satisfies

$$(\forall p, q, j:\ j < i:\ \mathrm{dist}(p,q) \leq j \ \Leftrightarrow\ (\exists \langle q, x_q, \mathrm{dist}(p,q)\rangle \in e_p))$$

The claim follows by induction on i. The basis of the induction is the first round, which trivially establishes $\langle p, x_p, 0\rangle \in e_p$ for every processor. The induction step follows because field e_p is assigned anew in each round and based on tuples that, by the induction hypothesis, have the required property.

Claim 2: Following round i, $i \geq 1$, the e_p field of every processor satisfies

$$(\forall p, q, j:\ j < i:\ (\exists \langle q, y, k\rangle \in e_p ::\ k \leq j \ \Rightarrow\ (\mathrm{dist}(p,q) = k \ \wedge\ y = x_q)))$$

This claim follows by same inductive argument presented for Claim 1.

Claim 3: Following round $d+1$, $(\forall p ::\ (\forall \langle , , k\rangle \in e_p ::\ k \leq d))$

The claim is shown by contradiction. Suppose e_p contains a tuple $\langle , , j\rangle$ where $j > d$. Observe that if $j > d+1$ then, by the construction of the *initseq* function, at the end of round $d+1$ the field e_p also contains some tuple $\langle q, , k\rangle$ where $k = d+1$. Thus to show the claim, it suffices to show a contradiction for $k = d+1$. Since p assigned the tuple $\langle q, , d+1\rangle$ to e_p during round $d+1$, it must be that p found at some neighbour s the tuple $\langle \bar{q}, , d\rangle$ and found no tuple with q as first component at a smaller distance. However, the tuple located at s having distance d represents the shortest distance to q by Claim 2. And since d bounds the maximum possible shortest path, by Claim 1 all shortest paths between p and q are visible to p at the end of round d. We conclude that $\mathrm{dist}(p,q) = d+1$, which contradicts the definition of diameter d.

Claims 1–3 together imply that, following $d+1$ rounds, each processor correctly has a tuple for every other processor at distance d and that every tuple in an *e*-field correctly refers to a processor □

Paper Number 4

Space-Effecient Distributed Self-Stabilizing Depth-First Token Circulation

Colette Johnen and Joffroy Beauquier

SPACE-EFFICIENT DISTRIBUTED SELF-STABILIZING DEPTH-FIRST TOKEN CIRCULATION

Colette Johnen, Joffroy Beauquier
L.R.I./C.N.R.S.
Université de Paris-Sud
Bat. 490, Campus d'Orsay
F-91405 Orsay Cedex, France.
tel : (+33) 1 69 41 66 29
fax : (+33) 1 69 41 65 86
colette@lri.fr, jb@lri.fr

Abstract

The notion of self-stabilization was introduced by Dijkstra. He defined a system as self-stabilizing when "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps". Such a property is very desirable for any distributed system, because after any unexpected perturbation modifying the memory state, the system eventually recovers and returns to a legitimate state, without any outside intervention.

In this paper, we are interested in a distributed self-stabilizing depth-first token circulation protocol on an uniform rooted network (no identifiers, but a distinguished root).

As already noted, a search algorithm together with a deterministic enumeration of the node's neighbors yields an algorithm determining a spanning tree.

Our contribution is improving the best up to now known space complexity for this problem, from $O(\log(N))$ to $O(\log(D))$ where N is number of nodes and D is the network's degree. Moreover, we give a full proof of the algorithm correctness assuming the existence of a distributed demon.

Keywords : fault-tolerant distributed algorithms, self-stabilization, spanning tree, mutual-exclusion, distributed demon.

1 Introduction

The notion of self-stabilization was introduced by Dijkstra [5]. He defined a system as self-stabilizing when "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps". Such a property is very desirable for any distributed system, because after any unexpected perturbation modifying the memory state, the system eventually recovers and returns to a legitimate state, without any outside intervention. Self-stabilizing has been since studied by various researchers and Dijkstra's original notion, which had a very narrow scope of application, has proved to encompass a formal and unified approach to fault-tolerance, under a model of transient failures for distributed systems.

In this paper, we are interested in the construction of a distributed Self-stabilizing for depth-first token circulation in an uniform rooted network (no identifier, but a distinguished root). As a token circulation algorithm, our algorithm provides a fair mutual-exclusion protocol (the node having the token is the one authorized to enter into critical section).
Several authors [5], [6], and [3] have presented token circulation algorithm on ring networks; Brown, Gouda, and Wu [2] have presented one on linear chains; Kruijer [11] have presented one on tree networks. Huang and Chen have presented an algorithm [10] on general networks with a distributed demon.

As noted in [10], a token circulation algorithm together with a deterministic enumeration of the node's neighbors yields an algorithm determining a spanning tree. The task of spanning tree construction is a basic primitive in communication networks. Many crucial network tasks, such as network reset (and thus any input/output task), leader election, broadcast, topology update, and distributed database maintenance, can be efficiently carried out in the presence of a tree defined on the network nodes spanning the entire network. Improving the efficiency of the underlying spanning tree algorithm usually also correspondingly improves the efficiency of the particular task at hand.

Note that other constructions of spanning trees in a self-stabilizing way are known. Some authors (as in [1] and [4]) have presented algorithms with a central demon. Huang and Chen [9] construct a minimal spanning tree with a distributed demon. Sur and Srimani [12] have presented a similar algorithm but the correctness proof is substantially simpler, based on graph theoretical reasoning. Dolev, Israeli, and Moran [7] have reported a minimal spanning tree construction with read-write atomicity (then the system is fully asynchronous). Finally, Tsai, and Huang [13] have presented an algorithm that constructs a minimal spanning tree with a fully distributed demon.

There are two principal measures of efficiency for self-stabilizing algorithms : stabilization time, which is the maximum time taken for the algorithm to converge to a legitimate state, starting from an arbitrary state and the space required at each node (e.g. size of local memory needed). We are interested here in reducing the value of the second parameter. The goal of producing systems with a small number of states per processor/node is of particular interest because such processors may have direct implementations in hardware.

The existing solutions for token circulation or spanning tree construction on general network topology have a space complexity in $O(\log(N))$, N being the number of nodes. Our contribution is a new algorithm that achieves the goal in $O(\log(D))$ states per node, D being the upper bound of node's degree.

On the other hand, Burns and Pachl [3] showed that does not exist uniform self-stabilizing token circulation on a composite ring. The best that can be proposed is a semi-uniform algorithm, as our algorithm.

Moreover, our protocol does not need to know the number of nodes in the network. Therefore, it works for any connected network and even for dynamic networks, in which the topology of the network may change during the execution (nevertheless, the upper bound of the node's degree should not increase to keep constant the required memory space at each node).

We give the extensive proof of our algorithm within the distributed model where several nodes can simultaneously perform a move.

The remainder of the paper has been organized as follows; an informal description of the proposed protocol is provided in section 2. The formal model is described in section 3; protocol formal description is given in section 4; its correctness is proven in section 5.

2 Informal description of the protocol

As a model of computation, we choose the following model, that is an extension of Dijkstra's original model for rings to arbitrary graphs. Consider a connected graph G(V, E), in which V is a set of nodes and E is a set of edges. Such a graph is used to model a distributed system with N nodes, N = |V|, in which each node represents a processor. In the graph, directly connected nodes are called each other's neighbors. Our goal is to design a self-stabilizing algorithm that performs a depth-first search on the graph.

The proposed self-stabilizing algorithm is encoded as a set of rules. Each processor has several rules. Each rule has two parts : the privilege (condition) part, and the move part. The privilege part is defined as a boolean function of the processor's own state and of the states of its neighbors. When the privilege of a rule on a processor is true, we say that the processor has the privilege. A processor having the privilege may then make the corresponding move which changes the processor state into a new one that is a function of its old state and of the states of its neighbors.

We assume the existence of a *distributed* demon and we assume that the computation proceeds in steps. The distributed demon [5] chooses *several* privileged nodes and one enabled rule on each chosen node at a time. Hence, in each computation step, several processors make a move. The privileges for the next move depend on the states resulting from the previous moves. The rules are atomic : the processors cannot evaluate their privilege at a time and then make the move later with in between other moves.
To ensure the correctness of the protocol, the demon is regarded as an adversary and the protocol is required to be correct in all possible executions. Nevertheless, the demon is fair, a node does not hold forever a privilege on a rule without being chosen by the demon.

The proposed algorithm has two parts. One circulates the token among the nodes in an indeterministic depth-first order. This part is identical to the one in the Huang-Chen algorithm [10]. The other part handles abnormal situation due to unpredictable initial states or transient failures.

We name r the distinguished node that initiates the depth-first circulation rounds, and chooses the

round color (0 or 1). Each node has a color among three values : 0, 1, and E (for error). The node having the token, takes the round color and searches among its neighbors one which has not been visited during this round (an isolated node having the color different of the round color and of E color). If it finds a suitable node then it passes to this node the token; else it backtracks the token to its parent. When the token has backtrack to r; the round is over. r initiates a new round with the other color.

There are two error-handling strategies : one for destroying the illegal branches that are not cycles and the other for the cycles. The treatment of illegal branches (branches which are not cycles and which are not rooted to the legal root) is similar to the one used by Huang and Chen. The illegal roots detect their abnormal situation and color themselves to E. The E color is propagated to their leaf; then, the E-colored leaves are dropped; and the detached E-colored nodes are recovered by changing color. The repetition of dropping and recovering processes will correct all nodes inside illegal branches (there is a finite number of creations of new illegal branches). The cycle destruction

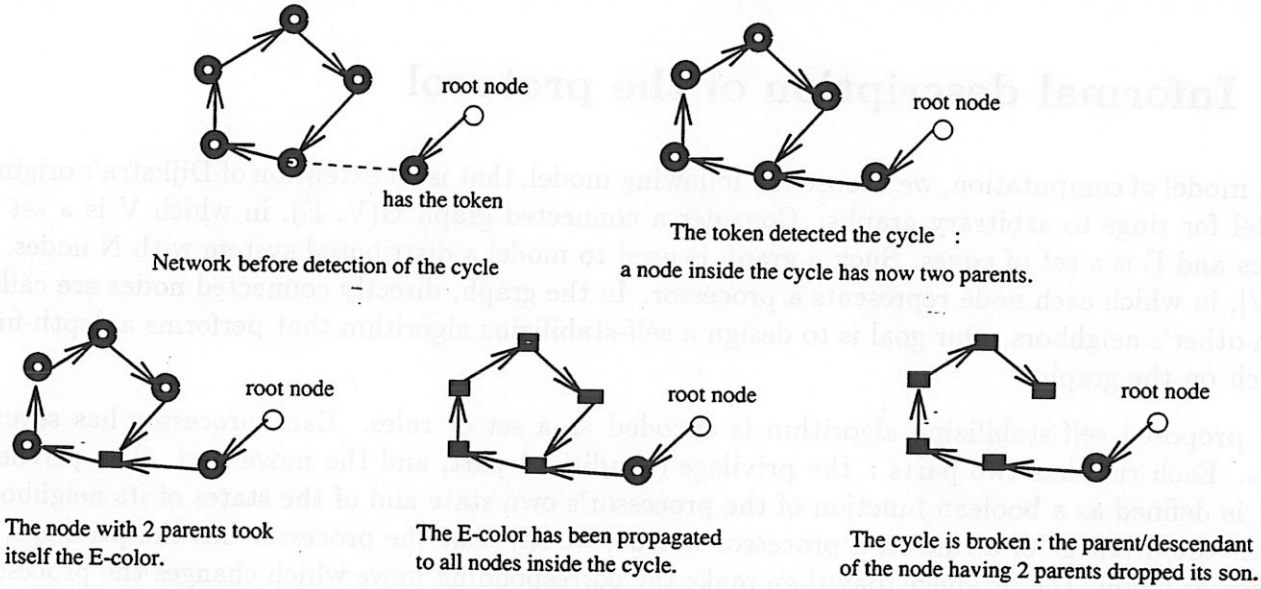

Figure 1: Destruction of a cycle

strategy is completely different from the one used in [10]. Our solution does not use a variable level. The key point is the detection of cycles by outside nodes that will provoke the correction. The root initializes successive depth-first searches alternatively colored 0 and 1. Note that, due to a bad initialization, such a depth-first search can only be partial.
During a 0-colored round, all nodes inside the branch are 0-colored. If during a 0-colored round, the leaf has an 1-colored neighbor which is inside a cycle, there must be an error somewhere. Then the leaf chooses the faulty node as son (figure 1). The faulty node detects that it has two parents, and then colors itself E. The E color propagates to the descendant-parent of the faulty node. At this point, this node can drop its son (the faulty node) and break the cycle.
Obviously, the same holds if a node inside a cycle during an 1-colored round is 0-colored.

The nodes inside a cycle can change their color only to become E-colored (by a R8 move). This move can be performed at most once on a node inside a cycle. Thus, such nodes stop changing color. We can also prove that the cycles are eventually destroyed.

3 Formal model

Let S be a system defined by a set of states and a set of transitions where each transition is an ordered pair of states.

A *computation* is a sequence of system states (s_1, s_2, ... , s_n, ...) where each couple (s_i, s_{i+1}) is a system transition.
A system state is defined by the local variable values of each node. If the simultaneous moves of several rules modify the system state from s_1 to s_2 then (s_1, s_2) is a system transition in the case where (i) at most one move by a node is performed; (ii) and in s_1 the rule privileges are satisfied on the nodes which perform the corresponding rule moves.

A *region* of a system is a subset of system states. A region *REG* is *closed* if for every transition (s_1, s_2) where s_1 in *REG* then s_2 is in *REG*.
A computation *C leads* to a region *REG*, if *C* has a state in *REG*.
A region *REG* is an *attractor of a computation* (s_1, s_2, ...) if there is an integer n such that for all i > n, s_i belongs to *REG*.
A region *REG* is an *attractor* if it is closed and all computations lead to *REG*.
A predicate P over system states defines the region *REG*(P) as the set of states where P is satisfied. Shorten, we said that P is closed (resp. attractor) if and only if *REG*(P) is closed (resp. an attractor).
A predicate P_n over node states is a *trap* if for any node i, the predicate "$P_n(i)$ = true" over system states is closed.

A *legitimate states set* verifies several properties [5] : (i) it is closed, (ii) in each legitimate state, one and only one node holds one privilege, (iii) each legitimate state is reachable from any other legitimate state, and (iv) each node has a legitimate state where it holds a privilege.

We call a system *self-stabilizing* if and only if regardless of the initial state and regardless of the computation, the system is guaranteed to reach the legitimate states set after a finite number of moves.

4 Protocol formal specification

Notation X.i is read X of i; notation X.Y.i : X of Y of i.
Each node i maintains the following variables :

- D.i : a pointer pointing to one of its neighbors (called i's son) or pointing to *NULL*.
- C.i : the color of node i taking value in the set {0, 1, E}.

The required space at each node can be evaluated. Under the hypothesis that the graph under consideration has a fixed upper bounded degree D, independent from the number N of nodes, the size of son variable is log(D); the color variable has also a fixed size (2 bits). Then, the space complexity of the algorithm at each node is O(log(D)).

Other used notations are :

- P.i : the set of i's parents.
- NB.i : the set of i's neighbors with r excluded.
- NP.i : the number of i's parents.

4.1 Token circulation rules

We define some predicates used in the definition of the token circulation rules :
Token.i holds if i is a live leaf and i's color is not the same as i's parent color. A live leaf is a leaf whose color is not E.
BToken.i holds if i's son is a live leaf whose color is the same as i's color.
Anomalous(i,k) holds if k has a parent and does not have the expected color for an inside node with respect to i (the expected color is either C.i if BToken.i or C.i+1 mod2 if Token.i).
Detached.i holds if i is a node without son and without parent.
PotentialFirstSon(i,k) holds if Token.i holds, there is no anomalous node with respect to i, and k is a potential first i's son (k is a detached node with the right color : i's color).
DeadEnd.i holds if Token.i holds, there is no anomalous node with respect to i, and it does not exist a potential first i's son.
PotentialNewSon(i,k) holds if BToken.i holds, there is no anomalous node with respect to i, and k is a potential new i's son. (k is a live detached node with the right color : different from i's color).
Backtrack.i holds if BToken.i holds, there is no anomalous node with respect to i, and it does not exist a potential new i's son.

We formally define the predicates :

- $\text{Token}.i = [((i = r) \wedge (\text{C}.i \neq \text{E}) \wedge (\text{D}.i = \text{NULL})) \vee ((i \neq r) \wedge (\text{D}.i = \text{NULL}) \wedge (\text{NP}.i = 1) \wedge (\text{C}.i \neq \text{E}) \wedge (\text{C.P}.i \neq \text{E}) \wedge (\text{C.P}.i \neq \text{C}.i))]$
- $\text{BToken}.i = [(\text{D}.i \neq \text{NULL}) \wedge (\text{D.D}.i = \text{NULL}) \wedge (\text{C.D}.i = \text{C}.i) \wedge (\text{C}.i \neq \text{E})]$
- $\text{Anomalous}(i,k) = [\exists\, k \in \text{NB}.i \mid (\text{NP}.k \geq 1) \wedge ((\text{Token}.i \wedge (\text{C}.k \neq \text{C}.i{+}1 \bmod 2)) \vee (\text{BToken}.i \wedge (\text{C}.k \neq \text{C}.i)))]$
- $\text{Detached}.i = [(\text{D}.i = \text{NULL}) \wedge (\text{NP}.i = 0)]$
- $\text{PotentialFirstSon}(i,k) = [\text{Token}.i \wedge (\forall\, \text{j} \in \text{NB}.i \mid \neg\text{Anomalous}(i,k)) \wedge (\exists\, k \in \text{NB}.i \mid \text{Detached}.k \wedge (\text{C}.k = \text{C}.i))]$
- $\text{DeadEnd}.i = [\text{Token}.i \wedge (\forall\, \text{j} \in \text{NB}.i \mid \neg\text{Anomalous}(i,k) \wedge \neg\text{PotentialFirstSon}(i,k))]$
- $\text{PotentialNewSon}(i,k) = [\text{BToken}.i \wedge (\forall\, \text{j} \in \text{NB}.i \mid \neg\text{Anomalous}(i,k)) \wedge (\exists\, k \in \text{NB}.i \mid \text{Detached}.k \wedge (\text{C}.k = \text{C}.i{+}1 \bmod 2))]$
- $\text{Backtrack}.i = [\text{BToken}.i \wedge (\forall\, \text{j} \in \text{NB}.i \mid \neg\text{Anomalous}(i,k) \wedge \neg\text{PotentialNewSon}(i,k))]$

On a node i, the token circulation rules are :

R0 : $\text{PotentialFirstSon}(i,k) \wedge (i = r) \rightarrow \text{C}.r = \text{C}.r{+}1 \bmod 2;\ \text{D}.r = k$
R1 : $\text{PotentialFirstSon}(i,k) \wedge (i \neq r) \rightarrow \text{C}.i = \text{C.P}.i;\ \text{D}.i = k$
R2 : $\text{DeadEnd}.i \wedge (i \neq r) \rightarrow \text{C}.i = \text{C.P}.i;\ \text{D}.i = \text{NULL}$

R3 : PotentialNewSon(i,k) $\rightarrow$ D.$i = k$

R4 : Backtrack.$i \rightarrow$ D.i = NULL

The rule R0 initiates a regular circulation round : the node r changes the round color and chooses a son that gains the token. By a R1 move, the token passes from the previous leaf to its new son (that is now the leaf). So, the branch lengthens. If the leaf cannot find a suitable son (a neighbor that had not been visited during the current round) the leaf drops its token, by a R2 move. A R3 move, substitutes a new leaf (a node that had not been visited during the current round) for the current one (that does not have the token); this new leaf gains the token. If the current leaf does have the token and a new suitable leaf cannot be found, the branch is shrunk by a R4 move. When the branch is completely destroyed (e.g. the round is over), the node r has the token, and can perform a R0 move.

Evaluation of any privilege necessitates two communications round : each node has to get the two local variable values from its neighbors. Then, each node can compute its number of parents and transmit this value to all its neighbors.

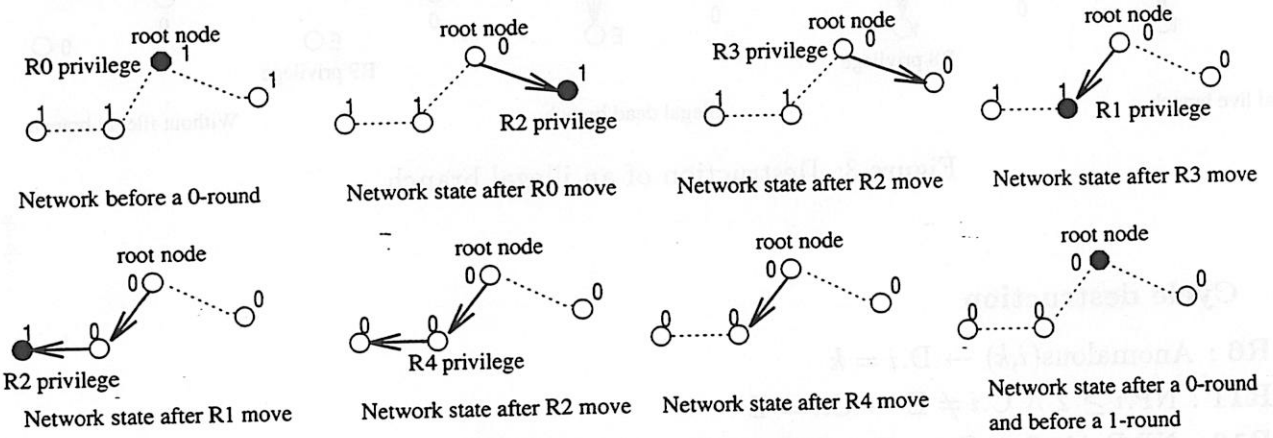

Figure 2: Token circulation

4.2 Error handling rules

A self-stabilizing system has an unpredictable initial state. In such a state, the D pointers point to any neighbors or NULL. Thus, illegal branches or cycles can exist in the initial state. The following rules delete illegal branches and transform cycles into branches. Thus, the system eventually reaches a legitimate state.

4.2.1 Illegal branch destruction

We define some predicates used in the definition of the illegal branch destruction rules :

- FBToken.i = [(D.$i \neq$ NULL) $\wedge$ (D.D.i = NULL) $\wedge$ (C.D.i = E)]
- IllegalRoot.i = [($i \neq r$) $\wedge$ (D.$i \neq$ NULL) $\wedge$ (NP.i = 0)]

FBToken.i holds if i's son is a dead leaf (an E-colored leaf).
IllegalRoot.i holds if i is a branch root without being the node r.

On a node i, the rules that destruct the illegal branches are :

R7 : FBToken.$i \rightarrow$ C.i = E; D.i = NULL
R8 : $\exists\ k \in$ NB.i | D.$k = i \wedge$ C.k = E $\wedge$ C.$i \neq$ E $\rightarrow$ C.i = E
R9 : Detached.$i \wedge$ C.i = E $\rightarrow$ C.i = 0
R10 : IllegalRoot.$i \wedge$ C.$i \neq$ E $\rightarrow$ C.i = E

The illegal branch destructions are processed as follows : R10 colors illegal roots E. R8 propagates the E color toward the leaf (the E color is propagated only from parent to son). R7 drops the E-colored leaf (the new leaf will have also the color E). R9 recovers the detached erroneous nodes.

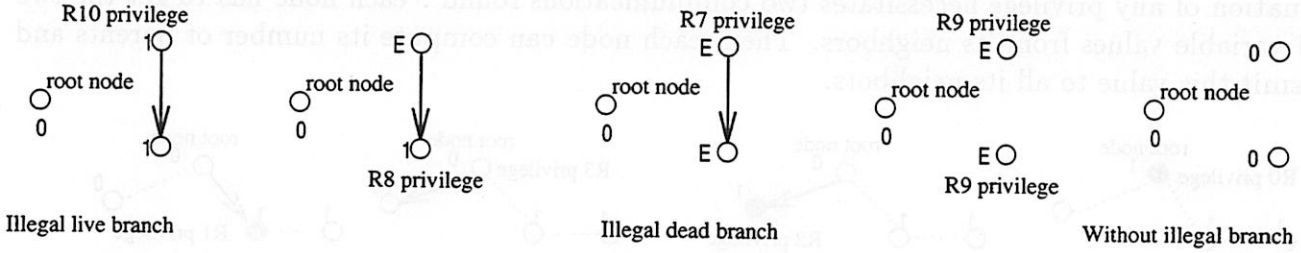

Figure 3: Destruction of an illegal branch

4.2.2 Cycle destruction

R6 : Anomalous(i,k) $\rightarrow$ D.$i = k$
R11 : NP.$i \geq 2 \wedge$ C.$i \neq$ E $\rightarrow$ C.i = E
R12 : NP.D.$i \geq 2 \wedge$ C.i = E $\wedge$ C.D.i = E $\rightarrow$ D.i = NULL

The cycle destructions are done as follows : R6 detects an anomalous node, and becomes its new parent (an anomalous node has a parent and does not have the expected color for a node having a parent). Now, R11 can color E the anomalous node (R11 colors E a node having several parents). The E color is propagated to the descendants of the anomalous node by R8 moves. Either the anomalous node is inside a branch (see above), or the anomalous node is inside a cycle. Then a R12 move on the parent/descendant of the anomalous node breaks the cycle (R12 disconnects an E-colored node with its son if its son has several parents and is also E-colored). After that, we have a branch whose leaf is E-colored.

4.2.3 Miscellaneous error handling

R5 : DeadEnd.$i \wedge (i = r) \rightarrow$ C.r = C.r+1 mod2; D.r = NULL
R13 : D.$i = r \rightarrow$ D.i = NULL

The rule R5 initiates a quick round (the only move is the r's color changing). R13 breaks the links parent-son with the node r (r should not have parent).

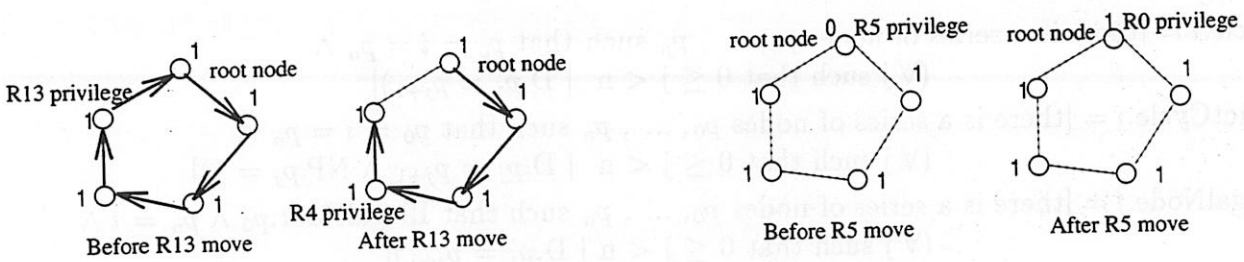

Figure 4: Miscellaneous error handling rules

5 Correctness of the protocol

We name *LS* the set of states where (i) only one node holds a privilege (ii) the satisfied privilege is R0, R1, R2, R3 or R4 (iii) there is no cycle and no illegal branch.

We will prove that *LS* is a valid legitimate states set, and that, in *LS*, the token circulates in depth-first order.

To prove the correctness of our algorithm, we use the *convergent stair* [8] method. We show that there is a sequence of predicates on the system states such that all computations lead to the regions defined by these predicates step by step (w.r.t. each region is an attractor, and each region is a subset of the previous one).

First, we prove that all computations are infinite. Then, we establish that there is a finite number of creations of illegal and live branches (e.g. branch whose root is r and whose leaf is not E-colored), in any computation. The fairness scheduling of the rules R10 and R8 provokes the dead of the illegal branches (e.g. their leaves get the E color). At this point, we show that no more node will join an illegal branch; and the illegal branches will eventually destroy themselves (by fairness scheduling of the rule R7). We prove that the legal branch will unavoidably become and stay sound (see the following predicate definition). We demonstrate that after the legal branch is sound, no more cycle is created; and that the cycles are eventually destroyed.

We conclude in showing that in *LS* the protocol provides a token circulation in depth-first order.

5.1 Predicate definitions

We define some predicates used in the correctness proofs :
Cycle.i holds if i belongs to a cycle : one of i's descendant is a i's parent.
StrictCycle.i holds if i belongs to a cycle and all nodes in this cycle have only one parent.
IllegalNode.i holds if i belongs to a branch whose root is not r.
IllegalLiveRoot.i holds if i is an illegal root whose branch ends in a dead leaf.
DeadLeaf.i holds if i is an erroneous leaf.
Unsound.i holds if i is an inside node (no leaf) of the legal branch that does have the same color as its parent and the legal branch ends in a live leaf. If a such node i exists, we said that the legal branch is unsound.

- Cycle.i = [there is a series of nodes p_0, ... , p_n such that $p_0 = i = p_n \wedge$
  $(\forall$ j such that $0 \leq$ j $<$ n $\mid$ D.$p_j = p_{j+1})$]
- StrictCycle.i = [there is a series of nodes p_0, ... , p_n such that $p_0 = i = p_n \wedge$
  $(\forall$ j such that $0 \leq$ j $<$ n $\mid$ D.$p_j = p_{j+1} \wedge$ NP.$p_j = 1)$]
- IllegalNode.i = [there is a series of nodes p_0, ... , p_n such that IllegalRoot.$p_0 \wedge p_n = i \wedge$
  $(\forall$ j such that $0 \leq$ j $<$ n $\mid$ D.$p_j = p_{j+1})$]
- IllegalLiveRoot.i = [IllegalRoot.$i \wedge$ there is a series of nodes p_0, ..., p_n such that
  $p_0 = i \wedge$ D.p_n = NULL $\wedge$ C.$p_n \neq$ E $\wedge (\forall$ j such that $0 \leq$ j $<$ n $\mid$ D.$p_j = p_{j+1})$]
- DeadLeaf.$i = \big[$(D.i = NULL) $\wedge$ (C.i = E) $\wedge \big($(i = r) $\vee$ (N.P.$i \geq 1$))$\big]$
- Unsound.i = [there is a series of nodes p_0, ... , p_n such that $p_0 = r \wedge$ D.p_n = NULL $\wedge$ C.$p_n \neq$ E
  $\wedge (\forall$ j such that $0 \leq$ j $<$ n $\mid$ D.$p_j = p_{j+1}) \wedge (\exists$ j $\mid 0 \leq$ j $<$ n $\wedge i = p_j) \wedge$ C.$i \neq$ C.P.i]

5.1.1 Algorithm Liveness

Theorem 1 *In any system state, at least one node holds a privilege.*

Proof : There are two kinds of configurations : either there is a leaf, or no. In these two configuration kinds, a move is possible. □

5.2 Destruction of illegal branches

5.2.1 Destruction of illegal and live branches

We present how all illegal and live branches are eventually destroyed, whatever computation is performed.

Lemma 1 *REG1 = { NP.r = 0 } is an attractor.*

Let us define CorrectLegalBranch as a boolean function of the system state. This function is true if there is a series of nodes p_0, ..., p_n such that

$p_0 = r \wedge (\forall$ i such that $0 \leq$ i $<$ n $\mid$ D.$p_i = p_{i+1}) \wedge \big($DeadLeaf.$p_n \vee (\exists$ i $\mid 0 \leq$ i $<$ n $\wedge p_i = p_n) \vee$
$\big($(NP.$p_n > 1 \vee$ D.p_n = NULL) $\wedge (\forall$ i such that $0 \leq$ i $<$ n $\mid$ C.$p_i \neq$ E)$\big)\big)$

CorrectLegalBranch is true when the branch whose root is r ends in a dead leaf or in a cycle, or when all nodes of this branch between the root and a suitable node are not E-colored (a node is suitable if it is a leaf or if it has several parents).

Let us define the number X as following (if CorrectLegalBranch is false then IncorrectLegalBranch = 1 otherwise IncorrectLegalBranch = 0) :

X = number of illegal and live roots + IncorrectLegalBranch

First, we will prove that X value decreases at each creation of an illegal live root; then we will establish that X value never increases. We will conclude that there is a finite number of illegal live root creations, in any computation. At this point, we will be able to prove that each computation is attracted by system states where there is no live and illegal branch.

Lemma 2 *In REG1, at each creation of an illegal and live root, X decreases.*

Proof : There is creation of an illegal root, only when all parents of one node perform a R12 move. This node becomes a new illegal root by losing all its parents. This node was inside a cycle, inside a dead branch, inside several illegal live branches, or inside only one illegal and live branch. In all cases, we prove that X decreases. □

Lemma 3 *In REG1, X never increases.*

Proof : There is only two cases where X increases : Either the number of illegal and live roots increases; thus a new illegal and live root has been created. The lemma 2 establishes that X does not increase in this case. Or, the legal branch reaches an incorrect state from a correct one. But in this case, the number of illegal live roots decreases. Thus X does not increase. □

Theorem 2 *REG2* $= REG1 \cap \{ \forall\, i\ \neg IllegalLiveRoot.i \}$ *is an attractor.*

Proof : X decreases at each creation of illegal and live root (lemma 2) and X never increases. At some point, there will be no more creation of illegal and live roots. Then, by fairness scheduling of the R8 and R10 moves, *REG2* will be reached. □

In *REG2* there is at most one live leaf (the leaf of the legal branch). There are some illegal branches but all of them are dead.

5.2.2 Destruction of illegal and dead branches

We present how all illegal and dead branches are eventually destroyed, whatever computation be performed.

Lemma 4 *REG3* $= REG2 \cap \{ X = 0 \}$ *is an attractor.*

Proof : *REG3* is closed (lemma 3). By fairness scheduling of the R8 moves, the legal branch will end in a dead leaf (X = 0). □

Lemma 5 *In REG3, the predicate* $\neg IllegalNode$ *is a trap.*

Proof : The illegal branches cannot be extended because they are dead. The only way that nodes become illegal is the creation of a new illegal branch whose root was not already an illegal node. This can only append when the legal branch is an incorrect state. □

Theorem 3 *REG4* $= REG3 \cap \{ \forall\, i\ \neg IllegalNode.i \}$ *is an attractor.*

Proof : By fairness scheduling of the R7 moves, the illegal branches will destroy themselves. □

5.3 Destruction of cycles

We show that in all computations, the cycles are eventually destroyed.

5.3.1 Soundness of the legal branch

We show that whatever computation be performed, it leads to system states where the legal branch is and stays sound (when the legal branch ends in a live leaf, all inside nodes of the legal branch have the same color).

Lemma 6 *In REG4, The predicate ¬Unsound is a trap.*

Proof : Any move does not change a sound node into an unsound one. □

Remark : $REG5 = REG4 \cap \{ \forall \text{ i} : \neg\text{Unsound.i} \}$ is closed.

In order to prove that any execution leads to *REG5*; we will prove that all computations have to lead to *REG5*.

Let C be a computation which does not reach *REG5*. Thus, there is a no-empty set of nodes which are unsound all along C. Let is name $\mathcal{N}_C$ this set. These nodes are and stay in the legal branch along C. We call $\mathcal{REG}4$ the subregion of *REG4* where all nodes of $\mathcal{N}_C$ are unsound and others are not.

In $\mathcal{REG}4$, R6 privilege holds only on the live leaf of the legal branch. As described in the proof of the lemma 6, after a R6 move on the legal leaf, the legal branch is sound. Therefore, C does not contain a R6 move.

Lemma 7 $\mathcal{REG}4a = \mathcal{REG}4 \cap \{ \forall\, i : NP.i \leq 1 \}$ *is an attractor of C.*

Proof : In $\mathcal{REG}4$, there is only one leaf. At a time, only one node can pick up a new son. Thus, any node cannot gain several parents, in one step. Only after a R6 move, a node having a parent get a second one. A R6 move is never performed by C; thus $\mathcal{REG}4$ is closed. By fairness scheduling of the rules R11, R8, or R12, $\mathcal{REG}4a$ will be reached by C. □

Lemma 8 $\mathcal{REG}4b = \mathcal{REG}4a \cap \{ \forall\, i : Cycle.i \vee C.i \neq E \}$ *is an attractor of C.*

Proof : By fairness scheduling of R7, R8 and R9 rules, $\mathcal{REG}4b$ will be reached in C. Any rule moves that can be performed in $\mathcal{REG}4b$ does not color E a node outside cycles. □

Remark : in $\mathcal{REG}4b$, C does not contain R6, R7, R9, R10, R11, R12, and R13 moves. in $\mathcal{REG}4b$, the move R8 is performed a finite number of times (at most one time on each node inside a cycle). After a R0, or R5 move the legal branch is sound. C contains only an infinity of R1, R2, R3, or R4 moves in $\mathcal{REG}4b$.

Let I_i be an integer function of system states defined as :

I_i = 4 x number of detached nodes of color different from C.i
\+ 3 x number of nodes in legal branch after i that have a son
\+ 2 x number of leaf whose color differs from C.i
\+ 1 x number of leaf whose color is C.i

Lemma 9 *Let* i *be the farthest node of* $\mathcal{N}_C$ *on the legal branch. In* $\mathcal{REG}4b$, *the* C *computation contains a R4 move on* i.

Proof : All nodes inside the legal branch after i have the same color as i, except the leaf. Until a R4 move on i, I_i is strictly decreased by R1, R2, R3, and other R4 moves. Assume that C does not contain a R4 move on i, the C computation would be finite, in contradiction with the theorem 1. □

After this R4 move, the farthest node of $\mathcal{N}_C$ is the leaf and is sound. Thus, there is a contradiction with the hypothesis *the nodes $\mathcal{N}_C$ are unsound all along C.* We conclude that all computations reach *REG5.* The following theorem is a consequence of the lemmas 6 and 9.

Theorem 4 *REG5 is an attractor.*

5.3.2 Destruction of strict cycles

We show that in all computations, the strict circles are eventually destroyed.

Lemma 10 *In REG5, the predicates ¬Cycle and ¬StrictCycle are traps.*

Remark : *REG6* = REG5 ∩ {∀ i : ¬StrictCycle.i} is closed (lemma 10).

In order to prove that any execution lead to *REG6*, we prove that it does not exist a computation not leading to *REG6.*

Let C be a computation which does not lead to *REG6.* Thus there is a no-empty set of nodes which are and stay inside a strict cycle along C; let us name $\mathcal{N}_C$ this set. We call $\mathcal{REG}5$ the subregion of *REG5* where all nodes of $\mathcal{N}_C$ are inside a cycle and others nodes are not inside a cycle. In *REG5*, after a R6(i,k) move, k which was previously inside a strict cycle, is no more within a strict cycle. Therefore, C cannot contain R6 move in $\mathcal{REG}5$.

The proof of the following lemma is similar to the proof of the lemmas 7 and 8.

Lemma 11 $\mathcal{REG}5b = \mathcal{REG}5 \cap \{ \forall\, i : NP.i \leq 1 \} \cap \{ \forall\, i : Cycle.i \vee C.i \neq E \}$ *is an attractor of C.*

Remark : C contains only an infinity of R0, R1, R2, R3, R4 moves in $\mathcal{REG}5b$.

Lemma 12 *In $\mathcal{REG}5b$, the C computation contains an infinity of R0 moves.*

Proof : Between two R4 moves on r, there is a R0 move. Assume that C contains a finite number of R0 moves on r. At some point, C does not contain R0 and R4 move on r. After that I_r is strictly decreased by all possible moves. Thus the C computation would be finite, in contradiction with the theorem 1. □

Let $\mathcal{D}_i$ be the the minimal distance between r and i defined as :

$$\mathcal{D}_i = \text{Min } \{\, n \in \mathcal{N} \mid \exists \text{ a node series } p_0, \ldots, p_n \text{ such that } p_0 = r \wedge p_n = i \wedge (\forall\, j \text{ such that } 0 \leq j < n \mid p_{j+1} \in \text{NB}.p_j) \,\}$$

Let $\mathcal{D}_C$ be the minimal distance between r and a node of $\mathcal{N}_C$. Formally, we define $\mathcal{D}_C$ as :

$$\mathcal{D}_C = \text{Min } \{\, n \in \mathcal{N} \mid \exists \text{ a node series } p_0, \ldots, p_n \text{ such that } p_0 = r \wedge \text{StrictCycle}.p_n \wedge (\forall\, j \text{ such that } 0 \leq j < n \mid \neg\text{StrictCycle}.p_j \wedge p_{j+1} \in \text{NB}.p_j) \,\}$$

Lemma 13 *If $\mathcal{D}_C > 1$, then in $\mathcal{REG}5b$, the C computation contains an infinity of R1 or R2 moves performed on each r's neighbor.*

Proof : After a R0 move, r cannot perform a new R0 move until all its neighbors have the same color as its own. (i) There are an infinity of R0 moves; (ii) the R0 moves are the only moves to change the color of r; (iii) and the only moves which change the r's neighbors color are R1 and R2 moves. □

Similarly, we prove the following lemma.

Lemma 14 *Let i be a node such that $\mathcal{D}_C > \mathcal{D}_i$ and such that the C computation contains an infinity of R1 or R2 moves performed by i. C contains also an infinity of R1 or R2 moves performed by each i's neighbor.*

Lemma 15 *There does not exist a computation which does not lead to REG6.*

Proof : Let i be a node such that $\mathcal{D}_C = D_i+1$ and such that i has a neighbor k verifying Strict-Cycle.k. In $\mathcal{REG}5b$, system states where Token.i is satisfied, are infinity often reached along C. At some point, k cannot change its color (the moves that are performed infinitely often along C cannot be performed on a node inside a cycle). After that, i will eventually performed a R6 move. □

The following theorem is a consequence of the lemmas 10 and 15.

Theorem 5 *REG6 is an attractor.*

5.3.3 Destruction of un-strict cycles

The strict cycle have been deleted; Thus, the is at most one (un-strict) cycle. Now, we establish that the last circle is eventually destroyed.

Theorem 6 *$REG7 = REG6 \cap \{ \forall\, i : \neg Cycle.i\}$ is an attractor.*

Proof : The lemma 10 establishes that the region *REG7* is closed. Let i be a node belonging to a cycle. i does not belong to a strict cycle. Thus, a node of its cycle holds the R11, R8 or R12 privilege. At each system state of *REG6*, only one node holds a privilege. At each step, the only enable move (R11, R8 or R12) is performed until the cycle is destroyed by the R12 move. □

5.4 Legitimate state set

The proof of the following theorem is similar to the proof of lemma 8.

Theorem 7 *$LS = REG7 \cap \{ \forall\, i : C.i \neq E\}$ is an attractor.*

In *LS*, (i) only one node has a privilege; (ii) only the R0, R1, R2, R3, or R4 moves are performed; and (iii) any node does not verify Cycle or IllegalNode predicates.
From any state of *LS*, we can reach the system state s_0 where all nodes are detached and have the color 0. It is quite obvious that from s_0, any state of *LS* can be reached.

The lemmas 12, 13, and 14 establish that each node i has several legitimate states where Token.i is true. In these states, i holds the R0, R1, or R2 privilege.

The privilege of R2, the rule that stops the branch growing is held if and only if the branch cannot lengthen. The privilege of R4, the rule that shrinks the branch, is held if and only if the branch cannot lengthen and cannot change its way (e.g. to change leaf). Thus, as long as it is possible, the current branch lengthens and the token goes further off r. In LS, the token circulation is done in a depth-first order.

We have proved that (i) *LS* is a valid legitimate state set; (ii) *LS* is an attractor; and (iii) in *LS*, our protocol provides a token circulating in the network in depth-first order.

References

[1] Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self-stabilization on general networks. In *Proc. 4th Int. Workshop on Distributed Algorithms*, volume 486, pages 15–28. Springer-Verlag, 1990.

[2] Geoffrey M. Brown, Mohamed G. Gouda, and Chuan lin Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38(6):845–852, 1989.

[3] James E. Burns and Jan Pachl. Uniform self-stabilizing rings. *ACM Trans. on Programming Languages and Systems*, 11(2):330–344, 1989.

[4] Nian-Shing Chen, Hwey-Pyng Yu, and Shing-Tsaan Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.

[5] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Com-munications of the A.C.M.*, 17(11):643–644, 1974.

[6] Edsger W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1:5–6, 1986.

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

[8] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

[9] Shing-Tsan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41:109–117, 1992.

[10] Shing-Tsan Huang and Nian-Shing Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7:61–66, 1993.

[11] H.S.M. Kruijer. Self-stabilizing (in spite of distributed control) in tree-structured systems. *Information Processing Letters*, 29:91–95, 1979.

[12] Sumit Sur and Pradip K. Srimani. A self-stabilizing distributed algorithm to construct bfs spanning trees on a symetric graph. *Parallel Processing Letters*, 2(2/3):171–179, 1992.

[13] Ming-Shin Tsai and Shing-Tsaan Huang. A self-stabilizing algorithm for the shortest paths problem with a fully distributed demon. *Parallel Processing Letters*, 4(1):65–72, 1994.

Paper Number 5

A Self-Stabilizing Distributed Heap Maintenance Protocol

Brian Bourgon and Ajoy Kumar Datta

A Self-Stabilizing Distributed Heap Maintenance Protocol *

Brian BOURGON and Ajoy Kumar DATTA
Department of Computer Science,
University of Nevada, Las Vegas

Abstract

This paper presents an $O(nh)$ self-stabilizing distributed heap maintenance protocol in a tree network, where h is the height and n is the number of nodes of the tree. The underlying paradigm of counter flushing due to Varghese is used to stabilize the synchronization between the nodes in the system. The heap maintenance problem is the problem of maintaining each node in the tree to have a larger value than any of its children. This heap is maintained on a general tree structure rather than the traditional balanced binary tree. The protocol presented handles all transient failures such as nodes entering and leaving the system, random corruption of variables, and message loss or corruption on the links. There is currently no known distributed protocol for this problem in the literature.

Keywords: broadcast-convergecast, counter flushing, distributed algorithms, heap maintenance, self-stabilization

1 Introduction

In recent work in the well-known area of self-stabilization [9, 14], there have been a number of paradigms designed for achieving self-stabilization [1, 2, 3, 4, 5, 6, 12, 15, 16]. Two paradigms in particular are of particular interest in solving those problems which require information to travel to a central leader and then be disbursed to all nodes in the network [3, 16]. The former due to Arora and Gouda is used to reset some distributed application protocol in the shared memory model, while the latter due to Varghese is used to disseminate the network information in a message passing system model. The problem of heap maintenance is such a problem in which information needs to be propagated throughout the network in a broadcast-convergecast manner. We have chosen to model the system for this protocol in the message passing environment, and are therefore using the counter flushing paradigm. The basic idea behind counter flushing is to synchronize the passing of information through the network by attaching a counter value in the range $0..Max$ to every message. Nodes in the network will only accept messages from parent nodes with counter values that differ from their own, and will only accept messages from children with the same counter values. New counter values can only be produced at the special root node. To simplify presentation, the I/O Automaton model of Lynch and Tuttle is used to model the system [13, 16]. In this model, each node can be represented by a single *automaton* that affects the rest of the system, the *environment*. In turn, the environment will affect each automaton. Three types of actions are required in this model: *input actions*, *output actions*, and *internal actions*. Input actions are those actions which have the environment affecting the automaton. Output actions are those which have the automaton affecting the environment. Internal

*Contact author: Ajoy Kumar Datta, Department of Computer Science, University of Nevada, Las Vegas, Las Vegas, NV 89154-4019, USA, Email: datta@cs.unlv.edu, Phone: (702) 895-0870, Fax: (702) 895-4075

actions are those which have the automaton affecting itself. The formal definition of an I/O automaton is a five tuple (S, A, G, R, I) where S is the set of states, A is the set of actions, G is an action signature (that classifies the action set into input, output, and internal actions), $R \subseteq S \times A \times S$ is the transition relation or set of valid state transitions, and $I \subseteq S$ is the set of initial states. In self-stabilizing algorithms, $I = S$ since no initialization is assumed. This model is called a UIOA or *uninitialized* I/O automaton. More details about the I/O automaton model can be seen in [13, 16]. Additionally, our system model assumes that links are initially bounded. In that we take a constant bound k to be the maximum number of messages in any given link when the system starts. This assumption is reasonable, since in practice all links must be bounded.

The problem of heap maintenance as defined in this paper is what is widely considered max-heap maintenance in the literature [8, 11]. This type of heap is one in which any node in the network has a larger value than its child node. This type of heap has several uses including efficient sorting. Additionally, the problem in the context of this paper will allow for a general tree network rather than the traditional balanced binary tree of the sequential heap maintenance algorithm. This is mainly due to the efficiency at which such a structure can be created on a general network topology. More formally, we define the problem to be:

Definition 1.1 *The distributed heap maintenance problem consists of the following instance and question:*

Instance: *A tree network $T = (V, E)$ with a set of nodes V such that all nodes know their parent (stored in $parent_i$) and their set of neighbors in the spanning tree (stored in N_i), and a set of edges E. Each node $i \in V$ has a hard coded value val_i.*

Problem: *To have each node $i \in V$ take a final value $final\_val_i$ with the following properties:*

(i) $\exists j \in V$ such that $final\_val_i = val_j$

(ii) $final\_val_i < final\_val_k$ such that $k \in V \wedge parent_i = k$

We are not aware of any existing solution of the problem of maintaining such a heap in a dynamic distributed environment. This paper presents a protocol that will first create a max-heap (in the sequel called simply a heap) in a tree structured network with no initialization of variables, and then maintain this heap in the presence of transient faults. It is assumed that any of a number of self-stabilizing spanning tree protocols is used as an initial wave in the protocol [3, 7, 10]. Such a protocol will stabilize the N_i sets and $parent_i$ variables. This assumption allows the algorithm to be assumed applicable to general network topologies. Each node has an internal value val_i which is assumed non-corruptible either by inclusion in non-volatile storage or by being hardware coded in each node. The root node will take the largest value in the system, and all of its children will have larger values than their children. In this same manner, all nodes will take larger values than their children.

The protocol presented here stabilizes in time proportional with the the number of nodes in the system times the height of the tree. Time complexity assumes the common presumptions for distributed computations: local computation is negligible, and sent messages are received in constant time.

The remainder of the paper is organized as follows. Section 2 covers the data structures and protocol for the heap maintenance problem, while Section 3 provides the correctness arguments of the given protocol, and Section 4 provides some summary information and conclusions concerning the given protocol.

2 The Protocol

The nodes are synchronized through the use of counters c_i at each node. A given node i will only accept a *Token* message from its parent if the counter passed is different from c_i. Likewise, node i will only accept a *Token* message from its child if the counter passed is the same as c_i. Since only the root r can create new counter values, once stabilized the protocol works in *waves*. A wave can be defined as the set of *Tokens* starting from the root with a specific counter value, broadcast down the tree to the leaves, and final broadcast back up the tree to the root. Each new correctly synchronized wave will begin with a *fresh counter*. A fresh counter is one that does not exist anywhere in the system, either at the nodes or in the links. If node r believes that nothing has changed in the tree during the last wave, then r initiates a *reset wave*. During a reset wave all nodes reset their local values to insure that only *true values* exist in the system. *True values* are those based on correct system information rather than the possibly false initial values that exist in the system due to random initialization of the protocol. In doing this, r initializes its *working final value* to its own hard coded value, val_r. The working final value, $sr\_final\_val_i$, is the variable where the heap is constructed at each node. This variable may change a number of times during each heap building process. Node r then passes the reset information on by sending the information to its children that nothing has changed in the tree. These nodes then reset themselves in the same manner; they set their working final value to their hard coded value val_i. This passes down the tree where at the leaves the working values are set the hard coded values and then a change in their own values is indicated. The leaves then pass along their_values with the fact that a change has occured. Internal nodes check to see if there has been any change in the subtree rooted at them. If there is a change, these nodes heapify by determining the new maximum value in that subtree. If the maximum came from one of its descendents, an exchange with that child the value was received from is made. In this way, the smaller values cascade to the bottom of the tree while larger values filter up to the top of the tree. Once all nodes have come to a point where they have greater values than all of their children, the heap is created. At this point a *false change status* will filter up the tree to the root who will initiate another reset. *Change status* refers to whether or not a change has been made in the subtree rooted at a particular node i. It is kept in the variable sr_change_i.

In order to keep the heap information around during resets and subsequent rebuilding of the heap, during the reset process the working final values are copied to an actual final value, $final\_val_i$ that is not touched except in a reset.

The reset wave is what allows the information to stabilize. When each node finds the reset status to be *true*, it resets its working final value to its actual hard coded value. In this way, all false final values are flushed out of the system. At the leaves, additionally the maximum value of their subtree (which consists of only themselves) is set to this reset working final value. As this information travels up the tree and the internal nodes heapify, the internal nodes change their maximum values to the reset working final value before comparing with descendent values. In this way, *all* false values are flushed out of the system. Due to the nature of self-stabilizing algorithms, this reset must repeat continuously. This is because there is no way of knowing if the current reset is rebuilding the correct heap or flushing out false values. Additionally, if new nodes enter the system, they will have to have their information added into the heap process. This can only occur if the heap is constantly being rebuilt.

Between resets, a series of waves work to filter the proper values to each node. At the end of the the reset wave, the root will have the highest value in the system as its final value. In the first wave following a reset, the children of the root will take the highest values in the subtrees rooted at them (excluding that value already taken by the root). Recursively, after each wave the next deeper set of nodes in the tree will take the maximum value occurring within the subtree rooted at themselves (excluding any values taken by nodes above them). In this way, the heap is built in a number of waves proportional to the height of the tree.

The variables used in this protocol can be seen in Figure 1. The functions used in this protocol as seen in Figure 2 work as follows: The function **CHOOSE**(Max, c) takes a value between 0 and Max to be the new counter value. This choosing can be done in three separate ways detailed in [16]. We will assume the *increment* instance of this function because it makes the proofs easier to follow, and does not adversely effect the overall stabilization time of the protocol.

The function **FINISHED** is a boolean function that determines whether or not a particular node has received a message back from all of its children in the current wave. Leaves always return *true* on this function since they have no children.

The function **COMPUTE_MAXIMUM** works by making the maximum value, max_val_i, at node i be its working final value. Then, i compares this maximum value with each value received from its children j, $r\_val\_c_i[j]$. If it finds any child's value to be greater than the current maximum value, it takes that value as its maximum and remembers which child the value came from in the max_c variable.

The **RESET** function first has the node i set its change status to *true* so that another reset does not occur prematurely. Then, i copies the working final value over to the actual final value in order to maintain the heap just created. The working final value is initialized to the hard coded value at node i. Finally, in leaf nodes only, the sr_change_i variable is set to *true* ending the reset wave.

The function **EXCHANGE** will prepare to send the current working value at node i to that child from whom the greatest value higher than this working final value came. Then, node i prepares to send back the values last received by all other children. Finally, node i takes the value it stored using the **COMPUTE_MAXIMUM** function and places it in the working final value.

In the function **HEAPIFY**, node i first determines whether or not to actually heapify the subtree rooted at it by checking the change status received from each of its children ($r\_change\_c_i[j]$). If the change status is *false* from all of its children, it prepares to pass the *false* change status to its parent. If, however, the change status is *true* from at least one child, node i will prepare to pass on the *true* change status to its parent and run the function **COMPUTE_MAXIMUM**. If no greater value came from the subtree, then node i prepares to send back the same values that came from all of its children. If indeed, a new maximum value was sent up the subtree, node i will run the function **EXCHANGE**.

Given these supporting functions, the actual actions at each node given in Figure 3 are as follows: **ROOT_START**$_r$ is only used by the root and upon receiving all expected messages from its children (as determined by running the **FINISHED** function), node r will check to see if any change has been made in the tree. If no change has occured, a reset wave is initiated in r by setting r's reset status to *true* and calling the function **RESET**. A new counter value is created by using the **CHOOSE**(Max, c) function. Finally, r will set itself to expect new values from its children thus starting a new wave.

The **SEND**$_{i,j}$ action will set the counter value to be sent from a node i to be the actual counter value

c_i at that node i. Then, there are two cases. One case for sending to a child, and one case for sending to a parent. In the first case, node i will simply send that value either last sent by the child in question, or the last working final value for node i. This is determined by whether or not the last **HEAPIFY** ran the **EXCHANGE** function or not. The most recent change status as well as the current reset status are sent. In the second case, internal nodes will merely send up the change status along with the maximum value from the subtree of i and the current reset status. If, however, the node is a leaf, then it has to additionally set the change status to *false* if it was *true* since after sending the *true* value the leaves will remain as they are. If this is a reset wave, the leaf will set its reset status to $false$, and reset its maximum value to its own hard coded value. If it is not a reset wave, then the leaf will set its max value to its working final value.

The $\mathbf{RECEIVE}_{j,i}$ action also has two cases: for receiving from a parent, and for receiving from a child. In the first case, the node will only take the information if the counter value is different. If this is so, then the node i will take the value, change status, reset status, and counter value from the $Token$. The node will then set itself to expect new information from all of its children propagating the current wave. If the current wave is a reset wave then node i will reset using the **RESET** function. If it is not a reset wave, then node i will check to see if it is receiving back the same value or some new smaller value. If it is the same value, node i will indicate no change has been made in its subtree. If it is a new value, then that value is accepted, and a change is indicated. In the second case, if the counter value is the same as c_i, node i will check to see if the received message is expected. If not, i will discard the message so as to avoid receiving new information before the next wave breaking synchrony. If the message is expected, i will set itself to no longer expect a value from its child since it is now receiving it. Then, it sets its received value from that child as well as the status of its child's subtree. Finally, i will take its child's reset status and if i has received information from all of its children, then it will run the **HEAPIFY** function.

3 Correctness Reasoning

Lemma 3.1 *Any counter value c sent by the root node will reach and be accepted by all nodes in the tree within $O(h)$ time, where h is the height of the tree.*

Proof: By induction on the distance from the root. We will consider the distance from the root to be the number of links between a particular node i and the root node r.

Basis: Distance of zero means that i is the root node r. It is trivial to see that the root has the value c since that value is created at the root node using the $\mathbf{CHOOSE}(Max, c)$ function.

Induction: The induction hypothesis is that all nodes at a distance $\delta - 1$ from the root will have received and accepted the value c. It must be shown that all nodes at a distance δ receive the counter value c. All nodes at distance $\delta - 1$ will send their current counter value c to their children. These children will accept these $Tokens$ carrying the counter value c only if their own counter value c_i is different. However, if a child does not accept the $Token$ message, then $c_i = c$. If a child does accept the $Token$, it will immediately set its counter value c_i to be c. Therefore, since all nodes at distance δ from the root must be children of nodes at distance $\delta - 1$ by definition, all nodes at distance δ will receive and accept the counter value c. Since the counter value is sent and received in constant time, all nodes will have received the new counter

value in time proportional to the height of the tree, $O(h)$. □

Lemma 3.2 *A new counter will be created at the root within $O(h)$ time from any arbitrary initial state.*

Proof: A new counter will be produced at the root node r by the **CHOOSE**(Max, c) function whenever **FINISHED** is *true* at r. This yields two cases.

Case 1: The root is initialized with **FINISHED** *true*. In this case, a new counter is produced in $O(1)$ time.

Case 2: The root is expecting at least one message from a child and thus **FINISHED** is $false$ at r. Thus, the root will continually send $Tokens$ with some old value $c = c_{old}$. By Lemma 3.1 $c = c_{old}$ will have reached all nodes in the network within $O(h)$ time. Once a leaf node receives the value $c = c_{old}$ it will begin sending $Tokens$ to its parent with the same counter value. All parents will accept these values since they will also hold the counter value $c = c_{old}$ by Lemma 3.1. In the same way all of these nodes will send to their parents and by induction on the maximum distance of a node from a leaf, all nodes up to the root will receive $Tokens$ with the value $c = c_{old}$. Once the root has received $Tokens$ with value $c = c_{old}$ from all of its children, **FINISHED** is *true* at r. Clearly, the $Tokens$ travel up the tree in the same time as they travel down the tree, $O(h)$. Therefore, the new counter is created in $O(h)$ time from any arbitrary state of the network. □

Lemma 3.3 *A fresh counter will be produced at the root within $O(nh)$ time from any arbitrary initial state if the increment version of* **CHOOSE**(Max, c) *is used.*

Proof: A new counter is produced at the root every $O(h)$ time by Lemma 3.2. Since links are initially bounded, at most k counter values exist in any link, where k is the constant bound on any link. Given n nodes in the network, at most n values exist at the nodes. There will be at most $n - 1$ edges in the tree. Therefore, there will be at most $c_{max} = k(n-1) + n$ values in the network when started. If the range of possible values for counters is set to be $0..Max$ with $Max > c_{max}$, then we can say that there exists a value c' that is not yet in the system. Therefore, at most c_{max} new counter values can be created at the root using the increment function before c' is created at the root. Since c' was not previously in the system it is a fresh counter by definition. Since a new counter is produced in $O(h)$ time by Lemma 3.2, and at most c_{max} new counters will be created before a fresh counter is created, $c_{max} \times O(h)$ time is needed to create a fresh counter. Since $c_{max} = k(n-1) + n$ a fresh counter is produced at the root in $O(nh)$ time. It is assumed that the constant $k << n$ and therefore is ignored in the overall complexity. □

Lemma 3.4 *The root will initiate a reset wave within $O(h^2)$ time from the time a fresh counter is produced.*

Once a fresh counter c_f is produced, all nodes in the system will be participating in the same wave. This is because no node will have the value c_f by definition, and all nodes in the system will receive the value c_f by Lemma 3.1. Thus, the nodes can only receive the value c_f by accepting tokens from their parents containing the value c_f.

A reset will occur only when the root believes that there is no change in the tree (i.e. when sr_change_r is $false$). This can happen in two ways. One case is when the root is initialized to believe this, the other

is when all of its children believe that the change status is $false$ and they forward this information to r so that r sets sr_change_r to $false$.

Case 1: If the root is initialized to believe that there is no change in the tree, then it will initialize a reset in constant time.

Case 2: If at least one child i of r believes a change has occured, then it will propagate downward the $Token$ with counter value c_f to a leaf node. Once the leaf node receives this, it will send back its working final value as its maximum to its parent. There are two sub-cases for every parent j as information propagates back up the tree. Either node j is told that its sub-tree has not changed or that it has.

Sub-case 1: If node j is told by a child that the sub-tree has not changed, then j will simply propagate the same information back up the tree as it previously did. Also, the indication of no change is forwarded to j's parent.

Sub-case 2: If node j is told by a child that the sub-tree has changed, j will determine if the change has created a new maximum value. If a new maximum was created, j takes that maximum, and forwards it to its parent, otherwise it sends up its previous maximum. Also, the indication of a change is forwarded up to j's parent.

We can now proceed with an inductive argument based on the maximum distance a node is from a leaf.

Basis: Leaf nodes will have no change immediately after sending their working final value up to their parents. A leaf will send this no change indication up to its parent during the next wave.

Induction: The induction hypothesis says that all nodes at most $\delta - 1$ links from a leaf will have taken no change in δ waves. Thus, these nodes will send this indication to their parent along with the same information as was previously sent. Thus, this parent will find no change when the **HEAPIFY** function is run and send that change status to its parent in the following wave.

Clearly, the root will find no change in $O(h)$ waves. Each wave will take $O(h)$ time by Lemma 3.2 above. Thus, the total time for the no change to be decided is $O(h^2)$. □

Lemma 3.5 *A reset wave will always have all nodes i set all their variables except for the $final_i$ variable to true system values.*

Proof: Starting at the root ($i = r$), node i will run the **RESET** function. This function does the following: $r\_change\_c_i[k]$ for all children k of i is set to $true$. The working final value will be set to the hard coded value val_i. At the leaves, sr_change_i will be set to $true$ indicating a change in its sub-tree (i.e. itself). As the reset propagates down the tree, the $token\_expected$ arrays are set to $true$.

Now, we need to use an inductive argument based on the maximum distance from a leaf node.

Basis: Once a leaf node resets, it will set its $reset\_status_i$ to $false$ and its max_val_i to be its hard coded val_i. This information is then sent to its parent. The arrays $r\_val\_c_i$ and $s\_val\_c_i$ and the variable max_c_i have no meaning. So, without the loss of generality it can be assumed that these variables do not exist in leaf nodes.

Induction: The induction hypothesis is that all nodes at a distance at most $\delta - 1$ from a leaf node have taken true system values for all of their variables. Thus, when a node i at level δ receives $Tokens$ from its children, the following will happen. The $r\_val\_c_i$ array will take the true system values sent from its children. Node i will also take $reset\_status_i$ to be $false$. Additionally, node i will **HEAPIFY** computing

a new maximum value since its child will indicate a change in the tree. In doing this, i will first set its max_val_i to be its already reset working final value. If necessary, i will take its maximum to be a reset r_val from one of its children. Either i will set the values in the $s\_val\_c_i$ array to be the reset r_vals sent to i, or the reset max_val_i in i. In either case, the values are true. If a new maximum is taken at i, it will set its max_c variable to be that child it received the new maximum from. If not, max_c will not be used and therefore will of no consequence if it is not reset.

Thus, all variables are reset to true values in the system except for the actual final values. □

Lemma 3.6 *The maximum value in the sub-tree rooted at a node i will reach i within $O(h)$ time following a reset.*

By induction based on maximum distance from a leaf.

Basis: The leaf will receive the maximum value in its sub-tree, since it will be reset to taking its hard coded value as its working final value and its maximum value as shown in Lemma 3.5. Additionally, a leaf is the only node in its sub-tree.

Induction: The induction hypothesis is that all nodes at distance $\delta - 1$ will have the maximum value in their sub-tree. The nodes at level $\delta - 1$ will send these maximum values to their parents. The parents upon receiving these values will compare them with their current maximum values to see if any of them are greater, if they are, the parent will take that value and send its smaller value back down the tree to the node that the maximum value came from. Since a node at maximum distance δ from a leaf will only have nodes with their sub-tree's maximum as children, by the induction hypothesis, these nodes indeed take the maximum value in their sub-tree. It has been established in Lemma 3.3 that information travels up the tree in $O(h)$ time. □

Lemma 3.7 *The correct heap will be created at the working final values within $O(h^2)$ time from the initiation of a reset wave.*

Proof: By induction on distance from the root.

Basis: By Lemma 3.5, all values are reset to true values, and if a new maximum is found in a node i's sub-tree, node i will take that value by Lemma 3.6. In this way, the root will take the largest value in the tree by the end of the reset wave. This is r's proper value in the heap.

Induction: The induction hypothesis is that all nodes at a distance $\delta - 1$ from the root have taken their proper value in the heap. By Lemma 3.6, the maximum value in i's subtree will reach i. Since all nodes at a distance $\delta - 1$ from the root are okay by the induction hypothesis, no exchange will occur.

This inductive process works such that at each wave, one greater distance from the root will take their proper value in the heap. Thus, the time complexity of $O(h^2)$. □

Lemma 3.8 *Once the correct heap is created in the working final values at the nodes, the correct heap will be copied to and remain in the final value variables barring any perturbations to the network.*

Proof: Once the heap is created at the working final values at the nodes, no more exchanges are made and a reset occurs. During the reset process, the correct heap values are copied from the working final values to the actual final values at each node. The only time the actual final values are touched is during

the reset process. By lemmas 3.5 and 3.7 the correct heap will again be created before the next reset. Thus, the proper heap will again be copied into the final values. □

Theorem 3.1 *The protocol given in Section 2 is a correct distributed heap maintenance protocol for tree structured networks with $O(nh)$ stabilization time.*

Proof: Synchronization is achieved in $O(nh)$ by Lemma 3.1. Once synchronization is achieved through the first fresh counter, a reset will occur within $O(h^2)$ time by Lemma 3.4. The reset initializes the system to contain only true system values by Lemma 3.5. After this initialization, another $O(h^2)$ time is needed to build the heap in the working final values at each node by Lemma 3.7. Finally, during the second reset the actual final values take the proper heap values and these values hold by Lemma 3.8. This last step only takes $O(h)$ time since it is only one wave in length. Thus, the algorithm is correct and stabilizes in $O(nh) + O(h^2) + O(h^2) + O(h) = O(nh)$ time, since $h \leq n$. □

4 Conclusions

This paper provides the first distributed heap maintenance protocol on a tree network. It should be noted that this protocol works on a general tree structure rather than the traditional balanced binary tree. Additionally, the protocol presented is self-stabilizing in nature which guarantees that the protocol handles transient faults. In doing so, no human intervention is ever required to insure eventual recovery from such errors as machines leaving and entering the system, random variable corruption at nodes, message loss, message corruption, message reorder, and loss of synchronization. The stabilization is achieved due to an underlying use of the counter flushing paradigm of Varghese [16]. This algorithm runs in $O(nh)$ time.

References

[1] Y. Afek, S. Kutten, and M. Yung, "Memory-Efficient Self-Stabilizing Protocols for General Networks," *Proceedings of the 4th International Workshop on Distributed Algorithms*, Bari, Italy, September 1990, pp. 15-28.

[2] A. Arora and M. Gouda, "Closure and Convergence: A Foundation for Fault-Tolerant Computing," *IEEE Transactions on Software Engineering*, Vol. 19, No. 11, November 1993, pp. 1015-1027.

[3] A. Arora and M. Gouda, "Distributed Reset," *Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science*, Bangalore, India, December, 1990, pp. 316-331; also to appear in *IEEE Transactions on Computers.*

[4] A. Arora, M. Gouda, and G. Varghese, "Constraint Satisfaction as a Basis for Designing Nonmasking Fault-Tolerance," *Proceedings of the 14th International Conference on Distributed Computing Systems*, Poznan, Poland, June 1994, pp. 424-431; also to appear in *J. High Speed Networks.*

[5] B. Awerbach, B. Patt-Shamir, and G. Varghese, "Self-Stabilization by Local Checking and Correction," *Proceedings of the 32nd Symposium on Foundations of Computer Science*, San Juan, Puerto Rico, October 1991, pp. 268-277.

[6] B. Awerbach and G. Varghese, "Distributed Program Checking: a Paradigm for Building Self-Stabilizing Distributed Protocols," *Proceedings of the 32nd Symposium on Foundations of Computer Science*, San Juan, Puerto Rico, October 1991, pp. 258-267.

[7] N. Chen, H. Yu, and S. Huang, "A Self-Stabilizing Algorithm for Constructing Spanning Trees," *Information Processing Letters*, Vol. 39, 1991, pp. 147-151.

[8] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 1990.

[9] E. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Communications of the ACM 17*, pp. 643-644, 1974.

[10] S. Dolev, A. Israeli, and S. Moran, "Self-Stabilization of Dynamic Systems Assuming only Read/Write Atomicity," *9th Annual ACM Symposium on Principles of Distributed Computing*, Quebec City, Canada, 1990, pp. 103-117; also *Distributed Computing*, 1993.

[11] E. Horowitz, S. Sahni, and S. Anderson-Freed, *Fundamentals of Data Structures in C*, Computer Science Press, New York, 1993.

[12] S. Katz and K. Perry, "Self-Stabilizing Extensions for Message-passing Systems," *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, Quebec City, Quebec, August 1990, pp. 91-101; also *Distributed Computing*, 1993.

[13] N. Lynch and M. Tuttle, "An introduction to input/output automata," *CWI Quarterly*, Vol. 2, No. 3, 1989, pp. 219-246.

[14] M. Schneider, "Self-Stabilization," *ACM Computing Surveys*, Vol. 25, No. 1, March 1993, pp. 45-67.

[15] G. Varghese, "Self-stabilization by local checking and correction," Ph.D. thesis, MIT, 1992.

[16] G. Varghese, "Self-Stabilization by Counter Flushing," *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, Los Angeles, California, August 1994.

The state of each node i consists of:

N_i	the set of neighbors of the node i in the spanning tree.
$parent_i$	the id of the parent node of i
c_i	a counter
$token\_expected_i[j]$	a boolean flag for each child j of i
$reset\_status_i$	the status of whether or not the current wave is a reset wave
val_i	the hard coded value at the node
sr_val_i	a value being sent/received
max_val_i	the maximum *value* of the subtree rooted at i.
$final\_val_i$	the final $value_i$ after the heap is constructed
$sr\_final\_val_i$	the working final value used during heap construction
sr_change_i	the change status of the sub-tree rooted at node i
$r\_val\_c_i[j]$	the last value received from child j of i
$s\_val\_c_i[j]$	the next value to be sent to child j of i
$r\_change\_c_i[j]$	the change status of j's sub-tree as sent to i
max_c	the child of the node from which the maximum value was received

Figure 1. The variables and data structures used by the heap maintenance protocol.

A token message is encoded as a tuple $(Token, c, sr\_val, sr\_change, reset\_status)$ where the variables $Token, c, sr\_val$, and sr_change contain the values of a node being sent/received.

FINISHED (* boolean function; set to *true* when not expecting tokens from any children *)
Return *true* if for all children k of i: $token\_expected_i[k] = false$
Return *true* if i is a leaf node

COMPUTE_MAXIMUM (* computes maximum value of the tree rooted at i *)
$max\_val_i = sr\_final\_val_i$
For all children k of i
If $r\_val\_c_i[k] > max\_val_i$
$max\_val_i = r\_val\_c_i[k]$; $max\_c = k$

RESET (* *)
For all children k of i : $r\_change\_c_i[k] = true$
$final\_val_i = sr\_final\_val_i$
$sr\_final\_val_i = val_i$
If ($|N_i| = 1$ and $i \neq r$) (* i is a leaf node *)
$sr\_change_i = true$

EXCHANGE (* exchanges $sr\_final\_val_i$ with the largest value received from the children *)
$s\_val\_c_i[max\_c] = sr\_final\_val_i$
For all children $k \neq max\_c$ of i : $s\_val\_c_i[k] = r\_val\_c_i[k]$
$sr\_final\_val_i = max\_val_i$

HEAPIFY (* *)
(* update max_val_i and sr_change_i *)
If ($r\_change\_c_i[k] = false$ for all children k of i)
$sr\_change_i = false$
Else (* at least one node in the tree of i changed *)
$sr\_change_i = true$
COMPUTE_MAXIMUM
If $max\_val_i = sr\_final\_val_i$ (* no exchange of values *)
For all children k of i : $s\_val\_c_i[k] = r\_val\_c_i[k]$
Else EXCHANGE (* exchange $sr\_final\_val_i$ with the largest value received from the children *)

Figure 2. The supporting functions used by the heap maintenance protocol.

ROOT_START$_r$ (* Leader starts a new cycle of broadcasting values. *)
Preconditions:
FINISHED
Effects:
(* if there is *no change* of values in the tree then broadcast *no change*
and reset various variables *)
If $sr\_change_r = false$ (* initiate a reset process *)
$reset\_status_r = true$; RESET
c_r =CHOOSE(Max, c) (* choose new counter value *)
For all children of k of r : $token\_expected_r[k] = true$

SEND$_{i,j}$($Token, c, sr\_val, sr\_change, reset\_status$) (* Node i sends token to node j *)
Preconditions:
$c = c_i$ (* counter of token matches node counter *)
If $j \neq parent_i$ (* j is a child of i *) (* send values equal to stored values *)
$sr\_val = s\_val\_c_i[j]$
$sr\_change = sr\_change_i$; $reset\_status = reset\_status_i$
Else If ($j = parent_i$ and FINISHED) (* j is the parent of i *)
(* send value equal to the current maximum values *)
$sr\_change = sr\_change_i$
If ($|N_i| = 1$ and $i \neq r$) (* i is a leaf node *)
If sr_change_i
$sr\_change_i = false$
If $reset\_status_i$
$reset\_status_i = false$; $max\_val_i = val_i$
Else $max\_val_i = sr\_final\_val_i$
$sr\_val = max\_val_i$; $reset\_status = reset\_status_i$

RECEIVE$_{j,i}$($Token, c, sr\_val, sr\_change, reset\_status$) (* Node i receives token from node j *)
Effects:
If ($j = parent_i$ and $c \neq c_i$) (* new counter from parent *)
(* set stored values equal to values in token message *)
$sr\_val_i = sr\_val$; $sr\_change_i = sr\_change$; $reset\_status_i = reset\_status$
$c_i = c$ (* set local counter equal to counter in token message *)
For all children k of i : $token\_expected_i[k] = true$
If $reset\_status_i$ (* root initiated a reset process *)
RESET
Else If $sr\_final\_val_i = sr\_val$
$sr\_change_i = false$
Else $sr\_final\_val_i = sr\_val_i$; $sr\_change_i = true$
Else If ($j \neq parent_i$ and $c = c_i$)
If $token\_expected_i[j]$
$token\_expected_i[j] = false$; $r\_val\_c_i[j] = sr\_val$
$r\_change\_c_i[j] = sr\_change$; $reset\_status_i = reset\_status$
If FINISHED
HEAPIFY

Figure 3. The heap maintenance protocol.

Paper Number 6

Asynchronous Fault-Tolerant One-Dimensional Cellular Automata

Peter Gacs

Asynchronous fault-tolerant one-dimensional cellular automata (Abstract)

Peter Gács*
Boston University

Abstract

In a probabilistic cellular automaton in which all local transitions have positive probability, the problem of keeping a bit of information for more than a constant number of steps is nontrivial, even in an infinite automaton. Still, there is a solution in 2 dimensions, and this solution can be used to construct a simple 3-dimensional discrete-time universal fault-tolerant cellular automaton. This technique does not help much to solve the following problems: remembering a bit of information in 1 dimension; computing in dimensions lower than 3; computing in any dimension with non-synchronized transitions.

Our more complex technique organizes the cells in blocks that perform a reliable simulation of a second (generalized) cellular automaton. The cells of the latter automaton are also organized in blocks, simulating even more reliably a third automaton, etc. Since all this (a possibly infinite hierarchy) is organized in "software", it must be under repair all the time from damage caused by errors. A large part of the problem is essentially self-stabilization recovering from a mess of arbitrary-size and content caused by the faults.

The present paper outlines the construction of an asynchronous one-dimensional fault-tolerant cellular automaton. It is more modular than our earlier papers on the subject and its construction is more amenable to the addition of further features like self-organization.

1 Introduction

Fault-tolerant computation and information storage in cellular automata is a natural and challenging mathematical problem but there are also some arguments indicating an eventual practical significance of the subject since there are avantages in uniform structure for parallel computers.

Since large groups of errors can destroy large parts of any kind of structure, self-stabilization techniques are needed in conjunction with traditional error-correction.

1.1 Previous work

The problem of reliable computation with an unbounded number of unreliable components and constant error probability was addressed by John von Neumann in [15] in the context of Boolean circuits. Von Neumann's solution, as well as its improved versions in [3] and [13], relies on high connectivity and non-uniform constructs.

Of particular interest to us are those probabilistic cellular automata in which all local transition probabilities are positive, since such an automaton is obtained by way of "perturbation" from a deterministic cellular automaton. The automaton may have e.g. two distinguished initial configurations: say X_0 in which all cells have state 0 and X_1 in which all have state 1. Let $p_i(x,t)$ be the probability that, starting from initial configuration X_i, the state of cell x is i. If $p_i(x,t) > 2/3$ for all x,t then we can say that our automaton remembers the initial configuration forever.

Informally speaking, a probabilistic cellular automaton is called **ergodic** if it eventually forgets all information about its initial configuration. (Ergodicity implies that this information will be forgotten but it means more: we pospone its technical definition.) Finite cellular automata with all positive transition probabilities are always ergodic. In the example above, one can define a "relaxation time" as the time by which the probability decreases below

*Email: gacs@cs.bu.edu, Fax: (617) 353-6457, Phone: (617) 353-2015. This research was supported in part by NSF grant CCR-9204284.

2/3. If an infinite automaton is ergodic then the relaxation time of the corresponding finite automaton is bounded independently of size. A minimal requirement of fault-tolerance is therefore that the infinite automaton be nonergodic.

The difficulty in constructing nonergodic one-dimensional cellular automata is that due to the positive local transition probabilities eventually large islands will randomly occur. We can try to design the a transition rule that (except for a small error probability) attempts to decrease these islands. It is a natural idea that the rule should replace the state of each cell, at each transition time, with the majority of the cell states in some neighborhood. However, majority voting among the five nearest neighbors (including the cell itself) seems to lead to an ergodic rule if the "failure" probabilities are not symmetric with respect to the interchange of 0's and 1's, and has not been proved to be nonergodic even in the symmetric case. Perturbations of the one-dimensional majority voting rule were actually shown to be ergodic in [8] and [9].

Nonergodic cellular automata with positive transition probabilities, for dimensions 2 and higher were constructed in [14]. The paper [7] applies Toom's work to design a simple three-dimensional fault-tolerant cellular automaton that simulates arbitrary one-dimensional arrays. The original proof was simplified and adapted to strengthen these results in [1].

A simple one-dimensional deterministic cellular automaton eliminating finite islands in the absence of failures was defined in [6]. See also [2]. It is now believed that perturbation makes this automaton ergodic. A similar rule that is also in some sense monotonic was proposed lately by Andrei Toom.

The paper [4] constructs a nonergodic one-dimensional cellular automaton working in discrete time, using some ideas from the very informal paper [11] of Kurdyumov. These constructions rely on an infinite hierarchy of simulations. The paper [5] constructs a two-dimensional fault-tolerant cellular automaton. In the two-dimensional work, the space requirement of the reliable implementation of a computation is only a constant times greater than that of the original version. (The time requirement increases by a logarithmic factor.)

Asynchrony In the three-dimensional fault-tolerant cellular automaton of [7], the components work in discrete time (with a global clock) and switch simultaneously to their next state. This requirement is unrealistic for arbitrarily large arrays. The natural model for asynchronous probabilistic cellular automata is that of a continuous-time Markov process. This is a much stronger assumption than allowing an adversary scheduler but it still leaves a lot of technical problems to be solved.

The paper [1] gives a simple method to implement arbitrary computations on asynchronous machines with otherwise perfectly reliable components. A two-dimensional asynchronous fault-tolerant cellular automaton was constructed in [16].

The present paper constructs a one-dimensional asynchronous fault-tolerant cellular automaton, thus completing the refutation of the so-called Positive Rates Conjecture in [12].

Proof method simplification A major point of interest in the one-and two-dimensional results is the technique of handling a large and complex construction and the proof of its correctness. Several methods have emerged that help the harnessing of complexity, but the following two are the most important.

A number of "interface" concepts is introduced (generalized simulation, generalized cellular automaton) helping to separate the levels of the infinite hierarchy, and making it possible to speak meaningfully of a single pair of adjacent levels.

Though the construction is large, we present its problems more-or-less one at a time and give a construction for solving each. E.g. the messiest part of the self-stabilization is the so-called Attribution Lemma, showing how after a while all cells can be attributed to some large organized group (colony). This lemma relies mainly on the Purge and Decay rules, and will be proved before introducing other major rules. It is not possible to ignore the other parts of the construction in this method, but we use the undefined parts only through "interface conditions" (specifications).

The new result and the new method of presentation will serve as a firm basis for other new results. One feature that will be easy to add can be called *self-organization.* This means that the system can be started from a simple initial configuration in which, essentially, the input of the computation is periodically repeated.

An other new problem likely to yield to the new framework is the *growth rate of the relaxation time* as a function of the size of a finite cellular automaton. At present, the relaxation time of all known cellular automata is either

bounded (ergodic case) or grows exponentially. We believe that our constructions will yield examples for other, intermediate growth rates as well.

2 Cellular automata

In this paper, we confine ourselves to one-dimensional cellular automata, mostly infinite ones.

2.1 Deterministic cellular automata

We define cellular automata in a slightly more general way than usual. The set **C** of **sites** is an additive subgroup of the set of all real numbers, or of the set of real numbers modulo a fixed number R. Usually, **C** is defined as the set of integers, but we will also use the case when **C** is the set of rational numbers, or the set of multiples of a real number B. In a space-time vector (x,t), we will always write the space coordinate first. A **configuration** is a function $\xi(x)$ assigning a state to each site from a finite set **S** of states. One of these states is a special one called *Vacant*. If $\xi(x) = Vacant$ then we will say that there is no cell at site x. There is a positive number B such that in each configuration, the distances between cells (nonvacant sites) are multiples of B. For a cell x, we will call the interval $[x - B/2, x + B/2)$ with center x its **body** and the number B its **body size**.

An **evolution** is a partial funcion $\eta(x,t)$ assigning a state to each site at each time within the interval of interest. In discrete-time cellular automata, state transitions occur only at integer times. More generally, it is useful to allow the time between neighboring cell state transitions to be a positive number T not neccessarily equal to 1.

A **deterministic cellular automaton** is determined by B, T and a local transition rule *Trans*(): we can denote it by

$$CA(Trans, B, T).$$

For such a cellular automaton, **trajectories** are those evolutions that obey the rule everywhere: An evolution η is a trajectory if

$$\eta(x,t) = Trans(\eta(x - B, t - T), \eta(x,t), \eta(x + B, t - T))$$

holds for all x, t. Given a configuration ξ over the space **C** and a transition function, there is a unique trajectory η with the given transition function and the initial configuration $\eta(\cdot, 0) = \xi$.

2.2 Fields of a cell

The evolution of a deterministic cellular automaton can be viewed as a "computation". Moreover, every imaginable computation can be performed by an appropriately chosen cellular automaton rule. Indeed, the Turing machine can be considered a special cellular automaton.

We will deal only with cellular automata in which the states of the cells are binary strings of some fixed length. The length of these strings (i.e. the binary logarithm of the number of states) will be called the **capacity** of the cellular automaton, and will be denoted by *Cap*. If a cellular automaton has capacity greater than 1 then the information represented by the state can be broken up naturally into parts. If e.g. the capacity is 12 we could subdivide these bits into strings of lengths 2,2,1,1,2,4 respectively called **fields**, and call these the input, output, mail coming from left, mail coming from right, memory and workspace. We can denote these as Input, Output, Mail_j $(j = -1, 1)$, Mem and Wk. If s is a state then s.Input denotes the first two bits of s, $s.\mathsf{Mail}_+$ means the sixth bit of s, etc.

Fields are always either disjoint or contained in each other. When we join e.g. the input fields of the cells at different sites we can speak about the input **track**, like a track of some magnetic tape. Fields will sometimes be also called **variables**. This suggests to view the state of each cell as the internal configuration of a little computer during the execution of a program: a variable in a programming language would refer to a certain part of the memory.

2.3 Probabilistic cellular automata

A **random evolution** is a pair μ, η where μ is a probability measure over some space Ω, together with a measurable function $\eta(x,t,\omega)$ which is an evolution for all $\omega \in \Omega$. We will generally omit ω from the arguments of η.

A **probabilistic cellular automaton** M is characterized by saying which random evolutions are considered trajectories. Thus, now a trajectory is not a single evolution but a distribution. A random evolution will be called a trajectory if it satisfies a certain local condition on the distribution. The condition depends on a **transition matrix**

$$\mathit{Trans\_prob}(s \mid r_{-1}r_0, r_1).$$

It says that the random evolution η is a trajectory if and only if the following holds. For all $x, t, s, r_{-1}, r_0, r_1$, if $\eta(x + jB, t - T) = r_j$ $(j = -1, 0, 1)$ and $\eta(x', t')$ is otherwise fixed arbitrarily for all $t' < t$ and for all $x' \neq x$, $t' = t$, then the conditional probability of $\eta(x, t) = s$ is equal to $\mathit{Trans}(s, r_{-1}, r_0, r_1)$.

A trajectory of a probabilistic cellular automaton is a discrete-time Markov process. If the set of cells is finite, moreover it consists of a single cell, then $\mathit{Trans\_prob}(s \mid r)$ is the transition probability matrix of this finite Markov chain.

2.4 Perturbation

Intuitively, a deterministic cellular automaton is fault-tolerant if even after it is "perturbed" into a probabilistic cellular automaton, its trajectories can keep the most important properties of the original trajectories.

Let $M = CA(\mathit{Trans}, B, T)$. We will say that a random evolution μ, η is a trajectory of the **ε-perturbation** $\mathit{Ptrb}_\varepsilon(M)$ of M if the following holds. For all x, t, r_{-1}, r_0, r_1, if $\eta(x + jB, t - T) = r_j$ $(j = -1, 0, 1)$ and $\eta(x', t')$ is otherwise fixed arbitrarily for all $t' < t$ and for all $x' \neq x$, $t' = t$, then the conditional probability of $\eta(x, t) = \mathit{Trans}(r_{-1}, r_0, r_1)$ is at least $1 - \varepsilon$.

Deterministic cellular automaton, probabilistic cellular automaton and ε-perturbation are special cases of a more general concept called **medium**. A medium will be defined later, more generally, by a certain set of "local" restrictions on random evolutions. Those random evolutions that satisfy these restrictions will be called trajectories.

Note that the medium $\mathit{Ptrb}_\varepsilon(M)$ is not a probabilistic cellular automaton. Rather, no matter how we obtain a probabilistic cellular automaton by a sufficiently small (possibly not even homogenous) local perturbation from M, its trajectories will also be trajectories of $\mathit{Ptrb}_\varepsilon(M)$.

Remembering a few bits Let D be a deterministic cellular automaton and suppose that the bit string that is a cell state has some field F (it can e.g. be the first two bits of the state). We will say that D **remembers** field F if there is an $\varepsilon > 0$ such that for each string $s \in \{0,1\}^{|\mathsf{F}|}$ there is a configuration ξ_s such that for all trajectories (μ, η) of $\mathit{Ptrb}_\varepsilon(D)$ with $\eta(\cdot, 0).\mathsf{F} = \xi_s$, for all x, t we have

$$\mu\{\eta(x, t).\mathsf{F} = s\} > 2/3.$$

In these terms, one result of the paper [4] says that there is a one-dimensional deterministic cellular automaton that remembers a field.

3 Codes

3.1 Colonies

For the moment, let us concentrate on the task of remembering a single bit in a field called Main_bit of a cellular automaton. Our non-local organization will be based on the concept of colonies. Let c be a cell and Q a positive integer with

$$Q' = \frac{Q - 1}{2}.$$

The set of Q cells $c - Q'B + iB$ for $i \in [0, Q)$ will be called the **Q-colony with center c**, and cell $c - Q'B + iB$ will be said to have **address i** in this colony. Cell b belongs to the colony iff i is an integer in $[0, Q)$. The union of the cell bodies of this colony occupies an interval of size QB with center c.

Let us be given a configuration ξ of a cellular automaton M with state set $\mathbf{S}$, and Q some positive integer. The fact that ξ is "organized into colonies" will mean that one can break up the set of all cells into non-overlapping colonies of size Q, using an **address field** Addr. The value $\xi(x).\mathsf{Addr}$ is a binary string which can be interpreted as an integer in $[0, Q)$. We will say that a certain Q-colony $\mathcal{C}$ is a "real" colony of ξ if for each element x of $\mathcal{C}$ with address i we have $\xi(x).\mathsf{Addr} = i$.

Our cellular automata will not change the value of the address field unless it seems to require correction. In the absence of faults, if such a cellular automaton is started up with a configuration that is grouped into colonies then this grouping will survive and the cells can use the Addr field to identify their colleagues withing their colony.

Grouping into colonies seems to help preserving the Main_bit field since each colony has this information in Q-fold redundancy. The transition rule may somehow involve the colony members in a coordinated periodic activity, repeated after a period of T steps for some integer T, of restoring this information from the gradual degradation caused by faults (e.g. with the help of some majority operation).

Let us call T steps of work of a colony of cells a **work period**. The best we can expect from a transition rule of the kind described above is that unless too many faults happen during some colony work period the Main_bit field of most cells in the colony will always be the original one. Such rules can indeed be written: they are not too complex. But they do not accomplish qualitatively much more than a simple local majority vote for the Main_bit field among three neighbors.

Suppose that a group of failures changes the original content of the Main_bit field in some colony, in so many cells that an internal correction is no more possible. The information is not really lost since most probably, neighbor colonies still have it. But correcting with the help or other colonies requires some organization reaching wider than a single colony. To arrange this broader activity also in the form of a cellular automaton we need the notion of simulation.

Let us denote by M_1 the fault-tolerant cellular automaton to be built. In this automaton, a colony $\mathcal{C}$ with center c will be involved in two kinds of activity during each of its work periods.

Simulation: Manipulating the collective information of the colony in a way that can be interpreted as the simulation of a single state transition of the cell c (of body size QB) of some cellular automaton M_2.

Trickle-down: Using the collective information (the state of c in M_2) to rewrite the Main_bit field of each cell within the colony if necessary.

The expression "trickle-down" suggests a view in which the simulated cells are the "higher, bigger entities", similar to institutions. Trickle-down will be an uncomplicated operation; let us first concentrate on the simulation.

Of course, even the cells of the simulated automaton M_2 will not be immune to errors. They must also be grouped into colonies simulating an automaton M_3, etc.; the organization will therefore have to be a *hierarchy of simulations.*

Reliable computation itself can be considered a kind of simulation of a deterministic cellular automaton by a probabilistic one.

3.2 Block codes

Codes on strings The notion of simulation relies on the notion of a **code**, since the way the simulation works is that the simulated evolution can be decoded from the simulating evolution. A code, φ between two sets X, Y is, in general, just a pair φ_*, φ^* where $\varphi_* : X \to Y$ is the **encoding function** and $\varphi^* : Y \to X$ is the **decoding function** and the relation

$$\varphi^*(\varphi_*(x)) = x$$

holds. A simple example would be when $X = \{0,1\}$, $Y = X^3$, and $\varphi_*(x) = (x,x,x)$ while $\varphi^*((x,y,z))$ is the majority of x, y, z.

This example can be generalized to a case when $\mathbf{S}_1, \mathbf{S}_2$ are finite cell state sets, $X = \mathbf{S}_2$, $Y = \mathbf{S}_1^Q$ where the positive integer Q is called the **block size**. Such a code is called a **block code**. Strings of the form $\varphi_*(r)$ are called **codewords**. The elements of a codeword $s = \varphi_*(r)$ are numbered as $s(0), \ldots, s(Q-1)$. The following block code can be considered our main simple example of codes.

(3.1) **Example** Suppose that $\mathbf{S}_1 = \mathbf{S}_2 = \{0,1\}^{12}$ is the state set of both media M_1 and M_2. As mentioned above, it is useful to break up the information contained in a state into various functional parts, called fields. Let us introduce the fields r.Addr and r.Info of a state $r = r_1 \cdots r_{12}$ in $\mathbf{S}_1$. The Addr field consists of the first 4 bits $r_1 \cdots r_4$, while the Info field is the last bit r_{12}. The other bits do not belong to any named field. Let $Q = 31$. Thus, we will use codewords of size 31, formed of the cells of M_1, to encode cells of M_2. The encoding funcion φ_* assigns a codeword $\varphi_*(r) = s(0) \cdots s(30)$ of elements of $\mathbf{S}_1$ to each element r of $\mathbf{S}_1$. Let $r = r_1 \cdots r_{12}$. We will set $s(i-1).\mathsf{Info} = r_i$ for $i = 1, \ldots, 12$. The 4 bits in $s(i).\mathsf{Addr}$ will denote the number i in binary notation. This did not determine all bits of the cells $s(0) \cdots s(30)$ in the codeword. In particular, the bits not belonging to neither the Addr nor the Info field are

not determined, and the values of the Info field for the cells $s(i)$ with $i \notin [0, 12)$ are not determined. To determine $\varphi_*(r)$ completely, we could set these bits to 0.

The decoding function is simpler. Given a word $s = s(0) \cdots s(30)$ we first check whether it is a "normal" codeword: if it does not have $s(0).\mathsf{Addr} = 0$ and $s(i).\mathsf{Addr} \neq 0$ for $i \neq 0$ then $r = \varphi^*(s)$ is the special value $\mathit{Vacant} \in \mathbf{S}_2$. Otherwise, $r_i = s(i-1).\mathsf{Info}$ for $i \in [1, 12]$.

Informally, the cells of the codeword use their first 4 bits to mark their address within the codeword. The last bit is used to remember their part of the information about the encoded cell. $\diamond$

A codeword w will be called **accepted** if $\varphi^*(w) \neq \mathit{Vacant}$, otherwise it is called **rejected**. A block code will be called **overlap-free** if for every string $s(1) \cdots s(n)$, if both $s(1) \cdots s(Q)$ and $s(i+1) \cdots s(i+Q)$ are accepted then $i \geq Q$. In other words, a code is overlap-free if two accepted words cannot overlap in a nontrivial way. The code in Example 3.1 is overlap-free. All block-codes considered from now on will be overlap-free. Overlap-free codes are used, among others, in [10].

Codes on configurations An overlap-free block code φ can be used to define a code on configurations between a medium M_1 of some cell size B and a medium M_2 of cell size QB. Suppose that a configuration ξ of medium M_2 is given. Then we define the configuration $\xi_* = \varphi_*(\xi)$ of M_1 by setting for each cell c of ξ and $i \in [0, Q)$,

$$\xi_*(c - Q'B + iB) = \varphi_*(\xi(c))(i).$$

The decoding function is defined correspondingly. Suppose that a configuration ξ of M_1 is given. We define the configuration $\xi^* = \varphi^*(\xi)$ of the medium M_2 as follows: for cell x, we set $\xi^*(x) = \varphi^*(s)$ where s is the string

$$\xi(x - Q'B)\xi(x - Q'B + B) \cdots \xi(x + Q'B) \tag{3.2}$$

of states of the cells of the colony with center c. If $\xi = \varphi_*(\zeta)$ then, due to the overlap-free nature of the code, the value $\xi^*(x)$ is nonvacant only at positions x where $\zeta(x)$ is defined. It is interesting to note that if ξ is not the code of any configuration ζ then it may happen that in the decoded configuration $\varphi^*(\xi)$, the cells will be not exactly at a distance QB apart. The overlap-free nature of the code still garantees that the distance of cells in $\varphi^*(\xi)$ is at least QB. This situation cannot be helped. Eventually, we will have to legalize these "illegal" decoded configurations; for the while, let us just ignore them.

3.3 Block simulations

Suppose that media M_1 and M_2 are deterministic cellular automata where $M_i = CA(\mathit{Trans}_i, B_i, T_i)$, and φ is a block code with

$$B_1 = B, \ B_2 = QB.$$

Suppose that the decoding function is as simple as in our Example 3.1: there is an Info track and once the colony is accepted the decoding function depends only on this part of the information in it.

For each evolution η of M_1, we can define an evolution $\eta^* = \varphi^*(\eta)$ of M_2 by setting

$$\eta^*(\cdot, t) = \varphi^*(\eta(\cdot, t)). \tag{3.3}$$

We will say that the code φ is a **simulation** if for each configuration ξ of M_2, for the trajectory μ, η of M_1, such that $\eta(\cdot, 0, \omega) = \varphi_*(\xi)$ for almost all ω, the random evolution μ, η^* is a trajectory of M_2. (We do not have to change μ here since the ω in $\eta^*(x, t, \omega)$ is still coming from the same space as the one in $\eta(x, t, \omega)$.)

We can view φ_* as an encoding of the initial configuration of M_2 into that of M_1. An evolution η of M_1 will be viewed to have a "good initial condition" $\eta(\cdot, 0)$ if the latter has the form $\varphi_*(\xi)$ for some configuration of M_2. Our requirements say that from every trajectory of M_1 with good initial conditions, the simulation decodes a trajectory of M_2.

Let us show one particular way in which the code φ can be a simulation. For this, the function Trans_1 must behave in a certain way which we describe here. Assume that

$$T_1 = T, \ T_2 = UT$$

for some positive integer U called the **work period size**. Each cell of M_1 will go through a period consisting of U steps in such a way that the Info field will be changed only in the last step of this period. The initial configuration $\eta(\cdot, 0) = \varphi_*(\xi)$ is chosen in such a way that each cell is at the beginning of its work period (this imposes a condition on the encoding function φ_*). By the nature of the code, in the initial configuration, cells of M_1 are grouped into colonies.

Once started from such an initial configuration, during each work period, each colony, in cooperation with its two neighbor colonies, does the following. Let us denote by r_-, r_0, r_+ the value in the first 12 bits of the Info track in the left neighbor colony, in the colony itself and in the right neighbor colony respectively. First, the three colonies "compute" $s = \mathit{Trans}_2(r_-, r_0, r_+)$ where Trans_2 is the transition function or M_2 and stores it on a memory track. (We are not interested now in how Trans_1 manages to perform this computation: just assume that it happens. It may help understanding how it happens if we think of the possibilities of using some mail, memory and workspace tracks.) Then, in the last step, s will be copied onto the Info track.

Clearly, if Trans_1 is like this then our code is a simulation. Such a simulation is called a **block simulation**.

There are many ways to construct block simulations. Moreover, there is a cellular automaton that can simulate every other cellular automaton by a block simulation. A transition function *Trans* is **universal** if for every other transition function *Trans'* there are Q, U and a block code φ such that φ is a block simulation of $CA(\mathit{Trans}', Q, U)$ by $CA(\mathit{Trans}, 1, 1)$. If *Trans* is universal then we will call every cellular automaton $CA(\mathit{Trans}, B, T)$ universal.

It is known that there is a universal transition function. This is proved somewhat analogously to the theorem on the existence of universal Turing machines.

Note that a universal cellular automaton cannot use codes similar to our example 3.1. Indeed, in that example, the capacity of the cells of M_1 is at least the binary logarithm of the colony size, since each colony cell contained its own address within the colony. But if M_1 is universal then the various simulations in which it participates will have arbitrarily large colony sizes.

The size Q of the simulating colony will generally be very large also since the latter contains the whole table of the simulated transition function. There are many special cellular automata M_2, however, whose transition function can be described by a small program and computed in relatively little space and time (linear in the size s_2). The universal transition function will simulate these with correspondingly small Q and U. We will only deal with such automata.

3.4 Single-fault-tolerant block simulation

Here, we outline a cellular automaton M_1 that block-simulates a cellular automaton M_2 correctly as long as at most a single error occurs in a colony work period of size U. The outline is very informal: it is only intended to give some mental model to refer to.

The automaton M_1 is not universal, i.e. the automaton M_2 cannot be chosen arbitrarily. Among others, this is due to the fact that the address field of a cell of M_1 will hold its address within its colony. But we will see later that universality is not needed in this context.

The cells of M_1 will have, besides the Addr field, also a field Age. If no errors occur then in the i-th step of the colony work period, each cell will have the number i in the field Age. There are also fields called Mail, Info, Wk, Hold, Prog.

The Info field holds the state of the represented cell of M_2 in three copies. The Hold field will hold parts of the final result before it will be, in the last step of the work period, copied into Info. The role of the other fields is clear.

The program will be described from the point of view of a certain colony $\mathcal{C}$. Here is an informal description of the activities taking place in the first third of the work period.

1. From the three thirds of the Info field, by majority vote, a single string is computed. Let us call it the **input string**. This computation, as all others, takes place in the workspace field Wk; the Info field is not affected. The result is also stored in the workspace.

2. The input strings computed in the two neighbor colonies are shipped in into $\mathcal{C}$ and stored in the workspace separately from each other and the original input string.

3. The workspace field behaves as some kind of universal automaton, and from the three input strings and the Prog field, computes the string that would be obtained by the transition function of M_2 from them. This string will be copied to the first third of the Hold track.

In the second part of the work period, the same activities will be performed, except that the result will be stored in the second part of the Hold track. Similarly with the third part of the work period. In a final step, the Hold field is copied into the Info field.

The computation is coordinated with the help of the Addr and Age fields. It is therefore important that these are correct. Fortunately, if a single fault changes such a field of a cell then the cell can easily restore it using the Addr and Age fields of its neighbors.

It is not hard to see that with such a program (transition rule), if the colony started with "perfect" information then a single fault will not corrupt more than a third of the colony at the end of the work period. On the other hand, if two thirds of the colony was correct at the beginning of the colony work period and there is no fault during the colony work period then the result will be "perfect".

4 Hierarchy

4.1 General simulations

In its most general form, a **simulation** of medium M_2 by medium M_1 is given by two mappings: a mapping Φ^* of the set of evolutions of M_1 into the set of evolutions of M_2 (the decoding) and a mapping φ_* of the set of configurations of M_2 to the set of configurations of M_1 (the encoding). The following properties are required for each configuration ξ and each trajectory η with starting time 0 and $\eta(\cdot, 0) = \varphi_*(\xi)$:

(a) The evolution $\Phi^*(\eta)$ is a trajectory of M_2.

(b) $\Phi^*(\eta)(\cdot, 0) = \xi$.

Our simulations will be invariant with respect to shifts in space-time. For this property, we will have to choose the cell sizes of the two media M_1 and M_2 appropriately, to avoid scaling.

A simulation will be called **local**, if there is a finite space-time window $V^* = I \times [-u, 0]$ such that $\Phi^*(\eta)(w, t)$ depends only on $\eta[\vec{w, t} + V^*]$. Together with the shift-invariance property, the locality property implies that a simulation is determined by a function defined over $Evols(V^*)$. Our simulations will be local unless stated otherwise.

If $u = 0$ then the configuration $\eta^*(\cdot, t)$ depends only on the configuration $\eta(\cdot, t)$. In this case, the simulation could be called "memoryless". For memoryless simulations, the simulation property is identical to the one we gave at the beginning of Subsection 3.3. If $u > 0$ then the decoding looks back on the whole evolution in the interval $[t - u, t]$.

Our real goal is to find nontrivial simulations between media M_1 and M_2, especially when these are not deterministic cellular automata. If e.g. M_1, M_2 are probabilistic cellular automata then the simulation property would mean that whenever we have a trajectory $\vec{\mu, \eta}$ of M_1 the evolution η^* decoded from η would be a trajectory of M_2. There are hardly any nontrivial examples of this sort since in order to be a trajectory of M_2, the conditional probabilities of $\varphi^*(\eta)$ must satisfy certain equations defined by $Trans\_prob_2$, while the conditional probabilitiees of η satisfy equations defined by $Trans\_prob_1$.

Simulation between perturbations There is more hope in the case when M_1 and M_2 are perturbations of some deterministic media since in this case, only some inequalities must be satisfied. Our modest goal of improving reliability could be this. For some universal transition function $Trans_2$, find $Trans_1, Q, U, c$ with $B_1 = B$, $B_2 = BQ$, $T_1 = T$, $T_2 = TU$ and a block simulation φ_1 such that for all $\varepsilon > 0$, if $\varepsilon_1 = \varepsilon$, $\varepsilon_2 = c\varepsilon^2$ and M_i is the perturbation $Ptrb_{\varepsilon_i}(CA(Trans_i, B_i, T_i))$ then φ_1 is a simulation of M_2 by M_1.

The meaning of this would be that our simulation will compute $Trans_2$ with a much smaller fault probability $c\varepsilon^2$, and this amounts to a kind of error correction. The hope is not unreasonable since in Subsection 3.4, we outlined a single-fault-tolerant block simulation while the probability of several faults happening during one work period is only of the order of $(QU\varepsilon)^2$.

Trickle-down Let us be sanguine for a moment and assume that our goal can be achieved (though in this form, it cannot yet); moreover, that we can define a whole sequence $M_1, M_2, \ldots$ of media and a sequence $\varphi_1, \varphi_2, \ldots$ of simulations such that φ_i is a simulation of M_{i+1} by M_i. Such a structure is called an **amplifier**. If $\eta = \eta^1$ is an evolution of M_1 then for $k > 1$ we define

$$\eta^k = \varphi^*_{k-1}\varphi^*_{k-2}\cdots\varphi^*_1\eta. \tag{4.1}$$

If $M_k = Ptrb_{\varepsilon_k}(CA(Trans_k, Q_k, U_k))$ then we can hope that ε_{k+1} is only about $(U_k Q_k \varepsilon_k)^2$, and that U_k, Q_k does not grow fast (they may even be constant). This is as close as a probabilistic automaton can get to simulating a deterministic one.

Amplifiers seem to be the generalization a renormalization groups in statistical physics (however, the semigroup property holds here only in a trivial sense).

When combined with trickle-down (mentioned above in Subsection 3.1), such a sequence of simulations would solve the problem of remembering a bit. We need to define trickle-down more formally for a simulation φ of medium M_2 by medium M_1. (The completely formal definition is not essentially different.) For an evolution η of M_1, let η^* denote the evolution $\varphi(\eta)$ of M_2. We will say that φ has the **ε-trickle-down** property with blocksize Q if for all trajectories μ, η of M_1, for each pair of sites x_1, x_2 and time t the following holds. Suppose that x_1 is an element of the Q-colony with center x_2 and $\eta^*(x_2, t)$ is nonvacant; then with conditional probability $1 - \varepsilon$, we have

$$\eta(x_1, t).\mathsf{Main\_bit} = \eta^*(x_2, t).\mathsf{Main\_bit}. \tag{4.2}$$

4.2 Hierarchical codes

Concatenated codes Let us discuss the hierarchical structure arising in an amplifier. If φ, ψ are two codes then $\varphi \circ \psi$ is defined by $(\varphi \circ \psi)_*(\xi) = \varphi_*(\psi_*(\xi))$ and $(\varphi \circ \psi)^*(\zeta) = \psi^*(\varphi^*(\zeta))$. It is assumed that ξ and ζ are here configurations of the appropriate media, i.e. the cell body sizes are in the corresponding relation.

For example, let M_1, M_2, M_3 have cell sizes $1, 31, 31^2$ respectively. Let us use the code φ from Example 3.1. The code $\varphi^2 = \varphi \circ \varphi$ maps each cell c of medium M_3 with body size $(31)^2$ into a "supercolony" of 31×31 cells of size 1 in M_1. Suppose that $\zeta = \varphi_*^2(\xi)$ is a configuration obtained by encoding from a lattice configuration of body size 31^2 in M_3. Then ζ can be broken up into colonies of size 31 around the cells $31i$ for $i \in \mathbf{Z}$. Cell 55 belongs to the colony with center 62 and has address 8 in it. Therefore the address field of $\zeta(55)$ contains a binary representation of 8. Its last bit encodes the 8-th bit of the cell 62 of M_2 represented by this colony. If we read together all 12 bits represented by the Info fields of the first 12 cells in this colony we get a state $\zeta^*(62)$ (we count from 0). The cells $31j$ for $j \in \mathbf{Z}$ with states $\zeta^*(31j)$ obtained this way are also broken up into colonies. In them, the first 4 bits of each $\zeta^*(31j)$ form the address and the last bits of the first 12 cells, when put together, give back the state of the cell represented by this colony. Notice that these 12 bits were really drawn from 31^2 cells of M_1. Even the address bits in $\zeta^*(62)$ come from different cells of the colony with center 62. Therefore cell with state $\zeta(55)$ does not contain information allowing to conclude that it is cell 55. It only "knows" that it is the 8-th cell within its own colony (with center 62) but not that its colony has address 17 within its supercolony (with center 0).

This process can be repeated, a code can be concatenated an arbitrary number of times with itself or other codes. In this way, a hierarchical, i.e. highly nonhomogenous, structure can be defined using cells that have only a small number of states.

Infinite hierarchy We will even need an infinite concatenated code since the the definition of the initial configuration for M_1 in the amplifier depends on all codes φ_i. What is the meaning of such a code? Suppose that we have a sequence $\varphi_1, \varphi_2, \ldots$ of codes with word sizes $Q_1, Q_2, \ldots$. We will want to concatenate them "backwards", i.e. in such a way that from a configuration ξ^1 of medium M_1 with cell body size 1, we can decode the configuration $\xi^2 = \varphi_1^*(\xi^1)$ of medium M_2 with cell body size Q_1, configuration $\xi^3 = \varphi_2^*(\xi^2)$, of M_3 with body size $Q_1 Q_2$, etc.

The repeated decoding process is well-defined by our formulas; however, how will we *encode* anything here? It seems that we have to begin at infinity. For simplicity, let us assume that all code sizes Q_i are odd. There is an easy solution, if our block codes φ_i have the following property: for codewords $\varphi_i(r) = s(0) \cdots s(Q-1)$, the middle symbol $s(Q_i')$ does not depend on r. Notice that our example code has this property. Let s_i be the common middle symbol of all codewords of the code φ_i. Let ξ_0^i be the configuration of M_i which has a single nonvacant cell at site 0 in state s_i. Let $R_i = Q_1 Q_2 \cdots Q_i$. Now, define the configurations ξ_i of M_1 which are nonvacant over integer sites x in $[-R_i', R_i')$, as follows:

$$\xi_i = \varphi_{1*}(\varphi_{2*}(\cdots \varphi_{i-1*}(\xi_0^i) \cdots))$$

for $i > 1$, and $\xi^1 = \xi_0^1$. Then each configuration in this sequence is an extension, to the left and right, of the previous one and so they converge to a limit.

Riders The infinite code defined above is not very satisfactory since the result of the encoding does not contain any information that is not in the codes themselves. To fix this problem let us agree that all cellular automata considered will have a field Rider. We generalize our code to a two argument function $\varphi_*(f, r)$ where r is a symbol of M_2 and f is a string of length Q from some new state space $\mathbf{K}$. For each symbol $r \in \mathbf{S}_2$ and string $f = f_0 \dots f_{Q-1}$ with $f_i \in \mathbf{K}$, the codeword $\varphi_*(f, r)$ is a string $s = s(0) \cdots s(Q-1)$ with $s(i) \in \mathbf{S}_1$, with the property that $\varphi^*(s) = r$ and $s(i).\mathsf{Rider} = \gamma^*(s(i)) = f_i$. Such a code will be called a block code with a **rider argument**. The string f gets a "ride" symbol-for-symbol on the codeword s. In Example 3.1, the set $\mathbf{K}$ could be the set of bit pairs, the field Rider could be bits 10 and 11 of the 12-bit cell state. The encoding $\varphi_*(f, r)$ would be almost the same as $\varphi_*(r)$, except that bits 10 and 11 of each $s(i)$ in $s = \varphi_*(f, r)$ receive, instead of 0's, the two bits of f_i. The decoding φ^* is the same, and the decoding γ^* is simply the reading out of the field Rider from a symbol of M_1.

Let us call a code $\varphi_*(f, r)$ with a rider **upward compatible** if its middle symbol (we still assume that Q is odd) depends only on the rider f (of course, then it depends only on the middle symbol of f). From now on, all our codes will be assumed (and constructed) to be upward compatible. Let us resume our attempt to define an infinite concatenation of codes whose decoding is $\varphi_1^* \circ \varphi_2^* \circ \cdots$. Assume that the code $\varphi_{i*}(f, r)$ is defined with rider state space $\mathbf{K}_i$. For each $i = 1, 2, \dots$, let us be given a rider configuration χ_i and let χ denote the whole sequence. Upward compatibility implies that for all configurations ξ of M_{i+1}, the state $\varphi_{i*}(\chi_i, \xi)(0)$ depends only on the rider configuration χ_i and not on the encoded configuration ξ. Let ξ_0^i be the configuration of M_i which has the single nonvacant cell at site 0 in this state and

$$
\begin{aligned}
\psi_{1*}(\chi, \xi) &= \varphi_{1*}(\chi_1, \xi), \\
\psi_{i*}(\chi, \xi) &= \psi_{(i-1)*}(\chi, \varphi_i(\chi_i, \xi)) \\
\xi_i &= \psi_{(i-1)*}(\chi, \xi_0^i).
\end{aligned} \tag{4.3}
$$

Configuration ξ_{i+1} is an extension, to the left and right, of ξ_i and so the sequence converges to a limit ξ. This limit is not a trivial infinite code anymore since all the rider configurations χ_i can be decoded from it via ψ_i^* and γ_i^*.

Note that though the configuration ξ above is obtained by an infinite process of encoding, there is no infinite process of decoding yielding a single configuration from it. At the k-th stage of the decoding, we get the configuration ξ^k and all these configurations have different cell sizes.

It helps to visualize this infinite code if we think of each cell c of a configuration ξ^i as a clerk who has a "main job", for which she is paid: to participate in the common task of her colony which is to encode all information (including the Rider field) of a cell of ξ^{i+1}. But the state $\xi^i(c)$ has some free capacity which can be used for some hobby: this hobby will simply be to store a symbol $\chi_i(c)$ of the i-th rider configuration in her own rider field. Thus, the only information in the configuration ξ contained in the infinite code is the one derivable from the hobbies of these clerks on various levels. It is a crucial feature of this system that the information that is just hobby on levels $i+1$ and higher is part of the main information to be encoded on levels i and lower. If therefore, as it will turn out, our codes lend increasing degree of reliability to the information on higher levels, the information contained in the rider strings increases in reliability as we climb higher in the hierarchy. This reliability will be exploited. In the simplest task, in the construction to remember the field Main_bit, the latter field will be essentially identical to Rider.

4.3 Amplifiers with trickle-down

Suppose that we succeed constructing an amplifier $M_1, M_2, \dots, \varphi_1, \varphi_2, \dots$ such that M_k is a perturbed medium with fault probability ε_k and the simulation φ_k has colony size Q_k and the ε_k'-trickle-down property with this colony-size. Here, both $\sum_k \varepsilon_k$ and $\sum_k \varepsilon_k'$ are less than 0.1. Also, the Main_bit field is the Rider field of the simulations φ_k.

Let $\mu, \vec{\eta}^1$ be a trajectory of M_1 starting from the initial configuration that is the infinite code described above, in which $\eta^1(x, 0).\mathsf{Main\_bit} = 0$ for all cells x, and let $\eta^{i+1} = \varphi_k^*(\eta^i)$. Then $\mu, \vec{\eta}^k$ is a trajectory of M_k. Take a cell x_1 of η_1 and a time t. This cell belongs to some colony with center x_2 in the simulation φ_1. Site x_2 of M_2 belongs to some colony with center x_3 in φ_2, etc. Since $\sum_k \varepsilon_k < 0.1$, for some k, the probability will be at least 0.9 that $\eta^i(x_i, t).\mathsf{Main\_bit} = 0$ for all $i \geq k$. From here, with repeated application of the trickle-down property, we can conclude that the probability is at least 0.8 that $\eta(x_1, t).\mathsf{Main\_bit} = 0$, which will show that the medium M_1 indeed remembers Main_bit.

4.4 Major difficulties

The idea of a simulation between two perturbed cellular automata is, unfortunately, flawed in the original form: the mapping defined in the naive way is not a simulation in the strict sense we need. A group of failures can destroy not only the information but also the organization into colonies in the area where it occurs. This kind of event cannot therefore be mapped by the simulation into a transient fault unless destroyed colonies "magically recover". The recovery is not trivial since "destruction" can also mean replacement with something else that looks locally as legitimate as healthy neighbor colonies but is incompatible with them. Rather than give up hope let us examine the different kinds of disruption that the faults can cause in a block simulation by a perturbed cellular automaton M_1.

Let us take as our model the informally described automaton of subsection 3.4. The information in the current state of a colony can be divided into the following parts:

"information": an example is the content of the Info track.

"structure": the Addr and Age tracks.

"program": the Prog track.

The "structure" does not represent any data for the decoding but is needed for coordinating cooperation of the colony members. The "program" determines which transition function will be simulated. The "information" determines what will be in the state of the simulated cell: it is the "stuff" that the colony processes.

Disruptions are of the following kinds (or a combination of these):

(1) Local change in the "information";

(2) Locally recognizable change in the "structure";

(3) Program change;

(4) Locally unrecognizable change in "structure";

A locally recognizable structure change would be a change in the address field. A locally unrecognizable change would be to erase two neighbor colonies sitting, say, at positions BQ and $2BQ$ and to put a new colony in the middle of the gap of size $2BQ$ obtained this way, at position $1.5BQ$ (remember that each colony is positioned around its center). Cells within the two new colonies and the remaining old colonies will be locally consistent with their neighbors; on the boundary, the cells have no way of deciding whether they belong to a new (and wrong) colony or an old (and correct) one.

The only kind of disruption whose correction can be attempted along the lines of traditional error-correcting codes and repetition is the first one: a way of its correction was indicated in Subsection 3.4. The three other kinds are new and we will deal with them in different ways.

To fight locally recognizable changes in the structure, we will use destruction and rebuilding. Cells that find themselves in structural conflict with their neighbors will become vacant. Vacant cells will eventually be restored if this can be done in a way structurally consistent with their neighbors.

To fight program changes, our solution will be that our simulation will not use any "program". We will not lose universality this way: our automata will still be universal, i.e. capable of simulating every other automaton by appropriate block simulation; but this particular simulation will differ from the others in that the transition rule will perform it without looking at any program, whenever the local consistency conditions are satisfied.

To fight locally unrecognizable changes, we will "legalize" all the structures brought in this way. In our example, the decoding function is already defined even for the configuration in which a single colony sits in a gap of size $2BQ$. In this decoded configuration, the cell at site 0 is followed by a cell at site $1.5BQ$ which is followed by cells at sites $3BQ, 4BQ$, etc. Earlier, we could not make any sense of these illegal configurations. We must legalize them now. Indeed, since they can be eliminated only with their own active participation, we must have rules (trajectory conditions) applying to them.

Our generalized cellular automaton will be called a **robust medium**. The generalization of the notion of the medium does not weaken the original theorem: the fault-tolerant cellular automaton that we eventually build is a cellular automaton in the old sense. The more general media are only needed to make sense of all the structures that arise in simulations by a random process.

4.5 Annotated contents

Here is an outline of the whole work.

Defining the general notion of a variable-period (asynchronous) medium, and stating the main results. We connect this notion of media with discrete-time and continuous-time Markov processes.

Description of a simple block simulation in order to fix concepts.

A **robust medium** is defined: this is similar to a perturbed cellular automaton, with the following differences:

Cells are not necessarily adjacent to each other.

The dwell period (time between state switchings) of cells is not constant, it has upper and lower bounds.

There is a special set of space-time points called the **damage**.

The local conditions on trajectories will be expressed in form of **axioms**.

The **Restoration Axiom** requires that in a trajectory, damage appear with small probability (ε) and disappear with large probability.

The **Computation Axiom** requires that the trajectory obey the transition function in the absence of damage. The transition function can erase and create cells.

The damage is defined in the simulated evolution of a robust medium as follows:

Let M_1 simulate M_2. We will say that damage occurs at a certain point $\vec{x},\bar{t}$ of space-time in η^* if within a certain space-time window $\vec{x},\bar{t} + (-w, w) \times [-u, 0]$ in the past of $\vec{x},\bar{t}$, the damage of η cannot be covered by a small rectangle of a certain size. This is saying, essentially, that damage occurs at least "twice". The restoration axiom for η with ε will then guarantee that the damage in the simulated medium η^* also satisfies a restoration axiom with with $\bar{\approx} \varepsilon^2$.

The notion of **amplifier** frame (for robust media) is defined. The **Amplifier Lemma** is the main lemma: it states for each frame the existence of an amplifier $M_1, M_2, \ldots$ fitting into it that has the trickle-down property.

The Amplifier Lemma is applied to the proof of the main theorems.

The simulation program (which defines the amplifier) is outlined. It will be given in terms of a number of **rules** and **conditions** (to be satisfied by rules to be defined later).

Rules concerning killing, creation and purge. Due to asynchrony, the main tool of analyzing arbitrary evolutions will be space-time paths of live cells. We prove the basic lemmas about paths that help trace back the pedigree of live cells.

The Decay rule. We prove the lemmas stating that once a gap has not been healed for a while then it will eat up a whole colony.

The crucial Attribution Lemma traces back each non-germ cell to a full colony. This lemma expresses best what can be called the **self-stabilization** property.

The Healing Rule and the Healing Lemma, showing how the effect of a small amount of damage will be corrected. Due to the need to restore some local clock values consistently with the neighbors, the healing rule is rather elaborate.

The error-correcting computation rules. The parts of the computation axiom not dependent on communication.

The communication rules. These are rather elaborate, due to the need to communicate with not completely reliable neighbor colonies asynchronously. We finish by proving the rest of the Computation Axiom.

References

[1] Piotr Berman and Janos Simon. Investigations of fault-tolerant networks of computers. In *Proc. of the 20-th Annual ACM Symp. on the Theory of Computing*, pages 66 – 77, 1988.

[2] Paula Gonzaga de Sá and Christian Maes. The Gács-Kurdyumov-Levin Automaton revisited. *Journal of Statistical Physics*, 67(3/4):607–622, May 1992.

[3] R. L. Dobrushin and S. I. Ortyukov. Upper bound on the redundancy of self-correcting arrangements of unreliable elements. *Problems of Information Transmission*, 13(3):201–208, 1977.

[4] Peter Gács. Reliable computation with cellular automata. *Journal of Computer System Science*, 32(1):15–78, February 1986.

[5] Peter Gács. Self-correcting two-dimensional arrays. In Silvio Micali, editor, *Randomness in Computation*, volume 5 of *Advances in Computing Research (a scientific annual)*, pages 223–326. JAI Press, Greenwich, Conn., 1989.

[6] Peter Gács, Georgii L. Kurdyumov, and Leonid A. Levin. One-dimensional homogenuous media dissolving finite islands. *Problems of Inf. Transm.*, 14(3):92–96, 1978.

[7] Peter Gács and John Reif. A simple three-dimensional real-time reliable cellular array. *Journal of Computer and System Sciences*, 36(2):125–147, April 1988.

[8] Lawrence F. Gray. The positive rates problem for attractive nearest neighbor spin systems on z. *Z. Wahrscheinlichkeitstheorie verw. Gebiete*, 61:389–404, 1982.

[9] Lawrence F. Gray. The behavior of processes with statistical mechanical properties. In *Percolation Theory and Ergodic Theory of Infinite Particle Systems*, pages 131–167. Springer-Verlag, 1987.

[10] Gene Itkis and Leonid Levin. Fast and lean self-stabilizing asynchronous protocols. In *Proc. of the IEEE Symp. on Foundations of Computer Science*, pages 226–239, 1994.

[11] G. L. Kurdyumov. An example of a nonergodic homogenous one-dimensional random medium with positive transition probabilities. *Soviet Mathematical Doklady*, 19(1):211–214, 1978.

[12] Thomas M. Liggett. *Interacting Particle Systems.* Number 276 in Grundlehren der mathematischen Wissenschaften. Springer Verlag, New York, 1985.

[13] Nicholas Pippenger. On networks of noisy gates. In *Proc. of the 26-th IEEE FOCS Symposium*, pages 30–38, 1985.

[14] Andrei L. Toom. Stable and attractive trajectories in multicomponent systems. In R. L. Dobrushin, editor, *Multicomponent Systems*, volume 6 of *Advances in Probability*, pages 549–575. Dekker, New York, 1980. Translation from Russian.

[15] John von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C. Shannon and McCarthy, editors, *Automata Studies.* Princeton University Press, Princeton, NJ., 1956.

[16] Weiguo Wang. *An Asynchronous Two-Dimensional Self-Correcting Cellular Automaton.* PhD thesis, Boston University, Boston, MA 02215, 1990. Short version: *Proc. 32nd IEEE Symposium on the Foundations of Computer Science*, 1991.

Paper Number 7

Observations on Self-Stabilizing Graph Algorithms for Anonymous Networks

Sandeep K. Shukla, Daniel J. Rosenkrantz and S. S. Ravi

Observations on Self-Stabilizing Graph Algorithms for Anonymous Networks[1]

(Extended Abstract)

Sandeep K. Shukla *Daniel J. Rosenkrantz* *S. S. Ravi*

Department of Computer Science
University at Albany - State University of New York
Albany, NY 12222

Abstract

We investigate the existence of **deterministic uniform** self-stabilizing algorithms (DUSSAs) for a number of problems on **anonymous** networks. This investigation is carried out under three models of parallelism, namely **central daemon**, **restricted parallelism**, and **maximal parallelism**. We show that many problems, including finding a maximum matching in an arbitrary graph and 2-coloring an odd degree bipartite graph, do not have DUSSAs even under the most restricted model, namely central daemon. We also observe that techniques due to Dana Angluin lead to general results that establish the non-existence of DUSSAs for a large collection of graph problems under any of the parallelism models. The problems in this collection include determining the parity of the number of nodes in a graph, determining the diameter of the graph, determining the eccentricities of the vertices of a graph and membership testing for various graph classes (for example, planar graphs, chordal graphs, and interval graphs).

On the positive side, we present DUSSAs for 2-coloring odd degree complete bipartite graphs, 2-coloring trees and finding maximal independent sets in general graphs. We also present randomized USSAs under maximal parallelism for some problems.

1 Introduction

The concept of **self-stabilization**, introduced by Dijkstra [Dij74], has been of considerable interest to researchers in the area of fault-tolerant distributed systems. Self-stabilization provides a uniform approach to fault-tolerance [Sch93]. Due to transient faults or arbitrary initialization, a distributed system may enter an undesirable or **illegitimate** global state [Dij74]. In such situations, a self-stabilizing algorithm (protocol) enables the system to recover to a legitimate global state in a finite amount of time. Self-stabilizing algorithms have been developed for a number of problems (see [Sch93, EK89] and the references cited therein).

In this paper we study self-stabilizing algorithms for graph problems on anonymous networks. A graph problem is specified by an *undirected* graph G and a requirement. Following previous work in this area (see for example, [GK93, KPBG94]), we assume that the distributed system has a processor for each node of the input graph; there is a communication link between two processors if and only if the corresponding nodes are adjacent in the input graph. The problem is to design a distributed algorithm such that when the system stabilizes, the resulting global state satisfies the specified requirement. As an example, consider the problem of 2-coloring an even ring. Here the graph is

[1]Research Supported by NSF Grants CCR-90-06396 and CCR-94-06611. Email addresses: {sandeep, djr, ravi}@cs.albany.edu

an unoriented ring with an even number of nodes. The requirement is that all the nodes of the ring are colored either 1 or 0 and no two adjacent nodes have the same color. We are required to design a distributed algorithm that will restore the system to a state where the colors assigned to the nodes satisfy the 2-coloring requirement. Moreover, the algorithm must enable the system to reach such a state from *any* initial state.

We restrict our attention to **uniform algorithms**, where each processor in the distributed system executes the same program. Uniform self-stabilizing algorithms (USSAs) are known for some problems including 6-coloring planar graphs [GK93], finding centers and medians for trees [KPBG94], 2-coloring certain rings and chains [SRR94], orienting odd-length rings [Hoe94], and leader election in rings of prime size [Hua93]. For some problems, it has been shown [Ang80, BP89, IJ93] that *deterministic* uniform self-stabilizing algorithms (DUSSAs) are impossible because of the difficulties encountered in deterministic **symmetry breaking**. For several such problems, researchers have presented randomized algorithms that self-stabilize with high probability; see for example, [Her90, IJ93, SRR94].

It is generally desirable to develop DUSSAs for problems on anonymous networks rather than non-uniform algorithms for ID-based networks. (See Section 2.1 for an explanation.) Unfortunately, for anonymous networks, DUSSAs are impossible even for very simple problems (e.g., 2-coloring an anonymous network [SRR94]). Therefore, it is of interest to investigate whether there are DUSSAs for problems on special classes of anonymous networks.

We consider three models for selecting the set of processors which will execute during a time step. These three processor selection models are **central daemon** [Dij74] (where a central scheduler selects one of the enabled processors at each step), **restricted parallelism** (where there is a specified restriction on the set of processors that may execute at each step), and **maximal parallelism** (where *all* enabled processors may execute at each step). A particular form of restricted parallelism, namely neighborhood-restriction, has the *non-interfering* property defined in [BGW89]. Also, maximal parallelism is the same as the **synchronous** model used in [BGM89, HWT94]. For all the problems considered in this paper, it is straightforward to show impossibility results under maximal parallelism. Therefore, we focus on DUSSAs under central daemon and restricted parallelism models. We then use a randomization strategy presented in [SRR94] to obtain randomized uniform self-stabilizing algorithms (RUSSAs) which self-stabilize with probability 1 under the maximal parallelism model. For some problems, we observe that DUSSAs are impossible under any of the three models. Our results are summarized below.

1. We show that there is no DUSSA for 2-coloring an odd-degree bipartite graph. We present DUSSAs for 2-coloring two subclasses of bipartite graphs, namely odd-degree complete bipartite graphs and trees. The DUSSA for 2-coloring trees uses the center finding algorithm for trees given in [KPBG94] and the *fair composition* technique given in [DIM93]. We prove the correctness of these algorithms under central daemon and restricted parallelism models. It is easy to show that there is no DUSSA for these problems under the maximal parallelism model. So, we use the correctness under restricted parallelism to obtain RUSSAs for these problems.

2. We show that there is no DUSSA for finding a *maximum matching* in an arbitrary graph even under the central daemon model.

3. We present a DUSSA for finding maximal independent sets in arbitrary graphs. We prove the correctness of the algorithm under central daemon and restricted parallelism models. It is easy to show that there is no DUSSA for this problem under the maximum parallelism model. The correctness of the algorithm under a restricted parallelism model enables us to obtain a RUSSA for this problem under the maximal parallelism model.

⟨precondition⟩→ ⟨action⟩;
□
⟨precondition⟩→ ⟨action⟩;
⋮
□
⟨precondition⟩→ ⟨action⟩;

Figure 1: Syntax of a program at any node

4. It is easy to show that under the maximal parallelism model, there is no DUSSA for producing a valid coloring of a planar graph. A DUSSA for 6-coloring planar graphs under the central daemon model can be derived from an algorithm in [GK93]. We observe the correctness of this algorithm under a restricted parallelism model and use this observation to develop a RUSSA for the problem.

5. We observe that techniques due to Angluin [Ang80] lead to general results that establish the non-existence of DUSSAs for a large collection of graph problems under any of the parallelism models. The problems in this collection include finding the parity of the number of nodes in a graph, finding the diameter of a graph, finding the eccentricities of the vertices of a graph and membership testing for various graph classes (for example, planar graphs, chordal graphs and interval graphs).

The remainder of this paper is organized as follows. In Section 2 we elaborate on our models for distributed systems. Section 3 addresses the coloring problems mentioned above. Section 4 discusses the impossibility result for maximum matching. Section 5 presents our maximal independent set algorithm. In Section 6 we discuss general results that establish the non-existence of DUSSAs for a large collection of graph problems.

2 Preliminary Definitions

2.1 Model of Distributed System

We model a distributed system as a network of processes or processors. The network is represented as a graph whose nodes represent the processors and whose edges represent communication or logical links. Henceforth we shall use the words 'processor' and 'node' interchangeably.

Since only uniform algorithms are considered in this paper, we describe a distributed algorithm by specifying the program run on a typical processor say, P_i. The syntax of the program executed by each node is as shown in Figure 1. Following [Dij74], each statement is referred to as a **rule**. For example, the first statement is Rule 1, and so on.

The semantics of such a notation is that whenever a processor executes, it selects the action corresponding to a rule whose precondition (or guard) evaluates to true. However, if two or more of the preconditions are true, then one of them is chosen nondeterministically. A processor is said to be **enabled** if at least one of the preconditions is true; otherwise, the processor is **disabled**.

The preconditions are Boolean functions of the state of a processor and the states of its neighbors. If all the preconditions can be evaluated deterministically, then we say that the algorithm is **deterministic**. If the value of a precondition depends on the outcome of a random experiment (e.g. generating a random number), then we say

that the algorithm is **randomized**. In an adversary oriented view of randomized algorithms, this definition might seem weak. If the adversary does not favor the preconditions in any of the nodes then it will not activate any of the processors. However, we randomize by making each processor generate a random number in $\{0, 1\}$ and then choosing all the processors that generated a 1 for execution. The probability that no processor will execute even though some are enabled is very low. Thus, the adversary (i.e., the scheduler) is not strong enough to prevent stabilization with high probability.

It is assumed that each processor has knowledge of its neighbors and this information is not corrupted even in the presence of transient errors. This assumption is not unreasonable because the neighbor information can be hardwired. We also assume that reading of the states of the neighbors is *atomic*. The execution of the action corresponding to the selected statement is also assumed to be atomic. In some literature this has been referred to as the **state-based** model as opposed to the **link-register** model [IJ93]. The state-based model is used by Dijkstra in [Dij74].

As in [SRR94] we consider three different kinds of adversaries, namely **central daemon**, **distributed daemon** and a new kind of adversary. Under the central daemon model, a central scheduler schedules one processor in every step. This model is used in [Dij74]. Under the maximal parallelism model, at any time step, *all* the enabled processors *may* execute their actions in parallel. The restricted parallelism model [SRR94] requires that only a certain proper subset of all the enabled processors may execute at any time step. It is difficult to implement this model in practice. However, we introduced this model for theoretical reasons. In [SRR94] we considered two varieties of restricted parallelism, namely *pairing restricted parallelism* (where, no pair of nodes which are adjacent can execute together) and *subset restricted parallelism* (where if more than one processors are enabled, any proper subset set of all enabled processors can execute together, but not all of them). The pairing restriction has the non-interfering property [BGW89]. We showed that starting from a DUSSA under a suitable restricted parallelism model, it is possible to obtain an RUSSA which stabilizes with probability 1 under the maximal parallelism model.

All of our results are for **anonymous** networks, where the processors do not have any identification numbers. It is also assumed that the processors in the network do not have any knowledge regarding the number of nodes in the network. In our correctness proofs, we refer to the processors by numbers, but that is just for notational convenience. In distributed systems where processors have unique identification numbers, it is often easier to design distributed algorithms for many problems. (However, this is not always the case [IR81].) Designing distributed algorithms for anonymous networks is more important for the following reasons. Since the number of nodes in the network is not known a priori, the memory required for storing the unique ID is not known. If there is an a priori bound on the number of nodes in the network, we cannot allow arbitrary dynamic addition of nodes to the system. However, if an arbitrary growth in the number of nodes must be accommodated, it must be possible to dynamically increase the space needed for storing the IDs.

The graph theoretic definitions used in this paper can be found in [Har69, Gol80].

2.2 Nature of Impossibility Results

Two types of impossibility results concerning DUSSAs have appeared in the literature.

1. There is no DUSSA for a problem even under the weakest computation model, namely central daemon. Examples of such problems include mutual exclusion in a bidirectional token ring of composite size [Dij74, BP89], anonymous leader election [Ang80], constructing a spanning tree of an anonymous connected graph [Ang80], 2-coloring an anonymous bidirectional even ring [SRR94], and 2-coloring an anonymous bipartite graph (this

paper). For these problems, the randomization strategy of [SRR94] cannot be used to obtain an RUSSA under maximal parallelism. Several problem-specific randomization techniques have been developed by various researchers. For example, Herman gave a randomized self-stabilizing mutual exclusion for an odd size token ring that self-stabilizes with probability 1 [Her90]. In [SRR94] we obtained a 2-coloring algorithm for even rings as a corollary of Herman's result. Other researchers have developed randomized algorithms for leader election [DIM91] and spanning tree construction [AK93].

2. There is a DUSSA for a problem under weaker models such as central daemon and some form of restricted parallelism, but there is no DUSSA for the problem under maximal parallelism. Examples of such problems include 2-coloring of an anonymous unoriented even chain and 3-coloring an anonymous bidirectional ring [SRR94]. In this paper, we point out several additional problems in this category; for example, finding a maximal independent set of an anonymous network, 2-coloring an anonymous odd-degree complete bipartite graph, and obtaining a valid coloring of an anonymous planar graph. In these cases, RUSSAs under maximal parallelism can be obtained using the technique of [SRR94] if a DUSSA can be designed under a suitable restricted parallelism model.

2.3 Techniques Used in Proving Impossibility Results

The impossibility results presented in this paper are based on two types of arguments. One type of argument relies on the impossibility of symmetry breaking. The second type of argument is based on covering relations between graphs, as introduced in [Ang80]. In both types of arguments, we also rely on the fact that a self-stabilizing algorithm must enable the system to reach a legitimate state from an arbitrary initial state.

In arguments based on symmetry breaking, we begin with a symmetric initialization of the system and show that there is an adversarial scheduling strategy that prevents the algorithm from breaking the symmetry. In arguments based on covering relations, we show two topologies such that if an algorithm achieves stabilization for one topology, it cannot achieve stabilization for the other topology. Here also, we specify suitable initial states, say S_1 and S_2, for the two topologies. We then argue that if the algorithm achieves stabilization for the first topology starting from state S_1, then there is an adversarial scheduling strategy that prevents the algorithm from achieving stabilization for the second topology starting from state S_2.

We note that because self-stabilization requires an algorithm to bring a system to a legitimate state starting from an arbitrary initial state, the impossibility proofs here are much simpler. The impossibility proofs for distributed algorithms given in [Ang80] involve much deeper arguments as they do not rely on arbitrary initialization.

3 Coloring Results

3.1 2-Coloring Bipartite Graphs

3.1.1 Impossibility result for Odd Degree Bipartite Graphs

In [SRR94] we proved the impossibility of DUSSA for 2-coloring an arbitrary anonymous even ring under all the parallelism models. It immediately follows that there is no 2-coloring DUSSA for arbitrary anonymous bipartite graphs. Therefore, it is of interest to identify subclasses of bipartite graphs for which DUSSAs may be designed. Since the general impossibility result depended on even rings (which are even-degree bipartite graphs), it might seem

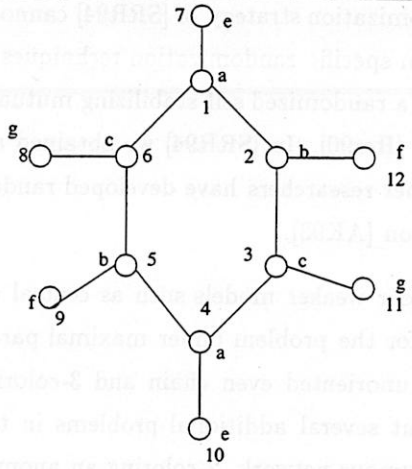

Figure 2: The initial configuration of the odd-degree bipartite graph in the proof of Proposition 3.1

that for the class of odd-degree bipartite graphs, DUSSAs may exist. However, the following proposition shows that such is not the case.

Proposition 3.1 *There is no DUSSA even under the central daemon model for 2-coloring an arbitrary odd-degree bipartite graph.*

Proof: Consider the anonymous odd-degree bipartite graph as shown in Figure 2. We denote the processors by numbers for notational convenience. (They are not available to the algorithm.) Let us partition the nodes of the graph into six disjoint classes namely, $\{(1,4),(2,5),(3,6),(7,10),(8,11),(9,12)\}$. Our choice of these classes is motivated by the following properties of these classes.

1. No two nodes in the same class are adjacent to each other. Hence, if a node changes its state, it does not have any immediate impact on the other node in its class.
2. No two nodes in the same class have a common neighbor.
3. If (p_1, p_2) is a class then either both p_1 and p_2 have three neighbors or both have one neighbor only.
4. For any class (p_1, p_2), the classes that contain neighbors of p_1 also contain the neighbors of p_2.

Now assume that there is a DUSSA for 2-coloring any odd degree bipartite graph. For the initialization shown in Figure 2 (states being indicated by letters a,b,c,e,f,g not necessarily all distinct), the members of the same class have the same initial state. However, in a valid 2-coloring of this graph, the members of the same class cannot have the same color. As a result, the initial state shown in Figure 2 is not a valid 2-coloring.

Now consider a central daemon adversary which schedules two nodes in the same class in succession all the time. Notice that by properties 3, 4 and 5 above, a member of a class is enabled, if and only if the other member is also enabled at the beginning and after an even number of steps. As a result, such an adversarial schedule is possible. Further in such a schedule, the members of the same class have the same state after an even number of steps. Notice, that such a *fair* schedule exists. Thus there is an infinite computation in which after an even number of steps (for each even number), the members of the same class are in the same state. By property 6, that implies under such an

1. $\langle Color = MajorityColor \rangle \rightarrow \langle Color := \overline{Color} \rangle$;

Figure 3: Program for any processor in Algorithm Odd-Complete

adversarial scheduling, the graph will never be 2-colored. □

Proposition 3.1 raises the question of whether there are subclasses of bipartite graphs for which DUSSAs can be designed under the central daemon model. We now present two such subclasses, namely odd-degree complete bipartite graphs and trees.

3.1.2 Coloring Complete Odd-degree Bipartite Graphs

In Figure 3 we present a DUSSA for 2-coloring a complete odd degree bipartite graph, denoted by K_{d_1,d_2}, where d_1, d_2 are odd positive integers. In the following discussion, we refer to the two disjoint subsets of nodes in the given complete bipartite graphs as the two sides. The algorithm is simple and contains only one rule. Each processor contains a one bit variable called *Color* which takes values from $\{0, 1\}$. *MajorityColor* is a function that each processor computes in its atomic step by looking at the *Color* of all its neighbors; if the number of neighbors with *Color* 0 is more than the number of neighbors with *Color* 1, then the *MajorityColor* is 0 otherwise it is 1. Note that each node has an odd number of neighbors and thus *MajorityColor* is well defined at each node. Since the graph is a complete bipartite graph K_{d_1,d_2}, when a node on one side computes *MajorityColor*, it computes the majority color on the other side. The correctness of the algorithm is an immediate consequence of the following two lemmas whose proofs are straightforward.

Lemma 3.2 *If the majority color on the two sides are complementary to each other, then under maximal parallelism Algorithm Odd-Complete self-stabilizes with the corresponding colors on each side.* □

Lemma 3.3 *If the majority colors on both sides are the same, then under central daemon they will eventually differ by the execution of Algorithm Odd-Complete.* □

Lemma 3.2 and Lemma 3.3 together show that starting from an arbitrary initial coloring of an odd degree complete bipartite graph, the algorithm self-stabilizes with a valid 2-coloring under the central daemon model.

It can be shown that Algorithm Odd-Complete also stabilizes under the following restricted parallelism model: No two adjacent nodes may execute together during any step. We call this restriction the **pairing restriction** [SRR94]. The identification of this restriction enables us to randomize the algorithm. Each processor generates a random number r such that $Prob(r = 0) = p$ and $Prob(r = 1) = 1 - p$, and decides to execute if it generates 1. Then under maximal parallelism the probability that all the nodes on one side of K_{d_1,d_2} generate 1 and all the nodes on the other side generates 0 is given by $\beta = p^{d_1}(1-p)^{d_2} + p^{d_2}(1-p)^{d_1}$. The probability that this event does not happen in a maximal parallel step is $(1 - \beta)$, which is less than 1. As a result, the probability that in k subsequent steps this event does not happen is $(1 - \beta)^k$ which decreases towards 0 as k increases. Thus the probability that eventually this restriction is imposed at least once tends to 1. The majority color on both sides may remain the same until the time when this restricted parallelism gets imposed; once the restriction is imposed the majority on the two sides are different. Then by Lemma 3.2, the system self stabilizes with valid a 2-coloring. Therefore we conclude:

$\langle \exists_{l \in Ne}((h(l) \geq h) \wedge color(l) = color \rangle \rightarrow \langle color := \overline{color} \rangle;$

Figure 4: Program for any processor in Algorithm Tree-color

Proposition 3.4 *Under maximal parallelism there is an RUSSA that self-stabilizingly 2-colors a complete odd-degree bipartite graph with probability 1.* □.

3.1.3 2-Coloring Trees

Trees are a subclass of bipartite graphs for which it is possible to design a 2-coloring DUSSA under central daemon. We use a *fair composition* [DIM93] of the DUSSA (called algorithm Tree-color) in Figure 4 with the DUSSA for center finding given in [KPBG94]. Henceforth, we refer to the center finding algorithm in [KPBG94] as Algorithm KPBG. In Algorithm KPBG each node of the tree has a variable h, and for each node v, the set of values of h corresponding to all its neighbors is denoted by $N_h(v)$. We denote the maximum value in $N_h(v)$ by $max\, N_h$.

When Algorithm KPBG stabilizes, each center node has $h \geq max\, N_h$. When there are two centers, their h values satisfy the condition $h = max\, N_h$. For all other nodes $h < max\, N_h$. We assume that each node contains a one bit variable called *color*. A fair composition of algorithm Tree-color with Algorithm KPBG will guarantee that after the composite algorithm stabilizes the tree is properly 2-colored.

It can be shown that a fair composition of KPBG and Algorithm Tree-color shown in Figure 4 will self stabilize with a valid 2-coloring of the tree. In the description of the algorithm Tree-color in Figure 4, we use symbol l to denote the port numbers through which a neighbor may be connected. Since we are designing algorithms for anonymous networks, no node has any globally known identifiers. However, each node can distinguish locally, the port numbers through which its neighbors are connected. Also Ne denotes the set of all port numbers through which some neighbor is connected.

We now informally explain how the fair composition of KPBG and algorithm Tree-color achieves self-stabilization. After KPBG stabilizes, there are one or two nodes which are centers. These nodes have the property that their h values satisfy the condition $\forall_{l \in Ne} h \geq h(l)$. If there are two centers then they will have the same h value. By the rule in Figure 4, if there are two centers (they must be adjacent [Har69]) which have the same color, whichever executes first under the central daemon model, will complement its color. subsequently, the immediate neighbors of a center change their colors to the complement of the corresponding center's color. Then the ones which are at a distance of two from the centers set their colors correctly and so on. A proof that when the fair composition stabilizes, the coloring is a valid 2-coloring of the tree can be given using induction on the distances of the nodes from the centers. This proof is omitted here due to space limitations.

Proposition 3.5 *A fair composition of KPBG and Tree-color self-stabilizes with a valid two coloring of the tree under the central daemon model.* □

Under maximal parallelism, the above algorithm may not self-stabilize. Consider the following scenario. The two centers of the tree have the same color and all the processors are executing together under maximal parallelism. The two centers will never have different colors under such a scenario. However, if we apply our randomization strategy, so that each node executes with probability $p < 1$, the probability that the two center nodes both execute together

or abstain from executing during a step is $p^2 + (1-p)^2$ which is less than 1. Thus the probability that their colors remain the same after k steps is given by $(p^2 + (1-p)^2)^k$. This quantity decreases with k and tends to 0. Thus with probability 1, the two centers eventually complement their colors and the algorithm self-stabilizes subsequently. We thus conclude:

Proposition 3.6 *There is an RUSSA obtainable from the DUSSA under central daemon for the 2-coloring of a tree which self-stabilizes with probability* 1. □

3.2 6-Coloring Planar Graphs

It is easy to see that there is no DUSSA for producing a valid coloring of an anonymous planar graph under maximal parallelism. In [GK93], a DUSSA for 6-coloring of planar graphs under the central daemon model has been presented. The algorithm presented there has two phases. In one phase, the edges of the graph are directed so that no node has out-degree more than 5. The second phase 6-colors the directed graph that results after the first phase stabilizes. They use a *fair composition* [DIM93] of the two phases to obtain a DUSSA for the 6-coloring problem. In the first phase, unique identifiers for the nodes are used to break symmetry. However, in a proof of correctness (attributed to J. Misra in [GK93]) it is noted that it is possible to carry out this phase without assuming unique identifiers. The second phase does not use node identifiers.

It can be shown that once the edge directions as described above are achieved, the algorithm will stabilize with a valid 6-coloring under a **neighborhood-restricted** (where a node executes if and only if none of its neighbors execute) parallelism. Then as described in the context of maximal independent set (see Section 5), it is possible to devise an RUSSA for the problem under maximal parallelism.

Proposition 3.7 *There is an RUSSA for 6-coloring anonymous planar graphs under maximal parallelism.* □

4 Impossibility of Self-Stabilizing Maximum Matching

In this section we consider the problem of developing DUSSAs for finding a maximum matching for arbitrary anonymous networks.

Definition 4.1 *A* **matching** *in a graph G is a set of edges $M \subseteq E$ such that no two edges in M are adjacent. A matching M is a* **maximum matching** *if M has the highest cardinality among all the possible matchings in G.*

The following proposition proves that there is no DUSSA for the maximum matching problem.

Proposition 4.2 *There is no DUSSA under the central daemon model for finding a maximum matching in an arbitrary anonymous network.*

Proof sketch: In [SRR94] we proved that there is no DUSSA for 2-coloring a bidirectional anonymous even ring. Suppose there is a DUSSA for finding a maximum matching for an arbitrary anonymous network. We can use the DUSSA to 2-color any even ring self-stabilizingly as follows. In an even ring of size $2n$, the size of the maximum matching is n. (Such a matching consists of every other edge in the ring.) We run a fair composition of the DUSSA for maximum matching and the following algorithm. A node can find whether it is at the tail or the head of a matched edge (recall that the ring is directed). Each tail node chooses color 0 and each head node chooses color 1. It is easy to see that fair the composition technique will guarantee that this will self-stabilizingly produce a valid 2-coloring of any even ring, contradicting the impossibility result proved in [SRR94]. □

1. $\langle AllZero \wedge Ind = 0 \rangle \rightarrow \langle Ind := 1 \rangle$;
□
2. $\langle \text{not}(AllZero) \wedge Ind = 1 \rangle \rightarrow \langle Ind := 0 \rangle$;

Figure 5: Program for any processor in Algorithm MaxInd

5 Self-Stabilizing Maximal Independent Set

Definition 5.1 *Given an undirected graph $G(V, E)$, an* **independent set** *of G is a subset V' of nodes such that there is no edge between any pair of nodes in V'. An independent set V' is* **maximal** *if no proper superset of V' is also an independent set.*

In this section we consider the problem of developing a DUSSA for constructing a **maximal independent set** of an anonymous network. In other words, we want to develop a DUSSA such that when the system stabilizes, a variable in each processor indicates whether that processor is in the constructed maximal independent set or not. For that purpose we assume that each processor maintains a one bit variable *Ind* that takes on values from $\{0, 1\}$. After stabilization, if a processor finds that its $Ind = 1$ then it is in the constructed maximal independent set, otherwise not.

First we note the following impossibility result under maximal parallelism. This can be proven by using the ring C_3 with suitable symmetric initialization.

Proposition 5.2 *There is no DUSSA for the maximal independent set problem for arbitrary anonymous networks under the maximal parallelism model.* □

Our DUSSA for this problem under central daemon is shown in Figure 5. The algorithm uses the predicate $AllZero(v) \equiv \forall_{x \in neighbors(v)}(Ind(x) = 0)$. For a node v, *AllZero* is true if currently none of its neighbors are in the maximal independent set. Although the processors do not have unique identifiers, we use $Ind(v)$ or $AllZero(v)$ etc. for notational convenience.

Now we prove that this algorithm **MaxInd** stabilizes under central daemon and when it stabilizes (i.e., when all the processors are disabled) it constructs a *maximal independent set.*

Lemma 5.3 *Under the central daemon model, if a processor executes Rule 1 of the algorithm MaxInd, it will be permanently disabled and all its neighbors will also be permanently disabled.*

Proof: A processor P can execute Rule 1 only if all its neighbors have $Ind = 0$ (otherwise $AllZero(P)$ would not be true). When P sets its *Ind* to 1, all of its neighbors have $not(AllZero)$ true but since none of them has $Ind = 1$ they all are disabled. Since all the neighbors get disabled, they cannot change their *Ind* value any more, thus P also cannot have its guard true ever again. □

Lemma 5.4 *Algorithm MaxInd stabilizes in $O(n)$ steps under the central daemon model, where n is the number of processors in the network.*

Proof: By Lemma 5.3, if a processor executes Rule 1 then it is permanently disabled. If a processor executes Rule 2, then its *Ind* becomes 0. So the only rule that it might execute again is Rule 1. But by Lemma 5.3, if it executes rule

1 again, it will be permanently disabled. Thus each node may execute at most twice. Hence the maximum number of steps needed for the algorithm to stabilize is $2n$. □

Using the above lemma, it can be shown that when **MaxInd** stabilizes (i.e., none the processors is enabled) a *maximal* independent set is constructed. The proof is omitted due to space limitations.

Lemma 5.5 *When the guards of all the rules of Algorithm Max-Ind are false for each processor then the processors with $Ind = 1$ form a maximal independent set for the given network.* □

Proposition 5.6 *Algorithm MaxInd is self-stabilizing and constructs a maximal independent set for any arbitrary anonymous network in $O(n)$ steps under the central daemon model.*

Now we identify the following restriction on parallelism which will allow the same algorithm to self-stabilize. Suppose we impose a restriction that in each step, there is at least one node which executes with none of its neighbors executing (we call it a **neighborhood-restriction**) then the following proposition shows that **MaxInd** will self-stabilize under this restricted model of parallelism.

Proposition 5.7 *Under* **neighborhood-restricted** *parallelism, Algorithm MaxInd self-stabilizes with a maximal independent set for an arbitrary anonymous network.*[2]

Proof sketch: Let us define two predicates on the nodes of the graph. For any node v in the graph let

$$Strong_1(v) \equiv Ind(v) = 1 \wedge \forall_{u \in neighbors(v)} Ind(u) = 0$$
$$Strong_0(v) \equiv Ind(v) = 0 \wedge \exists_{u \in neighbors(v)} Strong_1(u)$$

Informally a node is a $Strong_1$ node at a given time if its Ind is 1 and all its neighbors have their Ind value as 0. A node is a $Strong_0$ node if its Ind value is 0 and at least one of its neighbor is a $Strong_1$ node. After a maximal independent set is constructed, each node is either a $Strong_1$ node or a $Strong_0$ node.

In neighborhood-restricted parallelism, the scheduler chooses at least one neighborhood around a node in each step such that the node executes and its neighbors do not execute. If the scheduler chooses node v at a particular step then we call that node **blessed** for that step. When a node becomes blessed, if it executes Rule 1, then it becomes permanently disabled and becomes a $Strong_1$ node and all its neighbors become $Strong_0$ nodes. However, if it executes Rule 2, then it may need to be blessed again in a subsequent step unless one of its neighbor is already a $Strong_1$ node, or one of its neighbors become a $Strong_1$ node. But if it becomes blessed again then it executes Rule 1 and never executes again. Thus, each node needs to be blessed at most twice for self-stabilization. Given a fair neighborhood-restriction scheduler, each enabled node will be blessed twice if necessary, and the system will self-stabilize. □

Proposition 5.7 enables us to use randomization under maximal parallelism to implicitly implement such a scheduler. Suppose we provide in each processor a random number generator. Each time a processor wants to execute, it produces a random number r in $\{0, 1\}$ and if the number is 1 then it executes. Let $Prob(r = 1) = p$ and $Prob(r = 0) = 1 - p$. Consider an enabled node v of degree d. The probability that at a maximal parallel step, v

[2] Neighborhood restriction implies non-interference as defined in [BGW89]. Hence, any parallel execution can be serialized and this proves this proposition directly. However, the proof we give is helpful in understanding how the randomization works.

generates random number 1 and all its enabled neighbors generate 0 is at least $\beta = p(1-p)^d$. This is a lower bound on the probability that in a maximal parallel step, an enabled node v is **blessed**. Thus the probability that it is not blessed in a maximal parallel step is at most $(1-\beta)$. A weak node v will become a *Strong* node, if it is blessed at most twice. The probability that in k subsequent maximal parallel steps in which it is enabled but remains weak (i.e.,it does not become blessed twice) is at most $(1-\beta)^k + k\beta(1-\beta)^{k-1} = (1-\beta)^{k-1}(1+(k-1)\beta)$. As k increases this quantity decreases towards 0. So the probability that the node v becomes a strong node approaches 1 as the number of parallel steps increases.

If the algorithm does not self-stabilize, then there must be one node that executes infinitely often. But if that node ever becomes a *Strong* node then it cannot execute infinitely often. But as argued above, the probability that it does not become a strong node decreases with the increase in the number of steps. Thus the probability that there is no infinite execution increases towards one with increasing number of steps. We thus conclude:

Proposition 5.8 *There is an RUSSA for the maximal independent set problem under the maximal parallelism model which self-stabilizes with probability* 1. □

6 Impossibility Results

In Section 2, we mentioned two kinds of impossibility results for DUSSAs for anonymous networks. Impossibilities under maximal parallelism are easier to establish because maximal parallelism is a stronger adversary. Such impossibility results are established by finding a graph and a suitable illegitimate initial state such that on that graph, no deterministic algorithm can take the system to a legitimate state. In previous sections, we showed impossibilities of existence of DUSSAs under maximal parallelism in this manner.

Impossibility results under weaker adversaries such as central daemon allow us to reach stronger conclusions about the difficulty of self-stabilization for a problem. In [Ang80], a number of techniques for proving the nonexistence of deterministic distributed algorithms for various graph problems are presented. Although [Ang80] does not mention "self-stabilization", most of the proof techniques are applicable in the context of self-stabilization more easily. In [Ang80], the impossibility of leader election and specific graph property recognition problems are presented using graph theoretic methods. The main proofs in [Ang80] involve deeper ideas because impossibility results for general distributed algorithms cannot rely on arbitrary initialization. Due to their generality, these proof techniques can be readily applied in the context of self-stabilization to obtain impossibility results in an easy manner. We present two such results here.

A predicate $\mathcal{P}$ on natural numbers is *nontrivial* if there are two natural numbers i and j such that $\mathcal{P}(i)$ is true and $\mathcal{P}(j)$ is false. Examples of such predicates are $\mathcal{P}(n) \equiv n$ is even, or $\mathcal{P}(n) \equiv n \geq k$ for some fixed k, etc. Suppose we want a DUSSA for a problem of the following kind: A network of processors should decide if the number of nodes in the graph satisfies property $\mathcal{P}$. For example, suppose each node wants to know if the number nodes in the graph is odd or even, and accordingly set a flag to 0 or 1. We call this problem the self-stabilizing **parity determination problem** for general graphs. For any such property $\mathcal{P}$ of the number of nodes of the graph, we call the corresponding problem the self-stabilizing $\mathcal{P}$-determination problem. Our proof of the following proposition illustrates a proof technique in [Ang80].

Proposition 6.1 *Let $\mathcal{P}$ be a nontrivial predicate on natural numbers. Then there is no DUSSA for the $\mathcal{P}$-determination problem for general networks even under the central daemon model.*

Proof sketch: Suppose $\mathcal{P}(i)$ is true and $\mathcal{P}(j)$ is false. Consider the ring C_i of size i and the ring C_j of size j. The $\mathcal{P}$ determination DUSSA (if one exists) should stabilize in both the rings with the flags at the nodes set to 1 in C_i, and to 0 in C_j. Now consider the ring of size C_{i*j}. Consider an arbitrary initial state of C_i and a corresponding computation sequence under a central daemon. We can now construct a suitable initial state for C_{i*j} and an adversarial scheduling strategy under central daemon such that after stabilization, the flags at the nodes of C_{i*j} will be set to 1. By considering an arbitrary initial state of C_j and the corresponding computation sequence in C_j, we can construct a suitable initial state and another adversarial schedule such that C_{i*j} will stabilize with all flags set to 0. This shows that under two distinct initial states, under adversarial scheduling by the central daemon, the same size ring stabilizes with contradictory decisions about its size. □.

Proposition 6.1 shows that there is no DUSSA under the central daemon model which determines the parity of the size of the graph, for testing whether the size is greater than some fixed integer k, etc.

Another corollary of the results in [Ang80] concerns the recognition of graph properties. Suppose $\mathcal{T}$ is a predicate on graphs. Examples of such predicates are G is bipartite, G is an interval graph, G is a chordal graph etc. (See [Gol80] for definitions of these graph properties.)

Given any graph property $\mathcal{T}$, we call the corresponding self-stabilizing problems the $\mathcal{T}$-determination problem.

Proposition 6.2 *Let $\mathcal{T}$ be a predicate on graphs. Suppose there are two rings of size i and ki such that $\mathcal{T}(C_i)$ is true and $\mathcal{T}(C_{ki})$ is false. Then there is no DUSSA for deciding the property $\mathcal{T}$ for general network even under central daemon.*

Proof sketch: The proof involves the construction of a suitable initial state in C_{ki} and an adversarial schedule for it from an arbitrary state and a corresponding computation to stabilization in C_i. It can be shown that in both C_i and C_{ki} the system stabilizes with the same decision. □

Examples of $\mathcal{T}$-determination problems for which Proposition 6.2 applies are the following: determining whether a graph is bipartite (C_3 is not bipartite but C_6 is bipartite), determining whether a graph is an interval graph (C_3 is an interval graph but C_6 is not), determining whether a graph is chordal (C_3 is chordal but C_6 is not) and determining whether a graph is complete (C_3 is complete but C_6 is not) etc.

From the above proposition, we can derive corollaries that apply to problems involving a function computation rather than a $\mathcal{T}$-determination problem. To state these corollaries we need the following definitions.

Definition 6.3 *In a graph $G(V, E)$, let $d(i, j)$ be the length of a shortest path between vertices i and j.*

1. *$e(i) = max\{d(i, j) \mid j \in V\}$ is called the* **eccentricity** *of the vertex i.*
2. *$center(G) = \{i \in V \mid e(i) \leq e(j) \forall j \in V\}$ is the set of* **centers** *of G.*
3. *The maximum distance between any pairs of vertices in the graph is called the* **diameter** *of the graph, i.e., $diameter(G) = max_{(i,j)}\{d(i, j) \mid (i, j) \in V \times V\}$*

Corollary 6.3.1 *There is no DUSSA for computing the diameter of an anonymous ring, even under the central daemon model.* □

Proof: For a fixed integer k, let $\mathcal{T}(C)$ be the predicate "the diameter of C is k". The result follows from Proposition 6.2. □

Since in any ring, the eccentricity of any vertex is the diameter of the ring, we also obtain:

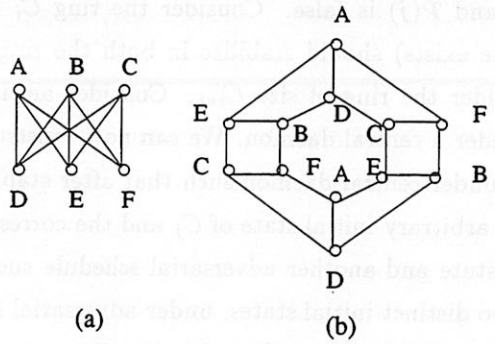

Figure 6: The initial configuration of $K_{3,3}$ and the other graph in the proof of Proposition 6.4

Corollary 6.3.2 *There is no DUSSA for computing the eccentricities of the vertices of an anonymous ring, even under the central daemon model.* □

Now since the centers of a graph are those vertices that have the minimum eccentricities, the above results raise the question whether there is a DUSSA for finding all the centers of an arbitrary anonymous network. However, it is certainly impossible to develop a DUSSA which finds a *single* center in an arbitrary graph because such a DUSSA will also serve as a DUSSA for leader election for anonymous arbitrary networks which is provably impossible [Ang80].

However, some other impossibility proofs will need other type of arguments. For example, consider the proof of nonexistence of a DUSSA for determining whether a graph is planar. It can be shown that there is a planar graph G (shown in Figure 6(b) which appeared in [AG81]) such that we can construct a suitable initialization, an adversarial schedule for it from an initial state (initialization is also shown in the Figure 6 where A,B,C,D,E,F are the initial states, not necessarily distinct) and a corresponding computation sequence for the complete bipartite graph $K_{3,3}$ (which is non-planar). The DUSSA (if one exists) will have to reach the same decision in both G and $K_{3,3}$ which is a contradiction. This leads to the following result:

Proposition 6.4 *There is no DUSSA for determining whether an arbitrary anonymous network is planar even under the central daemon model.* □

Acknowledgement: We are grateful to the anonymous reviewers for valuable suggestions. In particular, we considered the maximum matching and diameter finding problems at the suggestion of one of the reviewers.

References

[AG81] D. Angluin and A. Gardiner. Finite common covering of pairs of regular graphs. *Journal of Combinatorial Theory, Series B*, 30:184–187, 1981.

[AK93] S. Aggarwal and S. Kutten. Time optimal self-stabilizing spanning tree algorithm. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, LNCS, 1993.

[Ang80] D. Angluin. Local and Global properties in networks of processes. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, University of California at Los Angeles, 1980. ACM.

[BGM89] J.E. Burns, M. G. Gouda, and R. E. Miller. On relaxing interleaving assumption. In *Proceedings of MCC Workshop on Self-Stabilization*, Texas, Austin, 1989. MCC.

[BGW89] G. M. Brown, M. G. Gouda, and C. L. Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38(6):845–852, June 1989.

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

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

[DIM91] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic leader election. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, pages 163–180. Springer-Verlag, LNCS 579, 1991.

[DIM93] S. Dolev, A. Israeli, and S. Moran. Self-stabilizing Dynamic Systems Assuming Read/Write Atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[EK89] M. Evangelist and S. Katz, editors. *Proceedings of the MCC Workshop on Self-Stabilizing Systems.* MCC, Austin, TX, 1989.

[GK93] S. Ghosh and M. H. Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, 7(1):55–59, November 1993.

[Gol80] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press Inc., New York, NY, 1980.

[Har69] F. Harary. *Graph Theory.* Addison-Wesley Publishing Co., Reading, Mass., 1969.

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

[Hoe94] J.-H. Hoepman. Uniform deterministic self-stabilizing ring-orientation on odd-length rings. Technical Report CS-R9423, Computer Science Department, CWI, Amsterdam, April 1994.

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

[HWT94] S. T. Huang, L. C. Wuu, and M. S. Tsai. Distributed execution model for self-stabilization. In *Proceedings of 14th International Conference on Distributed Computing Systems*, pages 432–439, 1994.

[IJ93] J. Israeli and M. Jalfon. Self-stabilizing ring orientation. *Information and Computation*, 104(2):175–196, 1993.

[IR81] A. Itai and M. Rodeh. Symmetry breaking in distributive networks. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, pages 150–158, Nashville, Tennessee, 1981. IEEE Computer Society.

[KPBG94] M.H. Kaarata, S.V. Pemmaraju, S.C. Bruell, and S Ghosh. Self-stabilizing Algorithms for Finding Centers and Medians of Trees. TR- 94-03, University of Iowa, 1994.

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

[SRR94] S. K. Shukla, D. J. Rosenkrantz, and S. S. Ravi. Developing Self-Stabilizing Coloring Algorithms via Systematic Randomization. In *Proceedings of the 1st International Workshop on Parallel Processing*, pages 668–673, New Delhi, 1994. Tata McGraw-Hill.

Paper Number 8

On the Self-Stabilization of Processors with Continuous States

H. James Hoover

On the Self-Stabilization of Processors with Continuous States

H. James Hoover*

1 Motivation and Context

We are interested in the following question:

What is an appropriate model of computation for the continuous world?

The world we have in mind is asynchronous, states are continuous values, signals take time to propagate, and transitions between distinct state values take time. We seek a model of computation that captures the essence of implementable computation but that is just above those that actually model the physics of our computational devices.

We interested in this because we believe that it is useful to view the world as continuous when it comes to low level issues of models for distributed computation. Issues such as what is an atomic operation, what does asynchronous mean, and what is state, become especially important when one considers the problem of discrete self-stabilization. Thus this paper asks:

What are the continuous analogs of discrete self-stabilizing systems for networks of processors with continuous states?

The self-stabilizing systems we have in mind are those that have multiple equilibrium points, or have a dynamic steady-state behaviour. In a sense, such systems must "choose" to return to one of many equilibrium behaviours, and are often called upon to break symmetry to do so. This is a considerable extension to the usual domain of control theory which is geared toward keeping a system close to a single, externally specified, target steady-state.

If one believes that discrete systems are abstractions of a continuous world, then the following question also motivates this work:

Does the design of self-stabilizing protocols for the continuous case yield any insight into the models and protocols for the discrete case?

For example, even a fundamental notion such as "state" is not always clear in a self-stabilizing computation. Is it simply the values of the shared variables, or does it include the internal state of the protocol? If the latter, then one has to worry about the protocol "knowing" the state of the shared variable. For example, the protocol may write a 0 to a shared variable, and then enter an internal state assuming that 0 is in the shared variable. Meanwhile, a transient fault changes the shared variable to a 1, and thus the internal state of the protocol is inconsistent with the external value of the shared variable.

The general content of this paper is exploratory, not definitive. We define two notions of continuous self-stabilization: static (in which the system reaches a legitimate state and remains there) and dynamic (in which the system enters a legitimate pattern of state transitions and remains there). We then introduce a model of continuous computation that is a natural extension to the well-studied model of arithmetic circuits. Within this model we implement continuous versions of the following discrete distributed problems: two processors agreeing on distinct identities; orienting a ring; and the token passing problem.

In all cases our solutions will be *uniform*, that is, all processors will have the same state transition function, have no knowledge of their processor number, and have no global knowledge of the system they are in (eg. number of processors in a ring).

*Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1, e-mail address: hoover@cs.ualberta.ca, Web address: http://www.cs.ualberta.ca/

2 Introduction and Basic Notions

The prototypical distributed system is a collection of discrete state machines connected with a network. Such a system is *self-stabilizing* if it has the property that regardless of its current state, it will eventually enter and remain within a well-defined set of stable states. Self-stabilizing systems capture one aspect of fault tolerance — if some processors fault into an undesirable state then given sufficient fault-free time the systems will return to a desirable state. A restricted form of self-stabilization to a steady-state equilibrium point is a long-standing notion in control theory. The richness of the self-stabilization problem appears when it is applied to computing systems, with the first such case generally credited to Dijkstra [4, 5].

In reasoning about distributed systems, it is usually assumed that state changes cannot be observed while they are occurring. That is, the operations that change the state are atomic, and considered to occur instantaneously. Reasoning about the system then proceeds by analyzing how the system behaves when the execution of the atomic operations are scheduled by various styles of demon. For example the central demon (introduced by Dijkstra in [4]) schedules only one atomic shared variable update operation at a time, while the distributed demon (introduced by Brown, Gouda, and Wu in [1]) is permitted to schedule simultaneous atomic update operations from different processors. The highly adversarial read/write demon (introduced by Dolev, Israeli, and Moran) in [6] breaks shared variable update operations into read and write phases, and allows interleaving of the phases among processors.

The behaviour of a distributed algorithm subtly depends on the interaction between the granularity of its atomic operations and the nature of the scheduling demon. For example, under the distributed demon an algorithm may work properly if its basic atomic operation consists of a read of a shared variable followed by a write of a shared variable, but it may fail when this basic operation is subdivided into a atomic read operation followed by an atomic write operation. These kinds of issues are mentioned in [1], [7], [10], [12].

Given this diversity of models, it is reasonable to ask if there is some more fundamental underlying model. It is not obvious that a continuous model is the correct one. For example, it is not at all clear what the continuous analog of the scheduling demon should be.

2.1 Stability

Stability is a property of the *steady-state* response of a system. That is, it deals with the how the system behaves as time approaches infinity from its initial conditions.

There are two notions of stability for self-stabilizing systems: In the first kind, which we call *static stability*, the system reaches a legitimate configuration and remains there (for example, ring orientation). In the second kind, which we call *dynamic stability*, the system reaches a legitimate pattern of behaviour and continues to follow that pattern (for example, token passing).

To simplify the descriptions of the behaviour of a system it is useful to lump together states with equivalent meanings. This is especially true for continuous systems. For example, only the first component of a state vector may be important in deciding whether the state is legitimate, so all states with the same first component value could be lumped together. These sets of "equivalent" states we call targets.

Definition 2.1 *A* **target** *is any subset of the state space. A* **target space** *is any collection of targets.*

A system is in static stability if it has entered a target, and remains there:

Definition 2.2 *Let S be a target. A system is said to be* **statically stabilized** *at S if its current state and all future states are in target S.*

We use the term static, even though the system could be moving about inside the target and need not converge to a single point in the state space. This corresponds to the dynamical systems notion of a system being stable near an equilibrium point (see [8]).

Noise is crucial to the self-stabilization of continuous systems, so their behaviour must be expressed probabilistically.

Definition 2.3 *Let G be a target space. A system is* **Statically Self-stabilizing with respect to** *G if regardless of its current state, within an expected finite time (possibly depending on the system and current state), the system will statically stabilize at some target of G.*

Dynamic stability is much harder to define precisely, although the intuition is clear: the system must continue with its particular behaviour. For example, if it is passing a token clockwise around a ring, it cannot arbitrarily reverse direction, even though counter-clockwise passing may also be legitimate behaviour, and it cannot stop the passing process. Our problem is compounded by the fact that a continuous system may not be able to pass instantaneously between legitimate states, and instead we must describe how a particular *state trajectory* $q(t)$ encounters targets over time.

Definition 2.4 *Let G be a target space, and let $\rightarrow$ be a relation on G. The pair $B = (G, \rightarrow)$ defines a* **behaviour** *of the system. A sequence $S_0, S_1, S_2, \ldots$ of targets is an instance of behaviour B if each $S_i \in G$, and for all i, $S_i \rightarrow S_{i+1}$.*

A system is dynamically stabilized if it passes through exactly the targets of a behaviour:

Definition 2.5 *Let $q(t)$ be a state trajectory. The* **target sequence induced by** *$q(t)$ is the sequence of targets $S_0, S_1, \cdots$ such that $q(0)$ is in S_0, and for all $i \geq 0$ and $t \geq 0$, if $q(t)$ is in S_i, and $\Delta > 0$ is the smallest value such that $q(t + \Delta)$ is in a target different from S_i, then $q(t + \delta)$ is in S_{i+1}.*

Definition 2.6 *Let G be a target space. A system is said to be* **dynamically stabilized** *in behaviour B if the target sequence induced by the state trajectory beginning with the current state is an instance of behaviour B.*

Definition 2.7 *Let G be a target space, and B a behaviour. A system is* **Dynamically Self-stabilizing with respect to** *B if regardless of its current state, within an expected finite time (possibly depending on the system and current state), the system will dynamically stabilize in behaviour B.*

2.2 Discrete Model of Computation

No single model of discrete computation is common to all self-stabilization problems. For the purpose of this work, the following informal model of processors communicating by sharing registers will do.

Informal Definition 2.8 Discrete Model: *Processor inputs and outputs are integers (of fixed length), but interpreted as scaled fixed-point rational numbers of resolution ϵ, with $|\epsilon| < 1/2$. Processors can perform the usual arithmetic and logical operations, and in addition have a* `random` *operation that uniformly selects a value from the given set. A processor does atomic reads of its inputs, and atomic writes of its outputs. Between any two I/O operations, the processor computation is also atomic in the sense that it uses only local data, and so does not depend on the actions of any of the other processors. Scheduling of processors by the demon is in terms of the atomic I/O operations. If atomic operations are scheduled for two or more processors at the same time, then all read operations complete before any write operations begin. The* **state** *of a processor is the value of its outputs plus the position in the program just before the next I/O operation to be executed. A* **step** *of the computation is the simultaneous execution of one or more atomic operations. For the purpose of Definition 2.3, time is measured in steps.*

2.3 Continuous Model of Computation

Our goal is a computation model that is above the physical device layer. That is, we do not want to do a full simulation of a VLSI circuit. Instead we want a model that captures the (arguably) continuous nature of the various possible mechanisms for performing computations, be they electronic, mechanical, hydraulic, or pneumatic.

What does it mean to have a continuous computation? If we wish to eliminate operation atomicity, then we require both continuous time and continuous state. That is, our computation is described by a system of differential equations. The actual computation is the trajectory that the system follows over time, starting from given initial conditions. To pursue this approach we must address the following basic issues:

- What measurable quantities constitute the state of the system?
- What mechanism generates the instantaneous state change?
- What is a transient fault?
- What effects correspond to the activities of the scheduling demon?

The first issue is relatively simple to address. If a program for the continuous machine is a system of differential equations $\frac{d}{dt}\vec{x} = f(\vec{x}, t)$ then the state of the system is simply the vector of variables $\vec{x}$. A computation is the integration of this system with respect to time using an "analog" computer.

How the analog computer is "programmed" depends on how the instantaneous state change function f is represented? In particular, we need to be able to represent non-linear functions, for no computation involving decisions is possible without them.

One approach would be to build in special non-linear basis functions, for example the signum function or the delta function. Of course since these two functions are discontinuous, we would actually build in sets of increasingly accurate continuous approximations.

An alternative approach is to have a model that has a basic set of operations powerful enough to directly compute continuous approximations to the non-linear basis functions. Many possible approaches suggest themselves. For example one could use approximating polynomials, rational functions, or Fourier series. How do we chose among the alternatives?

The choice is simplified by two factors. The first is that there already exists one well-studied continuous model that is appropriate for representing f — arithmetic circuits over the reals. Such circuits consists of acyclic networks of $+$, $-$, $\times$, and $\cdot^{-1}$ gates. These circuits enable one to construct efficient (small circuit size) rational approximations to non-linear basis functions, in contrast to power series methods which typically converge too slowly to be computed efficiently. Special variants, called feasible-size-magnitude circuits have properties that enable their efficient numerical simulation (Hoover [9]).

Another factor that influences our choice of model is our desire to use mutual simulation in order to show equivalence between the discrete and continuous models under various conditions. This particular formulation is well suited to this, but further discussion is not appropriate here.

To complete our model of computation, we extend the notion of an arithmetic circuit over the reals (a circuit with $+$, $-$, $\times$, and $\cdot^{-1}$ gates) by relaxing the condition that the circuit be acyclic, and by adding another type of gate called an integration gate. An *integration gate* has a constant parameter x_i giving the initial condition of the gate at time $t = 0$, and a single input $\frac{d}{dt}x(t)$. At time t, the output of the gate is $x_i + \int_0^t \frac{d}{dt}x(t)\,dt$. Even though this model permits gates to have arbitrary real values, in practice no physical system is unbounded, and the circuit should be designed so that all gates, including the integration gate maintain bounded values over the computation.

Informal Definition 2.9 Continuous Model: *A* **continuous state circuit** *is a cyclic arithmetic circuit over the reals such that every cycle passes through at least one integration gate. The* **state** *of the computation at time t is the vector consisting of the outputs of all the integration gates in the circuit.*

It is important to note that the state of the circuit is defined by exactly the outputs of the integration gates. If f corresponds to the state change function in a discrete computation, then one can view integration gates as corresponding to memory.

A related model of analog computation has been around for some time. One of the first studies of the computational power of the General Purpose Analog Computer was by Shannon [15], which showed that GPACs compute exactly the solutions to algebraic differential equations. This proof had a flaw, corrected by Pour-El [14]. This was further corrected by Lipshitz and Rubel [13]. Our model is slightly different, allowing inversion gates, and exploiting noise.

Two issues remain to be discussed — transient faults and the scheduling demon. If the system of differential equations is autonomous (f only depends on $\vec{x}$ not t) then a transient fault can simply be modelled by an instantaneous change in the state variables. If the system is non-autonomous, then it has some memory of past state, and so a transient fault must be modelled as an external forcing function that operates over time to change the state. Depending on how the system has been designed, the magnitude of a fault can be bounded or unbounded.

The most serious weakness of this continuous model is in its ability to model the scheduling demon. We could possibly model scheduling effects by introducing non-constant signal propagation delays between processors. This, as one expects, seriously complicates reasoning about the behaviour of the system, for delays can introduce unstable oscillations in exactly the same manner as the scheduling demon. But the scheduling demon does more than delay signals. By suitably scheduling I/O operations, it can cause the slower processor to miss changes. For example, the fact that a processor wrote 1 then 0 then 1 may not even be perceived by its neighbour. It is not clear how such behaviour should should be modelled. One possibility is to vary the time scale associated with each processor so that a delayed processor operates very slowly relative to the active_processor.

3 Static Self-Stabilization: Distinguishing a Pair

Consider the following problem. Two processors P_0 and P_1 share two bits of memory b_0 and b_1. Processor P_0 reads b_0 as its *input* bit, and writes b_1 as its *output* bit. Processor P_1 does the opposite. The problem is for the two processors to set their bits to different values.

This problem has various solutions, depending on the atomicity and scheduling models, but all uniform solutions are essentially this: *Keep randomly setting your output bit so long as it matches your input bit.* (In order to break symmetry, discrete state uniform protocols require randomness [2], [3], [12].) Israeli and Jalfon give a solution to this problem under the read/write demon (a distributed demon with atomic reads and atomic writes) in [12].

To establish intuition for the continuous case, we present a slightly different version of the problem, with essentially the same solution as the Israeli and Jalfon protocol. The problem we wish to solve is the same for both discrete and continuous cases:

Informal Definition 3.1 *The* **Pair Distinguishing Problem** *is to give a self-stabilizing uniform protocol for two processors such that the processors agree on opposite signed states with magnitude near 1.*

3.1 Discrete State Case

We first consider the case of a discrete system.

For the Pair Distinguishing Problem we require two processors labelled P_0 and P_1. They each have one input and one output, with their outputs labelled b_0 and b_1. Processors are connected so that the output of one goes to the input of the other. The *state* of this system is the vector (b_0, b_1). The code for a processor is given in Figure 3.1.

```
Output x;
Input y;
loop {
    read y;
    dx := sgn(x-y) - x + random{-epsilon,0,+epsilon};
    x := x + dx;
    write x;
}
```

Figure 3.1: Discrete Pair Protocol

Our approach will be to view self-stabilization as a control problem. The set-point of a processor will be a function of its and its neighbour's state, and the state-change function will compute the numeric difference between the current state and the next state.

According to the protocol of Figure 3.1 when processors have different states, then their set-points are determined such that the processors change states in different directions. The set-point for a processor is $\text{sgn}(x-y)$, where the (discontinuous) signum function $\text{sgn}(z)$ is $-1, 0, +1$ depending on whether $z < 0, z = 0, z > 0$. For example, the set-point for a processor is -1 if its state x is closer to -1 than its neighbour's state y.

Thus the magnitude of the state change in order to reach the set-point is $\text{sgn}(x-y) - x$. A random perturbation is added to the state change to break symmetry in the event that both processors are in the same state. This results in the incremental state change of

$$dx = \text{sgn}(x-y) - x + \text{random}\{-\epsilon, 0, +\epsilon\}$$

Let G_d be the target space for the discrete pair protocol, consisting of the following two targets:

$$\{1, 1-\epsilon, 1+\epsilon\} \times \{-1, -1-\epsilon, -1+\epsilon\}$$

$$\{-1, -1-\epsilon, -1+\epsilon\} \times \{1, 1-\epsilon, 1+\epsilon\}$$

Then the following holds:

Proposition 3.2 *Under the Discrete Model the Discrete Pair Protocol with target space G_d is statically self-stabilizing. Furthermore, if the system is in a state outside of any target in G_d, then the expected number of steps before the system enters a target in G_d is $O(1)$.*

Proof Sketch. First observe that when $b_0 \neq b_1$ the processors must change to states near ± 1 and of opposite sign. Then suppose that $b_0 = b_1$ and examine all possible ways that the next few operations can be scheduled. In all cases, there is a non-zero probability that the two processors enter distinct states that are also different from their original equal values. □

3.2 Continuous State Case

Our continuous protocol for the pair distinguishing problem will be a direct analogy to the discrete protocol. Each processor consists of an integration gate, with a standard arithmetic circuit to compute the input to the gate. The processor has one input, and one output. The *state* of a processor in the continuous system is the value of the output of the integration gate. The state of the system at time t is the real-valued pair $(b_0(t), b_1(t))$. The two processors and their interconnection is shown in Figure 3.2.

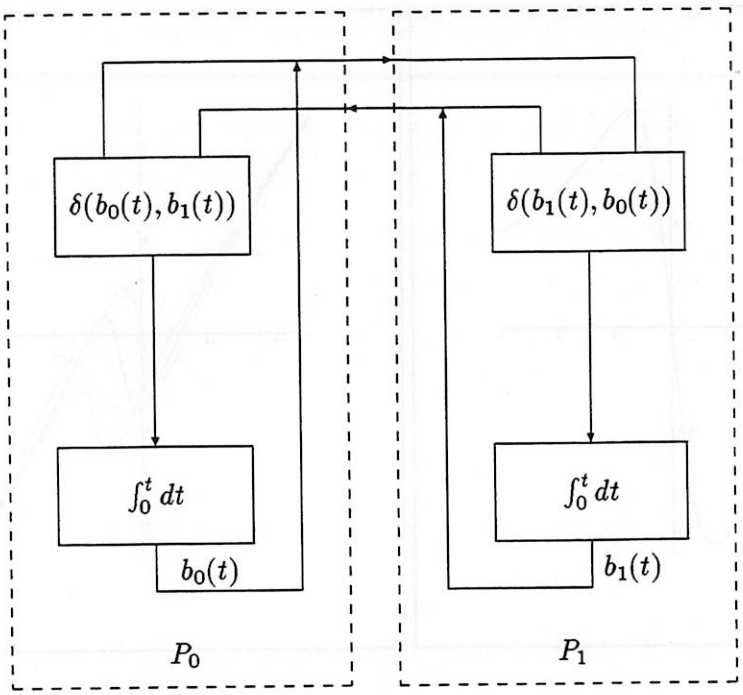

Figure 3.2: Continuous Model of the Pair Problem

In direct analogy to the discrete case, the (instantaneous) next-state of a processor, as a function of it current state, is given by

$$\frac{d}{dt}x(t) = \delta(x(t), y(t))$$

That is, the change in output x of a processor is a function of the current values of the output x, and of the input y. Connecting up the two processors we get the system of ordinary differential equations:

$$\frac{d}{dt}b_0(t) = \delta(b_0(t), b_1(t)), \quad \frac{d}{dt}b_1(t) = \delta(b_1(t), b_0(t))$$

The current state of a processor at time t is obtained by integrating the state-changes from time 0 to t.

What does δ look like? The intuition behind the function δ is exactly the same as for the discrete pair protocol computation of dx. The function δ must ensure that the system is driven towards stability regardless of the current state, and when the system is stable it remains so. We have two goals for δ. One is to keep the state values within reasonable bounds — if the current state of a processor is outside the interval $[-1, 1]$ it should tend to drive the state towards that interval. The other is to ensure that each processor enters a state opposite in sign to the other.

Since discontinuous functions cannot be computed by arithmetic circuits [9], we cannot actually use the sgn function in the computation. But we can use a continuous analogue:

$$Sn(x) = \frac{3\,x}{2\,x^2 + 1}$$

The graph of $Sn(x)$ is shown in Figure 3.3.

As for the random term that appears in the discrete protocol, this we drop and replace by noise. More on this later.

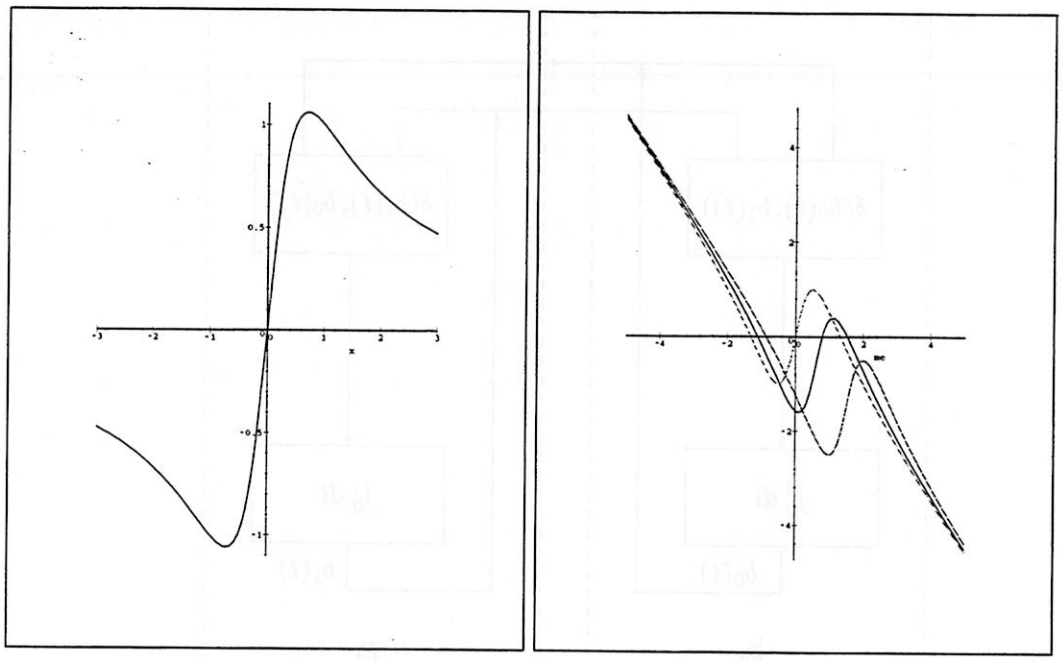

Figure 3.3: $Sn(x)$; and $\delta(x, y)$ at $y \in \{0, 0.6, 1.5\}$

Then δ can be written as the proportional-control function

$$\delta(x, y) = \frac{3}{2} Sn(x - y) - x = \frac{9(x - y)}{4(x - y)^2 + 2} - x$$

The graph of this function, for various values of y is shown in Figure 3.3. As a control function it has most of the features we desire. The state is driven towards $[-1, 1]$ if it is outside that interval, and inside the interval processors have their states driven in opposite directions.

But does the pair of processors stabilize? We define our continuous target space G_c, in an analogous way to the discrete case, by the two targets

$$\{(x, y) \mid x \in B_\epsilon(1),\ y \in B_\epsilon(-1)\}$$

$$\{(x, y) \mid x \in B_\epsilon(-1),\ y \in B_\epsilon(1)\}$$

for some fixed but arbitrary tolerance ϵ, and where $B_\epsilon(a) = \{x | a - \epsilon < x < a + \epsilon\}$.

Solving the system for its *equilibrium points* where δ is zero gives three solutions: $(0, 0)$, $(-1, 1)$, and $(1, -1)$. By examining the vector field plot of δ (Figure 3.4), and a tedious Liapunov analysis, not presented here, we find that the points $(-1, 1)$ and $(1, -1)$ are *asymptotically stable* (that is, attractors), the point $(0, 0)$ is *unstable*, and the system has no other limit cycles.

Thus two of the equilibrium points are in distinct targets of G_c. We would like to rule out the equilibrium point $(0, 0)$ since this is not in any target. Note that $(0, 0)$ is an attractor when approached on the line $x = y$ — the symmetric situation in which the two processors are in the same state. But since $(0, 0)$ is unstable, in any real physical system, noise would tend to push the system off of $(0, 0)$ and it would move toward a target in G_c. **Thus randomness in a discrete system is replaced by noise in a continuous system.**

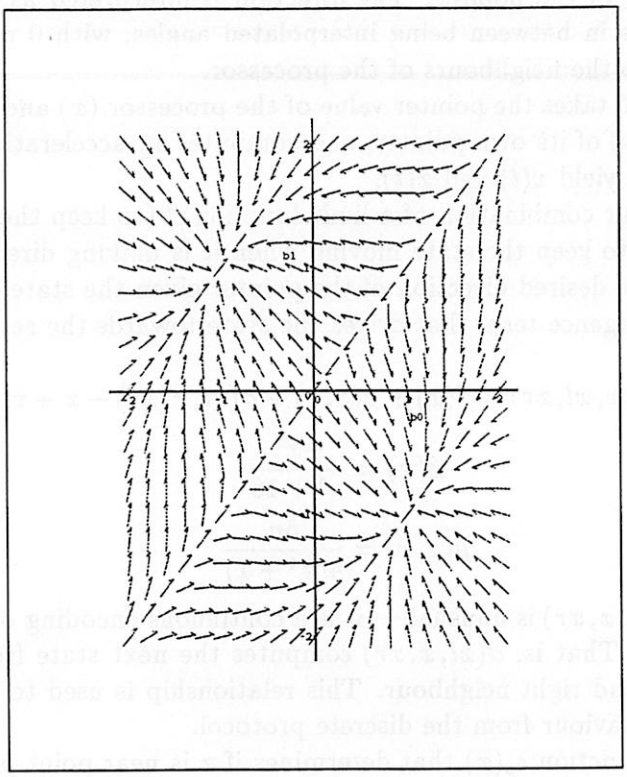

Figure 3.4: Vector field for Pair System

Proposition 3.3 *Under the Continuous Model the Continuous Pair Protocol with target space G_c is statically self-stabilizing. Furthermore, if the system is in a state (b_0, b_1) outside of a target of G_c, then the expected time before the system enters G_c is $O(\sqrt{b_0^2 + b_1^2})$.*

It is important to note that Sn is easy to compute, and has well behaved derivatives. This means that numerical solutions of the continuous system can be efficiently computed, and so the system can be converted into a discrete one directly via simulation.

4 Static Self-Stability: Orienting Rings of Processors

A more interesting example of static self-stability is ring orientation.

Informal Definition 4.1 *The* **Ring Orientation Problem** *is to give a self-stabilizing uniform protocol for a ring of processors such that the processors agree on a consistent orientation.*

One style of algorithm for the Ring Orientation Problem [10], [12] is this: *Associate an orientation arrow with each processor. Look at the neighbouring processor the arrow is pointing to. If that processor is pointing back, then randomly flip your pointer.*

The interesting thing about the continuous version of this problem is that seems to be unsolvable by a simple first-order system of differential equations. Because a processor is changing focus between its left and right neighbours, it needs some notion of velocity in order to survive a transient fault that alters its state (i.e. current integration gate values). Thus each processor has two state variables, its pointer

direction x, and the velocity v of the pointer. The direction is interpreted as -1 pointing fully left, $+1$ pointing fully right, and values in between being interpolated angles, with 0 pointing upward. Only the pointer direction is available to the neighbours of the processor.

The basic control function δ takes the pointer value of the processor (x) and its left (xl) and right (xr) neighbours; and the velocity (v) of its own pointer, and computes an acceleration (dv/dt) for the pointer. Two integration gates in series yield $v(t)$ and $x(t)$.

Control function δ is a linear combination of a limit function β (to keep the pointer state between -1 and 1), an inertia function μ (to keep the state moving when it is making direction changes), a set-point function σ (that determines the desired direction of the pointer given the state of the neighbour currently being pointed to), and a convergence term that drives the state towards the set-point.

$$\delta(x, v, xl, xr) = \beta(x) + \mu(x, v) + \sigma(xl, x, xr) - x - v$$

$$\beta(x) = \frac{-x^5}{x^4 + 16}$$

$$\mu(x, v) = \frac{5v}{4(x^4 + 1)}$$

The set-point function $\sigma(xl, x, xr)$ is unusual — it is a continuous encoding of the state transition table used by the discrete protocol. That is, $\sigma(xl, x, xr)$ computes the next state from the states xl, x, xr of the left neighbour, processor, and right neighbour. This relationship is used to argue that the continuous protocol inherits its proper behaviour from the discrete protocol.

To construct σ we need a function $\epsilon_p(x)$ that determines if x is near point p.

$$\epsilon_p(x) = \frac{1}{1 + (x - p)^4}$$

Then σ is the following linear combination:

$$\begin{aligned}\sigma(xl, x, xr) = \\ &\epsilon_1(x)\epsilon_1(xr) \; - \; \epsilon_{-1}(xl)\epsilon_{-1}(x) \; + \\ &\epsilon_1(x)\epsilon_{-1}(xr)Sn(x + xr) \; + \; \epsilon_1(xl)\epsilon_{-1}(x)Sn(xl + x)\end{aligned}$$

Which preserves the orientation of the processor unless it conflicts with its neighbour, in which case the processor with the smaller magnitude orintation switches direction.

Proposition 4.2 *There exists a uniform self-stabilizing protocol under the Continuous Model that solves the Ring Orientation Problem.*

Figure 4.1 illustrates a sample trajectory of four processors initially oriented symmetrically in two head-to-head pairs which converges to a common orientation.

5 Dynamic Self-Stability: Token Passing

The prototypical example of a dynamic self-stabilization problem is to create a unique token (representing some kind of privilege) that is passed among processors.

Informal Definition 5.1 *The* **Token Passing Problem** *is to give a self-stabilizing uniform protocol for a pair of processors such that exactly one token is continuously passed between the processors.*

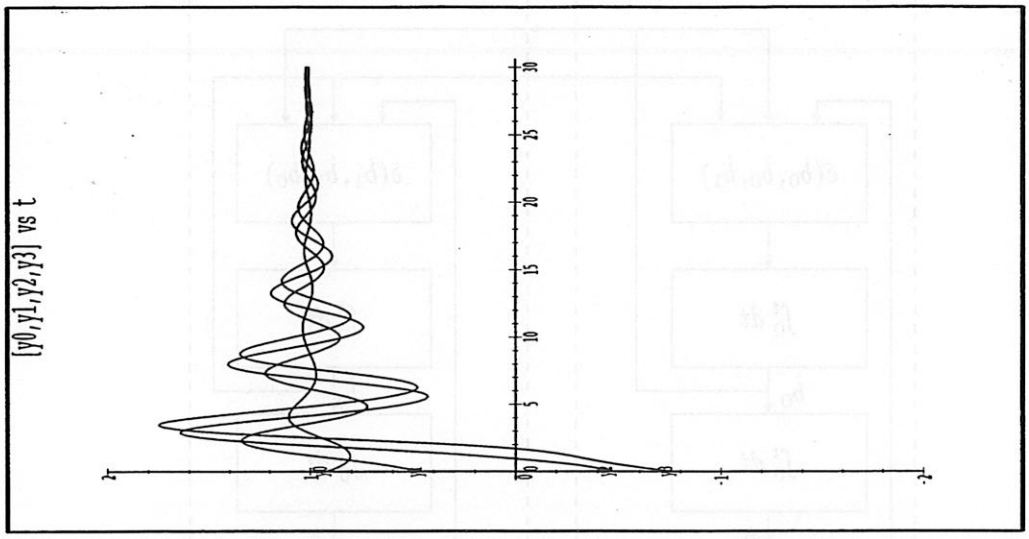

Figure 4.1: Example Trajectory of Ring Orientation
Time evolves on the vertical axis, the horizontal axis represents the starting state of 4 processors y_0 to y_3 in a ring initially oriented in two head-to-head pairs.

An example of a basic token passing protocol is given by Israeli and Jalfon [11]. Each processor has three states: idle (I), sending (S), and receiving (R). Processors examine their neighbour's state, and follow a basic cycle of

$$I \xrightarrow{S} R \xrightarrow{I} S \xrightarrow{R} I$$

where the label on the transition is the state of the other processor. To get the protocol to self-stabilize, the following transitions are added:

$$S \xrightarrow{S} I \ , \ R \xrightarrow{R} I \ , \ I \xrightarrow{I} \{I, S\}$$

That is, a processor moves to the idle state if its neighbour is in the same state, unless both processors are idle, in which case a processor attempts to break symmetry by randomly moving to state S or I.

The self-stabilization is achieved by phase-locking the cycles of the two processors, shifted out of phase by one transition. When stabilized, the system will move through the following limit cycle

$$IS,\ RS,\ RI,\ SI,\ SR,\ IR,\ IS,\ \cdots$$

Our continuous solution to this problem exploits this phase-locking intuition.

Each processor has two state variables, b and $\dot{b}$, which represent ownership of the token and its state of transfer. When b is positive, the processor thinks it holds the token, and when b is negative the processor thinks the other processor holds the token. When $\dot{b}$ is positive, the processor is taking possession of the token, and when $\dot{b}$ is negative, the processor is handing over the token.

The goal of the control function δ is to ensure that the processors have coordinated notions about the passing and ownership of the token. The two processors and their interconnection is shown in Figure 5.1.

Function δ takes the ownership state x, the local passing state v, and the neighbour's passing state v'. It is a linear combination of two limit terms $\beta(x)$ and $\beta(v)$ (which keep the ownership and passing values

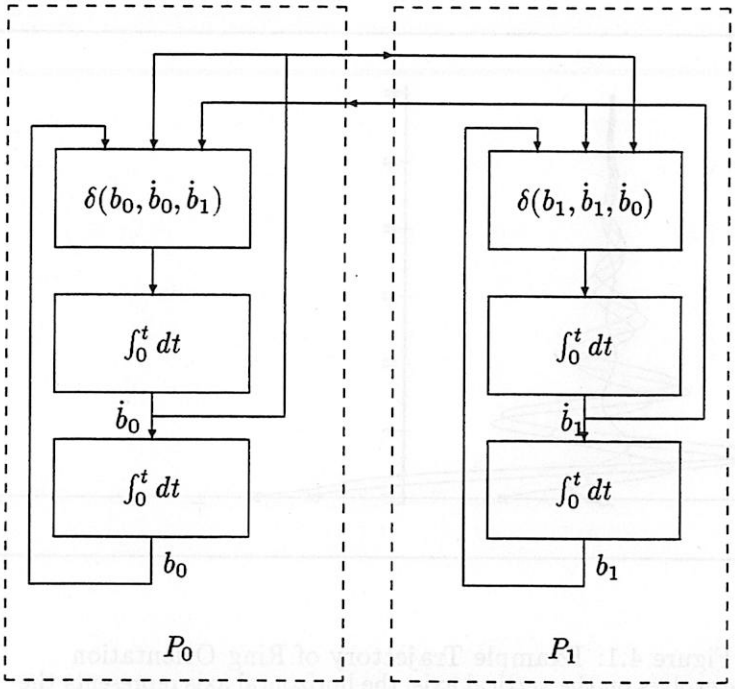

Figure 5.1: Continuous Model of the Token Passing Problem

roughly between -1.5 and 1.5), an inertia function μ (to keep the state moving when it is making a token transfer), a set-point function σ that determines the desired ownership state given the current ownership state, and a phase locking term that drives the processors to have the same magnitude but opposite sign passing states.

$$\delta(x, v, v') = \beta(x) + \beta(v) + \mu(x, v) + \sigma(x) - \frac{v + v'}{2}$$

$$\beta(x) = \frac{-x^5}{x^4 + 16}$$

$$\mu(x, v) = \frac{3v}{(16x^4 + 1)(2v^2 + 1)}$$

$$\sigma(x) = \frac{2}{16(x+1)^4 + 1} - \frac{2}{16(x-1)^4 + 1}$$

The set-point function $\sigma(x)$ is again a continuous encoding of a state transition table.
We define our target space G_p for the Token Passing Problem by the two targets

$$\{(b_0, \dot{b}_0) \mid b_0 > 0\} \times \{(b_1, \dot{b}_1) \mid b_1 < 0\}$$

$$\{(b_0, \dot{b}_0) \mid b_0 < 0\} \times \{(b_1, \dot{b}_1) \mid b_1 > 0\}$$

and our desired behaviour B by the continuous alternation of the two targets.

Proposition 5.2 *There exists a uniform self-stabilizing protocol under the Continuous Model that solves the Token Passing Problem.*

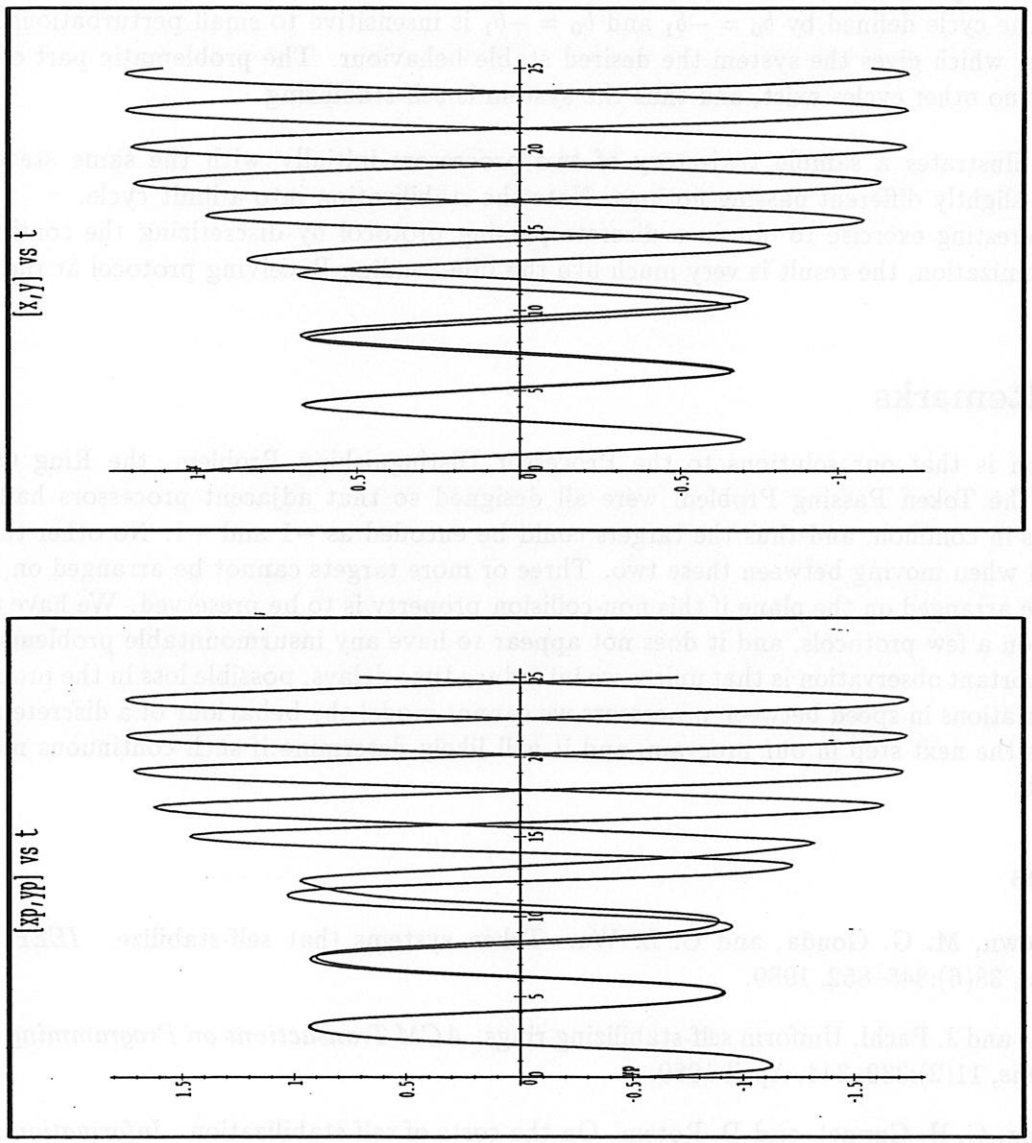

Figure 5.2: Token Passing - Ownership (top) and Passing State Trajectories
Time evolves on the vertical axis, the horizontal axis represents the starting state of the pair of processors. Both initially think they own the token and are about to pass it.

Proof Sketch. Solving $\delta(b_0, 0, 0) = 0$, $\delta(b_1, 0, 0) = 0$ yields only complex roots, so no equilibria exist for the system, and thus it must be continuously changing state. Since the system globally converges to a ball near the origin, its behaviour is asymptotically bounded, and thus it must have cyclic behaviour. What kind of limit cycles does the system have? If both processors start with exactly the same ownership and passing states, the the system is symmetric, and in the absence of noise both processors will follow exactly the same trajectories. But this cycle is not stable, in the sense that any noise kicks the system off of this cycle. The cycle defined by $b_0 = -b_1$ and $\dot{b}_0 = -\dot{b}_1$ is insensitive to small perturbations, and is an attracting cycle, which gives the system the desired stable behaviour. The problematic part of the proof is showing that no other cycles exist, and thus the system is self-stabilizing. □

Figure 5.2 illustrates a sample trajectory of two processors initially with the same state of token ownership, but slightly different passing notions. Note the stabilization into a limit cycle.

It is an interesting exercise to obtain a discrete passing protocol by discretizing the continuous one. After some optimization, the result is very much like the Idle-Sending-Receiving protocol at the beginning of this section.

6 Final Remarks

One observation is that our solutions to the Processor Distinguishing Problem, the Ring Orientation Problem, and the Token Passing Problem were all designed so that adjacent processors had only two possible targets in common, and thus the targets could be encoded as -1 and $+1$. No other target could be encountered when moving between these two. Three or more targets cannot be arranged on a line, but instead must be arranged on the plane if this non-collision property is to be preserved. We have attempted this technique on a few protocols, and it does not appear to have any insurmountable problems.

A more important observation is that unless we introduce time delays, possible loss in the propagation of signals, and variations in speed between processors we cannot model the behaviour of a discrete scheduling demon. This is the next step in our program, and it will likely determine if such continuous models have any merit at all.

References

[1] G. M. Brown, M. G. Gouda, and C. L. Wu. Token systems that self-stabilize. *IEEE Trans. on Computers*, 38(6):845–852, 1989.

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

[3] E. J. Chang, G. H. Gonnet, and D. Rotem. On the costs of self-stabilization. *Information Processing Letters*, 24:311–316, March 1986.

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

[5] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1:5–6, 1986.

[6] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In 9^{th} *Ann. ACM Symp. on Principles of Distributed Computation*, pages 103–118, August 1990.

[7] S. Ghosh. Binary self-stabilization in distributed systems. *Information Processing Letters*, 40:153–159, November 1991.

[8] J. Guckenheimer and P. Holmes. *Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields.* Springer-Verlag, 2nd revised edition, 1986.

[9] H. J. Hoover. Feasible real functions and arithmetic circuits. *SIAM Journal on Computing*, 19(1):182–204, 1990.

[10] H. J. Hoover and P. Rudnicki. The uniform self-stabilizing orientation of unicyclic networks. Technical Report TR 91–02, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G2H1, August 1991.

[11] A. Israeli and M. Jalfon. Modular constrcution of uniform self-stabilizing protocols. Technical report, June 1993.

[12] A. Israeli and M. Jalfon. Uniform self-stabilizing ring orientation. *Information and Computation*, 104:175–196, 1993.

[13] L. Lipshitz and L. A. Rubel. A differentially algebraic replacement theorem and analog computability. *Proceedings of the American Mathematical Society*, 99:367–372, 1987.

[14] M. B. Pour-El. Abstract computability and its relation to the general purpose analog computer (some connections between logic, differential equations and analog computers). *Transactions of the American Mathematical Society*, 199:1–28, 1974.

[15] C. E. Shannon. Mathematical theory of the differential analyser. *Journal of Mathematics and Physics of the Massachusetts Institute of Technology*, pages 337–354, 1941.

Paper Number 9

Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults

Shlomi Dolev and Jennifer L. Welch

Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults*

(Preliminary Version)

Shlomi Dolev† Jennifer L. Welch‡

Abstract

We initiate a study of bounded clock synchronization under a more severe fault model than that proposed by Lamport and Melliar-Smith [LM-85]. Realistic aspects of the problem of synchronizing clocks in the presence of faults are considered. One aspect is that clock synchronization is an on-going task, thus the assumption that in *any* period of the execution at least two thirds of the processors are nonfaulty is too optimistic. To cope with this reality we suggest *self-stabilizing* protocols that stabilize in any (long enough) period in which less than a third of the processors are faulty. Another aspect is that the clock value is bounded. A single transient fault may cause the clock to reach the upper bound. Therefore we suggest a bounded clock that wraps around when appropriate.

We present two randomized self-stabilizing protocols for synchronizing bounded clocks in the presence of Byzantine processor failures. The first protocol assumes that processors have a common pulse, while the second protocol does not. A new type of distributed counter based on the Chinese remainder theorem is used as part of the first protocol.

1 Introduction

In a distributed system, it is often necessary to keep the logical clocks of the processors synchronized. In such a system physical clocks may drift and messages could have varying delivery times. Moreover, processors may be faulty, and in many cases the type of failures is not predictable in advance. To handle this situation, the worst type of failures must be considered, namely *Byzantine* faults [LSP-82]. In the presence of Byzantine faults a processor can exhibit arbitrary "malicious", "two faced", behavior.

The problem of keeping clocks synchronized in the presence of Byzantine faults has been extensively studied (e.g., [HS+-84, LM-85, MS-85, DHS-86, ST-87, WL-88, RSB-90]). Lamport and Melliar-Smith [LM-85] were the first to present the problem and show that $3f + 1$ processors are sufficient to tolerate f Byzantine faults. The necessity of $3f + 1$ processors to tolerate f faults was later proved in [DHS-86]. A weaker fault model called authenticated Byzantine allows a protocol that can tolerate any number of faulty processors [HS+-84]. In that failure model reintegration of repaired processors is only possible if less than half the processors are faulty. Many of the protocols for this problem assume that the clocks are initially synchronized and thus focus on keeping them synchronized in the presence of clock drift.

The problem of how to ensure that the clocks are initially synchronized was addressed in, e.g., [ST-87, WL-88]. In these protocols, some mechanism is assumed that allows all the nonfaulty processors to begin the protocol within a bounded time period of each other. The mechanism essentially is that the processes

*Supported in part by TAMU Engineering Excellence funds and NSF Presidential Young Investigator Award CCR-9158478.

†Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva, 84105, Israel. e-mail: shlomi@cs.bgu.ac.il.

‡Department of Computer Science, Texas A&M University, College Station, TX 77843. e-mail: welch@cs.tamu.edu.

are assumed to wake up in a distinguished initial state, in which they can uniquely perform initializing actions, including communication with each other.

In this work we weaken the assumptions made for the design of clock synchronization protocols in the presence of Byzantine faults. Our goal is protocols that cope with a more severe (and realistic) fault model than the traditional Byzantine fault model [LSP-82]. Initially, protocols that tolerate Byzantine faults were designed for flight devices that need to be extremely robust. In such a device the traditional assumptions could be violated: Is it reasonable to assume that during *any* period of the execution less than one third of the processors are faulty? What happens if for a short period more than a third are faulty (perhaps experience a weaker fault than a Byzantine fault)? What happens if messages sent by nonfaulty processors are lost in one instant of time?

In this paper we present *self-stabilizing* protocols that can overcome these problems. Such temporary violations of the assumptions can be viewed as leaving the system in an arbitrary initial state from which the protocol resumes. Self-stabilizing protocols work correctly when started in any initial system state. Thus, even if the system loses its consistency due to an unexpected temporary violation of the assumptions made (e.g., more than one-third faulty, unexpected message loss) the system synchronizes the clocks when subsequently the assumptions hold (e.g., less than a third experience Byzantine faults).

Originally, Dijkstra defined (in [Dij-74]) a protocol to be *self-stabilizing* if, when started in an arbitrary system state, the system converges to a consistent global state that realizes the task. Self-stabilizing protocols are resilient to *transient faults* – faults that cause the state of a processor to change arbitrarily and then from the new state, the processor resumes operation according to its program. A *permanent fault* is a fault that causes a processor to permanently misbehave. A protocol tolerates *hybrid faults* if it is resilient to both transient and permanent faults (e.g., [DW-93, GP-93] which consider *napping* and *omission* faults, respectively). We are interested in clock synchronization protocols that can tolerate hybrid faults: they should work from an arbitrary initial configuration and they should tolerate less than a third of the processors exhibiting permanent Byzantine faults.

A realistic assumption for a clock synchronization protocol is that a 64-bit clock is "unbounded" for most possible applications. However, in the context of self-stabilizing protocols transient faults could cause the system to reach the upper bound of the clock at once. Thus, another aspect of the problem should be considered: the fact that the clocks are bounded.

In this paper we present two randomized self-stabilizing clock synchronization protocols that work in the presence of Byzantine faults. Both protocols work for bounded clocks. The first assumes the existence of a common pulse while the second does not make this assumption. The expected stabilization time of both protocols is exponential in n. This is a drawback when the number of processors is large. However, in addition to being of theoretical interest, we believe that our protocols could be of practical interest, at least when the number of backup processors is small.

One of the contributions of this paper is an interesting usage of the Chinese remainder theorem for implementing a distributed counter. This counter is used to accelerate the first protocol.

The remainder of the paper is organized as follows. In the next section we formalize the assumptions and requirements for the protocol. Section 3 presents a clock synchronization protocol under the assumption of a common pulse. In Section 4 we present a protocol that does not assume the existence of common pulses. Conclusions are in Section 5.

2 Definitions

A distributed system consists of a set of processors that communicate by sending messages to each other. Messages have a bounded delay. Each processor has a bounded physical clock that is constantly incre-

mented, wrapping around when appropriate; the physical clocks at the different processors run at approximately the same rates. Each processor also has a bounded logical clock, which is computed as a function of the current state and physical clock value. The goal is for the logical clocks of the nonfaulty processors to become and subsequently remain close to each other, while continuing to progress at a reasonable rate (wrapping around when appropriate). We consider two types of timing behavior of the system, synchronous and semi-synchronous. In both models, processors take steps either when they receive a message, or when their physical clocks reach some predetermined value. In addition, in the synchronous model, there is a common pulse that periodically occurs simultaneously at all processors, causing them to take a step. We now proceed more formally.

Each *processor* P_i, $1 \leq i \leq n$, is modeled as a state machine. Associated with the processor is its *physical clock*, which takes on integral values from 0 to $M_{pc} - 1$ for some M_{pc}. The state contains a distinguished *timer* variable that can take on the values 0 to $M_{pc} - 1$ and nil; it indicates that the processor wants to take a step the next time its physical clock has the given value. A transition takes the current state of the processor, the current value of its physical clock, and a message received (if any) and produces a new state of the processor and a set of messages to be sent. The *message system* holds all messages sent but not yet received. A *configuration* of the system is a set of processor states, one per processor, a set of physical clock values, one per processor, and a state for the message system.

An execution is an alternating sequence of configurations and events $C_0, e_1, C_1, \cdots$. In a *semi-synchronous execution*, events happen at real times, taking one configuration to the next. There are two types of events. One type is a *tick* of some processor's physical clock, causing it to increase by 1 mod M_{pc}. Nothing else changes. We require that the real time elapsed between two successive ticks of the same processor be between $1 - \rho$ and $1 + \rho$ for some fixed ρ.

The other type of event is a *step* of some processor. No processor can take more than one step at the same real time. In the step, the processor may or may not receive a message. The real time elapsed between the sending and receiving of any message must be in the range $[d - \epsilon, d + \epsilon]$ for some fixed d and ϵ. There is a fixed set of *faulty* processors of size f, where $n > 3f$. If the processor taking the step is nonfaulty, then the succeeding configuration must correctly reflect the processor's transition function acting on the message received and the state and physical clock in the preceding configuration. Thus the only changes are to the processor's state and the message system (removing the message received and adding the messages sent). If the processor taking the step is faulty, it can change state arbitrarily and add arbitrary messages (from itself) to the message system.

In a *synchronous execution*, in addition to the above constraints, there exists a value $\pi > 0$ such that, for all i, every processor P_i receives a special Pulse message (from a dummy processor) at time $i \cdot \pi$. (I.e., all the processors take a step at each pulse and the pulses occur regularly with period π.)

We require that for every processor P_i there exist a function $clock_i$ that, given a state of P_i and a value for P_i's physical clock, returns a value in the range 0 to $M_{lc} - 1$ for some fixed M_{lc}. This is the logical clock of P_i. Given a particular execution $C_0, e_1, \ldots$, we denote by $clock_i(t)$ the value of the function $clock_i$ applied to P_i's state and physical clock value in C_j, where j is the configuration in the execution whose real time of occurrence is the largest not exceeding t. We require that there exist a finite time t_s for which the following two conditions hold:

Clock Agreement: There exists $\gamma < M_{lc}/4$ such that for all $t \geq t_s$ and all nonfaulty processors P_i and P_j: $clock_i(t) - clock_j(t) \pmod{M_{lc}} \leq \gamma$.[1]

Clock Validity: There exists Δ, $0 < \Delta \leq M_{lc}/4$, and there exists $a \geq 0$ such that for all real times $t > t_s$ and all i, if $clock_i(t) = T$, then $T + \Delta/(1+a) \bmod (M_{lc}) \leq clock_i(t + \Delta) \leq T + (1+a)\Delta \bmod (M_{lc})$.

[1]The constant 4 is chosen for convenience; any constant larger than 2 is sufficient. Note that if the constant is 2 then this condition holds for any arbitrary configuration, since every two clock values are at most $M_{lc}/2$ apart.

Clock Agreement states that after t_s, the difference between any two nonfaulty processors' clocks is at most γ. Clock Validity states that after t_s, the amount of logical clock time that elapses during Δ real time is a linear function of Δ.

3 Synchronous Protocol

We first describe a protocol for the synchronous system, in which nonfaulty processors have access to a periodic common pulse. Each pulse triggers the processors to synchronize their clocks. The time between two successive pulses appears to be an important parameter to the problem. In case two successive pulses are farther apart than the time required to run a Byzantine agreement protocol, then the following scheme solves the problem: Every pulse starts a new version of the Byzantine agreement to agree on the common clock value. However, when the pulses are only on the order of the round trip message delay apart, this scheme cannot work.

We assume that the pulses are on the order of the round trip delay apart. Recall that π is the time between two successive pulses. Nonfaulty processors send messages and update their logical clocks only when a pulse occurs. We assume that π is long enough such that when a pulse takes place, no message sent by a nonfaulty processor in the previous pulse is present in the system. Whenever a nonfaulty processor P is triggered by a pulse, P sends a message with its clock value to all its neighbors. Then P waits to receive all the clock values of the other processors. P waits for a period $(1+\rho)(d+\epsilon)$ that is longer than the bound on the message delay and accounts for clock drift. If during that period P receives more than one message from some neighbor, say Q, then P uses the latest value that arrives from Q. Thus, at the end of such a period P has a set of at least $n-f$ logical clock values, at most one value for each nonfaulty processor including P. P uses the set of the logical clocks received in order to choose its own clock value.

The formal description of the protocol appears in Figure 1. We now describe the protocol informally. The protocol for a processor P works as follows: (1) if the value of P's clock appears less than $n-f$ times in the set of the received logical clocks then P assigns 0 to its clock. Otherwise, (2) in case that the value of P's clock appears at least $n-f$ times, we further distinguish between the case (2.1) in which P's clock value is not equal to 0 and the case (2.2) in which it is equal to 0. In case (2.1) P increments its clock by 1 (modulo the number of clock values M_{lc}). Case (2.2) is further subdivided into two cases: (2.2.1) in which (according to the state of P) in the previous pulse P incremented its clock by 1 (and the result was 0) and the case (2.2.2), otherwise. In case (2.2.1) P increments its clock by 1 (to be 1). In case (2.2.2) P tosses a coin and assigns the result (0 or 1) to its clock.

The protocol guarantees (with probability 1) that the system eventually reaches a global state in which all the nonfaulty processors have the clock value 1. Once such a global state is reached the clocks are synchronized: In every pulse, every nonfaulty processor P receives messages from at least $n-f-1$ processors containing a clock value that is identical to its own clock value. Moreover, a pulse in which all the nonfaulty processors set their clocks to 0 always follows a pulse in which every nonfaulty processor *increments* its clock value by 1 to set it to 0. Thus, case (2.2.2) is not applied.

The main idea of the protocol is to ensure that only when there are "enough" nonfaulty processors with the same clock value will this value be incremented. It is proved in the sequel that in any pulse at most one clock value of nonfaulty processors is incremented by 1 while the rest of the values are changed to be zero. This ensures that after the first pulse, the set of clock values of the nonfaulty processors contains at most two elements. Moreover, if two such elements indeed exist one of them is 0.

At first glance this seems to be sufficient and no coin toss is needed; the value that is incremented will eventually wrap around to 0 and at that time the clocks of all the nonfaulty processors will be 0. However, we now describe an infinite execution, E, that does not use coin tosses in which the clocks never become

synchronized. Consider a system with four processors P_1, P_2, P_3 and P_4 in which P_4 exhibits Byzantine behavior. Let $0,0,1$ be the clock values of P_1, P_2, P_3, respectively, in the first configuration of E. In the first pulse P_4 sends clock value 1 to P_1 and P_3 and clock value 0 to P_2. Thus, P_1 receives the clock values vector $0,0,1,1$, P_2 receives $0,0,1,0$ and P_3 $0,0,1,1$. P_2 is the only processor that finds $n-f=3$ processors with the same clock value (namely, the clock value 0) and increments its clock value by one (to be 1). At the same time, P_1 and P_3 find two clock values with value 1 and two with value 0 and assign 0 to their clocks. Hence, a configuration with clock values $0,1,0$ for P_1, P_2, P_3, respectively, is obtained. P_4 continue and sends the clock values $1,1,0$ to P_1, P_2, P_3, respectively. P_1 receives the clock values vector $0,1,0,1$, P_2 receives $0,1,0,1$ and P_3 receives $0,1,0,0$. Similarly, P_3 is the only processor that finds $n-f=3$ processors with the same clock value and assigns 1 to its clock while P_1 and P_2 assign 0. We reach a configuration with clock values $0,0,1$ for P_1, P_2, P_3 which are identical to the clock values in the first configuration. Therefore, an infinite execution in which nonfaulty processors never agree on their clock values is possible.

To overcome the above problem we use coin tosses. In a pulse in which a nonfaulty processor with clock value 0 receives $n-f$ clock values with value 0 the processor tosses a coin and decides whether to assign 0 or 1 to its clock. This leads to a possible scenario (that has some probability of occurring) in which the coin toss results cause all the nonfaulty processors to simultaneously assign 1 to their clocks.

```
01 when pulse occurs:
02   broadcast clock_i
03   collect clock values until (1+ρ)(d+ε) time has elapsed on the physical clock
04   if |{j|clock_i = clock_j}| < n − f then (*case (1)*)
05     {clock_i := 0; last_increment_i := false}
06   else (*case (2)*)
07       if clock_i ≠ 0 then (*case (2.1)*)
08         {clock_i := (clock_i + 1) mod M_lc; last_increment_i := true}
09       else (*case (2.2)*)
10           if last_increment_i = true then (*case (2.2.1)*) clock_i := 1
11           else (*case 2.2.2*) clock_i := toss(0,1)
12           if clock_i = 1 then last_increment_i := true
13           else last_increment_i := false
```

Figure 1: The Synchronous Protocol for P_i

3.1 Correctness Proof of the Synchronous Protocol

Throughout the proof we say that a processor P_i *increments* its clock by 1 in a certain pulse, if P_i assigns $last\_increment := true$ during this pulse. Otherwise, we say that P_i *assigns* 0 to $clock_i$.

Lemma 3.1 *If nonfaulty processors P_i and P_j increment their clocks by 1 during some pulse $\mathcal{P}$, then immediately after $\mathcal{P}$, $clock_i = clock_j$.*

Proof: Assume towards contradiction that $clock_i = (x+1) \bmod M_{lc} \neq clock_j = (y+1) \bmod M_{lc}$ following $\mathcal{P}$. Hence, during $\mathcal{P}$, P_i finds at least $n-f$ clock values that are equal to x. At least $n-2f$ of them belong to nonfaulty processors. Thus, P_j also receives $n-2f$ clock values that are equal to x. Hence, P_j receives at most $n-(n-2f)=2f$ clock values that are equal to y. Since $n>3f$, it holds that $n-f>2f$, which contradicts the possibility of P_j receiving at least $n-f$ clock values that are equal to y. ■

Lemma 3.1 implies in a straightforward manner the correctness of the next two corollaries.

Corollary 3.2 *After every pulse, the set of clock values of the nonfaulty processors contains at most two elements. In case there are such two values, one of them is* 0.

Corollary 3.3 *If during a pulse $\mathcal{P}$ a nonfaulty processor P increments its clock value by* 1 *and the result is* 0*, then immediately following $\mathcal{P}$ the clock values of all the nonfaulty processors are* 0.

Claim 3.4 *If during a pulse $\mathcal{P}$ that follows the first pulse, a nonfaulty processor P increments its clock to be* 1 *without tossing a coin, then just before $\mathcal{P}$ all the nonfaulty processors' clock values were* 0.

Proof: The variable *last_increment* is assigned during every pulse. Thus, since $\mathcal{P}$ follows the first pulse, P indeed increments during $\mathcal{Q}$, the pulse before $\mathcal{P}$. Thus by Lemma 3.1 all the nonfaulty processors have clock values 0 after $\mathcal{Q}$ and before $\mathcal{P}$. ■

The next theorem uses the scheduler-luck game of [DIM-91, DIM-95] to analyze the randomized protocol. The scheduler-luck game has two competitors, *scheduler* (adversary) and *luck*. The goal of the scheduler is to prevent the protocol from reaching a safe configuration while the goal of *luck* is to help the protocol reach a safe configuration. For the synchronous protocol a configuration is *safe* if for all nonfaulty processors, the logical clocks are equal and *last_increment* is true. For our system the scheduler chooses the message delays and clock drifts during the execution (within the predefined limitations). Each time the processor, activated by the scheduler, tosses a coin, *luck* may intervene and determine the result of the coin toss. It is proved in [DIM-91, DIM-95] that if, starting with any possible configuration c, *luck* has a strategy to win the scheduler-luck game within i interventions and expected time t, then the system reaches a safe configuration within expected time $t \cdot 2^i$. The main observation used for this proof is the fact that if a coin toss result differs from the desired result (according to *luck* strategy) a configuration is reached from which a new game can begin.

Theorem 3.5 *In expected $M_{lc} \cdot 2^{2(n-f)}$ pulses, the system reaches a configuration in which the value of every nonfaulty processor's clock is 1.*

Proof: The proof is by the use of Lemma 1 of [DIM-91] (Theorem 5 of [DIM-95]). We present a strategy for *luck* to win the scheduler-luck game with $2(n-f)$ interventions and within $M_{lc}+2\pi$ time. The strategy of *luck* is (1) wait for the first pulse to elapse. Thereafter, (2) *luck* waits till a pulse $\mathcal{P}$ in which a nonfaulty processor with clock value 0 receives $n-f$ clock values that are 0. This occurs within the next M_{lc} pulses (if it does not occur by then, there is at least one nonfaulty processor that does not assign 0 to its clock during M_{lc} successive pulses, which is impossible). In case (2.1) during this pulse all the nonfaulty processors are either tossing a coin or assigning 1 without tossing. Then *luck* intervenes at most $n-f$ times and fixes the coin toss results of all the nonfaulty processors to be 1. Otherwise, (2.2) if there is a nonfaulty processor P that is neither about to toss a coin nor about to assign 1 without tossing, then *luck* intervenes and fixes all the coin toss results (less than $n-f$) to be 0. Note that before $\mathcal{P}$, P's clock is not equal to 0. Thus, by Claim 3.4 no processor assigns 1 without tossing a coin. By Lemma 3.1 and the fact that some nonfaulty processor tosses a coin during $\mathcal{P}$, it holds that following $\mathcal{P}$ the clock values of all the nonfaulty processors are 0. Therefore, in the next pulse case (2.1) is reached and *luck* could intervene and fix at most $n-f$ coin toss results to ensure that the desired global state is reached. ■

By Theorem 3.5 the system reaches a configuration in which the value of every nonfaulty processor's clock is 1, in expected time $M_{lc} \cdot 2^{2(n-f)}$. It is easy to see that in any successive pulse, all the nonfaulty processors have the same clock value. Thus the clock agreement requirement holds with $\gamma = 0$. Since the clocks of the nonfaulty processors are incremented by 1 in every pulse and the pulses are constant time apart, the clock validity requirement also holds. Note that the clock value could be multiplied by π (if π is known), the time difference between two successive pulses, in order to yield a clock value that reflects real time. Otherwise, the value of a of the clock validity requirement encodes $1/\pi$.

3.2 Accelerating the Protocol

If $M_{lc} = 2^{64}$, our protocol converges after expected $2^{64} \cdot 2^{2(n-f)}$ synchronization pulses. Certainly, because of this time complexity this protocol cannot be used in practice. However, if M_{lc}, n, and f are all small[2] then the expected number of pulses required is reasonably small. For instance, if $M_{lc} = 2$, $n = 4$, and $f = 1$, then the expected number of pulses is 128. We use the above observation to accelerate our protocol. We achieve synchronization of clock values in the range of $M_{lc} = 2^{64}$ values within expected number of pulses that is less than $381 \cdot 2^{2(n-f)}$. (For $M_{lc} = 2^{16}$, synchronization occurs within expected number of pulses that is less than $58 \cdot 2^{2(n-f)}$ pulses).

We define the *Chinese remainder counter* by the use of the Chinese remainder theorem, which appears in [Kn-81] p. 270:

Theorem 3.6 *Let m_1, m_2, ..., m_r be positive integers that are relatively prime in pairs, i.e., $gcd(m_j, m_k)=1$ when $j \neq k$. Let $m = m_1 m_2 \cdots m_r$, and let a, u_1, u_2,...,u_r be integers. Then there is exactly one integer u that satisfies the conditions $a \leq u < a + m$, and $u \equiv u_j (modulo\ m_j)$ for $1 \leq j \leq r$.*

We use the Theorem for the case $a = 0$ and $m \geq M_{lc}$. Let $2, 3, 5, ..., p_j$ be the series of prime numbers up to the j-th prime such that $2 \cdot 3 \cdot 5 \cdot ... \cdot p_{j-1} < M_{lc} \leq 2 \cdot 3 \cdot 5 \cdot ... \cdot p_j$. We run j parallel versions of our protocol. The i-th version runs the protocol with $M_{lc} = p_i$. Each message carries the value of j clocks, one clock value for each version. The computation of the new clock value of some version i uses the values received for this particular version and is independent from the computation of all the other versions. Thus, the i-th version converges within expected $p_i \cdot 2^{2(n-f)}$ pulses. Therefore, the expected time for all the versions to be synchronized is less than $(p_1 + p_2 + \cdots + p_j) \cdot 2^{2(n-f)}$. This is an upper bound on the expectation since it corresponds to a scenario in which version i starts to synchronize after every version $k < i$ is already synchronized.

Now we apply the Chinese remainder theorem to show that every combination of those values is mapped to one and only one number in the range 0 to $2 \cdot 3 \cdot 5 \cdots p_j$. A well-known technique could be used in order to convert such a representation to its mapping (e.g., by Garner methods, c.f. p. 274 [Kn-81]).

The Chinese remainder theorem could be used for other implementations of distributed counters based on the number presentation method suggested in [ST-67]. One possible use is as a memory and communication efficient distributed counter. Let DC be a distributed counter that is maintained by a set of processors $P_1, P_2, \ldots, P_j$ that are triggered by a common pulse. P_i increments the counter mod p_i in every trigger. P_i does not need to store the entire bits of the clock or to send messages to indicate the carry (when its counter wraps around). Thus, when the counter is incremented no communication between processors is needed. Only when the value of the counter is to be scanned is communication required.

4 Semi-synchronous Protocol

In this section we drop the assumption of common pulses. We present a self-stabilizing randomized protocol for semi-synchronous systems. Due to space constraints, the formal description of the protocol and the full correctness proof are excluded from this section.

[2] It is reasonable to think of n and f as being small when a single processor can efficiently compute a task and additional processors are added only to ensure reliability. Let the *reliability* be $f/(n+f)$, the ratio of the number of faulty processors to the total number of processors. To reach a reliability of 0.25, the number of processors needed (and thus, in general terms, the blowup in the hardware and cost) is four. To improve the reliability to $2/7 \approx 0.28$ the blowup would be 7. Asymptotically, we need an infinite blowup to reach reliability of 1/3. Thus, most devices would use a relatively small number of processors for which our protocol stabilizes in a relatively short time.

Our protocol uses the fault-tolerant averaging function first introduced in [DL+-86] for solving approximate agreement and later used for clock synchronization in [WL-88]. Given a multiset of values, a processor applies the function by discarding the f highest and f lowest values and then taking the midpoint of the remaining values. It has been shown that this function, when used in the context of the protocols of [DL+-86, WL-88], approximately halves the range of values held by the nonfaulty processors.

In our situation, with bounded clocks, the notions of "highest" and "lowest" must be appropriately modified. But the real difficulty in directly applying the previous result is that the analysis showing the range is cut in half depends on all nonfaulty processors working with approximately the same multisets at each "round". The multisets can differ arbitrarily in the values corresponding to the faulty processors, but the values corresponding to nonfaulty processors must be close to the same (allowing for error introduced by clock drift and uncertain message delays). This "round" structure can be achieved because the actions of the processors are roughly synchronized in time in the [WL-88] protocols, due to the assumption of initial synchronization or of distinguished initial states.

Since our protocol is self-stabilizing, it cannot rely on either of those assumptions. Thus using the fault-tolerant averaging function in the obvious manner, with the processors starting with arbitrary information and collecting clock values at arbitrary times, would not ensure that the function is applied at the processors in rounds. For instance, P could apply the function to a multiset M, then subsequently Q could apply the function to a multiset M' that reflects P's new value instead of P's old value.

To achieve some sort of approximate rounds for applying the fault-tolerant averaging function, we first use randomization to bring all the clock values of the nonfaulty processors close to each other. Once this is achieved, all the nonfaulty processors collect (approximately) the same multisets from all the nonfaulty processors. In this stage the midpoint averaging function can be shown (cf. [WL-88]) to approximately halve the nonfaulty clock values, thus overcoming the ongoing effects of clock drift and uncertainty of message delay.

We now describe the protocol. A processor P_i has two synchronization procedures. The first is called the *averaging procedure* and the second is the *jumping procedure*. The averaging procedure is executed when the value of $clock_i$ is in a range greater than 0 and smaller than δ and T_a time has elapsed since the previous time that $clock_i$ had a value in this range. The jumping procedure is executed when T_j time has elapsed since the previous execution of the jumping procedure and P_i is not currently in the range dedicated for executing the averaging function. P_i measures T_a and T_j using its physical clock. Roughly speaking, the jumping procedure causes the clocks of the nonfaulty processors to be within a small range. Then the averaging procedure keeps the clocks of the nonfaulty processors in a small range by approximately halving the range each time the clock values wrap around.

Both the synchronization procedures of processor P_i start with a request for clock values. During the execution of the averaging procedure, a processor measures $2(d + \epsilon + \delta)$ time in order to make sure that all the requests for clock values arrive at their destinations and the responses return before it proceeds to decide on a new clock value. Thus each execution of the averaging procedure takes some period of time. We define the *symmetric clock* of $clock_i$ to be $clock_i + M_{lc}/2 \pmod{M_{lc}}$. In both procedures, if P_i finds $n - f$ clock values within a small range δ from $clock_i$, then P_i eliminates f values from each side of the symmetric clock value[3]. Then, in the jumping procedure, P_i chooses one of the clock values at random from the reduced clock values list, while in the averaging procedure, P_i chooses the midpoint of the reduced clock values list. In both procedures, if less than $n-f$ processors are found within δ from $clock_i$, P_i chooses randomly one of the clock values.

[3]For instance, if the collected values are 2,3,10,11, the symmetric clock value is 7 and $f = 1$, then 3 and 10 are eliminated.

4.1 Correctness Proof Sketch of the Semi-synchronous Protocol

A period of time is a *jumping period* if no nonfaulty processor executes the averaging procedure during this period. We choose T_a to be $2(n-f)(5T_j+d+\epsilon)(1+\rho)^2$. The next lemma proves that the above choice yields the existence of a period of length $5T_j(1+\rho)$ that is a jumping period.

Lemma 4.1 *Every T_a time there is a jumping period that is at least $5T_j(1+\rho)$ long.*

Proof: A processor measures time by the use of its physical clock, whose drift rate from real time is at most ρ. Thus, if a processor measures a period of time T on its physical clock, then the real time elapsed during the measurement is at least $T/(1+\rho)$ and at most $T(1+\rho)$. By the way T_a is chosen, in every period of length $2(n-f)(5T_j+d+\epsilon+\delta)(1+\rho)^2/(1+\rho) = 2(n-f)(5T_j+d+\epsilon+\delta)(1+\rho)$, every nonfaulty processor executes the averaging function at most once. A processor measures $2(d+\epsilon+\delta)$ time in order to make sure that the requests for clock values arrive at their destinations and the responses arrive before it decides on a new clock value. Thus, the time that the averaging function is executed by each processor in a period of $2(n-f)(5T_j+d+\epsilon+\delta)(1+\rho)$ is no more than $2(d+\epsilon+\delta)(1+\rho)$. Hence, the total time of averaging of all the processors during a period of $2(n-f)(5T_j+d+\epsilon+\delta)(1+\rho)$ is no more than $(n-f)2(d+\epsilon+\delta)(1+\rho)$. Therefore, the total non averaging time is at least $2(n-f)(5T_j+d+\epsilon+\delta)(1+\rho) - (n-f)2(d+\epsilon+\delta)(1+\rho) = 2(n-f)5T_j(1+\rho)$. By the pigeon hole principle at least one jumping period is of length $2(n-f)5T_j(1+\rho)/(n-f+1) > 5T_j(1+\rho)$. ■

A *safe configuration* is a system configuration in which the nonfaulty processors' clocks are within $\delta/8$ of each other. Moreover, in case a processor is in the middle of collecting clock values then all the clock values in transit sent by nonfaulty processors are within this range too.

We use the following assumptions in our correctness proof:

Assumption 1: $(1+\rho)^2 < 6/5$, thus $\rho < 0.095$.

Assumption 2: $(n-f)\epsilon + 2T_j(1+\rho)\rho < \delta/8$.

Lemma 4.2 *During any jumping period of length $5T_j(1+\rho)$, with probability at least $1/n^{6(n-f)}$, the system reaches a safe configuration.*

Sketch of proof: We prove the lemma by presenting a sequence of random choice results, that forces the system to reach a configuration in which the clocks of all the nonfaulty processors are less than $\delta/8$ apart. This sequence of random choice results has probability of at least $1/n^{6(n-f)}$ to occur. Let c be the configuration at the beginning of the jumping period.

Without loss of generality we assume that the number of faulty processors f is the maximal possible[4] that does not violate the inequality $n > 3f$. Let c be the first configuration in a choosing period. For every nonfaulty processor P, *luck* counts the number of other nonfaulty processors that have clocks within $T_r = \delta + 4(T_j(\rho+\rho^2)) + \epsilon$ of P's clock in the configuration c. Each nonfaulty processor that has at least $n-2f-1$ such surrounding clock values is called an *anchor*.

We claim that all the anchor processors are at most $2T_r$ apart. Assume towards contradiction that there are two nonfaulty anchor processors, P and Q, such that their clock values are more than $2T_r$ apart. Thus, P is surrounded by $n-2f-1$ nonfaulty processors and Q is surrounded by $n-2f-1$ *different* nonfaulty

[4]In case there are fewer faulty processors, one could assume that some of the nonfaulty processors "only behave" like nonfaulty processors.

processors. Therefore, the total number of nonfaulty processors is at least $2(n-2f) = 2n-4f > n-f$, contradiction.

Note that it is possible that no anchor processor exists. In this case *luck* chooses one nonfaulty processor to be an anchor processor.

Then *luck* chooses a single anchor processor A out of the anchor processors.

Until every nonfaulty processor executes the jump procedure twice *luck* uses the following strategy: Every time a processor, P_j, chooses a clock value and the value of the clock of A is a possible choice (i.e., either P_j does not find $n-f$ within δ range or A is in the reduced clock values list), this value is chosen; otherwise the value of $clock_j$ is not changed. Let c_1 be the first configuration reached from c after each processor executes the jump procedure at least twice with results according to the strategy of *luck*. Let E_1 be the execution that starts with c and ends with c_1. Since in a jumping period every nonfaulty processor chooses a clock value at least once in every period of length $T_j(1+\rho)$, c_1 occurs at most $2T_j(1+\rho)$ time after c.

We now show that in c_1 all the nonfaulty processors are within $2T_r + 2T_j(1+\rho)2\rho$ of each other. We first show that any nonanchor processor, P, assigns the value of A's clock to P's clock either in the first execution of the jump procedure or in the second one. Every processor collects the clock values during every execution of the jump procedure. In particular a nonanchor processor, P_j, receives the value of the clock of A before the second execution of the jump procedure. Next we show that, in the second execution of the jump procedure P_j can choose the value of A's clock.

The choice of P_j is restricted to a subset of the clock values that P_j read, only if P_j finds $n-f$ clock values within δ range of $clock_j$. Since P_j is a nonanchor processor it holds in c_1 that there are less than $n-f$ processors within δ range of $clock_j$. Moreover, no nonfaulty processor can assign a clock value within δ range of $clock_j$ since: (1) Every nonfaulty processor P_k that changes its clock value by the use of the jump procedure assigns the clock value of A (with up to ϵ range from the clock of A). (2) Every nonfaulty processor P_k that does not change its clock value by the use of the jump procedure can have a rate of drift from the clock of P_j of at most 2ρ. Thus, the difference between $clock_j$ and $clock_k$ can be shorten by at most $2T_j(1+\rho)2\rho = 4(T_j(\rho+\rho^2))$. Thus, if P_k was more than $T_r = \delta + 4(T_j(\rho+\rho^2)) + \epsilon$ apart from P_j in c then P_j cannot consider P_k to have a clock in δ range from $clock_j$ during E_1 (unless P_j assigns $clock_j$ by the value of the clock of A).

This proves that in c_1 all the nonanchor processors are within $\epsilon + 4T_j\rho$ from A's clock. The anchor processors that do not assign the clock value of A to their clock during E_1 were at most $2T_r$ apart in c, thus they are at most $2T_r + 2T_j(1+\rho)2\rho$ apart in c_1.

The fact that all the nonfaulty processors are within a small range of each other is used to define a new anchor processor A'. A' is the nonfaulty processor left after removing f nonfaulty processors with the highest clock values (mod M_{lc}) and f nonfaulty processors with the smallest values (mod M_{lc}).

¿From c_1 and until every processor executes the jump procedure at least twice, *luck* continues as follows: Any processor P_i that is in the process of collecting clock values in c_1 does not change $clock_i$ in the first execution of the jump procedure. For any other execution of the jump function, *luck* intervenes to fix the result to be the clock of A' or a clock of a processor that has already set its clock to the value of A''s clock since c_1. We have to prove that the above is a possible result of the jump function. This is obvious when the processor does not find $n-f$ processors within δ from its clock, since the choice is not restricted. It is also clear for the first set of processors that execute the jump procedure and use the clock values in c_1 as the base for the decision on the new clock value. Moreover, since *luck* intervenes and fixes all those results to be the value of the clock of A', the reduced list of every processor that uses the new clock values includes either the clock of A' or a clock of a processor that assigned its clock by the clock of A'.

Hence, in the first configuration, c_2, that follows the first two executions of the jump function of all the processors following c_1, all the nonfaulty processors are within $(n-f)\epsilon + 2T_j(1+\rho)\rho$ of each other, which, by assumption 2, is less than $\delta/8$.

Following c_2 any processor that is waiting for answers in the process of collecting clocks does not change its clock value. Thus, $(d+\epsilon)(1+\rho)$ time after c_2 a safe configuration is reached.

The length of the execution is $2T_j(1+\rho)$ until c_1 is reached, $2T_j(1+\rho)$ from c_1 to c_2 and additional $(d+\epsilon)(1+\rho)$ until a safe configuration is reached. Thus, a safe configuration is reached following $(4T_j + d+\epsilon)(1+\rho) < 5T_j(1+\rho)$ from c. By Assumption 1, $5T_j(1+\rho) < 6T_j/(1+\rho)$. Thus any processor could choose at most six times in such a range. Thus the total number of interventions is $6(n-f)$. ■

Lemma 4.3 *In any configuration of any execution that starts with a safe configuration, the clock values of all the nonfaulty processors are within at most $\delta/2$ of each other.*

The main observation made for the proof of the above lemma is that starting in a safe configuration every processor that either executes the jumping or the averaging procedure finds $n-f$ clock values within δ from its clock value. Thus, the new clock value chosen when jumping or averaging is in the range of clock values of the nonfaulty processors. The averaging procedure approximately halves the range of the clock values of the nonfaulty processors whenever they pass the zero clock value.

Theorem 4.4 *In expected $O(T_a n^{6(n-f)})$ time the system stabilizes.*

5 Concluding Remarks

Extensive research has been done to find efficient clock synchronization protocols in the presence of Byzantine faults. In this work we considered a more severe (and realistic) model of faults, i.e., one that takes into account transient faults as well as Byzantine faults. When arbitrary corruption of state is possible, as is often the case with transient faults, it is no longer reasonable to approximate unbounded clocks with bounded clocks, no matter how large. Consequently, clocks that can take on only a bounded number of values (and wrap around when appropriate) have been assumed in this paper. We presented two randomized self-stabilizing protocols for synchronizing bounded clocks in the presence of f Byzantine processor failures, where $n > 3f$.

We believe that our observations and definitions for the types of faults to be considered and the type of clocks (namely, bounded) reflect reality and open new directions for research. Protocols designed under our fault tolerance model are more robust than existing clock synchronization protocols. Therefore, such protocols might be preferred by the system implementer over protocols that cope with *only* Byzantine faults.

Acknowledgment: Many thanks to Brian Coan, Injong Rhee and Swami Natarajan for helpful discussions.

References

[Dij-74] E. W. Dijkstra, "Self stabilizing systems in spite of distributed control," *Communication of the ACM*, vol. 17, 1974, pp. 643–644.

[DHS-86] D. Dolev, J. Y. Halpern, and H. R. Strong, "On the possibility and impossibility of achieving clock synchronization," *Journal of Computer and Systems Science,* vol. 32, no. 2, 1986, pp. 230–250.

[DIM-91] S. Dolev, A. Israeli and S. Moran, "Uniform dynamic self stabilizing leader election," *Proc. of the 5th International Workshop on Distributed Algorithms*, 1991, pp. 167–180.

[DIM-95] S. Dolev, A. Israeli, and S. Moran, "Analyzing Expected Time by Scheduler-Luck Games," *IEEE Transactions on Software Engineering*, vol. 21, no. 5, May 1995.

[DL+-86] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl, "Reaching approximate agreement in the presence of faults," *Journal of the ACM,* vol. 33, 1986, pp. 499–516.

[DW-93] S. Dolev and J. L. Welch, "Wait-free clock synchronization," *Proc. of the Twelfth ACM Symp. on Principles of Distributed Computing,* 1993, pp. 97–108.

[GP-93] A. S. Gopal and K. J. Perry, "Unifying self-stabilization And fault-tolerance," *Proc. of the Twelfth ACM Symp. on Principles of Distributed Computing,* 1993, pp. 195–206.

[HS+-84] J. Halpern, B. Simons, R. Strong, and D. Dolev, "Fault-tolerant clock synchronization," *Proc. of the Third ACM Symp. on Principles of Distributed Computing,* 1984, pp. 89–102.

[Kn-81] D. E. Knuth, *The art of computer programming*, Vol. 2, 2nd edition, Addison-Wesley, 1981.

[LM-85] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *Journal of the ACM*, vol. 32, no. 1, 1985, pp. 1–36.

[LSP-82] L. Lamport, R. Shostak and M. Pease, "The Byzantine generals problem," *ACM Trans. on Prog. Lang. and Sys.*, vol. 4, no. 3, July 1982, 382-401.

[MS-85] S. Mahaney and F. Schneider, "Inexact agreement: accuracy, precision and graceful degradation," *Proc. of the Fourth ACM Symp. on Principles of Distributed Computing,* 1985, pp. 237–249.

[RSB-90] P. Ramanathan, K. G. Shin, and R. W. Butler, "Fault-tolerant clock synchronization in distributed systems," *IEEE Computer*, October, 1990, pp. 33–42.

[ST-87] T. K. Srikanth and S. Toueg, "Optimal clock synchronization," *Journal of the ACM*, vol. 34, no. 3, 1987, pp. 626–645.

[ST-67] S. Szabo, and R. I. Tanaka, *Residue arithmetic and its applications to computer technology*, McGraw-Hill, 1967.

[WL-88] J. L. Welch and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," *Information and Computation,* vol. 77, no. 1, 1988, pp. 1–36.

Paper Number 10

Possibility and Impossibility Results for Self-Stabilizing Phase Clocks on Synchronous Rings

Chengdian Lin and Janos Simon

Possibility and Impossibility Results for Self-Stabilizing Phase Clocks on Synchronous Rings

Chengdian Lin Janos Simon

Department of Computer Science
University of Chicago

Abstract

We consider the problem of obtaining self-stabilizing algorithms for finite phase clocks on uniform synchronous rings. In such a system, every processor is identical and has a local clock. A self-stabilizing protocol guarantees that all clocks will eventually have the same value regardless their initial values, and thereafter, each processor increments its clock by one in every step. We show that there is no solution to this problem if the processors are finite automata. On the other hand, we present such a protocol which works for any rings of odd size and for rings of size $2p$, where p is a prime. The idea yields a protocol of $O(\sqrt{n})$ states per processor for arbitrary rings. This improves the previous result of $O(n)$ states.

1 Introduction

We study self-stabilizing clocks in a synchronous setting. In such a system, there is a collection of identical processors, each with an internal, local, bounded clock. Computation proceeds synchronously: at discrete points in time every processor receives an external, global 'clock pulse'. Ideally, all local clocks – which from now on we will often call *phase clocks*, will have the same value, and will change at every global clock pulse. This is accomplished by a local protocol, which, besides the global pulse, may also use the state of the processor, as well as the states of adjacent processors. This is an interesting and important problem (see the references for motivation and applications.) Unless otherwise stated, we shall assume in this abstract that the processors are connected in a ring with a sense of direction. The task is to design a self-stabilizing protocol that guarantees that eventually all the phase clocks have the same value, regardless of the initial configuration of the system, and from that point on, they all change, in unison, at every global pulse. This *Phase Clock* problem can be specialized by specifying the bound on the clock (e.g. only finitely many values) and a bound on the number of states of the processor.

It is easy to see that the problem can be reduced to leader election: once a leader is elected, it can impose its clock to its neighbors, who impose it to theirs, until all clocks are synchronized. Part of the interest of the question is that it is *simpler* than leader election. There are deterministic phase clock algorithms for synchronous anonymous rings. But there is no deterministic leader election for synchronous anonymous rings (consider a ring configuration in which every processor has the same state.) The question is whether there exist solutions where each processor uses only finite memory – i.e. a ring of finite automata. In this paper, we show that there is no finite memory self-stabilizing solution to *Phase Clock* problem, either for unidirectional or bi-directional rings. On the other hand, we also show that such solution exists for many ring sizes.

We briefly review previous work on phase clocks. In [3], Gouda and Herman presented a simple and

elegant self-stabilizing clock unison protocol for a general graph: at each clock pulse the processor sets its local clock to 1+MAX{clocks of its neighbors}. Unfortunately this protocol requires unbounded clocks, so the size of each processor must be unbounded. The algorithm was strengthened by Arora, Dolev and Gouda in [1], where it is shown that if the number of processors n in the system is known, then an $O(n)$ clock size is sufficient: simply use MIN instead of MAX, and addition mod n. They also show that the scheme does not work if the modulus is too small. Herman and Ghosh [4] also presented a simple randomized protocol for processors that are finite automata. Their algorithm, as presented, is slow to stabilize: the convergence span (the expected time to attain a stable configuration in the worst case) is not bounded by a polynomial. In a ring with orientation their algorithm can be modified to run in expected polynomial time. Yet [10] has presented an algorithm for odd size rings.

In this paper, we study deterministic protocols for finite phase clocks. We first show that any deterministic self-stabilizing *Phase Clock* protocol for a ring of size n, where n an integer of some special form, has at least $\Omega(\sqrt[3]{\log n})$ states per processor. We then present a new deterministic strategy, that we call *Firing Generals*, which allows us to derive efficient protocols for rings of finite automata. We obtain self-stabilizing *binary-clock* protocols for rings of odd size and of size $2p$, where p is a prime. The technique can be extended to rings of length odd or $2p^k$ for $k \leq 2$. The idea also yields a deterministic protocol of $O(\sqrt{n})$ states for arbitrary size rings (compared to the $O(n)$ states protocol in [1].) We also present a easy lower bound on protocol speed, that it, any self-stabilizing protocol has convergence span of $\Omega(D)$, where D is the diameter of the graph.

2 Model and Impossibility Results

In a k-clock system, each processor has a state of form (l, c), where $l \in L$ is the label and $c \in C = \{0, 1, \cdots, k-1\}$ is the clock value. All states in the system form a (global) configuration. Let $\mathcal{P}$ be a protocol, in a transition of $\mathcal{P}$ at configuration γ, every processor P in the system simultaneously executes one step of $\mathcal{P}$ which is a function δ of P's old state and neighbors' states, thereby resulting the next state of P and the next (global) configuration $\delta(\gamma)$. A computation of the system is an infinite sequence of configurations resulting from an infinite sequence of transitions.

A *Phase Clock* system is a k-clock system for some finite integer k, and it satisfies the following two properties:

1. **[Unison]** There is a time t such that if all clocks in the system have the same value as c at time t, then for any $i > 0$, all clocks at time $t + i$ will have the same value as $c + i \pmod{k}$.
2. **[Self-Stabilization]** From any initial configuration, there exists a finite t such that the system will reach **Unison** at time t.

Definition. *Convergence Span* is the maximum number of transitions needed to reach a ***Unison*** configuration from an arbitrary configuration. For a self-stabilizing protocol $\mathcal{P}$, we denote its *Convergence Span* by $CS(\mathcal{P})$.

In the following, we will show that there is no constant space self-stabilizing protocol which solves *Phase Clock* problem deterministically. In particular, we show that: any self-stabilizing *Phase Clock* protocol for a ring of size n, where n an integer of some special form, has at least $\Omega(\sqrt[3]{\log n})$ states per processor.

2.1 Lower Bound for Unidirectional Rings

Lemma 1. Given any deterministic protocol $\mathcal{P}$ for synchronous uniform unidirectional ring. Let Q ($|Q| = m$) be the state set of each processor under $\mathcal{P}$. Then there exists a periodic sequence of states $s_0, s_1, \cdots, s_{k-1}$ where $s_i \in Q$ and $k \leq m^2$, such that

$$\delta(s_i, s_{i+1}) = s_{i+2} \quad \text{where addition is modulo } k$$

Proof. Let t_0 be some state in Q. Set $t_1 = t_0$. For $i > 1$, define t_i as

$$t_i \quad = \quad \delta(t_{i-2}, t_{i-1}) \tag{1}$$

There are at most m^2 distinct pairs on the right hand side of (1). Thus there exist u, v (both between 0 and $m^2 - 1$) such that $(t_u, t_{u+1}) = (t_v, t_{v+1})$. Set $k = v - u$, $s_0 = t_u$ and $s_i = t_{u+i}$ (for $1 \leq i \leq k-1$), and we are done. ∎

Definition. Let $\gamma = (a_0, a_1, \cdots, a_{n-1})$ be a configuration for a ring of size n. A configuration γ' is an i-shift of γ $(0 < i < n)$, if $\gamma' = (a_i, a_{i+1}, \cdots, a_{n-1}, a_0, a_1, \cdots, a_{i-1})$.

It is know from number theory that for any integer r, $lcm(1, 2, ..., r) < e^{2r}$ (see e.g. [11].)

Theorem 1. There are infinitely many n such that, any deterministic protocol $\mathcal{P}$ that solves phase clock problem for synchronous uniform unidirectional rings of size n must have at least $\Omega(\sqrt{\log n})$ states per processor.

Proof. For any given integer r, let $n = lcm(1, 2, ..., r)$. We want to show that $\mathcal{P}$ uses at least $\sqrt{r}$ states per processor for ring of size n. Suppose not, then $\mathcal{P}$ uses $m < \sqrt{r}$ states per processor. By **Lemma 1**, there is a periodic sequence $s_0, s_1, \cdots, s_{k-1}$ of length $k \leq m^2 < r$. Let $\gamma_{sub} = (s_0, s_1, \cdots, s_{k-1})$ be a sub-configuration. Since k divides n, we define a ring configuration

$$\gamma = (\gamma_{sub}, \gamma_{sub}, \cdots, \gamma_{sub})$$

Now it is easy to see that $\delta(\gamma)$ is a 1-shift of γ. If γ_{sub} has two or more different clock values, then any configuration after γ will have two or more different clock values and the unison will never be reached. If γ_{sub} represents a single clock value, this clock value will never change after γ, but then the system is not a clock.

Therefore $\mathcal{P}$ must use at least $\sqrt{r} = \Omega(\sqrt{\log n})$ states in each processor for this type of special n. ∎

Corollary 1. There is no constant space deterministic self-stabilizing protocol that solves the *Phase Clock* problem for unidirectional uniform rings.

2.2 Lower Bound for Bi-directional Rings

We proved the existence of periodic sequences on pairs in the unidirectional case. We are going to prove that a similar periodic sequence on triples also exists in the bi-directional case.

Lemma 2. Given any deterministic protocol $\mathcal{P}$ for synchronous uniform bi-directional ring. Let Q ($|Q| = m$) be the state set of each processor under $\mathcal{P}$. Then there exists a periodic sequence of states $s_0, s_1, \cdots, s_{k-1}$ where $s_i \in Q$ and $k \leq m^3$, such that

$$\delta(s_{i-1}, s_i, s_{i+1}) = s_{i+2} \quad \text{where addition is modulo } k$$

Proof. Let t_0 be some state in Q. Set $t_2 = t_1 = t_0$. For $i > 2$, define t_i as

$$t_i = \delta(t_{i-3}, t_{i-2}, t_{i-1}) \quad (2)$$

There are at most m^3 distinct triples on the right hand side of (2). Thus there exist u, v (both between 0 and $m^3 - 1$) such that $(t_u, t_{u+1}, t_{u+2}) = (t_v, t_{v+1}, t_{v+2})$. Set $k = v - u$, $s_0 = t_u$ and $s_i = t_{u+i}$ (for $1 \leq i \leq k-1$), and we are done. ∎

Theorem 2. There are infinitely many n such that, any deterministic protocol $\mathcal{P}$ solves phase clock problem for synchronous uniform bi-directional rings of size n must have at least $\Omega(\sqrt[3]{\log n})$ states per processor.

Proof. The proof is basically the same as for **Theorem 1**. For any given integer r, let $n = lcm(1, 2, ..., r)$. We want to show that $\mathcal{P}$ uses at least $\sqrt[3]{r}$ states per processor for ring of size n. Suppose not, then $\mathcal{P}$ uses $m < \sqrt[3]{r}$ states per processor. By **Lemma 2**, there is a periodic sequence $s_0, s_1, \cdots, s_{k-1}$ of length $k \leq m^3 < r$. Let $\gamma_{sub} = (s_0, s_1, \cdots, s_{k-1})$ be a sub-configuration. Since k divides n, we define a ring configuration

$$\gamma = (\gamma_{sub}, \gamma_{sub}, \cdots, \gamma_{sub})$$

Thus $\delta(\gamma)$ is a 2-shift of γ. The remainder of proof is the same as that of **Theorem 1**. ∎

Corollary 2. There is no constant space deterministic self-stabilizing protocol that solves the *Phase Clock* problem for bi-directional uniform rings.

The above lower bounds do not hold for all values of n. In the following section, we present a constant space deterministic self-stabilizing protocol for binary clocks on rings of certain size (odd and $2p$.)

3 The Firing Generals Protocol

3.1 The Basic Protocol: Informal Outline

We assume that the ring has a sense of orientation. Processors will check their neighbors: as long as the neighbors have the same phase clock value, the local clock is simply incremented (mod 2). A problem arises when a processor discovers that its neighbor has a different phase clock. We call an *army* or a *segment* a maximal group of consecutive members (processors) with the same clock. The two members at two ends are called *generals*, one is the *front-general* (FG) and the other is the *rear-general* (RG.) Other members of an army are called *soldiers*.

We need to merge the processors into a single segment. We will do this by requiring that armies fight, and eventually all but one disappear. Unlike most real warfare, we'll have the generals do all the fighting.

A general can be in one of two modes *Firing* or *Quiet*. A soldier should be in *Quiet* mode only (except in some illegal initial configurations.) At the border of two armies, there should be at most one firing general. Firing generals conquer quiet generals and soldiers, enlarging their armies.

As usual, there are two problems: generals should eventually fire, but no two adjacent generals (in different armies) should fire. We break the symmetry by allowing general to actually fire only when its local clock is 1. Since two adjacent enemy generals have different local clocks, exactly one of them will fire, and the other becomes *Quiet*.

If the front and rear generals of an army could communicate instantaneously with each other, the protocol above would suffice. The problem is to coordinate them, to make sure that both generals of an

army are firing, or both generals are quiet. We call an army where both generals are in *Firing* mode a *firing army*. A firing army will expand by conquering enemies at its two ends. It will increase its size and remain a firing army at next moment. This continues until the number of armies decreases. If a protocol guarantees the existence of a firing army, then the ring will converge to a single army.

In reality, two generals are apart, and could be in different states. Then one of them may expand, while the other is quiet, and is being conquered by its neighbor. To prevent this, two generals constantly check that they are in the same state. More precisely, a *Firing* general sends a *bullet* to the other general in the same army, while a *Quiet* general sends a *Stop-Firing* signal. Upon receiving a *bullet*, a *Firing* general will conquer an enemy and the newly surrendered general will become the new firing general. A *Firing* general becomes *Quiet* if instead a *Stop-Firing* signal is received.

If an army is not firing, then both generals will be *Quiet* soon (after at most the size of army ticks.) For convenience, we call a *Quiet* mode general *Peace* general if the neighbor enemy general is also *Quiet*. If both generals of an army $\mathcal{A}$ are *Peace* generals, then $\mathcal{A}$ will run a local synchronous protocol to let both generals of $\mathcal{A}$ enter the *Firing* mode at the same time. This protocol is essentially a firing squad synchronization protocol, hence the name of the algorithm.

In order to achieve the synchronization of two generals of an army $\mathcal{A}$, we ask $\mathcal{A}$ to locate its center (two centers if the size of $\mathcal{A}$ is even.) By doing this, the rear *Peace* general constantly sends *Request* signal to the front general. When the front *Peace* general receives a *Request* signal, the *Center Locating* protocol is started. The front general marks itself as the center C initially, and sends *Verify* signals to its generals and waits for signals to return. If two *Verify* signals are returned at different time, the marked center is adjusted to right and the newly marked center will send *Verify* signals again. If two *Verify* signals are returned at the same time, the marked center knows it is the real center and then broadcasts OK signals to two generals. This guarantees that the two generals will receive the OK signal at the same time, so they enter the *Firing* mode at the same time. In the case of two generals at the border entering *Firing* mode at the same time, they must have different clocks so we use our rule to break the symmetry by letting the firing general with clock value 0 become *Quiet* at the next clock tick.

Once an army is firing we want to keep its firing status until the number of armies decreases. To avoid an enemy general from becoming firing at the border, we ask a general to cancel the *Center Locating* protocol (by sending a *Stop-Firing* signal) as soon as it is no longer a *Peace* general.

This strategy almost always works. Except for a very special case that we will discuss below, it guarantees the existence of a firing army, and therefore the correctness of the protocol. The bad scenario is that only FG (or RG) of each army can enter *Firing* mode after the *Center Locating* protocol, because the other general, i.e. RG (FG) cancels the *Center Locating* protocol too late. Notice that this will not happen if there are only two armies, since both generals of an army enter the *Firing* mode at the same time. If one general cancels the *Center Locating* protocol, the other general must cancel the protocol as well. The problem is that there may an infinite chain of such restart/cancel moves that propagate along the ring. We show that the initial conditions must be very special for this unlucky scenario to happen. In particular, all armies must have the same size.

As a consequence, our protocol works for rings of odd size, and for rings of size $2p$ when p is prime, because there must be an even number of identical length armies for the deadlock to occur, and this is impossible for such ring sizes. Similar techniques, not covered in this abstract, extend the result to rings of size $2p^k$ for p prime and k a fixed exponent.

3.2 Basic Protocol: Details

Suppose that every soldier has a clock and a state, and it receives and sends signals. We consider the state as having 5 components:

1. Clock: $\{0, 1\}$
2. Status: $\{C, N\}$
3. Mode: $\{F, Q\}$
4. Left Signal: $\{\overleftarrow{B}, \overleftarrow{S}, \overleftarrow{R}, \overleftarrow{V}, \overleftarrow{RV}, \overleftarrow{K}, \overleftarrow{N}\}$
5. Right Signal: $\{\overrightarrow{B}, \overrightarrow{S}, \overrightarrow{R}, \overrightarrow{V}, \overrightarrow{RV}, \overrightarrow{K}, \overrightarrow{N}\}$

The first component is the binary clock component. The status component indicates whether a soldier is a center of the army. The third component is to indicate a soldier's mode, *Firing* or *Quiet*. The fourth and fifth component contains the signal a soldier has. Signals travel inside an army either from left to right or vise versa, and the they have the following meanings:

B is the *Bullet*.

S is the *Stop-Firing* signal.

R is the *Request* signal to ask for running the *Center Locating* protocol.

V is the *Verify* signal sent from center to generals.

RV is the *Returning Verify* signal sent from generals to center.

K is the *OK* signal sent from center(s) to generals to let them enter the *Firing* mode.

N means no signal.

We use P_i to denote the value in i-th component of processor P, and parentheses $(a, b, c, d, e)_P$ to denote the whole state. We often denote P's left neighbor by P^l, and P's right neighbor by P^r. We also use $[P_i^l, P_i, P_i^r]$ to denote the P's vicinity view of i-th component in three processors, and $[P_i^l, P_i]$ (or $[P_i, P_i^r]$) to denote the local view of two adjacent processors.

Definition. If two adjacent processors have different clocks ($[P_1^l, P_1] = [a, \bar{a}]$), then the left processor P^l is called a **Front General** (FG) and the right processor P is called a **Rear General** (RG.)

A processor P is a general if P is either a front general or a rear general, otherwise P is a soldier.

Definition. Assume P and P^l are generals:

1. P is **Firing** if P is in the firing mode ($P_3 = F$), otherwise P is **Quiet**.
2. A firing general P is called a **Conquering General** if P's clock is 1 ($(1, N, F, *, *)_P$.)
3. A quiet general P (P^l) is called a **Peace General** if the adjacent enemy general P^l (P) is also quiet, that is $(a, *, Q, *, *)_P$ and $(\bar{a}, *, Q, *, *)_{P^l}$.

Definition. An army $\mathcal{A}$ is a maximal contiguous sequence of processors with the same clock.

Definition. If $P_2 = C$, then processor P is called a marked center (or center.)

Definition. Signals have ranks: 6,5,4,3,2,1,0 are the ranks for S, K, V, RV, R, B, N respectively. Let X, Y be two signals of opposite directions, we say X kills Y if $Rank(X) > Rank(Y)$.

Normally, signals are passed according to their directions within an army. By above definition, when two signals (from left and right) of different ranks are received by a processor, the lower rank signal will be killed.

Our protocol has eight rules. Rule **R1** is for a singleton army. Rules **R2 – R4** are for the border which has firing general. Rules **R5 – R8** are for *Center Locating*.

A rule is *enabled* if the processor changes its state after applying it. Rules have priorities, that is, **Rule** I is applied before **Rule** J if $I < J$. So, at a pulse, a processor applies only one rule which is the first enabled rule by the priorities.

R1 (Singleton Army Rule):

If a general is the only member in the army, it will reinitialize its state. This action also causes a general to surrender if its clock is 0.

R1A01: **if** $[P_1^l, P_1, P_1^r] = [\bar{a}, a, \bar{a}]$ **then**
R1A02: $(0, N, Q, \overleftarrow{N}, \overrightarrow{N})_P$
R1A03: **endif;**

R2 (Conquering Rule):

Upon receiving a bullet B, a conquering general CG conquers the enemy general EG and EG becomes the new firing general. Since processors move synchronously, EG makes its clock the same as CG's clock and becomes firing, while CG becomes quiet.

Procedure Action-Of-Losing-Front-General
R2A01: **if** $([P_1^l, P_1, P_1^r] = [1, 0, 0])$ **and** $(1, *, F, *, \overrightarrow{B})_{P^l}$ **then**
R2A02: $(0, N, F, \overleftarrow{B}, \overrightarrow{N})_P$ **{also send** $\overleftarrow{B}$**, see R4}**
R2A03: **endif;**

Procedure Action-Of-Winning-Rear-General
R2B01: **if** $([P_1^l, P_1, P_1^r] = [1, 1, 0])$ **and** $(1, *, F, *, \overrightarrow{B})_P$ **then**
R2B02: $(0, N, Q, \overleftarrow{B}, P_5^l)_P$ **{send** $\overleftarrow{B}$**, receive** P_5^l **from left}**
R2B03: **endif;**

The code blocks for winning FG and losing RG are symmetric.

R3 (*S* **Signal Creation Rule**):

A quiet (but not peace) general sends out a *Stop-Firing* signal S at every tick. If two hostile generals at the border are both firing, then the general with clock 0 will be treated as if it is in *Quiet* mode. So it also sends a *Stop-Firing* signal. The logic for a rear general is symmetric.

If P^l (or P^r) is conquering, then this rule will be replaced by **Rule 2** due to rule priorities.

Procedure Creation-Of-S-Signal-At-Front-General
R3A01: **if** $([P_1^l, P_1] = [\bar{a}, a])$ **and** (P^l **is** *Firing*) **then**
R3A02: **if** (P **is** *Quiet*) **or** (P**'s clock is** 0) **then**
R3A03: $(\bar{a}, N, Q, \overleftarrow{N}, \overrightarrow{S})_P$
R3A04: **endif;**
R3A05: **endif;**

R4 (*B* **Signal Creation Rule**):

A firing front general FG sends a bullet $\overrightarrow{B}$ to the rear general, if there are no other higher rank signals than B. Otherwise, FG resets its state and becomes quiet. Symmetric logic applies for a firing rear general.

Procedure Creation-Of-Bullet-At-Front-General
R4A01: **if** $([P_1^l, P_1, P_1^r] = [\bar{a}, a, a])$ **and** (P **is** *Firing*) **then**
R4A02: **if** $(P_4 \in [\overleftarrow{B}, \overleftarrow{N}])$ **and** $(P_4^r \in [\overleftarrow{B}, \overleftarrow{N}])$ **then**
R4A03: $(\bar{a}, N, F, P_4^r, \overrightarrow{B})_P$ **{receive left from right, send** $\overrightarrow{B}$**}**
R4A04: **else**
R4A05: $(\bar{a}, N, Q, \overleftarrow{N}, \overrightarrow{N})_P$ **{reset}**
R4A06: **endif;**
R4A07: **endif;**

The following four rules form the *Center Locating* protocol.

When a front peace general receives an R signal, it starts the *Center Locating* protocol by marking itself as the center and sending *Verify* signals $\overleftarrow{V}, \overrightarrow{V}$ towards two generals in the army (**R7A02.**) A peace general will echo the RV signal when it receives the V signal (**Rule R7.**) Since the ***Center Locating*** protocol starts at the front general, a center should receive an $\overrightarrow{RV}$ before an $\overleftarrow{RV}$ for most the time. Because RV signal can not pass through a marked center (**R8B03**), when a center receives an $\overrightarrow{RV}$, it will keep the signal (**R6A05**) until the right neighbor receives an $\overleftarrow{RV}$ signal. In this case, the center need to be adjusted to right. So simultaneously, the center clears its C status and signals, and the right neighbor marks itself as the new center and starts the next phase of ***Center Locating*** by sending $\overleftarrow{V}, \overrightarrow{V}$ signals (**Rule R5.**)

If the size of the army is odd, then eventually the marked center will get $\overrightarrow{RV}, \overleftarrow{RV}$ signals back at the same time. The real center then clears the status and sends out the OK signals $\overleftarrow{K}$ and $\overrightarrow{K}$ to its generals (**R6A07.**) In the case of even size army, eventually, the marked center will get an $\overleftarrow{RV}$ before getting an $\overrightarrow{RV}$. At that time, both soldiers know they are center. Therefore at the next tick, both soldiers clear the status, and one sends an $\overleftarrow{K}$ while the other sends an $\overrightarrow{K}$ (**R5A06, R5A08.**)

Upon the arrival of an K signal, a peace general will enter the firing mode (**Rule R7.**)

R5 (**Center Adjusting Rule**):

Since the locating protocol starts at a front general, a marked center need to be adjust to the right if it receives signal $\overrightarrow{RV}$ before $\overleftarrow{RV}$. If the army size is even, the marked center P and its left neighbor are two real centers if P receives signal $\overleftarrow{RV}$ before $\overrightarrow{RV}$. So both centers broadcast K signal.

Procedure Center-Adjusting-Rule

R5A01: **if** $(a, C, Q, \overleftarrow{N}, \overrightarrow{RV})_P$ **and** $(a, N, Q, \overleftarrow{RV}, \overrightarrow{N})_{P^r}$ **then**
R5A02: $(\bar{a}, N, Q, \overleftarrow{N}, \overrightarrow{N})_P$ {**clear** *Center* **status**}
R5A03: **elseif** $(a, C, Q, \overleftarrow{N}, \overrightarrow{RV})_{P^l}$ **and** $(a, N, Q, \overleftarrow{RV}, \overrightarrow{N})_P$ **then**
R5A04: $(\bar{a}, C, Q, \overleftarrow{V}, \overrightarrow{V})_P$ {**become the new** *Center*}
R5A05: **elseif** $(a, N, Q, \overleftarrow{N}, *)_P$ **and** $(a, C, Q, \overleftarrow{RV}, \overrightarrow{N})_{P^r}$ **then**
R5A06: $(\bar{a}, N, Q, \overleftarrow{K}, \overrightarrow{N})_P$ {**one of centers, send** $\overleftarrow{K}$}
R5A07: **elseif** $(a, N, Q, \overleftarrow{N}, *)_{P^l}$ **and** $(a, C, Q, \overleftarrow{RV}, \overrightarrow{N})_P$ **then**
R5A08: $(\bar{a}, N, Q, \overleftarrow{N}, \overrightarrow{K})_P$ {**one of centers, send** $\overrightarrow{K}$}
R5A09: **endif;**

R6 (**Signals Processing at Center**):

A center should only receive N, RV signals, otherwise the system is in an illegal configuration, so the center reinitializes its components. A center will try to keep the $\overrightarrow{RV}$ signal until **Rule 5** is enabled. A center sends V or K signals only once, so it releases them after they have been sent.

Procedure Signals-Processing-At-Center

R6A01: **if** $(a, C, Q, *, *)_P$ **then**
R6A02: **if** $(P_5^l \notin [\overrightarrow{N}, \overrightarrow{RV}])$ **or** $(P_4^r \notin [\overleftarrow{N}, \overleftarrow{RV}])$ **then**
R6A03: $(\bar{a}, N, Q, \overleftarrow{N}, \overrightarrow{N})_P$ {**reset**}
R6A04: **elseif** $(P_5^l \in [\overrightarrow{N}, \overrightarrow{R}])$ **and** $(P_5 = \overrightarrow{RV})$ **then**
R6A05: $(\bar{a}, C, Q, \overleftarrow{N}, \overrightarrow{RV})_P$ {**keep** $\overrightarrow{RV}$ **signal**}
R6A06: **elseif** $(a, C, Q, \overleftarrow{RV}, \overrightarrow{RV})_P$ **then**
R6A07: $(\bar{a}, N, Q, \overleftarrow{K}, \overrightarrow{K})_P$ {**broadcast** OK **signals**}
R6A08: **elseif** $(a, C, Q, \overleftarrow{V}, \overrightarrow{V})_P$ **or** $(a, C, Q, \overleftarrow{K}, \overrightarrow{K})_P$ **then**
R6A09: $(\bar{a}, C, Q, \overleftarrow{N}, \overrightarrow{N})_P$ {**release** V **or** K **signals**}
R6A10: **endif;**
R6A11: **endif;**

R7 (**Signal Processing Rule for Peace Generals**):

A peace rear general will always try to send a request signal $\overleftarrow{R}$ to the front general. So, a peace front general starts the *Center Locating* protocol by broadcasting V signals when it receives the $\overleftarrow{R}$ signal. A peace front general also try to send an $\overrightarrow{R}$ signal to tell the rear general that they should not be firing.

A peace general P sends an RV signal towards center if a verify signal V is received, or enters *firing* mode if an K signal is received. Otherwise it receives the signal according to signal ranks.

Procedure Signals-Processing-At-Peace-Front-General
R7A01: **if** $P_4 = \overleftarrow{R}$ **then**
R7A02: $(\bar{a}, C, Q, \overleftarrow{V}, \overrightarrow{V})_P$ **{start *Center Locating* protocol}**
R7A03: **elseif** $P_4 = \overleftarrow{V}$ **then**
R7A04: $(\bar{a}, *, Q, \overleftarrow{N}, \overrightarrow{RV})_P$ **{echo V signal}**
R7A05: **elseif** $P_4 = \overleftarrow{K}$ **then**
R7A06: $(\bar{a}, N, F, \overleftarrow{N}, \overrightarrow{N})_P$ **{enter *Firing* mode}**
R7A07: **elseif** P_5 **kills** P_4^r **then**
R7A08: $(\bar{a}, N, Q, \overleftarrow{N}, \overrightarrow{R})_P$ **{send $\overrightarrow{R}$}**
R7A09: **else**
R7A10: $(\bar{a}, *, Q, P_4^r, \overrightarrow{R})_P$ **{also receive signal from right}**
R7A11: **endif;**

Procedure Signals-Processing-At-Peace-Rear-General
R7B01: **if** $P_5 = \overrightarrow{V}$ **then**
R7B02: $(\bar{a}, *, Q, \overleftarrow{RV}, \overrightarrow{N})_P$ **{echo V signal}**
R7B03: **elseif** $P_5 = \overrightarrow{K}$ **then**
R7B04: $(\bar{a}, N, F, \overleftarrow{N}, \overrightarrow{N})_P$ **{enter *Firing* mode}**
R7B05: **elseif** P_4 **kills** P_5^l **then**
R7B06: $(\bar{a}, N, Q, \overleftarrow{R}, \overrightarrow{N})_P$ **{request *Center Locating*}**
R7B07: **else**
R7B08: $(\bar{a}, *, Q, \overleftarrow{R}, P_5^l)_P$ **{also receive signal from left}**
R7B09: **endif;**

R8 (Signals Passing Rule for Soldier):

First of all, a soldier should not be firing, otherwise the soldier resets its state.

Normally, a soldier receives and passes signals according to their ranks. Signal $\overrightarrow{RV}$ can not pass through a center to the right, because the center will keep it until either **R5** is enabled or an error correction can be made by **R6A03**. But, a center P does not need to keep the $\overleftarrow{RV}$ signal because P and its left neighbor will know both of them are centers.

Procedure Soldier-Is-Quiet
R8A01: **if (P is Firing) then**
R8A02: $(\bar{a}, N, Q, \overleftarrow{N}, \overrightarrow{N})_P$ **{reset, soldier can not be firing}**
R8A03: **endif;**

Procedure Receiving-Left-Signal-At-Soldier
R8B01: **if** (P_4 **kills** P_5^l) **or** (P_4^r **kills** P_5^l) **then**
R8B02: $(\bar{a}, N, Q, \overleftarrow{N}, \overrightarrow{N})_P$ {**reset**}
R8B03: **elseif not** $(a, C, Q, *, \overrightarrow{RV})_{P^l}$ {$\overrightarrow{RV}$ **doesn't pass through a center**}
R8B04: **receive** P_5^l **from left**
R8B05: **endif;**

Procedure Receiving-Right-Signal-At-Soldier
R8C01: **if** (P_5 **kills** P_4^r) **or** (P_5^l **kills** P_4^r) **then**
R8C02: $(\bar{a}, N, Q, \overleftarrow{N}, \overrightarrow{N})_P$ {**reset**}
R8C03: **else**
R8C04: **receive** P_4^r **from right**
R8C05: **endif;**

3.3 Proof of Correctness

It is easy to see that the number of armies is nonincreasing (**Rules R1, R2**), so eventually the ring will reach a configuration after which the number of armies is constant. WLOG, we assume the number of armies is a constant that we call $2m$. We write these armies as $\mathcal{A}_1, \mathcal{A}_2, \cdots, \mathcal{A}_{2m}$. Members of an army change during the war, we use $\mathcal{A}_{i,t}$ to denote the army $\mathcal{A}_i$ at time t and $FG_{i,t}, RG_{i,t}$ to denote $\mathcal{A}_{i,t}$'s *Front General* and *Rear General* respectively. We will also use FR_i, RG_i whenever their meanings are clear. We also denote the number of members in army $\mathcal{A}_i$ at time t by $Size(\mathcal{A}_{i,t})$.

Definition. An army $\mathcal{A}_i$ is firing if the following conditions hold:

1. Both generals of $\mathcal{A}_i$ are firing, but RG_{i-1} and FG_{i+1} are both quiet.
2. $\mathcal{A}_i$ does not have centers ($P_2 = N$, $\forall\, P \in \mathcal{A}_i$.)
3. All signals inside $\mathcal{A}_i$ are either B or N.

Definition. We say that army $\mathcal{A}_{i,t}$ starts the **Center Locating** protocol at time t, and denote it by the predicate $SCL(\mathcal{A}_{i,t})$, if

1. Both generals of $\mathcal{A}_i$ are peace generals.
2. FG_i has state $(a, C, Q, \overleftarrow{V}, \overrightarrow{V})$ and it is the only center in the army.
3. All other signals inside $\mathcal{A}_i$ are either R or N.

Lemma 3. If there is a firing army, then it will continue to be firing until the number of armies decreases.

Proof. By definition and **Rules R2, R4.** ∎

Since the number of armies is a constant $2m$, there will be no firing army.

Lemma 4. For any i, there are infinitely many t such that both generals of $\mathcal{A}_i$ are quite at time t.

Proof. If one of generals is quiet, then this quiet general will always send a signal with rank higher than $rank(B)$. So eventually, both generals will be quiet by **R4.**

In order to keep both generals being firing, only B or N can be allowed in the army by **Rule R4.** Since a firing general only sends B signal and B signal clears any center (**R6**), the army is a firing army. Contradiction. ∎

Lemma 5. If one of $\mathcal{A}_i$'s general is firing for infinitely many times, then there are infinitely many t such that $SCL(\mathcal{A}_{i,t})$.

Proof. WLOG assume that front general FG is firing for infinitely many times. Since $\mathcal{A}_i$ can not be firing, both generals will be quiet by **Lemma 4.** Thus, sometimes FG is firing and sometimes it is quiet. A general can be firing again after being quiet only by receiving an K signal. Only center can generate K signal, and it is no longer a center after sending an K signal (see **Rule 5, 6.**) A center can only be originated at the front general when it receives an $\overleftarrow{R}$ signal. Eventually, the $\overleftarrow{R}$ signal has to be from the rear general. Since this $\overleftarrow{R}$ survives, the army does not have other higher rank signals. Thus, $SCL(\mathcal{A}_{i,t})$ is true for some t. By the same argument, $SCL(\mathcal{A}_{i,t'})$ will be true again some time t' after t. ∎

We now prove the correctness of the **Center Locating** protocol.

Definition. If an army $\mathcal{A}_{i,t}$ has centers, then the error-degree of the army is defined as

$$ED(\mathcal{A}_{i,t}) = |dist(FG_{i,t}, C_1) - dist(RG_{i,t}, C_2)|$$

where C_1 is the closest center from $FG_{i,t}$, C_2 is the closest center from $RG_{i,t}$ (C_1 may equal C_2.) If $\mathcal{A}_{i,t}$ has no centers, then $ED(\mathcal{A}_{i,t})$ is undefined.

Lemma 6. Suppose that $\mathcal{A}_{i,t}$ starts the **Center Locating** protocol at time t. Then for some $t' > t$ both peace generals will receive the OK signal at the same time t', if both $RG_{i-1,t'}$ and $FG_{i+1,t'}$ are quiet. Furthermore, we have following:

$$\begin{aligned} t' &= t + 3k^2 + 3k + 1, \quad \text{if } Size(\mathcal{A}_i) = 2k \\ t' &= t + 3k^2 + 5k + 2, \quad \text{if } Size(\mathcal{A}_i) = 2k+1 \end{aligned}$$

Proof. FG_i sends out both $\overleftarrow{V}$ and $\overrightarrow{V}$ at time t. Note that $R1 - R4$ are not enabled.

Case 1 $Size(\mathcal{A}_{i,t})$ is odd.

We first show that there is a $t_1 > t$ such that$ED(\mathcal{A}_{i,t_1}) = ED(\mathcal{A}_{i,t}) - 2$. RG_i will receive the $\overrightarrow{V}$ signal and send the returning signal $\overleftarrow{RV}$ at time $t + (2k+1)$. When RG_i's right neighbor receives $\overleftarrow{RV}$ at time $t + 2(2k+1) - 1$, the marked center will be adjusted to right by **Rule 5.** Let $t_1 = t + 2(2k+1)$, then $ED(\mathcal{A}_{i,t_1}) = ED(\mathcal{A}_{i,t}) - 2$.

The marked center need to be adjusted $k+1$ time for error degree ED to be zero. Then, the center will send OK signals when both returning verify signals are back at the same time. Because signal $\overrightarrow{K}$ will kill $\overleftarrow{R}$ signal, we know that both generals of $\mathcal{A}_i$ will receive the K signal at time $t' = t + x$, where

$$\begin{aligned} x &= 2(2k+1) - 1 + 2(2k) - 1 + \cdots + 2(k+1) - 1 + k + 1 \\ &= 2\sum_{i=k+1}^{2k+1} i = 3k^2 + 5k + 2 \end{aligned}$$

Case 2 $Size(\mathcal{A}_{i,t})$ is even.

Similarly, the ED will decrease to 1 after k adjustments of the marked center. The marked center will receive $\overleftarrow{RV}$ before $\overrightarrow{RV}$ after it sends out the V signals for the $k+1$ time. By **Rule R5**, both soldiers then broadcast the OK signal at the next pulse. Therefore, both generals of $\mathcal{A}_i$ will receive the K signal at time $t' = t + y$, where

$$\begin{aligned} y &= 2(2k) - 1 + 2(2k-1) - 1 + \cdots + 2(k+1) - 1 + 2k - 1 + k \\ &= 2\sum_{i=k}^{2k} i - 1 = 3k^2 + 3k - 1 \end{aligned}$$

■

For an army, the ***Center Locating*** protocol starts when the front general receives an $\overleftarrow{R}$ signal, and it ends when some general receives the K signal. Let $CL(x)$ denote the *Center Locating* time in a army of size x. By **Lemma 6**, it is easy to verify that $CL(y) - CL(x) > x$ if $y > x$.

Lemma 7. For some army $\mathcal{A}_i$, one of its generals will be firing for infinitely many times.

Proof. Otherwise, we may assume that all generals in the ring are quiet forever. So eventually there will be no S, K signals after some time t'. Since K can not be generated, eventually all centers will be cleared. And all V, RV signals will be gone too. But, generals will keep sending R signals, and $SCL(\mathcal{A}_{i,t})$ will be true when the front general receives an $\overleftarrow{R}$ signal at time t. But $\mathcal{A}_i$ will be firing after it finishes the center locating by **Lemma 6**, a contradiction. ■

Lemma 8. For any army $\mathcal{A}_i$, there are infinitely many t such that $SCL(\mathcal{A}_{i,t})$.

Proof. If for any army, one of its generals will be firing for infinitely many times, then we are done by **Lemma 5**.

Otherwise, by **Lemma 7**, there is i and t_0 such that after time t_0, one of $\mathcal{A}_i$'s general will be firing infinitely many times but both generals of $\mathcal{A}_{i-1}$ are quiet forever.

Whenever $\mathcal{A}_i$ finishes a run of the locating protocol, RG_i will not receive the K signal. Otherwise $\mathcal{A}_i$ will be firing. So FG_{i+1} will be firing infinitely many times. By **Lemma 5**, army $\mathcal{A}_{i+1}$ will start the locating protocol. Continue the argument, we know that army $\mathcal{A}_{i-2}$ will start the locating protocol after t_0. Since FG_{i-1} is always quiet, $\mathcal{A}_{i-2}$ will be firing after it finishes the center locating by **Lemma 6**. ■

Lemma 9. If there are only two armies, then one of them will be a firing army.

Proof. Let two armies be $\mathcal{A}$ and $\mathcal{B}$. Suppose otherwise, then $\mathcal{A}$ and $\mathcal{B}$ will start the *Center Locating* protocol by **Lemma 8**. WLOG, assume $\mathcal{A}$ finishes its *Center Locating* not later than $\mathcal{B}$ does. If $\mathcal{A}$ finishes it early, then $\mathcal{A}$ will be firing. If both $\mathcal{A}$ and $\mathcal{B}$ finish the locating protocol at the same time t, then generals of two armies have different clocks. By **Rule R3**, the army with clock 1 will be firing. ■

Lemma 10. If no firing army can be generated, then all armies will have the same size.

Proof. By **Lemma 9**, the number of armies is a $2m$, and $m > 1$. By **Lemma 8**, we may assume that we are at some time t' and all armies have run *Center Locating* protocol at least once (this gives us clean ring configurations.) Since no firing army can be created, the size of each army will not change. Let $a_1, a_2, \cdots, a_{2m}$ be the their sizes. We need the inequality that $CL(b) - CL(a) > a$ if $b > a$.

Assume that at time t_1, RG_1 receives the K signal but FG_1 does not receive the K because RG_{2m} receives the K at t_{2m} before t_1, so FG_1 starts to send cancel signal S from t_{2m}. Let p_{2m}, p_1 be the time

FG_1, RG_1 become peace general respectively. We will show that $a_{2m} \leq a_1$.

Assume by contradiction that $a_{2m} \geq a_1 + 1$. Then $p_{2m} < p_1$, otherwise $\mathcal{A}_1$ could have started the *Center Locating* protocol before $\mathcal{A}_{2m}$ starts its ***Center Locating*** protocol, so $\mathcal{A}_1$ should finish the ***Center Locating*** protocol and become the firing army before $\mathcal{A}_{2m}$ does. Since $t_1 = p_1 + a_1 + CL(a_1)$ and $t_{2m} \geq p_{2m} + a_{2m} + CL(a_{2m})$, we have

$$p_{2m} + a_{2m} + CL(a_{2m}) \leq t_{2m} < t_1 = p_1 + a_1 + CL(a_1)$$

$$a_1 < CL(a_{2m}) - CL(a_1) < p_1 - p_{2m} + a_1 - a_{2m} < p_1 - p_{2m}$$

This means that RG_1 is quiet at p_{2m}, otherwise RG_1 should receive an $\vec{R}$ signal and become peace general at time $p_{2m} + a_1$ ($< p_1$.) Therefore, FG_2 is firing at time p_{2m} until time p_1. But $\mathcal{A}_2$ itself is not firing, its general FG_2 can be firing for at most a_2 ticks. Hence we have

$$a_1 < CL(a_{2m}) - L(a_1) < p_1 - p_{2m} \leq a_2$$

$\mathcal{A}_1$ will finish another run of ***Center Locating*** at time $x = t_{2m} + 2a_1 + CL(a_1)$. $\mathcal{A}_{2m}$ will finish another run of ***Center Locating*** at time $y \geq t_{2m} + a_{2m} + CL(a_{2m})$ and $\mathcal{A}_2$ will finish another run of ***Center Locating*** at time $z \geq t_{2m} + a_1 + CL(a_2)$. But $x < y$ and $x < z$, this means $\mathcal{A}_1$ will be firing.

Thus, we have $a_{2m} \leq a_1$. Similarly, we can show that $a_i \leq a_{i+1}$ for all i. ∎

Theorem 3. If ring size is odd or $2p$ for prime p, then the *Firing Generals* protocol guarantees that the ring converges to a unison configuration in at most $O(n^3)$ steps.

Proof. If *Firing Generals* fails, then by **Lemma 9** and **10**, the ring size $n = 2am$, where $m > 1, a > 1$ and $2m$ is the number of armies and a is the size of each army.

The *Center Locating* protocol takes at most $O(n^2)$ steps, then the number of armies will decrease. Thus the convergence span is $O(n^3)$. ∎

3.4 Other Protocols

By the proof of **Lemma 10**, if *Firing Generals* protocol fails then every army have the same size a and $\mathcal{A}_i$ always finishes the *Center Locating* protocol some $b_i \leq a$ ticks before $\mathcal{A}_{i+1}$ finishes the ***Center Locating*** protocol. If $\mathcal{A}_1$ finishes a run of ***Center Locating*** protocol at time t, then $\mathcal{A}_1$ has one firing general at time $t_1 = t - \sum_1^{2m} b_i$. This means $\mathcal{A}_1$ starts the ***Center Locating*** protocol after t_1. Hence

$$CL(a) < \sum_1^{2m} b_i \leq 2ma = n$$

Corollary 3. If the *Center Locating* protocol can be modified to take more than n ticks, then the modified *Firing Generals* protocol is self-stabilizing.

In the light of above **Corollary**, we can derive other self-stabilizing protocols by allowing processors to have more states (in a function of n.) For instance, we add another component to the state of a processor and this new *speed control* component has $\sqrt{n}$ values. When a front general needs to run the *Center Locating* protocol, it first uses its and neighbor's *speed control* components to count to n ticks then actually starts the ***Center Locating*** protocol. Other protocols of less than $O(\sqrt{n})$ states per processor can be similarly obtained by using the *speed control* component.

We can slow down our *Center Locating* protocol by a factor of 4 ($CL(a) > 2a^2$), and obtain a protocol that works also for ring of size $2p^2$, where p is a prime. Because for this kind of rings to have all armies of same size, the army size must be p and $CL(p) > 2p^2 = n$.

4 Lower Bounds on Convergence Span

Theorem 4. For any self-stabilizing k-clock protocol SSP, $CS(SSP) \geq dia(G)deg(G)/2$.

Corollary 4. Any self-stabilizing k-clock protocol for ring of size n has convergence span of $n/4$ if ring is bi-directional, and has convergence span of $n/2$ if ring is unidirectional.

The easy proofs will be given in the full paper.

Acknowledgements: We want to thank referees for their valuable suggestions regarding the presentation of the paper.

References

[1] A. Arora, S. Dolev and M. Gouda. Maintaining Digital Clocks In Step. *Parallel Processing Letters* 1 (1991), pp. 11-18.

[2] B. Awerbuch and R. Ostrovsky. Memory-efficient and self-stabilizing network reset. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing* (1994)

[3] M. G. Gouda and T. Herman. Stabilizing Unison. *Information Processing Letters* 35 (1990), pp. 171-175.

[4] T. Herman and Sukuma Ghosh, Stabilizing Phase-Clocks, *Technical Report, University of Iowa* (9/1993).

[5] A. Israeli and M. Jalfon. Token management schemes and random walks yield self stabilizing mutual exclusion. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing* (1990), pages 119–130.

[6] G. Itkis and L. Levin. Fast and lean self-stabilizing asynchronous protocol. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science* (1994).

[7] G. Itkis, C. Lin and J. Simon. Deterministic, Constant Space, Self-Stabilizing Leader Election on Uniform Rings. *submitted to WDAG 95*

[8] C. Lin and J. Simon. Observing Self-Stabilization. *Proceedings of the 11-th Annual ACM Symposium on Principles of Distributed Computing* (1992) pp. 113-123.

[9] A. Mayer, Y. Ofek, R. Ostrovsky and M. Yung. Self-Stabilizing Symmetry Breaking in Constant-Space. *STOC'92*, pp. 667-678.

[10] A. Pancholi. Master Thesis. Department of Computer Science, Texas A&M, 1994.

[11] J.B. Rosser, L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Math. Vol 6(1962)* pp. 64-94.

We can slow down our Counter Counting protocol by a factor of 4 ($C(4n) > 2n^2$), and obtain a protocol that works also for ring of size $2n^2$, where p is a prime. Because for this kind of rings to have all agents of same size, the army size must be p and $C(p) > 2p^2 = n$.

4 Lower Bounds on Convergence Span

Theorem 4. For any self-stabilizing 4-clock protocol SSA, $CS(SSA) \geq \deg(G)\deg(G)/2$.

Corollary 4. Any self-stabilizing 4-clock protocol for ring of size n has convergence span of $n/4$ if ring is bi-directional, and has convergence span of $n/2$ if ring is unidirectional.

The easy proofs will be given in the full paper.

Acknowledgements: We want to thank referees for their valuable suggestions regarding the presentation of the paper.

References

[1] A. Arora, S. Dolev and M. Gouda. Maintaining Digital Clocks in Step. Parallel Processing Letters 1 (1991), pp. 11-18.

[2] B. Awerbuch and R. Ostrovsky. Memory-efficient and self-stabilizing network reset. In Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (1994)

[3] M. G. Gouda and T. Herman. Stabilizing Unison. Information Processing Letters 35 (1990), pp. 171-175.

[4] T. Herman and Sukumar Ghosh. Stabilizing Phase-Clocks. Technical Report, University of Iowa (91/1993).

[5] A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing (1990), pages 119-130.

[6] G. Itkis and L. Levin. Fast and lean self-stabilizing asynchronous protocol. In Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science (1994).

[7] C. Johnen, C. Tin and J. Simon. Deterministic, Constant Space, Self-Stabilizing Leader Election on Uniform Rings. submitted to DISC'95

[8] C. Lin and J. Simon. Observing Self-Stabilization. Proceedings of the 13-th Annual ACM Symposium on Principles of Distributed Computing (1994) pp. 113-123.

[9] A. Mayer, Y. Ofek, R. Ostrovsky and M. Yung. Self-Stabilizing Symmetry Breaking in Constant-Space. STOC'92, pp. 667-678.

[10] M. Pandeshi. Master Thesis. Department of Computer Science, Technion, 1994.

[11] R.E. Tarjan. Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1983.

Paper Number 11

Self-Stabilizing Dynamic Programming Algorithms on Trees

Sukumar Ghosh, Arobinda Gupta, Mehmet Hakan Karaata, and Sriram V. Pemmaraju

Self-Stabilizing Dynamic Programming Algorithms on Trees

Sukumar Ghosh* Arobinda Gupta
Mehmet Hakan Karaata Sriram V. Pemmaraju

April 29, 1995

Abstract

Dynamic programming is a bottom-up approach that is typically used for designing algorithms for optimization problems. Many graph-theoretic optimization problems that are NP-hard in general, can be efficiently solved, using dynamic programming, when restricted to trees. Examples of such problems include maximum weighted independent set and minimum weighted edge covering. In this paper, we present a technique to translate certain dynamic programming algorithms into distributed, self-stabilizing algorithms that run on trees. The resulting self-stabilizing algorithms are deterministic and uniform. We prove the correctness of the algorithms produced by our translator assuming a distributed scheduler with read-write atomicity. We also show that on a tree with radius r, our algorithms stabilize in no more than $2r + 3$ rounds.

Keywords: Centers, Distributed algorithms, Dynamic programming, Self-stabilization, Trees.

1 Introduction

Dynamic programming is a bottom-up approach that is typically used for designing algorithms for optimization problems. It is similar to the divide-and-conquer strategy in that a problem is solved by solving and combining solutions of subproblems. It differs from the divide-and-conquer strategy in that after a subproblem is solved, its solution is saved to be reused later, thus avoiding excess work in case subproblems are shared among several problems. Two common examples of problems that have efficient dynamic programming solutions are the *matrix-chain multiplication* problem and the *longest common subsequence* problem [7]. Many interesting graph-theoretic optimization problems, when restricted to trees, can also be efficiently solved using the dynamic programming paradigm. A few examples of such problems are *maximum weighted matching*, *maximum weighted independent set*, *minimum weighted edge covering*, and *minimum weighted dominating set*. In this paper, we present a general method for translating the dynamic programming solutions to these types of problems into *deterministic*, *uniform*, *self-stabilizing* distributed algorithms that run on trees. We then show that our translation scheme produces algorithms that are correct and time-optimal.

*This author's research was supported in part by the National Science Foundation under grant CCR-9402050.

Self-stabilizing algorithms are attractive because of their inherent tolerance to transient faults in the system. Since no initialization of variables is required for the correctness of the algorithm, the algorithm converges to a desired state starting from any arbitrary state in which the system may be in after a transient fault. In addition, some self-stabilizing algorithms may be capable of adapting to a dynamically changing topology. In particular, our self-stabilizing algorithm can adapt dynamically to arbitrary addition or deletion of nodes and edges in the tree, as long as the tree topology is maintained.

The rest of this paper is organized as follows. Section 2 describes the dynamic programming paradigm, focusing on two examples. Section 3 presents our model of computation. A brief description of related work and discussion of how it compares with our work is presented in Section 4. Section 5 features a general method for translating certain types of dynamic programming algorithms into distributed, uniform, deterministic, self-stabilizing algorithms that run on trees. In Section 6 we prove of correctness of the algorithms resulting from our translation scheme.

2 Dynamic Programming

A dynamic programming algorithm for a problem typically consists of two or more phases. In the first phase, the smallest instances of the problem, called *base problems*, are solved and their solutions are stored. In each successive phase, larger instances of problems are solved by combining solutions of smaller instances. Note that the solution of each instance is stored for future reuse and therefore when a solution to a smaller instance is required, it is simply fetched from a store. The process outlined above proceeds from the bottom, upwards, solving larger and larger instances, until the original instance of the problem is solved. The following is a simple example to illustrate the dynamic programming paradigm.

Problem: *Compute C_r^n, the number of subsets of size r of a set of size n, where $n \geq r$.*

The base problems are: *Compute* C_ℓ^k, $1 \leq k \leq n$, $\ell \in \{0, k\}$. The base problems have solutions: $C_0^k = 1$, and $C_k^k = 1$, for any k, $1 \leq k \leq n$. C_ℓ^k, for any k, ℓ, where $1 \leq \ell < k \leq n$, can be computed using the following recurrence: $C_\ell^k = C_\ell^{k-1} + C_{\ell-1}^{k-1}$. The computations are done in a bottom-up fashion starting with the base subproblems and computing C_ℓ^k only after C_ℓ^{k-1} and $C_{\ell-1}^{k-1}$ have been computed. Care is taken to store the result of each computation for later reuse. It is easy to see that the dynamic programming algorithm sketched above can be implemented in $O(nr)$ time using $O(r)$ space. Note that the recurrence relation expressing C_ℓ^k in terms of C_ℓ^{k-1} and $C_{\ell-1}^{k-1}$ suggests a simple divide-and-conquer algorithm that computes C_ℓ^{k-1} and $C_{\ell-1}^{k-1}$ *independently* and adds the results. But this naive approach fails to reuse solutions of subproblems and results in a huge amount of overlapped, and thus, unnecessary computations. It is easily verified that the time complexity of this divide-and-conquer algorithm is $\Omega(2^r)$ and it is therefore hardly a competitor for the dynamic programming approach. For more details on the dynamic programming paradigm and examples of this approach, refer to [4, 7].

We now turn our attention to a class $\mathcal{C}$ of problems that take as input an undirected, and possibly

weighted[1], tree $T = (V, E)$ and can be solved using the dynamic programming approach. In particular, we require that each problem $P \in \mathcal{C}$ satisfy the following *decomposability* condition:

> Let T be rooted at *any* node $r \in V$. There exists problem $P(i)$ with solution $S(i)$ associated with each node $i \in V$ such that
>
> (a) $P(i)$ can be solved by solving and combining solutions of $P(j)$ for all children j of v. More precisely, the following relationship holds:
>
> $$S(i) = f_k(S(j_1), S(j_2), \ldots, S(j_k)),$$
>
> where $j_1, j_2, \ldots, j_k$ are children of i and f_k is some computable function with k arguments. Note that since T is rooted, the parent-child relationship is well-defined.
>
> (b) $S(r)$ contains a solution of P.

Thus, assuming that P satisfies the decomposability condition, P can be solved by the following generic algorithm:

Step 1. *Arbitrarily* root T at a node $r \in V$.

Step 2. For each node i whose children are nodes $j_1, j_2, \ldots, j_k$, compute $S(i)$ using the equation $S(i) = f(S(j_1), S(j_2), \ldots, S(j_k))$.

Of course, to obtain an efficient version of the above algorithm, one must adopt a bottom-up approach, starting at the leaves and computing $S(i)$ only after $S(j)$ has been computed for all children j of i. The algorithm terminates when $S(r)$ is computed. Note that by part (b) of the decomposability condition, $S(r)$ contains a solution of P. Also note that for a leaf v, the number of children, $k = 0$, and f_0 is a constant function. As an example of this approach we provide a dynamic programming algorithm for the *maximum weighted independent set* problem, restricted to trees. Note that this problem is NP-hard for arbitrary graphs [9].

Let G be a node-weighted graph in which each node i has a positive, real *weight* $w(i)$. An independent set in G is a set of nodes such that no two nodes in the set are adjacent in G. The weight of an independent set is the sum of the weights of all nodes in the set. A *maximum weighted independent set* in G is an independent set with maximum weight. The *maximum weighted independent set* problem, restricted to trees, is defined as follows.

Problem: *The input is a node-weighted tree $T = (V, E)$ in which each node i has a positive, real weight $w(i)$. Find a maximum weighted independent set in T.*

Though the problem asks that we compute a maximum weighted independent set in T, we instead focus on the "smaller" problem of computing the *weight* of a maximum weighted independent set in T. We will

[1] Weights may be associated with edges, nodes, or both.

subsequently show that it is a simple step to go from the weight to the independent set itself. Define the following three variables for each node $i \in V$ as follows.

$W^+(i)$ = weight of maximum weighted independent set that includes i in the subtree rooted at i

$W^-(i)$ = weight of maximum weighted independent set that does not include i in the subtree rooted at i

$W(i)$ = weight of maximum weighted independent set in the subtree rooted at i

Thus to each node $i \in V$, we can associate a problem $P(i)$: Compute $W^+(i)$, $W^-(i)$, and $W(i)$. The solution $S(i)$ of $P(i)$ is the triple $(W^+(i), W^-(i), W(i))$. Note that since $W(r)$ is the solution of our problem, computing $S(r)$ does indeed solve our problem. The following equations provide a way of computing $S(i)$ using values of $S(j)$ for all children j of i. We use C_i to denote the set of children of a node i.

$$W^+(i) = \sum_{j \in C_i} W^-(j) + w(i)$$
$$W^-(i) = \sum_{j \in C_i} W(i)$$
$$W(i) = \max\left(W^+(i), W^-(i)\right)$$

Note that for a leaf node i, $C_i = \emptyset$ and therefore $W(i) = W^+(i) = w(i)$ and $W^-(i) = 0$.

Figure 1(a) shows a node-weighted tree with the node weights shown next to each node. Figure 1(b) shows the tree with the values of $(W(i), W^+(i), W^-(i))$ for each node i. It can be easily verified that the value of $W(r)$ is indeed the weight of a maximum weighted independent set in T. A particular maximum weighted independent set IS can be easily computed starting at the root and traversing the tree in a top-down fashion. More precisely, suppose that $flag(i)$ is a boolean variable associated with each node i, that indicates whether node $i \in IS$. Then, $flag(i)$, for all $i \in V$ can be computed as follows:

(a) $flag(r) = (W^+(r) > W^-(r))$.

(b) For any node i with parent p_i, $flag(i) = (\neg flag(p_i)) \wedge (W^+(i) > W^-(i))$.

Thus $IS = \{i \in V \mid flag(i) = true\}$.

3 Model of Computation

Suppose that a problem $P \in \mathcal{C}$ takes as input a tree T. Our goal is to translate the dynamic programming algorithm that solves P into a distributed, deterministic, uniform, self-stabilizing algorithm that runs on T. In other words, we use T as the underlying communication network. Therefore, each node i in T

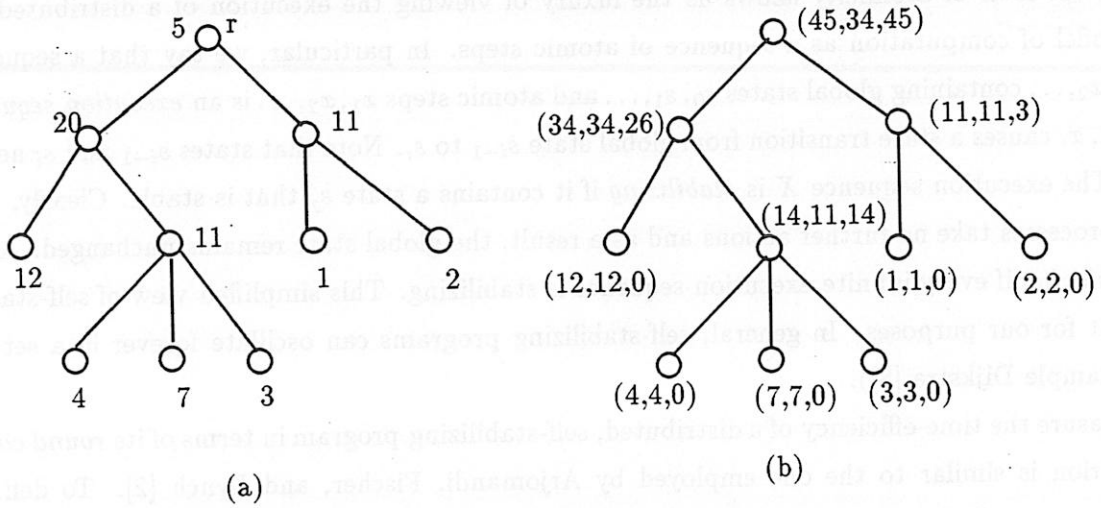

Figure 1: Solving maximum weighted independent set on a tree

corresponds to a process, called process i, that executes a program asynchronously and each edge $\{i, j\}$ in T corresponds to a bidirectional connection between process i and process j. Though we give processes names for notational convenience, processes are all identical and are therefore anonymous. Each process i contains a finite set of local shared variables. These local shared variables in process i can be read by process i and any of its neighbors, but can be written onto by process i alone. The *state* of a process is given by the values of its variables. If we allow S_i to denote the set of states of process i, then any element in the cartesian product $\prod_{i \in V} S_i$ is called a *global state*. Guarded statements are used to specify the program executed by each process. A guarded statement is of the form $G \rightarrow A$, where G is a boolean condition called a *guard* and A is the corresponding *action* that is executed only if G is true. A global state s is called *stable* if in state s all guards in all processes are false.

We assume the presence of a *distributed scheduler*. A distributed scheduler can be thought of as a collection of private or local schedulers — one for each process. The local scheduler for process i arbitrarily selects one guard from among all guards in process i. Process i then evaluates the selected guard and executes the corresponding action, if the guard evaluates to true. The local scheduler then chooses an arbitrary guard again and the cycle repeats. The distributed scheduler is assumed to be fair only in the weakest possible sense: in any infinite sequence of guard selections by a local scheduler, each guard that can be selected is selected infinitely often. The local schedulers in different processes are independent, thus allowing for overlapping execution of distinct processes. Therefore, we need to be precise about the level of atomicity being used. We assume *read-write atomicity*. That is, only the reading of a single variable or the writing onto a single variable are assumed to be atomic. This implies that the evaluation of a guard or the execution of an action by a process is a sequence of atomic steps that may be arbitrarily interleaved with atomic steps by other processes.

Fixing the level of atomicity allows us the luxury of viewing the execution of a distributed program in this model of computation as a sequence of atomic steps. In particular, we say that a sequence $X = s_0, x_1, s_1, x_2, \ldots$ containing global states $s_0, s_1, \ldots$ and atomic steps $x_1, x_2, \ldots$ is an *execution sequence* if for each $i \geq 1$, x_i causes a state transition from global state s_{i-1} to s_i. Note that states s_{i-1} and s_i need not be distinct. The execution sequence X is *stabilizing* if it contains a state s_j that is stable. Clearly, once s_j is reached, processes take no further actions and as a result, the global state remains unchanged. A program is *self-stabilizing* if every infinite execution sequence is stabilizing. This simplified view of self-stabilization is sufficient for our purposes. In general, self-stabilizing programs can oscillate forever in a set of states (see for example Dijkstra [8]).

We measure the time-efficiency of a distributed, self-stabilizing program in terms of its *round complexity*. Our definition is similar to the one employed by Arjomandi, Fischer, and Lynch [2]. To define round complexity, we need the notion of a *move*. A process is said to make a *move* when it evaluates a guard (selected by its local scheduler), finds the guard to be true, and then takes the corresponding action. Note that if a process evaluates a guard and finds it to be false, then it does not take the corresponding action, and is not considered to have made a move. Under the assumption of read-write atomicity, a move may consist of several atomic steps: the first few as part of guard evaluation and the next few as part of the action execution. A process i is said to be *alive* in a global state s_0, if there exists an execution sequence, $X = s_0, x_1, s_1, x_2, \ldots$ that contains a move by process i. The implication is that if a process i is not alive in a global state s_0, then from that point on, no matter which guards are selected by i's local scheduler, i takes no further action, and its state remains unchanged. Of course, moves by other process also cannot change the state of i. Also note that no process is alive in a stable state. Let $X = s_0, x_1, s_1, \ldots$ be an infinite execution sequence of a self-stabilizing program. The *round complexity*, $R(X)$, of X can be defined as follows. If X does not contain the execution of at least one move by each process that is alive in s_0, then $R(X) = 1$. Otherwise, express X as $X = X_p X_s$, where X_p is the minimal prefix of X such that every process that is alive in s_0 makes a move in X_p. Since X_p is minimal, its last element is an atomic step and therefore the first element of X_s is a global state. X_p is one round of the execution sequence. Define $R(X) = R(X_s) + 1$. The round complexity of the self-stabilizing program is the maximum $R(X)$ over all infinite execution sequences X. Round complexity is a realistic measure of time complexity in our model because it pays attention to the "slowest" process, by ensuring that a round is completed only when *all* processes, including the slowest process, have made at least one move.

4 Related work

In a related work, Collin, Dechter, and Katz (CDK) [5, 6] present distributed self-stabilizing solutions for a class of *constraint satisfaction* problems. The input to a constraint satisfaction problem consists of a finite set of variables along with a finite set of constraints among subsets of variables. The solution to the problem is an assignment of values to variables such that all constraints are satisfied. CDK restrict their attention

to constraint satisfaction problems in which all constraints are *binary*, that is, all constraints relate *pairs* of variables. In this case, a *constraint graph* can be associated with the constraint satisfaction problem. Each node in the constraint graph represents a variable, and an edge between two nodes represents a binary constraint between the corresponding variables. Assuming that the underlying communication network is the constraint graph of the problem, CDK provide a distributed, deterministic, self-stabilizing algorithm to solve any constraint satisfaction problem with binary constraints. This algorithm assumes a special node and is therefore not uniform. For trees, CDK provide a uniform solution. However, this solution requires the presence of a central scheduler and is not guaranteed to stabilize otherwise.

A dynamic programming algorithm can also be viewed as the systematic imposition of constraints among subsets of variables. But, typically these constraints are not binary. For example, in the *maximum weighted independent set* problem discussed earlier, the variable $W^+(i)$ is related to all variables in the set $\{W^-(j) \mid j \in C_i\}$. Thus the approach of CDK, that only applies to binary constraint-satisfaction problems, cannot be applied to the types of problems we are considering. In theory, the constraints in a dynamic programming problem on trees can be converted into binary constraints as follows. Choose a root for the tree and group all variables in a level into one variable. Notice that now all constraints are binary and we have a constraint graph that is just a path. However, since the dynamic programming paradigm is inherently bottom-up and CDK's algorithms adopt a top-down approach, this yields an extremely unnatural and inefficient solution to dynamic programming problems. Furthermore, as compared to CDK's algorithm, it is easy for our algorithm to adjust to changes in structure of T.

5 Self-stabilization and Dynamic Programming

In this section, we present a general method for translating the dynamic programming algorithm for any problem $P \in \mathcal{C}$ into a distributed, deterministic, uniform, self-stabilizing algorithm. If the input to P is a tree $T = (V, E)$, then our self-stabilizing algorithm assumes that T is its underlying communication graph.

The self-stabilizing algorithm for P can be viewed as having two phases, each phase roughly corresponding to a step in the generic algorithm (presented in Section 2) that solves P. In phase 1, T is converted into a tree that is either singly or doubly rooted and in which every node knows the identity of its children and its parent. We use a modification of the center-finding algorithm of Karaata, Pemmaraju, Bruell, and Ghosh (KPBG) [10] for this purpose [2]. There are several reasons for doing so. First, the center-finding algorithm roots T at one or *two adjacent roots*. As we will show later, since the two roots are adjacent, it is unnecessary to break symmetry and select one of them, before sending T on to Phase 2. Thus we do not have to resort to either randomization or unique process ids to break the symmetry. Second, we will show in Section 6, that the time-complexity of our algorithm depends on the height of the roots. By choosing centers as roots we obtain roots with minimum height and thereby provide a time optimal algorithm.

[2] CDK [5, 6] also use a similar phase in their work.

Program for each process i:
do
$(i \text{ is a leaf}) \ \wedge \ (h(i) \neq 0) \rightarrow \ h(i) := 0$
$\square$ $(i \text{ is not a leaf}) \ \wedge \ (h(i) \neq 1 + 2ndmax(N_h(i))) \rightarrow \ h(i) := 1 + 2ndmax(N_h(i))$
od

Figure 2: The center-finding algorithm

In phase 2, the problem P is solved on the rooted tree constructed in Phase 1. In this phase, each node attempts to assign values to its variables that are consistent with values of its childrens' variables. Note that the above view of our algorithm as having two distinct phases is only for purposes of explanation. In the self-stabilizing algorithm for P, the two phases are not distinct and occur in an arbitrarily interleaved fashion.

In the next subsection, we describe the center-finding algorithm of KPBG [10]. We then show how the constraints in a dynamic programming solution can be embedded on top of the center-finding algorithm to yield a distributed self-stabilizing algorithm that solves P.

5.1 Constructing the Rooted Tree

Let $d(i,j)$ denote the *distance*, the length of a shortest path in T, between nodes i and j. Then $e(i) = max\{d(i,j) \mid j \in V\}$ denotes the *eccentricity* of a node i, the distance between i and a farthest node from i in T. A *center* of T is a node with minimum eccentricity. The eccentricity of a center is called the *radius* of T. It is well-known that a tree has one or two adjacent centers [3].

The center-finding algorithm of KPBG [10] is shown in Figure 2. This algorithm runs on T and each node $i \in V$ corresponds to a process that has a local variable $h(i)$, referred to as the *h-value* of node i. The h-value of a node can initially take on any non-negative integer value. Let $N(i)$ denote the set of neighbors of i. In the algorithm in Figure 2, $N_h(i)$ denotes the multi-set of h-values of the neighbors of i and $2ndmax(N_h(i))$ denotes the second largest element in the multi-set $N_h(i)$ More precisely, $2ndmax(N_h(i))$ is equal to the second element in the sequence obtained by ordering the elements in $N_h(i)$ in descending order. Clearly, the h-values of the nodes in T constitute a stable state of the center-finding algorithm if for each leaf i, $h(i) = 0$, and for each non-leaf i, $h(i) = 1 + 2ndmax(N_h(i))$. To explain the meaning of h-values of nodes, we define the *height* of a node as follows. Let $T_0 = T$ and for any $\ell > 0$, let T_ℓ be the tree obtained from $T_{\ell-1}$ by removing all leaves (and the incident edges) from T_ℓ. For any $\ell \geq 0$, a node i is called a *height-ℓ* node if it is a leaf in T_ℓ. If i is a height-ℓ node, then we write $height(i) = \ell$. KPBG show that in a stable state of the center-finding algorithm, $h(i) = height(i)$ for all $i \in V$. Thus, the h-values of nodes eventually (after the state has become stable) represent the heights of the nodes. Heights of nodes in a tree have several useful properties [10].

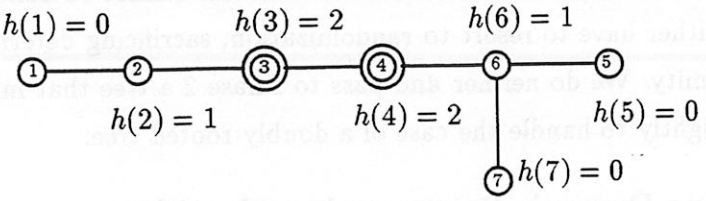

Figure 3: A tree whose h-values constitute a final state of the center-finding algorithm.

Proposition 1 *Suppose that i is a height-ℓ node in T.*

1. *If i is not a center of T, then i has exactly one height-$(\ell+1)$ neighbor. The rest of the neighbors all have height less than or equal to $\ell-1$. If $\ell > 0$, then i has at least one height-$(\ell-1)$ neighbor.*
2. *If i is a unique center of T, then all neighbors of i have height less than or equal to $\ell-1$. If $\ell > 0$, then i has at least two height-$(\ell-1)$ neighbors.*
3. *If i is one of two adjacent centers of T, then i has exactly one height-ℓ neighbor (the other center). All other neighbors have height less than or equal to $\ell-1$. If $\ell > 0$, then i has at least one height-$(\ell-1)$ neighbor.*

Proposition 1 has the following immediate corollary.

Corollary 2 *Let r be the radius of T and let H_ℓ be the height-ℓ nodes in T. The nodes in T are partitioned by $\{H_0, H_1, \ldots, H_r\}$.*

Figure 3 shows a tree whose h-values constitute a final state of the center-finding algorithm. Notice that the two adjacent centers, 3 and 4, have equal h-values. KPBG [10] proves the correctness of the center-finding algorithm and establishes an upper bound on the total number of moves made by the algorithm assuming a central scheduler. In Section 6, we establish an upper bound on the round complexity of the center-finding algorithm assuming a weaker model of computation: a distributed scheduler with read-write atomicity.

It is easy to use the h-values to view T as a rooted tree. For any node i, define

$$parent(i) = \{j \mid j \in N(i) \text{ and } h(j) > h(i)\}.$$

Proposition 1 implies that when the h-values constitute a stable state of the center-finding algorithm, then $parent(i) = \emptyset$ if i is a center; otherwise $parent(i)$ is a singleton. Furthermore, if $j \in parent(i)$, then $i \notin parent(j)$. Thus we have a tree rooted at either one or two adjacent nodes and every node has a consistent view of the identity of its parent and its children. If the length of the longest path in T is odd then T has a single root and we have successfully completed Step 1 of the generic algorithm presented in Section 2. However, if the longest path in T has even length, then we have two roots and to complete Step

1, we need to choose one of the two roots. However, since this cannot be done in the model of computation we have chosen, we either have to resort to randomization, sacrificing determinism, or to distinct process ids, sacrificing anonymity. We do neither and pass to Phase 2 a tree that might have two adjacent roots. We adjust Phase 2 slightly to handle the case of a doubly-rooted tree.

5.2 Self-stabilizing Dynamic Programming Algorithm

In our first attempt, we modify the center-finding algorithm by introducing an additional variable $S(i)$ in each process i. Note that we assume that $S(i)$ is a single variable only for the sake of simplicity. In general, $S(i)$ represents a finite collection of variables derived from the dynamic programming solution of P. We modify the center-finding algorithm into a self-stabilizing algorithm for solving P, by simply attaching a guarded statement that ensures that each process i sets $S(i)$ to a value that is consistent with the values of $S(j)$ for all children j. For a leaf process i define a predicate $correct(i)$ as: $correct(i) \equiv (i \text{ is a leaf}) \wedge (h(i) = 0)$. For a non-leaf process i, define $correct(i)$ as:

$$correct(i) \equiv \Big(i \text{ is not a leaf}\Big) \wedge \Big(h(i) = 1 + 2ndmax(N_h(i))\Big).$$

Let $C_i = \{j \mid j \in N(i) \text{ and } h(j) < h(i)\}$. Process i believes that if $j \in C_i$ then j is indeed its child in the rooted tree constructed in Phase 1. For notational convenience, we drop the subscript k in f_k and denote the sequence $S(j_1), S(j_2), \ldots, S(j_k)$ by $S(j_1, j_2, \ldots, j_k)$. After verifying that its h-value is consistent with the h-values of its neighbors (that is, $correct(i) = true$), process i attempts to make $S(i)$ consistent with $S(j)$ for all $j \in C_i$ by executing the action $S(i) := f(S(C_i))$. This is shown in guarded statement $S3$ in the program in Figure 4. While this approach is correct if T has a single root, it does not work if T has two adjacent roots, say r_1 and r_2. In this case, neither $S(r_1)$ nor $S(r_2)$ contain the solution to the instance of P that we are interested in solving. To get around this problem, we assume that each node i has an additional variable $S'(i)$. $S(i)$ is computed as described above while $S'(i)$ is computed only when process i detects that it is one of two roots. In particular, when process i detects that it is one of two roots, then it computes a value for $S'(i)$ that is consistent with $S(j)$ for all $j \in C_i$ *assuming that the adjacent root* is also its child. The intuition here is that each root attempts to act as the sole root of T. To be more precise, for each process i and for each neighbor $j \in N(i)$ define the predicate:

$$two\_roots(i,j) \equiv \Big(correct(i)\Big) \wedge \Big(\forall k \in N(i) : h(i) \geq h(k)\Big) \wedge \Big(h(i) = h(j)\Big).$$

Each process i, that finds the predicate $\exists j \in N(i) : two\_roots(i,j)$ true, executes the action $S'(i) = f(S(C(i) \cup \{j\}))$, if necessary. This is shown in guarded statement $S4$ in the program in Figure 4. As a result, if T has one root r then $S(r)$ contains a solution of P; otherwise if T has two roots r_1 and r_2, then both $S'(r_1)$ and $S'(r_2)$ contain a solution of P. The complete self-stabilizing algorithm, called the *dynamic programming algorithm*, that runs on T and solves problem P, is shown in Figure 4.

Program for each process i:

do

$S1$ $(i$ is a leaf$) \wedge (h(i) \neq 0)$ $\rightarrow$ $h(i) := 0$

$S2$ $\square$ $(i$ is not a leaf$) \wedge (h(i) \neq 1 + 2ndmax(N_h(i)))$ $\rightarrow$ $h(i) := 1 + 2ndmax(N_h(i))$

$S3$ $\square$ $correct(i) \wedge S(i) \neq f(S(C_i))$ $\rightarrow$ $S(i) := f(S(C_i))$

$S4$ $\square$ $\exists j \in N(i) : two\_roots(i,j) \wedge S'(i) \neq f(S(C_i \cup \{j\}))$ $\rightarrow$ $S'(i) := f(S(C_i \cup \{j\}))$

od

Figure 4: The dynamic programming algorithm

6 Proof of Correctness

In this section, we show that the round complexity of our dynamic programming algorithm is at most $2r+3$ rounds, where r is the radius of T. A global state s is said to be *h-stable*, if in state s, for each node $i \in V$, the predicate $correct(i)$ holds. In other words, in an h-stable state the h-values of all processes are correctly set and the guards corresponding to statements $S1$ and $S2$ are false in all processes. We first show, in Theorem 4, that the dynamic programming algorithm, starting from an arbitrary initial state, requires at most $r+1$ rounds to reach an h-stable state. Subsequently, in Theorem 6, we show that the dynamic programming algorithm, starting from an h-stable state, requires at most $r+2$ rounds to reach a stable state. Theorems 4 and 6, together, give the claimed upper bound of $2r+3$ on the round complexity of the dynamic programming algorithm.

For our proofs we assume that each move by a process consists of (a) a sequence of reads through which the process gathers enough information to evaluate a guard, followed by (b) a sequence of reads followed by a single write through which the process executes the corresponding action. Thus the last atomic step of each move is assumed to be a write-step. As in the RAM model [1], we assume that each process has sufficiently many local registers to store temporary results of calculations. As mentioned earlier, for simplicity we also assume that $S(i)$ is a single variable. Our proofs can be easily extended to the case in which $S(i)$ is a finite collection of variables.

To prove Theorem 4 we need the following lemma.

Lemma 3 *Let i be a height-ℓ node. For any $k \geq 0$, after at most k rounds of the dynamic programming algorithm,*

$$h(i) = \ell \quad \text{if } \ell < k$$
$$h(i) \geq k \quad \text{if } \ell \geq k.$$

Proof: The proof is by induction on k.

Base Case: Let $k = 0$. Any height-ℓ node $i \in V$ satisfies $\ell \geq k = 0$. Since we assumed that the initial h-values of all nodes in T are non-negative, it follows that after 0 rounds, $h(i) \geq k = 0$, for all $i \in V$.

Therefore, the lemma is true for $k = 0$.

Induction Hypothesis: Suppose that the lemma is true for some $k \geq 0$.

Induction Case: Suppose that round $k+1$ is the execution subsequence $s_0, x_1, s_1, \ldots, s_{p-1}, x_p$. Also suppose that atomic step x_p causes the global state to change from s_{p-1} to s_p. We prove the following claim about the h-values of nodes in states $s_0, s_1, \ldots, s_p$.

Claim: The h-value of any height-ℓ node i satisfies the following property in any state s_j, $0 \leq j \leq p$:

1. If $\ell < k$, then $h(i) = \ell$.
2. If $\ell = k$ and process i has completed at least one move in round $k+1$ prior to state s_j, then $h(i) = \ell$. Otherwise, if $\ell = k$, then $h(i) \geq k$.
3. If $\ell > k$ and process i has completed at least one move in round $k+1$ prior to state s_j, then $h(i) \geq k+1$. Otherwise, if $\ell > k$, then $h(i) \geq k$.

Proof of Claim: The proof is by induction on j.

Base Case: $j = 0$. Immediately follows from the induction hypothesis that claims the truth of the lemma after k rounds.

Induction hypothesis: Suppose that the claim is true for some $j \geq 0$.

Induction case: We want to show that atomic step x_{j+1}, that causes a state transition from state s_j to state s_{j+1}, preserves the truth of the above claim in state s_{j+1}. If x_{j+1} is a read-step then, $s_{j+1} = s_j$ and x_{j+1} is not the last step in a move by process i. Together, these two facts imply the continued truth of the claim in state s_{j+1}. If x_{j+1} is a write-step, then it can write onto variables $h(i)$, $S(i)$, or $S'(i)$. In the following, we treat writing onto $h(i)$ as one separate case and writing onto $S(i)$ or $S'(i)$ as one separate case.

Case 1. x_{j+1} writes onto $h(i)$. It is easily seen that if $\ell = 0$ (that is, node i is a leaf) then x_{j+1} preserves the truth of the claim in state s_{j+1}. So we suppose that $\ell > 0$. Prior to writing onto $h(i)$ in atomic step x_{j+1}, process i reads the h-values of neighbors in atomic read-steps. Since these atomic read-steps occur prior to state s_j, the h-values of neighbors that are read, satisfy the claim. We now consider three cases depending on whether i is a non-center, the unique center, or one of two centers in T.

1. i is not a center of T. Let j_1 be the height-$(\ell+1)$ neighbor and j_2 be a height-$(\ell-1)$ neighbor of i.
2. i is the unique center of T. Let j_1 and j_2 be two height-$(\ell-1)$ neighbors of i.
3. i is one of two centers of T. Let j_1 be the height-ℓ neighbor (which is the other center) and let j_2 be a height-$(\ell-1)$ neighbor of i.

The existence of j_1 and j_2 are guaranteed by Proposition 1 and by the fact that $\ell > 0$. It is easy to see by applying the induction hypothesis that node i calculates a value for $2ndmax(N_h(i))$ that is equal to the h-value of j_2 that it reads. In particular, we consider the following three cases:

1. $\ell < k$. By the induction hypothesis i reads $h(j_2)$ as $\ell - 1$. Therefore i calculates the value of $2ndmax(N_h(i))$ as $\ell - 1$. As a result x_{j+1} writes the value ℓ onto $h(i)$. Thus $h(i)$ satisfies item (1) of the claim in s_{j+1}.
2. $\ell = k$. By the induction hypothesis, process i reads a value for $h(j_2)$ that satisfies $h(j_2) = \ell - 1 = k - 1$. Therefore, i calculates $2ndmax(N_h(i))$ as $\ell - 1$ and x_{j+1} writes ℓ into $h(i)$. This, along with the fact that x_{j+1} is the last atomic step in a move by process i in round $k+1$, ensure that $h(i)$ satisfies item (2) of the claim in s_{j+1}.
3. $\ell > k$. By the induction hypothesis, process i reads a value of $h(j_2)$ that satisfies $h(j_2) \geq k$. This implies that the value calculated for $2ndmax(N_h(i))$ satisfies $2ndmax(N_h(i)) \geq k$. Therefore x_{j+1} writes a value greater than or equal to $k+1$ onto $h(i)$. This, along with the fact that x_{j+1} is the last atomic step in a move by process i in round $k+1$, ensure that $h(i)$ satisfies item (3) of the claim in s_{j+1}.

Case 2. x_{j+1} writes onto $S(i)$ or $S'(i)$. To write onto $S(i)$ or $S'(i)$, process i must be execute guarded statement $S3$ or $S4$. In order to execute the action in $S3$ or $S4$, process i must first verify that $correct(i)$ is true. Process i does this by reading the h-values of neighbors in a sequence of atomic steps, calculating $2ndmax(N_h(i))$, and then verifying that $h(i)$ is one more than the calculated value of $2ndmax(N_h(i))$. By using the same argument as in Case 1, we see that there exists a state $s_{j'}$, $0 \leq j' < j+1$, such that $h(i)$ satisfies the claim in state $s_{j'}$ and $h(i)$ remains unchanged from state $s_{j'}$ to s_{j+1}. **End of proof of claim.**

Since the execution of a round consists of a move by every process, the proof of the above claim ensures that in state s_p, we have $h(i) = \ell$ if $\ell < k+1$ and $h(i) \geq k+1$ if $\ell \geq k+1$. This proves the induction case and hence the lemma. □

Using the above lemma, we now prove the following upper bound on the number of rounds required by the dynamic programming algorithm to reach an h-stable state starting from an arbitrary initial state.

Theorem 4 *Starting from an arbitrary initial state, the dynamic programming algorithm requires at most $r+1$ rounds to reach an h-stable state.*

Proof: (by contradiction) Suppose that starting from some initial state, the dynamic programming algorithm does not reach an h-stable state in $r+1$ rounds. By Lemma 3, after $r+1$ rounds, for any height-ℓ node i, $h(i) = \ell$ if $\ell < r+1$ and $h(i) \geq r+1$ if $\ell \geq r+1$. However, by Corollary 2, every node in T has height less than or equal to r. Therefore, for any height-ℓ node i in T, $h(i) = \ell$. This implies that $correct(i)$ is true for all nodes $i \in V$ and therefore the dynamic programming algorithm reaches an h-stable state in no more than $r+1$ rounds. □

We now prove an upper bound on the number of rounds required by the dynamic programming algorithm to reach a stable state starting in an arbitrary h-stable state. The following lemma is easily proved by induction on k, the number of rounds.

Lemma 5 *Suppose that the initial state of the dynamic programming algorithm is an h-stable state. After k rounds, $S(i) = f(S(C_i))$ for any height-ℓ node i with $\ell < k$.*

Theorem 6 *Starting from an h-stable state, the dynamic programming algorithm requires at most $r + 2$ rounds to reach a stable state.*

Proof: (by contradiction) Suppose that starting from an h-stable initial state, the dynamic programming algorithm does not reach an stable state in $r+2$ rounds. By Lemma 5, after $r+1$ rounds, for any height-ℓ node with $\ell < r+1$, $S(i) = f(S(C_i))$. However, by Corollary 2, every node in T has height less than or equal to r. Therefore, after $r+1$ rounds, for any node i in T, $S(i) = f(S(C_i))$. If T has two adjacent roots r_1 and r_2, then in one additional round the values of $S'(r_1)$ and $S'(r_2)$ are correctly assigned. □

The upper bound of $2r + 3$ follows from Theorems 4 and 6.

7 Conclusion

The distributed self-stabilizing algorithms that we provide for certain types of dynamic programming problems, only calculate the optimal value of the objective function that is being optimized. However, we believe that it is possible to add another top-down self-stabilizing layer to our algorithm, so as to calculate not only the optimum value of the objective function, but also the solution that achieves this optimum. For example, consider the maximum weighted independent set problem (see Section 2). On top of the self-stabilizing distributed algorithm that calculates $(W^+(i), W^-(i), W(i))$ for all $i \in V$, we can easily impose a layer that calculates the boolean value of $flag(i)$ for all $i \in V$. Recall that computing $flag(i)$ for all $i \in V$ is equivalent to computing a particular maximum weighted independent set in a distributed fashion.

References

[1] A. Aho, E. J. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, MA, 1974.

[2] E. Arjomandi, M. Fisher, and N. Lynch, *Efficiency of synchronous versus asynchronous distributed systems*, Journal of the ACM, 30 (1983), pp. 449–456.

[3] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, Elsevier Science Publishing Co., Inc., 1976.

[4] G. Brassard and P. Bratley, *Algorithmics: Theory and Practice*, Prentice Hall, 1987.

[5] Z. Collin, R. Dechter, and S. Katz, *Constraint satisfaction.* Personal communication.

[6] ——, *Constraint satisfaction*, in Proceedings of the 11th International Joint Conference on Artificial Intelligence, 1991.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *An introduction to Algorithms*, The MIT Press, 1991.

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

[9] M. Garey and D. S. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*, Freeman, San Fransisco, 1979.

[10] M. H. Karaata, S. V. Pemmaraju, S. C. Bruell, and S. Ghosh, *Self-stabilizing algorithms for finding centers and medians of trees*, Tech. Report TR-94-03, Department of Computer Science, The University of Iowa, 1994. Abstract appeared in ACM Annual Symposium on Principles of Distributed Computing, 1994.

Paper Number 12

Self-Stabilization by Tree Correction

George Varghese, Anish Arora, and Mohamed Gouda

Self-Stabilization by Tree Correction

George Varghese*and Anish Arora†and Mohamed Gouda‡

May 11, 1995

Abstract

We describe a simple tree correction theorem that states that any locally checkable protocol that works on a tree can be efficiently stabilized in time proportional to the height of the tree. We show how new protocols can be designed, and how existing work can be easily understood using this theorem.

1 Introduction

In Dijkstra's [Dij74] model, a network protocol is modeled using a graph of finite state machines. In a single move, a *single* node is allowed to read the state of its neighbors, compute, and then possibly change its state. In a real distributed system such atomic communication is impossible. Typically communication has to proceed through channels. Such channels must be modeled explicitly as state machines that can store messages sent from one node to another. Also, in message passing models, the channel state machine is essentially fixed (with actions to send and deliver packets) but the node state machines can be arbitrarily specified by the protocol designer. However, in Dijkstra's model *all* state machines are node state machines and can be arbitrarily specified by the protocol designer.

While Dijkstra's original model is not very realistic, it is probably the simplest model of an asynchronous distributed system. This simple model provided an ideal vehicle for *introducing* [Dij74] the concept of stabilization without undue complexity. We will use this simple model to describe a simple technique for self-stabilization that we call *tree correction*. We use this to show that existing work in [Dij74] and [AG90] can be understood very succinctly using the framework of local checking and tree correction. We will also state (but not prove) that this theorem can be extended to message passing systems.

The main result of the paper is a theorem (Theorem 3.1) that states that any locally checkable protocol on a rooted tree can be efficiently stabilized. In Section 3 we obtain Theorem 3.1. In Section 4, we show briefly how a reset protocol [AG90] due to Arora and Gouda can also be simply understood in this framework. The tree correction theorem can be generalized to message passing systems [Var93] but is most simply stated and proved in the simple shared memory model. Finally we state our conclusions and

*Washington University in St. Louis

†Ohio State University

‡University of Texas, Austin

compare our work with related work on local checking and correction [APV91b] and distributed constraint satisfaction [AGV94, CDK91].

2 Modeling Shared Memory Protocols

We will use a version of the timed I/O Automaton model [MMT91]. How can we map Dijkstra's model into this model? Suppose each node in Dijkstra's model is a separate automaton. Then in the I/O automaton model, it is not possible to model the simultaneous reading of the state of neighboring nodes. The solution we use is to dispense with modularity and model *the entire network as a single automaton.* All actions, such as reading the state of neighbors and computing, are *internal actions.* The asynchrony in the system, which Dijkstra modeled using a "demon", is naturally a part of our model. Also, we will describe the correctness of Dijkstra's systems in terms of *executions* of the automaton.

The major reason for using the timed I/O automaton model is that it allows us to model time, and hence to provide precise definitions of stabilization time. However, in terms of asynchronous executions, our model behaves exactly like Dijkstra's model and hence our results apply to Dijkstra's original model. The I/O automaton model also provides standard notation (e.g., fairness using classes) that we find helpful.

Thus we model a network as a single automaton in which a node can read and write the state of its neighbors in a single move using an internal action. Formally:

A *shared memory network automaton* $\mathcal{N}$ for graph $G = (E, V)$ is an automaton in which:

- The state of $\mathcal{N}$ is the cross-product of a set of node states, $S_u(\mathcal{N})$, one for each node $u \in V$. For any state s of $\mathcal{N}$, we use $s|u$ to denote s projected onto S_u. This is also read as the state of node u in global state s.
- All actions of $\mathcal{N}$ are internal actions and are partitioned into sets, $A_u(\mathcal{N})$, one for each node $u \in V$
- Suppose $(s, \pi, \tilde{s})$ is a transition of $\mathcal{N}$ and π belongs to $A_u(\mathcal{N})$. Consider any state s' of $\mathcal{N}$ such that $s'|u = s|u$ and $s'|v = s|v$ for all neighbors v of u. Then there is some transition $(s', \pi, \tilde{s}')$ of $\mathcal{N}$ such that $\tilde{s}'|v = \tilde{s}|v$ for u and all u's neighbors in G.
- Suppose $(s, \pi, \tilde{s})$ is a transition of $\mathcal{N}$ and π belongs to $A_u(\mathcal{N})$. Then $s|v = \tilde{s}|v$ for all $v \neq u$.

Informally, the third condition requires that the transitions of a node $u \in V$ only depend on the state of node u and the states of of the neighbors of u in G. The fourth condition requires that the effect of a transition assigned to node $u \in V$ can only be to change the state of u.

A shared memory *tree automaton* is a shared memory network automaton where G is a rooted tree. Thus for any node i in a tree automaton, we assume there is a value *parent*(i) that points to the parent of node i in the tree. There is also a unique root node r that has *parent*$(r) = nil$. For our purposes, it is convenient to model the *parent* values as being part of the code at each node. More generally, the parent pointers could be variables that are set by a stabilizing spanning tree protocol as shown in [AG90]. We will often use the phrase "tree automaton" to mean a "shared memory tree automaton" and the phrase "network automaton" to mean a "shared memory network automaton".

3 Tree Correction for Shared Memory Systems

We start with some standard definitions. A *closed predicate* [1] of an automaton A is a predicate L such that for any transition $(s, \pi, \tilde{s})$ of A, if $s \in L$ then $\tilde{s} \in L$.

A *link subsystem* of a tree automaton is an ordered pair (u, v), such that u and v are neighbors in the tree. To distinguish states of the entire automaton from the states of its subsystems we will sometimes use the word *global state* to denote a state of the entire automaton. For any global state s of a network automaton, we define $(s|u, s|v)$ to be the state of the (u, v) link subsystem. Thus the state of the (v, u) link subsystem in global state s is $(s|v, s|u)$.

A *local predicate* $L_{u,v}$ of a tree automaton is a subset of the states of a (u, v) link subsystem. A link predicate set $\mathcal{L}$ for a tree automaton is a set that contains exactly one predicate for every link subsystem in the tree and which satisfies the following symmetry condition: for each pair of neighbors u and v, if $(a, b) \in L_{u,v}$, then $(b, a) \in L_{v,u}$. (i.e., while a link predicate set has two link predicates for each pair of neighbors, these two predicates are identical except for the order in which the states are written down.) We will also assume that every link predicate set is *non-trivial* in that there is at least one global state s such that $(s|u, s|v) \in L_{u,v}$ for all link subsystems (u, v) in the tree.

A tree automaton is *locally checkable* for predicate L if there is some link predicate set $\mathcal{L} = \{L_{u,v}\}$ such that: $L \supseteq \{s : (s|u, s|v) \in L_{u,v}$ for all link subsystems (u, v) in the tree.$\}$ In other words, the global state of the automaton satisfies L if every link subsystem (u, v) satisfies $L_{u,v}$.

We say that automaton A stabilizes to the executions of automaton B in time t if for every execution α of A there is some suffix of execution α (whose first state starts no less than t time units after the first state of α) that is an execution of B.

For any automaton A we define $U(A)$ (which can be read as the unrestricted version of A) to be the automaton that is identical to A except that any state of A can be a start state of $U(A)$. Conversely, we use $A|L$ to denote the automaton that is identical to A except that the start states of $A|L$ are the states in set L. For any rooted tree T, we let *height*(T) denote the maximum length of a path between the root and a leaf in T. We can now state a simple theorem.

Theorem 3.1 Tree Correction in Shared Memory Systems: *Consider any tree automaton $\mathcal{T}$ for tree T that is locally checkable for predicate L. Then there exists an unrestricted tree automaton $\mathcal{T}^+$ for T such that $\mathcal{T}^+$ stabilizes to the executions of $\mathcal{T}|L$ in time proportional to height(T).*

Thus after a time proportional to the height of the tree, any execution of the new automaton $\mathcal{T}^+$ will "look like" an execution of $\mathcal{T}$ that starts with a state in which L holds. To prove this theorem we first describe how to construct $\mathcal{T}^+$ from $\mathcal{T}$ and then show that $\mathcal{T}^+$ satisfies the requirements of the theorem.

Assume that $\mathcal{T}$ is *locally checkable* for predicate L using link predicate set $\mathcal{L} = \{L_{u,v}\}$. We start by defining the set of global states that satisfy all local predicates. Let $L' = \{s : (s|u, s|v) \in L_{u,v}$ for all link subsystems (u, v) in the tree.$\}$. Clearly $L \supseteq L'$. Also because of the non-triviality of the link predicate set, L' is not the empty set. To construct $\mathcal{T}^+$ from $\mathcal{T}$ we do the following:

[1] This is often called a stable predicate. We avoid this phrase because of potential confusion with stabilization.

The state of $\mathcal{T}^+$ is identical to $\mathcal{T}$ except that the state set of each node is normalized to $\{s|u : s \in L'\}$

MODIFIED ACTION $a_u, a_u \in A_u$ (*modification of action a_u in $\mathcal{T}$*)
  Preconditions:
    Exactly as in a_u except for the additional condition:
    For all neighbors v of u: $(s|u, s|v) \in L_{u,v}$
  Effects:
    Exactly as in a_u

CORRECT$_u$ (*extra correction action for all nodes except the root*)
  Preconditions: $(parent(u) = v)$ and $((s|u, s|v) \notin L_{u,v})$
  Effects:
    Let a be any state in $S_u(\mathcal{T}^+)$ such that $(a, s|v) \in L_{u,v}$
    Change the state of node u to a

Each CORRECT$_u$ action is in a separate class with upper bound t_n

Figure 1: Augmenting $\mathcal{T}$ to create $\mathcal{T}^+$

- We first *normalize* all node states in $\mathcal{T}$. Intuitively, we remove all states in the state set of a node u that are not part of a global state that satisfies L'. Thus $S_u(\mathcal{T}^+) := \{s|u : s \in L'\}$. The state set of $\mathcal{T}^+$ is just the cross-product of the normalized state sets of all nodes. Intuitively, this rules out useless node states that never occur in global states that satisfy all local predicates.
- We retain all the actions of $\mathcal{T}$ but we add an extra precondition (i.e., an extra guard) to each action $a_u \in A_u$ of $\mathcal{T}$ as shown in Figure 1. Intuitively, this extra guard ensures that a normal action of $\mathcal{T}$ is not taken at node u unless all links adjacent to u are in "good states". All actions of $\mathcal{T}$ remain in the same classes in $\mathcal{T}^+$. (Recall that in the timed IOA model, timing guarantees for internal actions are expressed by grouping these actions into classes. Each class c has an associated time bound t_c; intuitively, if some action is class c is enabled for t_c time, then some action in class c must occur in t_c time.)
- We add an extra correction action CORRECT$_u$ for every node u in the tree that is not the root. CORRECT$_u$ is also described in Figure 1. Intuitively, this extra action "corrects" the link between node u and its parent if this link is not in a "good" state. Each CORRECT$_u$ action is put in a separate class with upper bound equal to t_n.

We outline a proof of the theorem by a series of lemmas. The first thing a careful reader needs to be convinced about is that the code in Figure 1 is realizable. The careful reader will have noticed that we made two assumptions. First, in the CORRECT$_u$ action, we assumed that for any link subsystem (u, v) of $\mathcal{T}^+$ and any state b of node v there is some a such that $(a, b) \in L_{u,v}$. Second, we assumed that when a

modified action a_u is taken at node u, the resulting state of node u has not been removed as part of the normalization step.

We will begin with a lemma showing that the first assumption is a safe one. We show that the second assumption is safe later.

Lemma 3.2 *For any link subsystem (u, v) of $\mathcal{T}^+$ and for any state a of node u there is some b such that $(a, b) \in L_{u,v}$.*

Proof: We know that for any state a of v there is some state $s \in L'$ such that $s|u = a$. This follows because all node states have been normalized and because L' is not empty. Then we choose $b = s|v$. □

The next lemma shows a *local extensibility* property. It says that if any node u and its neighbors have node states such that the links between u and its neighbors are in good states, then this set of node states can be extended to form a good global state.

Lemma 3.3 *Consider a node u and some global state s of $\mathcal{T}^+$ such that for all neighbors v of u, $(s|u, s|v) \in L_{u,v}$. Then there is some global state $s' \in L'$ such that $s'|u = s|u$ and $s'|v = s|v$ for all neighbors v of u.*

Proof: We create a global state s' by assigning node states to each node in the tree such that for every link subsystem (u, v), the state of the subsystem is in $L_{u,v}$. Start by assigning node state $s|u$ to u and $s|v$ to all neighbors v of u. At every stage of the iteration we will label a node x that has not been assigned a state and is a neighbor of a node y that has been assigned a state. But, by Lemma 3.2, we can do this such that the state of the subsystem containing x and y is in $L_{x,y}$. Eventually we label all nodes in the tree and the resulting global state is in L'. Once again, this is because for every link subsystem (u, v), the state of the (u, v) subsystem is in $L_{u,v}$. The labeling procedure depends crucially on the fact that the topology is a tree. □

To prove the theorem, we will use an Execution Convergence Theorem stated and proved in [Var93]. To state the theorem we need the following definition.

We say that an automaton A is stabilized to predicate L using predicate set $\mathcal{L}$ and time constant t if:

1. $\mathcal{L} = \{L_i, i \in I\}$ of sets of states of A, where $(I, <)$ is a finite, partially ordered index set. We let *height*$(\mathcal{L})$ denote the maximum length chain in the partial order.
2. $\cap_{i \in I} L_i \subseteq L$.
3. For all $i \in I$ and for all steps $(s, \pi, \tilde{s})$ of A, if s belongs to $\cap_{j \leq i} L_j$, then $\tilde{s}$ belongs to L_i.
4. For every $i \in I$ and every execution α of A and every state s in α the following is true. Suppose that either $s \in \cap_{j<i} L_j$ or there is no $L_j < L_i$. Then there is some state $\tilde{s} \in L_i$ that occurs within time t of s in α.

The first condition says there is a partial order on the predicates in $\mathcal{L}$. The second says that L becomes "true", when all the predicates in $\mathcal{L}$ become true. The third is a stability condition. It says that any transition of A leaves a predicate L_i true, if all the predicates less than or equal to L_i are true in the previous state. Finally the last item is a liveness condition. It says that if all the predicates *strictly* less than L_i are true in a state, then within time t after this state, L_i will become true.

We define $height(L_i)$, the height of a predicate $L_i \in \mathcal{L}$, to be the maximum length of a chain that ends with L_i in the partial order. The value of $height(\mathcal{L})$ is, of course, the maximum height of any predicate $L_i \in \mathcal{L}$. By the liveness condition, within time t all predicates with height 1 become true; these predicates stay true for the rest of the execution because of the third stability condition. In general, we can prove by induction that within time $i \cdot t$ all predicates with height i become and stay true. This leads to a simple but useful theorem:

Theorem 3.4 Execution Convergence: *Suppose that automaton A is stabilized to predicate L using predicate set $\mathcal{L}$ and time constant t. Then, A stabilizes to the executions of $A|L$ in time $height(\mathcal{L}) \cdot t$.*

However, to apply that theorem we have to work with predicates of $\mathcal{T}^+$ (i.e., sets of states of $\mathcal{T}^+$) and not link predicates of $\mathcal{T}^+$ (i.e., sets of states of link subsystems of $\mathcal{T}^+$). This is just a technicality that we deal with as follows. For each link subsystem (u, v), we define the predicate $L'_{u,v} = \{s : (s|u, s|v) \in L_{u,v}\}$. Clearly, $L' = \cap L'_{u,v}$.

Next consider some u, v, w such that $v = parent(u)$ and $w = parent(v)$. We assume that $v \neq nil$. (But v may be the root in which case w is *nil*.) The next lemma states an important stability property. It states that if $L'_{u,v}$ holds in some global state s of $\mathcal{T}^+$ it will remain true in any successor state of s if either:

- v is the root OR
- $L'_{v,w}$ is also true in s.

Lemma 3.5 *Consider some u, v, w such that $v = parent(u) \neq nil$ and $w = parent(v)$. Suppose there is some global state s of $\mathcal{T}^+$ such that $s \in L'_{u,v}$ and $(w \neq nil) \rightarrow s \in L'_{v,w}$. Then for any transition $(s, \pi, \tilde{s})$, $\tilde{s} \in L'_{u,v}$.*

Proof: It suffices to consider all possible actions π that can be taken at either u or v in state s. It is easy to see that we don't have to consider correction actions because, by assumption, neither the CORRECT$_u$ or the CORRECT$_v$ action is enabled in state s.

Consider a modified action a_u of $\mathcal{T}^+$ that is taken at node u. Suppose action a_u occurs in state s and results in a state $\tilde{s}$. By the preconditions of action a_u, for all children x of u, $(s|x, s|u) \in L_{x,u}$. But in that case by Lemma 3.3 there is some other global state $s' \in L'$ such that: $s'|u = s|u$, $s'|v = s'|v$ and $s'|x = s|x$ for all children x of u. Thus by the third property of a network automaton, the action a_u is also enabled in s' and, if taken in s', will result in some state say $\tilde{s}'$. But since L' is closed, $\tilde{s}' \in L'$ and hence $(\tilde{s}'|u, \tilde{s}'|v) \in L_{u,v}$. But by the third property of a network automaton, $\tilde{s}|u = \tilde{s}'|u$ and $\tilde{s}|v = \tilde{s}'|v$. Thus $\tilde{s} \in L'_{u,v}$. The case of a modified action at v is similar. □

The previous lemma also shows that our second assumption is safe. If a modified action is taken at a node u, resulting in state $\tilde{s}$ then $\tilde{s} \in L'_{u,v}$ for some v. Thus by Lemma 3.3 there is some other state $\tilde{s}' \in L'$ such that $\tilde{s}'|u = \tilde{s}|u$. Thus $\tilde{s}|u$ cannot have been removed as part of the normalization step.

The next lemma states an obvious liveness property. If $L'_{u,v}$ does not hold in some global state of $\mathcal{T}^+$, then after at most t_n time units we will eventually reach some global state $\tilde{s}$ in which $L'_{u,v}$ holds. Clearly this is guaranteed by the correction actions (either CORRECT_u or CORRECT_v depending on whether u is the child of v or vice versa) and by the timing guarantees.

Lemma 3.6 *For any (u, v) link subsystem and any any execution α of $\mathcal{T}^+$ and any state s_i of α, if $s_i \notin L'_{u,v}$ then there is some later state $s_j \in L'_{u,v}$ that occurs within t_n time units of s_i.*

Proof: Suppose not. Then either CORRECT_u (if u is the child of v) or CORRECT_v (if v is the child of u) is continuously enabled for t_n time units after s_i. But then by the timing guarantees, either CORRECT_u or CORRECT_v must occur within t_n time after s_i, resulting in a state in which $L'_{u,v}$ (and, of course, $L'_{v,u}$) holds. □

We now return to the proof of the theorem. First we define a natural partial order on the predicates $L'_{u,v}$. For any link subsystem (u, v), define the *child node* of the subsystem to be u if $parent(u) = v$ and v otherwise. Define the ordering $<$ such that $L'_{u,v} < L'_{w,x}$ iff the child node of the (u, v) subsystem is an ancestor (in the tree T) of the child node of the (w, x) subsystem.

Using this partial order and Lemmas 3.5 and 3.6, we can show by induction that $\mathcal{T}^+$ stabilizes to the executions of $\mathcal{T}^+|L'$ in time $height(T) \cdot t_n$. But any execution α of $\mathcal{T}^+|L'$ is also an execution of $\mathcal{T}|L$. This follows from three observations. First, since L' is closed for $\mathcal{T}$, L' is closed for $\mathcal{T}^+$. Second, if L' holds in all states of an execution α of $\mathcal{T}^+|L'$, then no correction actions can occur in α. Third, any execution of $\mathcal{T}|L'$ is also an execution of $\mathcal{T}|L$ because $L \supseteq L'$. Thus we conclude that $\mathcal{T}^+$ stabilizes to the executions of $\mathcal{T}|L$ in time $height(T) \cdot t_n$.

This theorem can be used as the basis of a design technique. We start by designing a tree automaton T that is locally checkable for some L. Next we use the construction in the theorem to convert T into $\mathcal{T}^+$. $\mathcal{T}^+$ stabilizes to the executions of $T|L$ even when started from an arbitrary state.

Weakening the Fairness Requirement In the previous construction, we assigned each CORRECT_u action to a separate class. Actually the theorem only requires a property we call *eventual correction*: if a CORRECT_u action is continuously enabled, then a CORRECT_u action occurs within bounded time. This property can be established quite easily for the protocols in [Dij74] and [AG90], allowing all the actions in the entire automaton to be placed in a single class!

4 A Reset Protocol on a Tree

Before describing the reset protocol due to Arora and Gouda [AG90], we first describe the network reset problem. Recall that we have a collection of nodes that communicate by reading the state of their

neighbors. The interconnection topology is described by an arbitrary graph. Assume that we are given some application protocol that is being executed by the nodes. We wish to superimpose a reset protocol over this application such that when the reset protocol is executed the application protocol is "reset" to some "legal" global state. A "legal" global state is allowed to be any global state that is reachable by the application protocol after correct initialization. The problem is called distributed reset because reset requests may arrive at any node.

A simple and elegant network reset protocol is due to Finn [Fin79]. In this protocol each node i running the application protocol has a session number. When the reset protocol is not running, the session numbers at every node are the same. When a node receives a reset request, it resets the local state of the application (to some prespecified initial state) and increments its session number by 1. When a node sees that a neighbor has a higher session number, it changes its session number to the higher number and resets the application. Finally, the application protocol is modified so that a node cannot make a move until its session number is the same as that of its neighbors. This check prevents older instances of the application protocol from "communicating" with newer instances of the protocol. This protocol is shown to be correct [Fin79] if all the session numbers are initially zero and the session numbers are allowed to grow without bound.[2]

We rule out the use of unbounded session numbers as unrealistic. Also, in a stabilizing setting, having a "large enough" size for a session number does not work. This is because the reset protocol can be initialized with all session numbers at their maximum value. Thus, we are motivated to search for a reset protocol that uses bounded session numbers. Suppose we could design a reset protocol with unbounded numbers in which *the difference between the session numbers at any two nodes is at most one in any state.* Suppose also that for any pair of neighboring nodes u and v that compare session numbers, the session number of one of the nodes (say u) is always no less than the session number of the other node. Then, since the session numbers are only used for comparisons, it suffices to replace the session numbers by a single bit that we call $sbit_i$. This is the first idea in Arora and Gouda's reset protocol [AG90].

To realize this idea, we cannot allow a node to increment its session number as soon as it gets a reset request. Otherwise, multiple reset requests at the same node will cause the difference in session numbers to grow without bound. Thus nodes must coordinate before they increment session numbers.

In Arora and Gouda's reset protocol [AG90], the coordination is done over a rooted tree. Arora and Gouda first show how to build a rooted tree in a stabilizing fashion. In what follows we will assume that the tree has already been built. Thus every node i has a pointer called $parent(i)$ that points to its parent in the tree and the parent of the root is a special value *nil*.

Given a tree, an immediate idea is to funnel all reset requests to the root. On receipt of a request, the root could send reset grants down the tree. Nodes could increment their session number on receiving a grant. Unfortunately, this does not work either because a node A in the tree may send a reset request and receive a grant before some other node B in the tree receives a grant. After getting its first grant, A may

[2]Finn also considered the problem of using bounded sequence numbers. His solution was shown to be incorrect by Humblett and Soloway who proposed a fix in [HS91]. However, neither paper addresses the problem of designing a self-stabilizing reset protocol.

send another request and receive a second grant before B gets its first grant. Assuming that the session numbers are unbounded, the difference in the session numbers of A and B can grow without bound.

Instead, the reset task is broken into three phases. In the first phase, a node sends a reset request up the tree towards the root. In the second phase, the root sends a reset wave down the tree. In the third phase, the root waits until the reset wave has reached every node in the tree before starting a new reset phase. This ensures that after the system stabilizes, the use of three phases will guarantee that a single bit $sbit_i$ is sufficient to distinguish instances of the application protocol.

The three phases are implemented by a mode variable $mode_i$ at each node i. The mode at node i has one of three possible values: *init*, *reset*, and *normal*. All nodes are in the *normal* mode when no reset is in progress. To initiate a reset, a node i sets $mode_i$ to *init* (this can be done only if both i and its parent are in *normal* mode). A reset request is propagated upwards by the action PROPAGATE_REQUEST$_i$ which sets the mode of the parent to *init* when the mode of the child is *init*. A reset wave is begun by the root by the action START_RESET which sets the mode of the root to *reset*. The reset wave propagates downwards by PROPAGATE_RESET$_i$ which sets the mode of a child to *reset* if the mode of the parent is *reset*. When a node changes its mode to *reset*, it flips its session number bit, and resets the application protocol. Finally, the completion wave is propagated by the action PROPAGATE_COMPLETION$_i$ which sets a node's mode to *normal* when *all* the node's children have *normal* mode.

The automaton code for this implementation is shown in Figure 2 and Figure 3. Notice that besides the actions we have already described, there is a CORRECT$_i$ action in Figure 3. This action was used in an earlier version [AG90] to ensure that the reset protocol was stabilizing.

Informally, the reset protocol is stabilizing if after bounded time, any reset requests will cause the application protocol to be properly reset. The correction action in Figure 3 [AG90] ensures stabilization in a very ingenious way. However, the proof of stabilization is somewhat difficult and not as intuitive as one might like. The reader is referred to [AG90] for details. Instead, we will use local checking and tree correction to describe *another* correction procedure that is very intuitive. As a result, the proof of stabilization becomes transparent.

We start by writing down the "good" states of the reset system in terms of link predicates $L_{i,j}$. We say that the system is in a good state if for all neighboring nodes i and j, the predicate $L_{i,j}$ holds, where $L_{i,j}$ is the conjunction of the two predicates:

- If $(parent(i) = j)$ and $(mode_j \neq reset)$ then $(mode_i \neq reset)$ and $(sbit_i = sbit_j)$
- If $(parent(i) = j)$ and $(mode_j = reset)$ then either:
  - $(mode_i \neq reset)$ and $(sbit_i \neq sbit_j)$ OR
  - $(sbit_i = sbit_j)$

The predicates can be understood intuitively as describing states that occur when the reset system is working correctly. The first predicate says that if the parent's mode is not *reset*, then the child's mode is not *reset* and the two session bits are the same. This is true when the system is working correctly because

The state of the system consists of two variables for every process in the tree:
$mode_i \in \{init, normal, reset\}$
$sbit_i$, a bit

PROPAGATE_REQUEST$_i$ (*internal action to propagate a reset request upwards *)
Preconditions:
$mode_i = normal$
$i = parent(j)$ and $mode_j = init$
Effects: $mode_i := init$

START_RESET$_i$ (*internal action at root to start a reset wave*)
Preconditions: $mode_i = init$ and $parent(i) = nil$
Effects:
$mode_i := reset$; (*also reset application state at this node*)
$sbit_i :=\sim sbit_i$; (*flip bit*)

PROPAGATE_RESET$_i$ (*internal action to propagate reset downwards*)
Preconditions:
$mode_i \neq reset$
$j = parent(i)$ and $mode_j = reset$ and $sbit_j \neq sbit_i$
Effects:
$mode_i := reset$; (*also reset application state at this node*)
$sbit_i :=\sim sbit_i$; (*flip bit*)

PROPAGATE_COMPLETION$_i$ (*internal action to propagate completion wave upwards *)
Preconditions:
$mode_i = reset$
For all children j of i: $mode_j = normal$ and $sbit_i = sbit_j$
Effects: $mode_i := normal$

Every action is in a separate class with upper bound equal to
t_n

Figure 2: Normal Actions at node i in Arora and Gouda's Reset Protocol.[AG90]

CORRECT$_i$ (*extra internal action for correction at node i*)
Preconditions:
$j = parent(i) \neq nil$
$(mode_j = mode_i)$ and $(sbit_i \neq sbit_j)$
Effects: $sbit_i := sbit_j$

Every action is in a separate class with upper bound equal to t_n

Figure 3: Original correction action in Arora and Gouda's Reset Protocol [AG90].

of two reasons. First, the child enters *reset* mode only when its parent is in that mode, and the parent does not leave *reset* mode until the child has left *reset* mode. Second, if the parent changes its session bit, the parent also goes into *reset* mode; and the child only changes its session bit when the parent's mode is *reset*.

The second predicate describes the correct states during the second and third phases of the reset until the instant that the completion wave reaches j. It says that if the parent's mode is *reset*, then there are two possibilities. If the child has not "noticed" that the parent's state is *reset*, then the child's bit is not equal to the parent's bit. (This follows because when the parent changes its mode to *reset*, the parent also changes its bit; and just before such an action the second predicate assures us that the two bits are the same.) On the other hand, if the child has noticed that the parent's state is *reset*, then the two bits are the same. (This follows because when the child notices that the parent's mode is *reset*, the child sets its bit equal to the parent's bit and does not change its bit until the parent changes its mode.)

Suppose that in some state s these link predicates hold for all links in the tree. Then [AG90] show that the system will execute reset requests correctly in any state starting with s. This is not very hard to believe. But it means that all we have to do is to add correction actions so that all link predicates will become true in bounded time. But this can easily be done using the transformation underlying the Tree Correction Theorem that we just stated. An even simpler correction strategy is described below.

The tree topology once again suggests a simple strategy. We remove the old action CORRECT$_i$ in Figure 3 and add a new action CORRECT_CHILD$_i$ as shown in Figure 4. Basically, CORRECT_CHILD$_i$ checks whether the link predicate on the link between i and its parent is true. If not, i changes its state such that $L_{i,j}$ becomes true. Notice that CORRECT_CHILD$_i$ leaves the state of i's parent unchanged. Suppose j is the parent of i and k is the parent of j. Then CORRECT_CHILD$_i$ will leave $L_{j,k}$ true if $L_{j,k}$ was true in the previous state.

Thus we have an important stability property: correcting a link does not affect the correctness of links above it in the tree. Using this we can show that in bounded time, all links will be in a good state and so the system is in a good state.

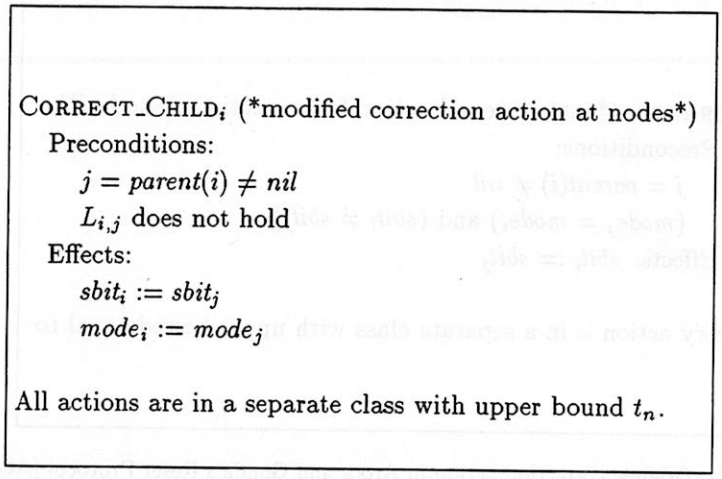

Figure 4: Modified Correction action for Arora and Gouda's Reset Protocol. All other actions are as in Figure 2.

5 Related Work

The original paper on local checking and correction [APV91b] proved a *local correction theorem*: any locally checkable protocol that met certain other constraints (referred to as *local correctability*) could be stabilized. However, that theorem *does not* imply our tree correction theorem. In order to apply the Local Correction Theorem, one has to explicitly exhibit a correction function with the appropriate properties; this is not needed in Tree Correction. Also, the explicit correction action used in local correction only depends on the previous state of a node; however, the implicit correction action used in the proof of Tree Correction depends on the state of a node's parent as well as the node's previous state. Finally, the definition of local checkability used in [APV91b] and [AGV94] is stronger than ours. The previous definitions require that each local predicate be a closed predicate. This is a strong assumption that we *do not require.*

Both local correction and tree correction are useful techniques that apply to different problems. For example, local correction has been used to provide stabilizing solutions to many problems on general graphs (e.g., synchronizers [Var93], end-to-end protocols [APV91b]) for which tree correction is inapplicable. On the other hand, local correction does not seem applicable to the reset protocol described in Section 4. This is because the correction action (Figure 4) depends on both the state of the parent and the child. There are still other problems (e.g., mutual exclusion on a tree [Var93]) for which both techniques are applicable.

There has also been considerable work in the Artificial Intelligence literature on *distributed constraint satisfaction.* A constraint can be considered to be a local predicate between two neighboring nodes, and the goal is to find a solution that meets all constraints. A seminal paper[3] by Collin, Dechter, and Katz [CDK91] considers the feasibility of self-stabilizing, distributed constraint satisfaction and shows that the problem is impossible in general graphs without some form of symmetry breaking. However, they also show that the problem is solvable in tree networks. Their basic procedure is similar to ours: they use what they call an *arc consistency step* (similar to our normalization step) and then each node chooses a value

[3]We are grateful to one of the referees for pointing out this reference.

in its domain that is consistent with its parent's value.

However, their tree protocol is limited to finding *static* solutions to constraint satisfaction. Thus constraint satisfaction can be used to solve a static problem like coloring but not a *dynamic* problem like mutual exclusion or network reset. In other words, constraint satisfaction seeks solutions that converge to a fixed point; our formulation seeks solutions that converge to a closed predicate. Even after convergence to a closed predicate, protocol actions continue to occur. In order to allow this, we added an extra modification beyond the the normalization and local correction modifications needed by [CDK91]: we modified the original protocol actions to add an extra guard to check whether all local predicates were satisfied before the action is executed. This modification is crucial to ensure stability of the local predicates (Lemma 3.5) which in turn prevents oscillatory behavior (in which predicates are corrected for and then become falsified by protocol actions). The proof of stability requires the notion of local extensibility (Lemma 3.3). None of these notions are required for the result in [CDK91].

Thus our theorem does not follow from the result in [CDK91]. However, the result in [CDK91] applies to uniform tree networks — i.e., without symmetry breaking in the form of parent pointers, as we have used. This is done by a protocol that finds tree centers and directs each node's parent pointer towards its closest center. We believe that their method could be applied to make our theorem applicable to uniform tree networks. [CDK91] also has a number of other results on general networks, including a self-stabilizing procedure to find a feasible solution using distributed backtracking.

Finally, [AGV94] also expresses local checkability conditions using *constraint graphs*. The formulation in [AGV94] applies to more general protocols than the constraint satisfaction protocols of [CDK91]. However, none of the theorems in [AGV94] or [APV91b] imply our tree correction theorem.

6 Conclusions

It may seem "obvious" that any locally checkable protocol on a tree can be stabilized. However, it is a different matter to state and prove a precise result that embodies this intuition. The proof requires some subtle notions such as normalizing each automaton, the notion of local extensibility, and the need to defend against unexpected transitions.

Much of the initial work in self-stabilization was done in the context of Dijkstra's shared memory model of networks. Later, the work on local checking and correction was introduced [APV91b] in a message passing model. A contribution of this paper is to show that existing work in the shared memory model can be understood crisply in terms of local checking and correction. Protocols that appeared to be somewhat *ad hoc* are shown to have a common underlying principle.

Although we have not described it here for lack of space, Dijkstra's first protocol in [Dij74] can also shown to be a locally checkable protocol that works on a line graph, and thus is amenable to tree correction. The correction actions we add are once again different from the original actions in [Dij74]. However, the corrections actions we add (and consequently the proofs) are much more transparent than the original version. See [Var93] for details.

As we have argued at the beginning of this paper, we believe that message passing models are more useful and realistic. The definitions of network automata, local predicates, local checkability, local correctability, and link subsystems that we used in this paper are specific to shared memory systems. The main theorem in this paper states that any locally checkable protocol that uses a tree topology can be efficiently stabilized. As the reader might expect, there is a corresponding Tree Correction theorem for message passing systems. This theorem is described in [Var93].

References

[AG90] Anish Arora and Mohamed G. Gouda. Distributed reset. In *Proc. 10th Conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 316–331. Spinger-Verlag (LNCS 472), 1990.

[AGV94] Anish Arora and Mohamed G. Gouda and George Varghese. Constraint Satisfaction as a Basis for Designing Nonmasking Fault-tolerance. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol 18, American Mathematical Society, 1994.

[APV91b] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, October 1991.

[CDK91] Z. Collin, R. Dechter, and S. Katz. On the Feasibility of Distributed Constraint Satisfaction. *Proceedings of the 12th IJCAI*, August 1991.

[Dij74] Edsger W. Dijkstra. Self stabilization in spite of distributed control. *Comm. of the ACM*, 17:643–644, 1974.

[Fin79] Steven G. Finn. Resynch procedures and a fail-safe network protocol. *IEEE Trans. on Commun.*, COM-27(6):840–845, June 1979.

[MMT91] M. Merritt, F. Modugno, and M.R. Tuttle. Time constrained automata. In *CONCUR 91*, pages 408–423, 1991.

[HS91] P. Humblett and S. Soloway. A Fail Safe Layer for Distributed Network Algorithms and Changing Topologies. Techical Report LIDS-P-1702, Lab for Information and Decision Sciences, MIT, May 1987.

[Var93] George Varghese. Self-stabilization by local checking and correction. Ph.D. Thesis MIT/LCS/TR-583, Massachusetts Institute of Technology, 1993.

Paper Number 13

Formal Derivation of a Probabilistically Self-Stabilizing Program: Leader Election on a Uniform Tree

Takashi Amisaki, Yoshihiro Tsujino, and Nobuki Tokura

Formal Derivation of a Probabilistically Self-Stabilizing Program: Leader Election on a Uniform Tree

Takashi Amisaki* Yoshihiro Tsujino† Nobuki Tokura†

Abstract

This paper concerns a formal derivation of a probabilistically self-stabilizing program that solves a leader election problem on a uniform tree-structured distributed system. The derivation method we use is the stepwise refinement of a specification in the UNITY formalism. An initial specification for the program is naturally introduced in terms of progress and safety assertions. The specification is just the definition of a self-stabilizing property. The specification is then repeatedly refined to finally obtain a corresponding UNITY program. Every verification at every refinement step is completely formal, since it is carried out within predicate calculus without considering the meanings of each assertion or each program text.

1 Introduction

Assertional reasoning with some temporal proof logic can be a formal method for verification and/or derivation of concurrent programs. In this method, we reason about concurrent programs using two kinds of assertions, i.e., safety and progress assertions. Roughly speaking, a safety assertion states a restriction which a program should obey, while a progress assertion states a requirement which a program should establish. With these two kinds of assertions, we can reason about various properties of concurrent programs, such as invariants, termination and reachability.

Self-stabilization is a property which makes a program tolerate a certain kind of failures. According to the definition by Schneider[1], a program is said to be self-stabilizing, if it can attain its legitimate state irrespective of its initial state (*convergence*), and remains in its legitimate state once the state is established (*closure*). Owing to these properties, a self-stabilizing program can recover from transient errors, in which its variables are corrupted, and then the program is stabilized, if the errors do not continue to occur.

We can similarly recognize the advantage of a self-stabilizing system. If a distributed system is self-stabilizing, then the system can recover from transient failures, which may change the global state of the system by corrupting the local state of a process or by corrupting communication links between processes. Self-stabilization is thus a desirable property for distributed systems.

Self-stabilization can be subject to assertional reasoning. Convergence and closure properties of a program can be stated with progress and safety assertions, respectively. Let P be the predicate

*Department of Computer and Information Science, Faculty of Science, Shimane University, 1060 Nishikawatsu, Matsue 690, JAPAN. E-mail: ami@cis.shimane-u.ac.jp. Fax: +81.852.32.6489.

†Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University, 1-3 Machikaneyama, Toyonaka 560, JAPAN. E-mail: {tsujino, tokura}@ics.es.osaka-u.ac.jp. Fax: +81.6.850.6582.

over the domains of program variables. Then using UNITY notation ([2]), for example, we can assert the convergence and closure properties of the program with respect to P by

$$(true \mapsto P) \text{ and } (\textbf{stable}\ P),$$

respectively. These assertions are an obvious specification for a self-stabilizing program, and such simple specification can be a common starting point for formal derivation of any self-stabilizing programs.

Chandy and Misra[2] introduced UNITY, a formalism to aid in both verification and derivation of concurrent algorithms, while many of other such methods have intended only for a posteriori verification of the programs. In this sense, UNITY is appropriate for formal derivation of self-stabilizing programs. Program derivation in UNITY formalism proceeds by a series of refinement steps, each of which constitutes a strengthening of the specification at the previous refinement step. The method can be completely formal in the sense that all reasoning steps are performed within predicate calculus.

Recently, Rao[3] has extended the UNITY formalism for reasoning about a class of properties of concurrent programs, a property that hold either deterministically or with probability 1. As stated in his paper, the method is applicable to reasoning about probabilistic programs, involving self-stabilization, mutual exclusion or leader election programs. However, few derivation of such programs has been reported, at least to our knowledge.

In this paper, we present an example showing how a probabilistically self-stabilizing program can be derived from the above specification using the extended UNITY formalism. We derive a program for leader election on a tree-structured distributed system, assuming the following architectural constraint: (a) the distributed system is uniform, i.e., all processes are identical and execute the same program; and (b) all communications are made via registers associated with each communication link. The basic idea behind the algorithm that we are deriving is same as the one which we previously reported in [4] with rather informal proof. In the present paper, on the other hand, the correctness of the formally derived program is guaranteed by its construction.

We construct the program from two component programs using *union*, which is one of the program structuring techniques of UNITY. The component programs are so selected that both of them can be self-stabilizing. Accordingly, a composite program that is union of the component programs is also self-stabilizing. We use one program for finding centers of a tree and the other for electing a process to leader of two processes. If the center of the tree is exactly one, we choose the process as a leader. Otherwise, if two processes are centers of the tree, then we employ the latter program to choose exactly one leader. Because of the two assumption that we have made, i.e., uniformity of the system and communication via registers, the latter component program should be necessarily probabilistic, and we use a kind of randomization that can be originally found in a protocol for self-stabilizing ring orientation by Israeli and Jalfon[5].

The remainder of this paper is organized as follows: Section 2 contains description for a notation which is intensively used to express quantification throughout this paper. Section 3 describes a brief introduction to UNITY. Section 4 formally specifies our uniform self-stabilizing program which comprises the other two uniform self-stabilizing programs. These two component programs are formally derived in sections 5 and 6, where the verification at each refinement step can be performed within predicate calculus. Since the entire proof are not presented in this paper, interested reader is referred to [6]. Finally, we conclude our paper with some remarks on our derivation process and on the derived program.

2 Notational Conventions

We use the following convention for quantified expressions[7, 2]:

$\langle \oplus x : r.x : t.x \rangle$,

where $\oplus$ is any binary, associative and symmetric operator, such as $+, \wedge, \vee$ and max. And $x, r.x, t.x$ are called the dummy, the range, the term of the quantification, respectively. An infix dot is used to denote function application. If the quantification is either a predicate or an arithmetic expression, the result is obtained by evaluating the expression, $t.x_1 \oplus t.x_2 \oplus \cdots \oplus t.x_n$, where $x_1, x_2, \cdots, x_n$ are the all possible instantiations of the dummy which satisfy the range. For example, $\langle \exists i : 0 \leq i \wedge i \leq N : A[i] \rangle$ is true if some element of $A[0..N]$ is true, and $\langle \sum i : 0 \leq i \wedge i \leq N \wedge A[i] < A[j] : 1 \rangle$ is the number of elements smaller than $A[j]$ in $A[0..N]$. As shown in these examples, when $\oplus$ is one of $+, \wedge$ and $\vee$, it is written as $\sum, \forall$ and $\exists$, respectively. In case $\oplus$ has an identity element e, $\langle \oplus x : \mathit{false} : t.x \rangle$ is well-defined and equal to e. The range of the quantification is occasionally omitted when the range is obvious from the context.

The binary operator max gives the maximum of two numbers, formally defined as:

$\langle \forall z :: z \geq x \max y \equiv z \geq x \wedge z \geq y \rangle$.

The domain of max is restricted to the set of natural numbers, and, thus, its identity element is 0.

3 An Overview of UNITY

In this section, we briefly describe the extended UNITY ([2, 3]), as simplified for our purposes.

UNITY Program

A UNITY program, say F, is a set of multiple, probabilistic assignment statements, S_i. It is represented as

$F \equiv \langle \, [] \, i :: S_i \rangle$,

where $[]$ is a binary operator which gives union of two sets of statements, or simply separates two statements in a UNITY program. A probabilistic assignment statement S_i can be expressed as

$x := e_0 \mid e_1 \mid \cdots \mid e_{k-1}$ **if** b,

where x is a list of variables, each $e_j (0 \leq j \wedge j < k)$ a list of expressions, and b a Boolean expression. When $k = 1$, S_i is a usual conditional multiple assignment statement.

Execution of a UNITY program proceeds by repeatedly selecting a statement S_i with some i and executing it. The only constraint we impose on this selection process is *unconditional fairness*, that is, in an infinite execution, each statement is selected infinitely often. Execution of probabilistic assignment statement S_i is to execute a conditional multiple assignment $x := e_j$ **if** b for some j. Each of these k possible assignments is called a *mode* of S_i. The constraint we impose on this mode selection is *extreme fairness*: For any first-order predicate over program variables, X, it holds that, if S_i is executed infinitely often from the state satisfying X, then every mode of S_i is executed infinitely often from that state. On the other hand, we do not concern with the actual probability p_j associated with each mode j. We only require that $\langle \forall j : 0 \leq j \wedge j < k : p_j \neq 0 \rangle$ and $\langle \sum j : 0 \leq j \wedge j < k : p_j \rangle = 1$.

UNITY Operators

We first define two predicate transformers because we use the predicate transformer semantics of S_i to define the UNITY operators. Let X be a predicate. A predicate transformer $\mathbf{wp}.S_i$ is defined as

$$\mathbf{wp}.S_i.X \equiv b \Rightarrow \langle \forall j : 0 \leq j \wedge j < k : X[x := e_j] \rangle,$$

where $X[x := e_j]$ stands for X with all occurrences of variables in x simultaneously replaced by the matching expressions in e_j. The predicate $\mathbf{wp}.S_i.X$ is called the *weakest precondition* of S_i with respect to X. The other predicate transformer $\mathbf{wpp}.S_i$ is defined as

$$\mathbf{wpp}.S_i.X \equiv b \Rightarrow \langle \exists j : 0 \leq j \wedge j < k : X[x := e_j] \rangle.$$

The predicate $\mathbf{wpp}.S_i.X$ is called the *weakest probabilistic precondition* of S_i with respect to X.

Safety assertions in UNITY are written using operators **unless** , **upto** or **stable** , defined as:

$$X \textbf{ unless } Y \equiv \langle \forall S_i : S_i \in F : X \wedge \neg Y \Rightarrow \mathbf{wp}.S_i.(X \vee Y) \rangle,$$
$$X \textbf{ upto } Y \equiv \langle \forall S_i : S_i \in F : X \wedge \neg Y \Rightarrow \mathbf{wp}.S_i.X \vee \mathbf{wpp}.S_i.Y \rangle,$$
$$\textbf{stable } X \equiv X \textbf{ unless } \mathit{false}.$$

Intuitively, X **unless** Y means that whenever predicate X holds for a state, X continues to hold at least until Y holds. On the other hand, X **upto** Y means that either (X **unless** Y) holds, or X is falsified by executing a statement containing a mode that could have made Y establish. **Stable** X means that, once X is established, it will never be falsified. For example, both **stable** $x \geq k$ and $x = k$ **unless** $x > k$ state that x never decrease. (In this paper, aassertions are often written without explicit quantification; these are universally quantified over all values of free variable occuring in them. For example, **stable** $x \geq k$, where x is a program variable and k is a free variable, should be understood as $\langle \forall k :: \textbf{stable } x \geq k \rangle$.)

Under the unconditional fairness constraint, if there exists a statements that can contribute progress to the program, and the other statements never disturb its contribution, then the progress property of the program is guaranteed. **Ensures** and **entails** are operators employed in asserting such steady progress properties. These are defined as:

$$X \textbf{ ensures } Y \equiv X \textbf{ unless } Y \wedge \langle \exists S_i : S_i \in F : X \wedge \neg Y \Rightarrow \mathbf{wp}.S_i.Y \rangle,$$
$$X \textbf{ entails } Y \equiv X \textbf{ upto } Y \wedge \langle \exists S_i : S_i \in F : X \wedge \neg Y \Rightarrow \mathbf{wpp}.S_i.Y \rangle.$$

Intuitively, X **ensures** Y means that, if X holds in a state, then X continues to hold until Y holds, and Y will certainly hold. X **entails** Y means that, if X is infinitely often true in a computation, then Y is infinitely often true.

At the beginning of program development, program texts are not available, but to be determined later in the development. Hence, at early stage, abstract progress properties are expressed using the operators, $\mapsto$, $\rightsquigarrow$ and $\models\!\!\!\Rightarrow$ instead of **ensures** and **entails** . The $\mapsto$ operator is defined by

$$\frac{X \textbf{ ensures } Y}{X \mapsto Y} \text{ (Base)}, \quad \frac{X \mapsto Y \quad Y \mapsto Z}{X \mapsto Z} \text{ (Transitivity)},$$
$$\frac{\langle \forall X : X \in W : X \mapsto Y \rangle}{\langle \exists X : X \in W : X \rangle \mapsto Y} \text{ (Disjunctivity)},$$

where W is an arbitrary set of predicates. Then the $\rightsquigarrow$ operator is defined as

$\frac{X \textbf{ entails } Y}{X \leadsto Y}$ (Base), $\frac{X \mapsto Y}{X \leadsto Y}$ (Leads-to), $\frac{X \leadsto Y \quad Y \leadsto Z}{X \leadsto Z}$ (Transitivity).

Using $\leadsto$, the $\Longmapsto$ operator is defined as

$$\frac{X \textbf{ unless } Y \quad X \leadsto Y}{X \Longmapsto Y} \text{ (Base)}, \quad \frac{X \Longmapsto Y \quad Y \mapsto Z}{X \Longmapsto Z} \text{ (Transitivity)},$$
$$\frac{\langle \forall X : X \in W : X \Longmapsto Y \rangle}{\langle \exists X : X \in W : X \rangle \Longmapsto Y} \text{ (Disjunctivity)},$$

Intuitively, $X \mapsto Y$ means that, if X holds then Y will eventually hold, and $X \Longmapsto Y$ means that, if X holds then Y will eventually hold with probability 1.

We present here some useful properties of UNITY operators just we defined. The details and their proofs can be found in [2, 3, 6].

- PSP(Progress-Safety-Progress):
  $$\frac{X \mapsto Y \quad U \textbf{ unless } V}{(X \wedge U) \mapsto (Y \wedge U) \vee V}$$
- Completion: For i ranging over any finite set I,
  $$\frac{\langle \forall i :: X.i \mapsto Y.i \rangle \quad \langle \forall i :: Y.i \textbf{ unless } Z \rangle}{\langle \forall i :: X.i \rangle \mapsto \langle \forall i :: Y.i \rangle \vee Z}$$
- Induction Principle: Let $(W, \prec)$ be a well-founded set. Let M be a mapping from program states to W. Then,
  $$\frac{\langle \forall m : m \in W : (X \wedge M = m) \mapsto M \prec m \rangle}{true \mapsto \neg X}$$
- Union(program composition): The *union* of two programs F and G is simply a set union, denoted by $F \,[]\, G$. There are several useful theorems about union operation. For example,
  $$\frac{X \textbf{ unless } Y \textbf{ in } F \quad X \textbf{ unless } Y \textbf{ in } G}{X \textbf{ unless } Y \textbf{ in } F \,[]\, G} \quad \text{or} \quad \frac{X \textbf{ ensures } Y \textbf{ in } F \quad X \textbf{ unless } Y \textbf{ in } G}{X \textbf{ ensures } Y \textbf{ in } F \,[]\, G}.$$

Specification Refinement

In UNITY formalism, a specification is a collection of assertions written in terms of UNITY operators. Let P, Q be specifications. P is said to be a refinement of Q if and only if P implies Q. A program derivation proceeds by iterative refinements of the specification, until the translation from the specification to UNITY code is obvious. The correctness of each refinement step is verified using the theorems about the properties of UNITY operators.

4 Problem and Its Specification

Consider a distributed system composed of a finite set of processes, V, and a set of bidirectional links between them, E. A link is denoted by a two-element set, e.g., $\{u, v\} \in E$. Each process can communicate with its neighbors through links between them. Each communication is made via a single register associated with each direction of each link. The system's communication graph is

the graph formed by representing each process as a node and each link as an edge. We assume that the graph $N = (V, E)$ is an undirected tree, and that the system is uniform, i.e., all processes are identical and execute the same program. Hence the tree does not have root. Electing leader on such distributed system in a self-stabilizing manner is our problem.

As mentioned in section 1, we build the program L from two programs, F and G. The former elects leaders on every two adjacent processes. The latter finds centers of a tree. We can think of the program F as a composite progarm $\langle \Box e : e \in E : F_e \rangle$ where F_e is a program that elects a leader of two processes, say x and y, connected by a link e. F_e comprises a part of a program F_x, which a process x executes, and a part of a program F_y, which a process y executes. Because of the uniformity of the system, all F_e's are identical. Therefore, the independent component that we have to derive are only one F_e and G. In this section, initial specifications for the programs, F_e and G, are introduced. Note that if no variable are shared among the all programs, then the properties of every component program are preserved in the composite program L.

First, we give an initial specification for the program F_e. When discussing specification of F_e, we consider a two-process system for a while to avoid use of complicated names for program variables. We use x and $\bar{x}$ to denote the two adjacent processes. We define the leader of x and $\bar{x}$ as a process whose local variable s is greater than the others'. Namely, if $x.s > \bar{x}.s$ then x is a leader of the two, or if $x.s < \bar{x}.s$ then $\bar{x}$ is a leader of the two. Because of the uniformity of the system and communication via registers, the program F_e should be probabilistic rather than deterministic. Accordingly, we propose the following specification for the program F_e:

[Specification F1]
(F1.1) $\mathit{true} \models\!\Longrightarrow P_F$
(F1.2) **stable** P_F
(F1.3) $P_F \Rightarrow x.s \neq \bar{x}.s$

where P_F is a certain but unspecified predicate determined subsequently in the course of refinement.

Next, we consider the program G. The centers of a tree are nodes whose eccentricities are the smallest in the tree. We define the legitimate state of G as a state, such that each process x keeps its own eccentricity e_x in its local variable $x.e$. We then propose the following specification of the program G:

[Specification G1]
(G1.1) $\mathit{true} \mapsto P_G$
(G1.2) **stable** P_G
(G1.3) $P_G \Rightarrow \langle \forall x :: x.e = e_x \rangle$

where P_G is a certain predicate that characterizes the legitimate state of G.

The specifications F1 and G1 have the same form. It is just a definition of the self-stabilization. However, the other constraints, i.e., uniformity and communications via registers, are not explicitly stated in the specifications. These constraints are incorporated in the succeeding refinement steps.

In remainder of this section, we formally define eccentricity, and summarize some useful properties concerning it. The theorems are given without proofs, but can be mechanically proven with predicate calculus ([6]).

Definition 1 (Height of sub-tree) *Let $\{x, y\} \in E$. After being removed the edge $\{x, y\}$ a height of sub-tree rooted at x, denoted by h_x^y, is inductively defined as*

$$\langle \forall x, y : \{x, y\} \in E : h_x^y = \langle \max v : \{v, x\} \in E \wedge v \neq y : h_v^x + 1 \rangle \rangle.$$

Note that a height of sub-tree consisting of only one process is 0, since the identity element of max is 0.

Definition 2 (Eccentricity) *The eccentricity of x, e_x, is defined as*

$$\langle \forall x :: e_x = \langle \max v : \{v, x\} \in E : h_v^x + 1 \rangle \rangle.$$

Theorem 1 $\langle \forall x :: \langle \forall y : \{x, y\} \in E : e_x = h_x^y \rangle \equiv \langle \forall y : x \neq y : e_x < e_y \rangle \rangle.$

Theorem 2 $\langle \forall x, y : \{x, y\} \in E : e_x = h_x^y + 1 \equiv e_x = e_y \wedge \langle \forall z : z \neq x \wedge z \neq y : e_x < e_z \rangle \rangle.$

Theorem 3 $\langle \exists x :: \langle \forall y : y \neq x : e_x < e_y \rangle \rangle \not\equiv$
$\langle \exists x, y : \{x, y\} \in E : e_x = e_y \wedge \langle \forall z : z \neq x \wedge z \neq y : e_x < e_z \rangle \rangle.$

5 Leader Election on Two Processes

Introducing Register Variables

At this stage, we make two kinds of refinements. One is to introduce the register variables, $x.r$ and $\bar{x}.r$. The two processes communicate the values of $x.s$ or $\bar{x}.s$ to each other through $x.r$ and $\bar{x}.r$, respectively. The other refinement concerns progress property. The assertion F1.1 is conformable to a refinement that is naturally inspired with the shape of the definition of $\models\!\!\Rightarrow$. Namely, $(true \models\!\!\Rightarrow P_F)$ can be replaced with $(true \leadsto P_F)$ and the tautology $(true$ **unless** $p)$. In the following specification, the progress property has been further refined using transitivity of $\leadsto$.

[Specification F2]
(F2.1) $x.s = \bar{x}.s \leadsto x.s \neq \bar{x}.s$
(F2.2) $x.s \neq \bar{x}.s \leadsto P_F$
(F2.3) $x.s = i \wedge x.r = i \wedge \bar{x}.r \neq i$ **unless** $x.s = i \wedge x.r = i \wedge \bar{x}.r = i$
(F2.4) $P_F \equiv x.s \neq \bar{x}.s \wedge x.s = x.r \wedge \bar{x}.s = \bar{x}.r$

Since the system concerned is uniform, the names of processes should be interchangable. In case that an assertion is asymmetric with respect to x and $\bar{x}$, another assertion obtained by interchanging x and $\bar{x}$ should also hold. Therefore, the assertion F2.3 should be understood as

$$\langle \forall u, v : u \in \{x, \bar{x}\} \wedge v \in \{x, \bar{x}\} \wedge u \neq v :$$
$$u.s = i \wedge u.r = i \wedge v.r \neq i \textbf{ unless } u.s = i \wedge u.r = i \wedge v.r = i \rangle,$$

or more precisely, as mentioned in Section 3,

$$\langle \forall i :: \langle \forall u, v : u \in \{x, \bar{x}\} \wedge v \in \{x, \bar{x}\} \wedge u \neq v :$$
$$u.s = i \wedge u.r = i \wedge v.r \neq i \textbf{ unless } u.s = i \wedge u.r = i \wedge v.r = i \rangle \rangle.$$

(Verifications)

F2 $\Rightarrow$ F1.1:

1 $x.s = \bar{x}.s \leadsto P_F$
, transitivity of $\leadsto$ using F2.1 and F2.2

2 *true* $\leadsto P_F$
, disjunction of $\leadsto$ using 1 and F2.2

3 *true* **unless** P_F
, tautology for **unless**

4. *true* $\models\!\!\Longrightarrow P_F$ — (F1.1)
, from 2 and 3, definition of $\models\!\!\Longrightarrow$

We can prove F2 $\Rightarrow$ F1.2 using the property about general conjunction of **unless** ([6]). And F1.3 immediately follows from F2.4. □

Hereafter, we replace P_F with $x.s \neq \bar{x}.s \wedge x.s = x.r \wedge \bar{x}.s = \bar{x}.r$, and omit F2.4, for simplicity.

Refinement on Progress

A progress assertion that contains **ensures** or **entails** can correspond to a single statement in a UNITY program. We replace $\leadsto$ operators that occur in the specification F2 with **ensures** or **entails** , since each assertion that contains $\leadsto$ cannot be directly translated into program text.

[Specification F3]

(F3.1) $x.s = i \wedge x.r \neq i$ **ensures** $x.s = i \wedge x.r = i$

(F3.2) $x.s = i \wedge x.r = i \wedge \bar{x}.r \neq i$ **unless** $x.s = i \wedge x.r = i \wedge \bar{x}.r = i$

(F3.3) $x.s = i \wedge x.r = i \wedge \bar{x}.s = i \wedge \bar{x}.r = i$ **ensures** $x.s \neq \bar{x}.s$

(F3.4) $x.s = i \wedge x.r = i \wedge \bar{x}.s = j \wedge \bar{x}.r = i$ **entails**
$(x.s = i \wedge x.r = i \wedge \bar{x}.s = j \wedge \bar{x}.r = j) \vee (x.s = k \wedge x.r = i \wedge \bar{x}.s = j \wedge \bar{x}.r = i)$

where i, j and k are distinct natural numbers.

Distribution

We are thus far concerned with the program F_e executed jointly by x and $\bar{x}$. At this stage, we make the final refinement, so that the program and the data are distributed to each process. Taking account of the constraint on the communication, we obtain the following specification of the program F_x that *a process x executes.*

[Specification F4]

(F4.1) **stable** $\bar{x}.s = i$ **in** F_x

(F4.2) **stable** $\bar{x}.r = i$ **in** F_x

(F4.3) **stable** $x.s = i \wedge x.r = i \wedge \bar{x}.r \neq i$ **in** F_x

(F4.4) $x.s = i \wedge x.r \neq i$ **ensures** $x.s = i \wedge x.r = i$ **in** F_x

(F4.5) $x.s = i \wedge x.r = i \wedge \bar{x}.r = i$ **ensures** $x.s \neq i \wedge x.r = i \wedge \bar{x}.r = i$ **in** F_x

(F4.6) $x.s = i \wedge x.r = i$ **upto** $x.s \neq i \wedge x.r = i \wedge x.s \neq \bar{x}.s$ **in** F_x

The specification of the program $F_{\bar{x}}$ that *a process $\bar{x}$ executes* is obtained by substituting x and $\bar{x}$ with $\bar{x}$ and x, respectively. Note that $(F_e = F_x \,[]\, F_{\bar{x}})$.

Program F

It is now straightforward to translate the specification into UNITY codes. According to the specifications we obtained thus far, it appears that three-valued (thus bounded) variables are appropriate for any variables which occur in the specification. We therefore choose

$$x.r := x.s \textbf{ if } x.r \neq x.s \qquad \text{and}$$
$$x.s := x.s + 1 \mid x.s + 2 \textbf{ if } x.s = x.r \wedge x.s = \bar{x}.r,$$

for F4.4 and F4.5, respectively, where evaluations for operators, $+$, $=$ and $\neq$, are made modulo 3. (Verifications) It is obvious that both of these two statements do not violate the five assertions from F4.1 to F4.5. And it is also obvious that the first statement does not violate F4.6. We show the second statement, which we denote by s, does not violate the assertion F4.6. For simplicity, we denote lhs and rhs of **unless** in the assertion F4.6 by X and Y, respectively. Then, we have to show the validness of the predicate

$$X \wedge \neg Y \Rightarrow \mathbf{wp}.s.X \vee \mathbf{wpp}.s.Y.$$

Assume $X \wedge \neg Y$. Then, according to the definitions of **wp** and **wpp**, we can show that $\mathbf{wp}.s.X \equiv x.s \neq \bar{x}.r$ and $\mathbf{wpp}.s.Y \equiv x.s = \bar{x}.r$. Hence $\mathbf{wp}.s.X \vee \mathbf{wpp}.s.Y \equiv \mathit{true}$. □

Up to this point we have developped a program F_e for leader election on two-process system. The program is easily genaralized into the composite program $F(= \langle \Box e : e \in E : F_e \rangle)$ that elects leaders on every two adjacent processes. It is given below:

Program F
$\langle \Box x : x \in V :$
  $\langle \Box y : \{y, x\} \in E :$
    $x.r[y] := x.s[y]$ **if** $x.r[y] \neq x.s[y]$
    $\Box$ $x.s[y] := x.s[y] + 1 \mid x.s[y] + 2$ **if** $x.s[y] = x.r[y] \wedge x.s[y] = y.r[x]$
  $\rangle$
$\rangle$

The correctness of the program F is guaranteed by the locality of the program variables and by the correctness of the previous statements.

6 Centers of Tree

Introducing Variables for Height of Sub-Tree

As previously mentioned, if the program G has reached a state where each process keeps its own eccentricity in its local variables $x.e$, then there exists exactly one process or one pair of neighbors that keep the smallest $x.e$. Moreover, we can find whether a process contains the smallest $x.e$ by comparing its eccentricity with relevant sub-tree heights (Theorems 1, 2). Since associated with each link $\{x, y\} \in E$ are two heights of sub-tree, h_x^y and h_y^x, we introduce respective variables, $x.d[y]$ and $y.d[x]$.

In addition, we refine the progress assertion G1.1 using the induction principle.

[Specification G2]
(G2.1) $\neg P1 \wedge M = m \mapsto M \succ m$
(G2.2) $P1 \mapsto P1 \wedge P2$
(G2.3) **stable** $P1$
(G2.4) **stable** $P1 \wedge P2$
(G2.5) $P_G \equiv P1 \wedge P2$

where, $P1 \equiv \langle \forall x, y :: x.d[y] = \langle \max v : v \neq y : v.d[x] + 1 \rangle \rangle$,
$P2 \equiv \langle \forall x :: x.e = \langle \max v :: v.d[x] + 1 \rangle \rangle$, M a mapping over the program states, and $\succ$ is an ordering on its range. Since there does not yet seem to be an obvious choice for M, we postpone the construction of M, and only assume $(W, \succ)$ is well-founded.
(Verifications) We give a proof for G1.1 as an example.

$\neg P1 \wedge M = m \mapsto M \succ m$ — (G2.1)
$\Rightarrow$ {induction principle}
$true \mapsto P1$
$\Rightarrow$ {transitivity with G2.2}
$true \mapsto P1 \wedge P2$
$=$ {G2.5}
$true \mapsto P_G$ — (G1.1) □

Hereafter, we replace P_G with $P1 \wedge P2$, and omit the assertion G2.5, for simplicity.

Distribution(1)

At this stage we replace the assertions which contain $\mapsto$ operator with the assertions that contain **ensures** operator. In addition, we preliminaryly distribute the specification, that is, roughly speaking, we intend to distribute the UNITY operators over universal quantifier in every assertion in G2. The obtained specification will be actually ditributed to each process in the next step.

[Specification G3]
(G3.1) $\langle \forall x, y :: x.d[y] = i \wedge \beta_x^y = j \wedge M = m$ **ensures** $M \succ m \rangle$
(G3.2) $\langle \forall x :: x.e \neq \alpha_x$ **ensures** $x.e = \alpha_x \rangle$
(G3.3) $\langle \forall x, y : x.d[y] = K[x, y] \wedge \beta_x^y = K[x, y] \wedge \pi_x^y$ **unless** $x.d[y] = K[x, y] \wedge \neg \pi_x^y \rangle$
(G3.4) $\langle \forall x :: x.e = i \wedge \alpha_x = i$ **unless** $x.e = i \wedge \alpha_x \neq i \rangle$

where, $\alpha_x = \langle \max v :: v.d[x] + 1 \rangle$, $\beta_x^y = \langle \max v : v \neq y : v.d[x] + 1 \rangle$,
$\pi_x^y \equiv \langle \forall v : v \neq y : v.d[x] = K[v, x] \rangle$, and K a two-dimensional array of free variable. In the assertion G3.1, i and j are distinct natural numbers. Namely the assertion should be understood as $\langle \forall i, j, m : i \neq j : \langle \forall x, y :: x.d[y] = i \wedge \beta_x^y = j \wedge M = m$ **ensures** $M \succ m \rangle \rangle$.
(Verifications) We show that specification G3 implies progress assertions in specification G2.

G3 $\Rightarrow$ G2.1:
G3.1
$\Rightarrow$ {definition of $\mapsto$}
$\langle \forall x, y :: x.d[y] = i \wedge \beta_x^y = j \wedge M = m \mapsto M \succ m \rangle$
$\Rightarrow$ {disjunction}

$\langle \exists x, y :: x.d[y] = i \wedge \beta^y_x = j \rangle \wedge M = m \mapsto M \succ m$

$\Rightarrow$ {removing free variables}

$\langle \exists x, y :: x.d[y] \neq \beta^y_x \rangle \wedge M = m \mapsto M \succ m$

$=$ {de Morgan}

$\neg \langle \forall x, y :: x.d[y] = \beta^y_x \rangle \wedge M = m \mapsto M \succ m$ — (G2.1)

G3 $\Rightarrow$ G2.2:

1 $P1$ **unless** *false*
, from the proof for G3 $\Rightarrow$ G2.3 ([6])

2 $\langle \forall x :: x.e \neq \alpha_x \mapsto x.e = \alpha_x \rangle$
, from G3.2 using definition of $\mapsto$

3 $\langle \forall x :: P1 \wedge x.e \neq \alpha_x \mapsto P1 \wedge x.e = \alpha_x \rangle$
, from 1 and 2 using PSP

4 $\langle \forall x :: P1 \wedge x.e = \alpha_x \mapsto P1 \wedge x.e = \alpha_x \rangle$
, obvious

5 $\langle \forall x :: P1 \mapsto P1 \wedge x.e = \alpha_x \rangle$
, disjunction with 3 and 4

6 $\langle \forall x :: P1 \wedge x.e = \alpha_x$ **unless** *false*$\rangle$
, from the proof for G3 $\Rightarrow$ G2.4 ([6])

7 $P1 \mapsto P1 \wedge \langle \forall x :: x.e = \alpha_x \rangle$ — (G2.2)
, completion with 5 and 6

□

Distribution(2)

Taking account of the architectural constraint, we make final refinement. To this end, each variable should be allocated to a process either as its local variable or as a register variable. We already have a natural choice of the way of the allocation: heights of sub-tree as register variables, and the others as local variables. Then we obtain the final specification G4 for a program G_x that *a process* x *executes*. Note that the composite program $\langle \Box x : x \in V : G_x \rangle$ is just the program G.

[Specification G4]

(G4.1) $\langle \forall y : y \neq x :$ **stable** $y.e = i$ **in** $G_x \rangle$

(G4.2) $\langle \forall y, z : y \neq x :$ **stable** $y.d[z] = i$ **in** $G_x \rangle$

(G4.3) **stable** $x.e = i \wedge \alpha_x = i$ **in** G_x

(G4.4) $\langle \forall y ::$ **stable** $x.d[y] = i \wedge \beta^y_x = i$ **in** $G_x \rangle$

(G4.5) $\langle \forall y :: x.d[y] = i \wedge M = m$ **unless** $M \succ m$ **in** $G_x \rangle$

(G4.6) $M = m$ **unless** $M \succ m$ **in** G_x

(G4.7) $\langle \forall y :: x.d[y] = i \wedge \beta^y_x = j$ **ensures** $x.d[y] = j \wedge \beta^y_x = j$ **in** $G_x \rangle$

(G4.8) $x.e \neq \alpha_x$ **ensures** $x.e = \alpha_x$ **in** G_x

Program G

It is now straightforward to transform the specification G4 into a UNITY program G. The specification G4 contains only two progress assertions, G4.7 and G4.8, which implies the use of $x.d[y] := \beta^y_x$ and $x.e := \alpha_x$, respectively. Thus a natural choice of program G is:

Program G
$\langle \,[]\, x : x \in V :$
$\quad \langle \,[]\, y : \{y, x\} \in E : x.d[y] := \langle \max v : \{v, x\} \in E \wedge v \neq y : v.d[x] + 1 \rangle \rangle$
$\quad [] \quad x.e := \langle \max v : \{v, x\} \in E : v.d[x] + 1 \rangle$
$\rangle$

(Verifications) Let h_m be $\langle \max u, v : \{u, v\} \in E : h_u^v \rangle$. And let $\succ$ be a lexicographic ordering of $(h_m + 1)$-tuple of natural numbers. We adopt the following $(h_m + 1)$-tuple for a mapping M:

$$(\langle \sum u, v : h_u^v = 0 \wedge u.d[v] = \beta_u^v : 1 \rangle, \cdots, \langle \sum u, v : h_u^v = h_m \wedge u.d[v] = \beta_u^v : 1 \rangle).$$

Because we assume that the system is composed of finite set of processes, each component of M is bounded from above, and thus M is bounded. Hence, when W is a set of all possible values of M over the set of the program states, $(W, \succ)$ is well-founded. With this choice of M, an assignment statement $x.d[y] := \beta_x^y$, for example, does not violate the assertion G4.5. This is because we can write the proof obligation as

$$x.d[y] = l \wedge M = k \wedge \beta_x^y \neq l \Rightarrow M[x.d[y] := \beta_x^y] \succ k,$$

which is a valid predicate. Although the details are not presented, these verification are mechanically performed within predicate calculus. □

7 Final Program

The program L is given simply as the union of the programs F and G. When the composite program $F \,[]\, G$ has reached at its legitimate state (fixed point), then, according to Theorems 1 and 2, there exists exactly one process x with $x.e = x.d[y]$ for all its neighbors y, or with $x.e = x.d[y] + 1$ and $x.s[y] > y.r[x]$ for some neighbor y. The leader is the process.

8 Concluding Remarks

We have derived a probabilistically self-stabilizing program L in this paper. The program we obtained solves a leader election problem on a uniform distributed system whose communication graph is represented as a tree. The program is derived using the stepwise refinement technique in UNITY formalism which has been extended for reasoning about properties of probabilistic programs. An initial specification for this derivation process is naturally obtained from a self-stabilizing property of the program. Subsequent verifications at each refinement step are completely formal, since it is performed within predicate calculus without considering the meanings of each assertion or each program text. In this sense, our derivation is completely formal. Moreover, we do not reason about any execution sequence or we did not take account of any probability associated with each event of the probabilistic program.

When designing programs with UNITY formalism, architectural considerations are delayed as long as possible, so that concerns on algorithms are seperated from those on a particular architecture. The derivation of program G proceeds in this way. On the other hand, the derivation process of program F begins with the introduction of the register variables. This is because almost entire portion of the derivation process corresponds to refinement concerning architectural constraint.

Incorporating architectural constraints into a specification, which is called a *mapping*, is often complicated within UNITY formalism. The authors are now interested in the way how we can make a mapping of the program to other communication means, such as message passing.

Program atomicity can be treated as a kind of architectural constraint. Although we carry out the derivations presented in this paper without explicit consideration on program atomicity, program F is correct under weak atomicity that is *read/write* atomicity given in [8]. A program is said to be correct under read/write atomicity, if all shared registers are serializable with respect to read/write operations. Program G is no correct in this atomicity in the strict sence, since its atomic step contains a multiple operation on registers, i.e., 'max'. However, it is easy to improve the atomicity of program G by introducing additional local variables during the course of the refinement.

The program structuring method that we used is a simple union. The method is sufficient for our purpose. On the other hand, a fair protocol compostion, that is a method for structuring self-stabilizing programs, has been proposed by Dolev, Israeli and Moran [8, 9]. The formal treatment for this method can be found in the paper by Stomp [10]. The union method under the unconditional fairness and the fair protocol composition are roughly same. However, the latter method specifies the way of selecting component programs. A basic idea behind their approach is to build program components as if these were executed one after another. For example, the program concerned in this paper could be built in the following way: Design a program F which is (probabilistically) self-stabilizing with respect to the predicate P_F; Design a program G' such that G' never falsify P_F and G' is stabilizing with respect to $P_F \wedge P'_G$. In this paper, on the other hand, two self-stabilizing component programs are developped independently. This leads to simple verifications at each refinement step. However, there are many self-stabilizing algorithms designed with a concept similar to that in the fair protocol composition, and it seems difficult to select independent component programs (sharing no program variables) for some algorithms. In a future work, we will consider a formal structuring method for such cases.

References

[1] M. Schneider. Self-Stabilization, *ACM Computing Surveys*, 25(1):45-67, 1993.

[2] K. M. Chandy and J. Misra. *A Foundation of Parallel Program Design*, Addison-Wesley, Reading, Mass., 1988.

[3] J. R. Rao. Reasoning about Probabilistic Parallel Programs, *ACM Transactions on Programming Languages and Systems*, 16(3):798-842, 1994.

[4] N. Nishikawa, T. Masuzawa and N. Tokura. Uniform Self-Stabilizing Algorithms for Mutual Exclusion, *Transactions of IEICE D-I*, J75-D-I(4):201-209, 1992 (in Japanese).

[5] A. Israeli and M. Jalfon. Self Stabilizing Ring Orientation, In *Lecture Notes in Computer Science 486: Distributed Algorithms*, Van Leeuwen, J. and Santoro, N. Eds., pages 1-14, Springer-Verlag, 1991.

[6] T. Amisaki, Y. Tsujino and N. Tokura. Formal Derivation of a Probabilistically Self-Stabilizing Program: Leader Election on a Uniform Tree, manuscript, Osaka University, 1995.

[7] E. W. Dijkstra and C. S. Scholten. *Predicate calculus and program semantics*, Springer-Verlag, New York, 1990.

[8] S. Dolev, A. Israeli and S. Moran. Self Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity, In *Proceedings of the ACM Symposium on Principles of Distributed Computing*(Quebec City), pages 103-117, ACM, New York, 1990.

[9] S. Dolev, A. Israeli and S. Moran. Uniform Dynamic Self-Stabilizing Leader Election. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, pages 167-180, 1991.

[10] F. A. Stomp. Structured Design of Self-Stabilizing Programs. In *the Proceedings of the Second Israel Symposium on Theory of Computing and Systems*, pages 167-176, 1993.

Paper Number 14

Uniform Randomized Self-Stabilizing Mutual Exclusion on Unidirectional Ring under Unfair C-Daemon

Hirotsugu Kakugawa and Masafumi Yamashita

Uniform Randomized Self-Stabilizing Mutual Exclusion on Unidirectional Ring under Unfair C-Daemon

Hirotsugu Kakugawa and Masafumi Yamashita
Hiroshima University, Japan
e-mail: {kakugawa,mak}@se.hiroshima-u.ac.jp

Abstract

A distributed system consists of a set of processes and a set of communication links, each connecting a pair of processes. A distributed system is called self-stabilizing if it converges to a correct system state no matter which system state it is started with. A self-stabilizing system is considered to be an ideal fault tolerant system, since it can tolerate a finite number of transient failures.

In this paper, we investigate a randomized self-stabilizing mutual exclusion system, under the assumption that the underlying communication topology of system is a unidirectional ring. It is well-known that if the number of processes (i.e., ring size) is composite, there is no deterministic system, even if c-daemon is assumed. We present a randomized self-stabilizing mutual exclusion system working under c-daemon. We allow the system to have a composite number of processes, and c-daemon to produce an unfair schedule. The system has only $4(n-1)$ states, where n is the number of processes.

1 Introduction

A distributed system consists of a set of processes and a set of communication links, each connecting a pair of processes. Processes observe the current states of adjacent processes, and decide next activities. In distributed systems, fault tolerance is an important issue, and many different approaches have been examined. Pursuing self-stabilizing systems is one of the most promising approaches in them. A distributed system is called *self-stabilizing* if it converges to a correct system state no matter which system state it starts with. Hence, a self-stabilizing system does not need to intialize. Moreover, it tolerates a finite number of transient failures, since it regards the system state right after the last failure as a given inital state and converges to a correct system state.

In 1974, Dijkstra proposed the concept of self-stabilizing, and presented self-stabilizing bidirectional ring systems for solving the mutual exclusion problem[3]. Each of his systems contains a special process, on which a different algorithm executes from ones for other processes. In this sense, his systems are *non-uniform*. In 1989, Burns and Pachl first proposed a uniform self-stabilizing mutual exclusion unidirectional ring system whose size is prime[2]. Every process executes the same algorithm in their system and therefore their system is *uniform*. Their system contains approximately $\Theta(n^2/\ln n)$ process states, where n is the number of processes. They also showed that the composite size case is unsolvable. As for bidirectional rings, recently in 1993, Huang proposed a self-stabilizing mutual exclusion system[6]. The undecidability of the composite case was already suggested by Dijkstra.

Including the three works by Dijkstra, Burns and Pachl, and Huang, many of the previous works on self-stabilizing mutual exclusion problem assume *c-daemon* (central daemon) as a scheduler. In other words, they assume that at a time at most one process is selected for execution. It is worth noting that the system does not control c-daemon: C-daemon may select a malicious schedule for the system and try to keep the system from a correct system state forever. C-daemon is said to be *fair* if it selects every process infinitely many times; otherwise, it is *unfair*.

So far, we implicitly assume systems to be deterministic; a process must obey a deterministic algorithm. In this paper, we allow a process to take a random action. In 1990, such randomized solutions were first given by Herman for synchronous unidirectional odd rings[5], and by Israeli and Jalfon for bidirectional rings with d(istributed)-daemon[7]. Recently, Beauquier and Delaët presented a randomized solution for unidirectional rings with d-daemon, but they assume that d-daemon be fair[1].

Since Israeli and Jalfon's algorithm for bidirectional rings does not require d-daemon to be fair, one might imagine that we could devise a randomized algorithm for unidirectional rings allowing unfair d-daemon, modifying Israeli and Jalfon's random walk idea. Let us observe that a naive random walk approach does no work for unidirectional rings even if c-daemon is assumed. Consider a simple random walk of tokens t_1 and t_2. It is possible that c-daemon repeatedly selects the process P_i on which t_1 resides until it leaves P_i. If t_1 never leaves P_i, c-daemon can select it everytime and the system never stabilize. If t_1 moves to the right process P_{i+1}, we let c-daemon repeatedly select the process P_j on which t_2 resides until it leaves P_j. Token t_2 must eventually leave P_j if the system stabilize. By repeating the above two processes, we can construct an infinite execution sequence like "*a dog chasing its tail*" which never stabilizes. We need more sophisticated algorithm.

This paper presents a randomized solution for unidirectional rings with c-daemon. But we do not assume the fairness of c-daemon. The system requires $4(n-1)$ process states.

The organization of this paper is as follows: Section 2 prepares important concept such as self-stabilizing system and defines our computational model. In section 3, a randomized self-stabilizing mutual exclusion system under c-daemon is proposed and its correctness is shown. We then conclude the paper giving some concluding remarks.

2 Preliminary

In this section, we introduce useful concepts and terms, and define our computation model. A *unidirectional uniform ring system* is a triple $R = (n, \delta, Q)$, where n is the number of processes in the system, δ is a state transition function (or state transition algorithm since it is defined as a form of an algorithm), and Q is a finite set of process states. The n processes $P_0, P_1, ..., P_{n-1}$ form a unidirectional ring; for any $i = 0, 1, \ldots, n-1$, there is a unidirectional communication link from P_i to P_{i+1}, and therefore, the current state of P_i can be observed by P_{i+1}. Note that indices must be calculated modulo n. (We assume that $P_0, P_1, ..., P_{n-1}$ are arranged in a clockwise manner, and that right means the clockwise direction and left the counterclockwise direction.) Let Q_i be the state set of process P_i. Note that $Q_i = Q_j$ for all i, j, but we use this notation for the simplicity of explanation. The systems is called *uniform*, since δ and Q (i.e., the algorithm a process obeys) is the same for every process.

A *configuration* of R is an n-tuple of process states; if the current state of process P_i is $q_i \in Q_i$, the configuration of system is $\gamma = (q_0, q_1, ..., q_{n-1})$. Let Γ be the set of all configurations, i.e.,

$\Gamma = Q_1 \times Q_2 \times \cdots \times Q_{n-1}$.

State transition algorithm δ is given as a set of *guarded commands*:

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

A guard is a predicate $g_j(q_i, q_{i-1})$. A command is a sequence of assignment statements $q_i := f_j(q_i, q_{i-1})$, where f_j is either a usual function or a random bit generator RandomBit, which simply returns 0 or 1 with the equal probability. Since R is a unidirectional ring, P_i can observe only the current state q_{i-1} of P_{i-1} to decide its next state. That both functions g_j and f_j take only q_i and q_{i-1} as parameters reflects this restriction. A uniform ring system is called a *randomized uniform ring system*, if RandomBit is used.

We say that P_i has *privilege* at a configuration γ if and only if $g_j(q_i, q_{i-1})$ is true for some $1 \leq j \leq m$ at P_i. P_i can execute and change its state only when it has privilege. In general, more than one privileged process exists at a time. This paper assumes that exactly one of them is arbitrarily selected and is executed. This arbitration mechanism is called *c-daemon*. Thus, our c-daemon may not select a paticular process forever, despite that it obtains privilege infinitely many times.

At configuration γ, suppose that P_i has privilege (and therfore, it has a chance to be selected by c-daemon). We assume that the execution of P_i changes the state of P_i from q_i to q. Then, by its execution, the configuration changes from γ to γ', where

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

If γ yields γ' as above, we write $\gamma \rightarrow \gamma'$. The reflexive transitive closure of relation $\rightarrow$ is denoted by $\rightarrow^*$. In order to explicitly describe the fact that the transition is caused by the execution of a process P, we may write $\xrightarrow{P}$. A *computation* or a *transition sequence* Δ starting with $\gamma_0 \in \Gamma$ is an infinite sequence of configurations $\gamma_0, \gamma_1, ...$, where $\gamma_j \rightarrow \gamma_{j+1}$ for all $j \geq 0$.

Let Λ be a (sub)set of all configurations Γ called *legitimate configurations*. A randomized uniform ring system R is a *randomized self-stabilizing mutual exclusion system for Λ* if and only if all of the following conditions hold:

- **No Deadlock:** For any configuration $\gamma \in \Gamma$, there exists at least one $\gamma' \in \Gamma$ such that $\gamma \rightarrow \gamma'$.
- **Closure:** For any legitimate configuration $\lambda \in \Lambda$ and any configuration $\gamma \in \Gamma$, $\lambda \rightarrow \gamma$ implies that $\gamma \in \Lambda$.
- **Fairness:** For any legitimate configuration $\lambda_0 \in \Lambda$, any computation $\Delta = \lambda_0, \lambda_1, ...$ starting with λ_0, and any process P_i $(0 \leq i < n)$, there exist infinite many transitions caused by P_i in Δ.
- **Mutual Exclusion:** At any legitimate configuration $\lambda \in \Lambda$, exactly one process has privilege.

- **No Livelock:** For any configuration $\gamma_0 \in \Gamma$, the computation starting with γ_0 reaches a legitimate configuration in finite transitions with probability 1, and the expected number of transitions necessary to converge to a legitimate configuration $\lambda \in \Lambda$ is finite, no matter how maliciously c-daemon behaves.[1]

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

3 A Self-Stabilization under C-daemon

3.1 A Randomized Self-Stabilizing Mutual Exclusion Algorithm

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

The idea of the proposed algorithm is based on the algorithm by Burns and Pachl[2]. A state of a process is a 3-tuple $l.t.r$. The first field of a state is called *label*, the second is called *tag*, and the last is called *random signature*. To stabilize the ring, we add a toss-a-coin feature to each process to break a symmetry of ring (with high probability) in spite of c-daemon. The random signature is a signature of a segment[2] which is randomly generated. To break a symmetry of a configuration of a ring, signatures of segments are compared and a "weaker" segment is absorbed.

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

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

In the description of the algorithm, addition and subtract are computed modulo $n - 1$.

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

Rule A: IF $A_i \wedge \alpha_i$ THEN
$l_i := l_{i-1} + 1$

[1] For a more concrete explanation, let us fix any selection algorithm (i.e., schedule) A for c-daemon and follow the computation starting with $\gamma_0 \in \Gamma$ until it reaches a legitimate configuration. Since we fix the selection algorithm, the computation (2-way) branches only when a RandomBit operator is executed. It is easy to see that the whole computation can be represented by a rooted tree $T(A)$ with root γ_0, whose leaves correspond to legitimate states. For any leaf ℓ, by $b(\ell)$ and $d(\ell)$, we denote the number of branch nodes between the root and ℓ and the depth of ℓ, respectively. Because RandomBit produces either 0 or 1 with the equal probability, the probability that the computation reaches a leaf ℓ is $2^{-b(\ell)}$. No livelock condition guarantees that for any A, $\sum_\ell 2^{-b(\ell)} = 1$, i.e., the probability that the computation reaches a leaf is 1, and $\sum_\ell d(\ell) \times 2^{-b(\ell)} < \infty$, i.e., the expected number of transitions necessary to reach a legitimate configuration is finite.

[2] By a relation on labels of consecutive processes, we define a *segment* as a sequence of processes. The formal definition is given in the next section.

$t_i := l_i - l_{i-1}$
$r_i :=$ RandomBit()

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

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

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

In the above description of the algorithm, assignment statements are performed in parallel.

A legitimate configurations is a configuration such that

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

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

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

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

$$5.0.0, 6.0.0, 0.0.1, 1.0.1, 2.0.1, \underline{2.0.0}, 3.0.0, 4.0.0,$$

is a legitimate configuration when $n = 8$. (The underlined state is the state of the privileged process.)

3.2 Correctness Proof

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

Now, we give the correctness proof of the proposed algorithm.

Lemma 1 *For any configuration $\gamma \in \Gamma$, the number of segments is at least one.*

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

Lemma 2 *The algorithm is deadlock free.*

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

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

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

  By definition of Rule A and Rule A', the head process has privilege by Rule A or Rule A'; a contradiction.
- Otherwise:
  Because the number of segments is more than one and every head process has label 0, a process whose label is 0 is the head process. Since no processes has privilege by Rule B, every process in a segment has the same tag and random signature. Because $t_i \geq t_{i-1}$ is true for all head processes, every process (including a non-head process) has the same tag. Now we consider C_i. By assumption, $\neg C_i = A_i \vee (l_i = l_{i-1} + 1) \vee (r_i > r_{i-1}) \equiv r_i > r_{i-1}$ is true at every head process. But this is a contradiction because all processes in a segment s_j have the same random signature for each segment s_j. (See Rule B.) □

Lemma 3 *Closure property holds.*

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

Lemma 4 *Fairness property holds.*

(Proof) Let λ be any legitimate configuration. By the definition of legitimate configurations, the number of processes which have privilege is one and by lemma 3, privilege moves to a right process by an execution. Therefore, privilege circulates the ring. □

Lemma 5 *Mutual exclusion is guaranteed.*

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

Above lemmas prove four properties of a self-stabilizing system. We prove that the expected steps that the system stabilizes is finite.

Lemma 6 *Let $\gamma_0 \in \Gamma$ be any configuration and $\Delta = \gamma_0, \gamma_1, \gamma_2, ...$ be any infinite computation starting from γ_0. Then, there exists $0 \leq I < \infty$ such that a transition $\gamma_I \rightarrow \gamma_{I+1}$ is an application of Rule A, A', or C.*

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

Because the algorithm is deadlock free (by lemma 2), there exists a process that has privilege by one of Rule A, A', and C. Therefore, one of these rules are applied. □

Lemma 7 *The configuration reaches a legitimate configuration within a finite steps if the number of segments at an initial configuration γ is one.*

(Proof) Let $\gamma \in \Gamma$ be any configuration such that $\sharp(\gamma) = 1$. It is easy to see that the number of segments is non-increasing by the definition of the algorithm. Thus, we have $\sharp(\gamma') = 1$ for any γ' such that $\gamma \rightarrow^* \gamma'$. By lemma 6, the head process executes one of Rule A, A', and C. By an execution of any of these rules, the head process changes and the label of the (new) head process of the segment increases by one. Therefore, within a finite steps from γ, the configuration becomes a configuration such that the label of the head process is 0. Let this configuration be γ_0 and let processes be $P_0, P_1, ..., P_{n-1}$ in clockwise order in the ring and assume P_0 is the head process at γ_0.

By lemma 6, P_0 executes one of Rule A, A', and C and its tag and random signature become the same as P_{n-1}'s. Note that the tag is zero. Let the configuration after P_0 executes a rule be γ_1. Similarly, P_1 executes a rule and its tag and signature become as the same as P_0's. Repeating this argument, it is easy to see that the configuration becomes a legitimate configuration. □

Lemma 8 *For any configuration $\gamma \in \Gamma$ such that $\sharp(\gamma) = n$, the number of segments becomes $n - 1$ by an execution of a rule.*

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

Lemma 9 *Let γ_0 be any configuration such that $\sharp(\gamma_0) > 1$ and s be any segment at γ_0. Then, there exist no computations starting γ_0 such that the number of applications of Rule B by processes in s is infinite if the set of processes consisting of s does not change forever.*

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

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

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

Lemma 10 *Let γ_0 be any configuration such that $\sharp(\gamma_0) > 1$. Assume that there exists a computation starting from γ_0 such that the number of segments does not change. Then, there exist no segments whose head process does not change.*

(Proof) Assume that there exists such a segment s. By lemma 9, there exist no computations such that only Rule B is applied. Thus, Rule A, A', or C is applied within a finite steps, which implies that there exists a segment s' whose member changes infinitely many times during an infinite computation. Let P be the head process of s. Then, P also becomes a member of s' infinitely often because a segment moves to only one direction on a ring; this is a contradiction. □

The next lemma shows that any schedule which tries to keep the number of segments leads to a configuration in which all segments are well formed.

Lemma 11 *Let $\gamma_0 \in \Gamma$ be any configuration such that $2 \leq \sharp(\gamma_0) \leq n-1$. Assume that there exists an infinite computation $\Delta = \gamma_0, \gamma_1, \gamma_2, ...$ such that $\sharp(\gamma_0) = \sharp(\gamma_j)$ for all $j \geq 0$. Then, there exists I such that every segment at γ_i is well formed for all $i \geq I$.*

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

Let l be the label of the tail process of s_1 at γ_0. Then, by lemma 10, the head process of s_2 executes a rule and becomes the tail process of s_1 within a finite steps and its label becomes $l+1$. Repeating this discussion, it is easy to see that a configuration reaches a configuration such that the label of the tail process of s_1 becomes 0 within a finite steps. Let this configuration be γ_1 and $P_1 = \mathsf{Tail}(s_1)$.

Consider a configuration μ_2 such that P_1 becomes the head process of s_1 for the first time after γ_1. (It is easy to see that such a configuration exists by lemma 10.) By the definition of rules, P_1 never changes its tag and random signature between γ_1 and γ_2. Thus, the right processes of P_1 in the same segment inherit P_1's tag and random signature. Therefore, the segment s_1 is well formed at γ_2.

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

The range of labels and definition of gap is the same as the ones in [2]. Lin and Simon showed the next lemma in [9] for the algorithm in [2]. Thus, the next lemma also holds for our algorithm.

Lemma 12 *Let $\gamma \in \Gamma$ be any configuration and $s_1, ..., s_L$ be segments at γ and g_i be a gap of s_i, where $L = \sharp(\gamma)$. Then, $\sum_{1 \leq i \leq L} g_i = L - 1 \bmod n - 1$*

(Proof) Proof can be found in [9]. □

Lemma 13 *Let γ_0 be any configuration such that $2 \leq \sharp(\gamma_0) \leq n - 1$ and every segment is well formed at γ_0 and $\Delta = \gamma_0, \gamma_1, ...$ be any infinite computation. Then, there exist a configuration γ_k and a head process, say P, of a segment at γ_k such that P does not have privilege by Rule A nor Rule A' at γ_k,*

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

Because each segment is well formed and $t_i < t_{i-1}$ does not hold for all head processes, there exists j such that $t_j \geq t_{j-1}$. Thus, $t_j = 0$ holds. Now consider its right segment $s_{j'}$, where $j' = j + 1$. To $t_{j'} = 0 \vee t_{j'} < t_{j'-1}$ be true, $t_{j'} = 0$ holds since $t_{j'-1} = t_j = 0$ and tags are non-negative. Repeating this discussion, we have $t_i = 0$ for all i, which contradicts lemma 12. □

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

Lemma 14 *Let γ_0 be any initial configuration such that $2 \leq \sharp(\gamma_0) \leq n - 1$. Then, the expected steps that the number of segments decreases is finite.*

(Proof) Assume that $\Delta = \gamma_0, \gamma_1, \gamma_2, ...$ be any infinite computation such that the number of segments never decreases. By lemma 11, there exists I such that all segments are well formed at γ_i for any $i \geq I$. We consider configurations after γ_I.

Since every segments are well formed and the number of segments is kept, there is no processes that has privilege by Rule B at any configuration γ_i ($i \geq I$). By lemma 13, there exists a process P and a configuration γ_J ($J \geq I$) such that P is the head process of a segment s_j and does not have privilege by Rule A nor Rule A'. Since every head process has privilege by the assumption, $P = \mathsf{Head}(s_j)$ has privilege by Rule C.

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

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

$n-2$ by Rule A. Note that new random signature is not cleared by Rule B since its label is 0 and Rule B does not propagate tag and random signature over a prcess with label 0.

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

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

We have the theorem from above lemmas.

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

Note that the algorithm does not work under a d-daemon.[3] For example, consider a configuration such that a state of every process is 0.0.0 and a schedule such that all processes are executes at every step. Then, the number of segments never decreases.

3.3 Reduction of the Number of States

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

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

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

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

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

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

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

[3] A *d-daemon* is a scheduler which chooses any set of processes which have a true guard and let them execute in parallel.

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

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

Rule RB: If B_i then
$t_i := t_{i-1}$
$r_i := r_{i-1}$

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

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

Lemma 15 *The algorithm satisfies the (1) closure property, (2) fairness property, and (3) mutual exclusion.*

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

Lemma 16 *The algorithm satisfies no deadlock property.*

(Proof) The proof is the identical to the proof of lemma 2 except $t_i = l_i - l_{i-1}$ is replaced by $t_i = f(l_i - l_{i-1})$ and $t_i \neq l_i - l_{i-1}$ is replaced by $t_i \neq f(l_i - l_{i-1})$. □

Lemma 17 *The algorithm satisfies no livelock property.*

(Proof) Lemmas 6, 7, 8, 11 hold by the same proofs. Lemma 13 is shown by replacing $t_i = l_i - l_{i-1}$ by $t_i = f(l_i - l_{i-1})$ and $t_i \neq l_i - l_{i-1}$ by $t_i \neq f(l_i - l_{i-1})$ in the proof. Note that $t_i = 0$ does not hold at all head processes because $t_i = 0$ implies $l_i = l_{i-1}$ and it contradicts lemma 12. Lemma 14 is also shown by replacing $t_i = l_i - l_{i-1}$ by $t_i = f(l_i - l_{i-1})$ and $t_i \neq l_i - l_{i-1}$ by $t_i \neq f(l_i - l_{i-1})$ in the proof. □

Now we have the following theorem.

Theorem 2 *For each $n \geq 1$, there exists a randomized self-stabilizing mutual exclusion system which requires $4(n-1)$ states per process for a ring of size n under c-daemon.* □

4 Discussion

In this paper, we have proposed uniform randomized self-stabilizing mutual exclusion systems on unidirectional rings. We assume c-daemon, but it can be unfair. The number of states per process is $4(n-1)$. As mentioned in Section 1, Beauquier and Delaët proposed a randomized self-stabilizing mutual exclusion algorithm for uniform rings[1]. They assume d-daemon, which is less powerful than c-daemon. But their d-daemon must be fair.

As mentioned in Section 3, our systems do not work correctly under unfair d-daemon. Finding a system for unidirectional rings working under unfair d-daemon is left as an interesting future work.

In this paper, we assume that every random bit generator correctly produces random bits. In real situations however, preparing ideal random bit generators may be difficult, and some of them may be faulty. Discussing the minimum number of correct random bit generators necessary for a randomized algorithm to exist seems to be interesting.

References

[1] Joffroy Beauquier and Sylvie Dealöet. Probablistic self-stabilizing mutual exclusion in uniform rings. In *Proceedings of 13th Annual ACM Symposium on Principles of Distributed Computing*, page 378, 1994.

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

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

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

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

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

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

[8] J.L.W. Kessels. An exercise in proving self-stabilization with a variant function. *Information Processing Letters*, 29(1):39–42, September 1988.

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

[10] Prabhakar Raghavan. Lecture notes on randomized algorithms. Technical report, IBM Research Division, December 1989. Research Report RC 15340 (#68237) 1/9/90.

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

Paper Number 15

Optimum Probabilistic Self-Stabilization on Uniform Rings

Joffroy Beauquier, Stephane Cordier, and Sylvie Delaet

Optimum probabilistic self-stabilization on uniform rings *

Joffroy Beauquier
LRI CNRS
Université Paris sud
Orsay, F 91405
e-mail : jb@lri.fr

Stéphane Cordier
LAN
Université Paris 6
Paris, F 75232
e-mail : cordier@ann.jussieu.fr

Sylvie Delaët
LRI CNRS
Université Paris sud
Orsay, F 91405
e-mail : delaet@lri.fr

Abstract

We propose a probabilistic self-stabilizing mutual exclusion protocol on unidirectional uniform rings of size $n > 2$. Our solution uses only δ states per process where δ is the smallest integer which does not divide the ring size n ($\delta = 2$ if n is odd). We prove that this number is optimum on unidirectional uniform rings. In this paper, we give the complete detailed proofs of the optimality of the state number per process and of the proposed protocol.

1 Introduction

1.1 The problem

A system is self-stabilizing if it can correctly start from *any possible* configuration. To prove that a system self-stabilizes to a specification P, we must find a subset $\mathcal{L}$ of its configurations, called legitimate configurations, and show that P is satisfied by every execution starting from $C \in \mathcal{L}$ (**correction**) and that, starting from any configuration, the system eventually reaches a configuration of $\mathcal{L}$ (**convergence**).

In the same way than the pionnering Dijsktra's paper [Dij74], this paper deals with the self-stabilizing mutual exclusion on rings of size n ($n>2$). The protocol is a set of guarded rules. A process is privileged if it can apply a rule. An execution verifies the mutual exclusion specification if there is exactly one privilege in each configuration of the execution and if each process is privileged infinitely many times in the execution.

The parameters of the self-stabilizing mutual exclusion problem on ring, are the assumptions of the ring (uniform or not, unidirectional, bidirectional, of any size, of odd size only, etc.), the number of states per process, and the characteristic of the protocol (deterministic or probabilistic).

We are interested here in uniform unidirectional rings of any size n. We first prove that the minimal number of states per proces is δ (the smallest integer which does not divide the ring size n).

We propose a probabilistic solution that uses only δ states per process. This solution completes some previous work mentioned in the next section, and is optimal for the number of states per process. We give here the complete proofs of the protocol and of its optimality.

1.2 Previous Work

Dijsktra [Dij74] obtained some deterministic solutions in a non uniform ring. His best solutions use only 3 states per process in a bidirectional ring, and n states per process in a unidirectional ring. For uniform rings of size n, deterministic solutions are not possible when n is composite. Burns and Pachl [BP89] first proposed a deterministic protocol that uses approximately $(n^2/\ln n)$ states for each process in a

*This work was supported by the MEP project.

unidirectional uniform ring of prime size. Huang [Hua93] obtained a $3n$-state deterministic solution on a bidirectional ring. Probabilistic uniform solutions have been given by Herman [Her90] for synchronous unidirectional odd rings and by Israeli and Jalfon [IJ90] for bidirectional rings with *distributed daemon.* This last solution is based on random walk, then the bidirectional hypothesis is crucial. We propose here a probabilistic solution on unidirectional uniform ring which develops the ideas briefly presented in [BD94]. Our solution uses only δ states per process where δ is the smallest integer which does not divide the ring size. We prove that this number of states per process is optimum on unidirectional uniform rings.

1.3 Organization of the paper

In the section 2, we define the model that we use. This model is closed to Dijsktra's one, except that it allows to use some random bits for probabilistic protocols. We formally define the distributed daemon.

In the section 3, we prove that it is not possible to find a self-stabilizing mutual exclusion protocol for unidirectional uniform ring which uses less than δ states per process. This impossibility proof is based on combinatorial approach of possible configurations. To read this proof, the subsection 2.2.2 is not neccessary, the subsection 2.2.1 suffices.

In section 4, we give a optimum protocol and its formal proof. One can read the protocol and its proof independently of section 3 but the entire section 2 is useful.

2 The model

2.1 Informal description

Each process can read the state of its preceding neighbor on the ring. The processes are uniform and use no identities. But, for the proofs, we consider that each process is denoted by an integer in the range $[O, n-1]$. The preceding neighbor of a process i is the process $i-1$. All the computations on process names are made modulo n (in particular the preceding neighbor of the process 0 is the process $n-1$).

A protocol is a set of guarded rules. Since the ring is unidirectional, the guard of the rules depends only on the states of the process and its preceding neighbor. Each rule can change the state of the process which applies it (but it does not change the state of the neighbor). In the same way as in the classical model, when a process can apply a rule, we say that this process is privileged. Hence the problem is reduced to the case where the only way to change is state its to be in the critical section.

We use the classical model of *distributed daemon.* The distributed daemon is a scheduler which chooses a subset of the privileged processes. Each process of this subset can apply its rule.

We say that the protocol is probabilistic if there is a random bit generator on each process, and the rule depends on the generated random bit of the process which applies it.

The execution of a protocol is a sequence of steps. A step in a sequence consists of the neighbor's state reading, the choice of a subset of privileged processes by the distributed daemon, and the application of the rules by the processes which have been chosen. In a first time all the guard are evaluated, in a second time all rules of chosen processes are executed according to the values of the guard.

A legitimate configuration is a configuration with exactly one privilege.

2.2 Definitions and notations

In this section, we give the formal definitions and notations which are used along the paper. The section 2.2.1 suffices for the reading of optimality proof (section 3). But this whole section is necessary to understand precisely the proof of the protocol given in the section 4.

2.2.1 The expression of the protocol

Definition 1 We denote by S the set of all possible states for a process.

Definition 2 Let g be a boolean function on $S \times S$ and f a function from $S \times S$ to S. A rule $g \rightarrow f$ guarded by g is the restriction of f to $\{(s, s') \in S \times S \text{ such that } g(s, s')\}$.

Definition 3 We denote by R the set of all the guarded rules.

For defining a self-stabilizing mutual exclusion protocol, we must give the sets S and R.

Definition 4 $Priv$ is the set of pairs (state of a process, state of its neighbor) of $S \times S$ such that at least one guard is true.

$$Priv = \{(s, s') \in S \times S \ / \ \exists g \rightarrow f \in R \text{ and } g(s, s')\}.$$

Definition 5 A configuration c is a vector of n states.

$$c = (s_0, \cdots, s_{n-1}).$$

The number of privileges (or true guards) in a configuration $c = (s_0, \cdots, s_{n-1})$ is the number of pairs (s_{i-1}, s_i), $0 \leq i \leq n-1$ in the configuration c such that $(s_{i-1}, s_i) \in Priv$.

Definition 6 $G(c)$ is the set of integer i, $0 \leq i \leq n-1$ such that in the configuration c, (s_{i-1}, s_i) is in $Priv$.

$$G(c) = \{i \in [0, n-1] \ / \ \exists g \rightarrow f \in R \text{ and } g(s_{i-1}, s_i)\}.$$

$|G(c)|$ is the number of privileges in configuration c.

Definition 7 Leg is the set of legitimate configurations.

$$Leg = \{c \in S^n / |G(c)| = 1\}.$$

Definition 8 D_{min} is a function on unlegitimate configurations which gives the minimum distance between two privileges.

$$\begin{array}{rcl} D_{min} : S^n \setminus Leg & \longrightarrow & [1, \lfloor \frac{n}{2} \rfloor], \\ c & \longmapsto & \min\limits_{i,j \in G(c), i \neq j} i - j. \end{array}$$

2.2.2 The execution of the protocol

We modelize the behavior of the protocol by a sequence of steps leading from a configuration to another with respect to the applied rules.

We consider that a step is the reading of preceding neighbor state by all the processes, a choice by the distributed daemon and the application of rule by the processes which have been chosen. We denote by c^k the configuration after k steps (see figure 1). At each step k, the distributed daemon chooses a subset A^k of $G(c^k)$ to compute the configuration c^{k+1}.

The choice function is given before the execution of the protocol and depends only on the set of possible configurations.

It is convenient to consider the distributed daemon as an adversary. We assume that there exists an integer D, such that at each step the adversary knows at most the D preceding configurations of the current execution and remembers at most the D preceding choices it made. From this informations it makes deterministically the next choice. We modelize this behavior by a choice function. After the choice of A^k, each process i of A^k applies a rule (then uses its random bit x_i^k if the protocol is probabilistic).

Definition 9 An adversary is a set of function α^k $(k \leq D)$ such that :

$$\begin{array}{rcl} \alpha^k (k < D) : S^{nk+1} \times [\mathcal{P}art([0, n-1])]^k & \longrightarrow & \mathcal{P}art([0, n-1]) \\ ((c^0, \cdots, c^k), (A^0, \cdots, A^{k-1})) & \longmapsto & A^k / A^k \subset G(c^k). \end{array}$$

$$\begin{array}{rcl} \alpha^D : S^{nD} \times [\mathcal{P}art([0, n-1])]^{D-1} & \longrightarrow & \mathcal{P}art([0, n-1]) \\ ((c^{k-D}, \cdots, c^{k-1}), (A^{k-D-1}, \cdots, A^{k-1})) & \longmapsto & A^k / A^k \subset G(c^k). \end{array}$$

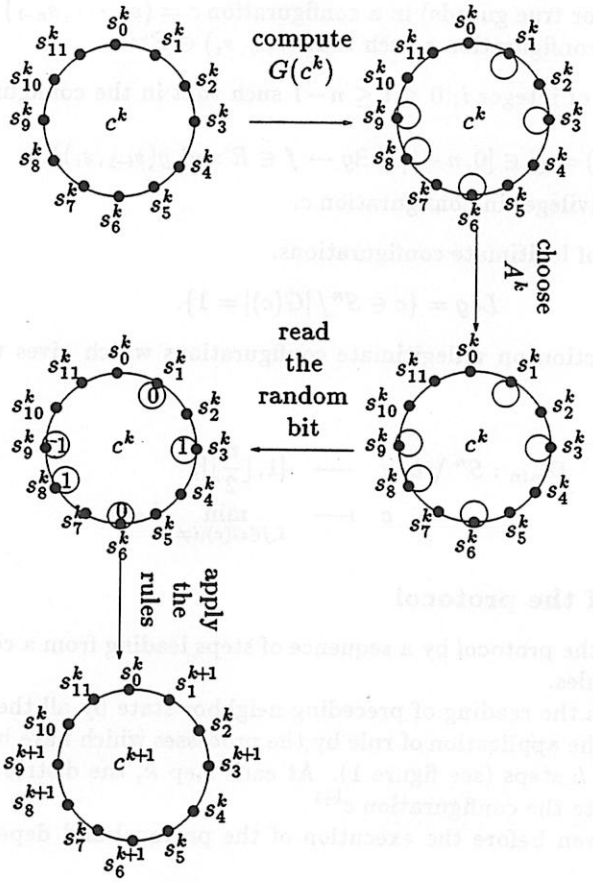

Figure 1: The step $k+1$

For a given configuration c^0 and for a given choice function, the set of all possible executions can be seen as a infinite tree (see figure 2), where an execution is the projection on the set of configurations of an infinite branch. If the protocol is not probabilistic, the tree is just a branch. In this case, there is only one possible execution for each pair of initial configuration and choice function.

The bounded memory hypothesis can be viewed on the tree as a property of its subtrees. In the tree if two inner distinct paths of lenght D are identical then the subtrees rooted at the extremity of this two path are identical.

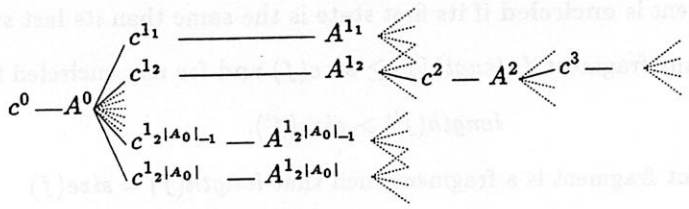

Figure 2: infinite execution tree

We assume that the adversary is fair. It means that if a process is infinitely many times privileged in a execution, it must be chosen infinitely many times by the adversary.

Definition 10 An adversary is **fair** if its choice function is such that

$$\text{for any infinite execution } c_0, A_0, \cdots, c_k, A_k, \cdots,$$
$$\exists i \in [0, n-1] \text{ and } \exists (k_l)_{l \in N},\ \forall l \in N\ i \in G(c^{k_l}) \Rightarrow \exists k > k_0, i \in A^k.$$

The property of fairness can be viewed on the tree in the following way : In each infinite branch such that i is infinitely privileged, i is infinitely often selected in this branch.

We consider that there is a random bit generator on each process. We denote by x_i the random bit for the process i and r_i $(0 < r_i < 1)$ the probability that $x_i = 0$. If the protocol is probabilistic, the rules depend on the randoms bits. Then, for a current configuration and a choice of a set of processes by the distributed daemon, the next configuration depends on the values of random bits. We just can determinate the set of reachable configurations. Each of them has a probability to become the current configuration equal to the probability of the random bits which can generated it. One can consider that each edge in the tree is labelled by the probability to use it. The probability of an execution is the produce of probabilities of its configurations.

3 Optimality of δ

In this section, we prove that on a unidirectional uniform ring of size n, it is not possible to find a self-stabilizing mutual exclusion protocol that uses less than δ states per process (where δ is the smallest integer which does not divide n).

This result comes from two important properties of self-stabilizing mutual exclusion protocols on ring : there exists at least one configuration with exactly one privilege (correction) and there exists no configuration without privilege (no deadlock). The proof is a combinatorial approach of the number of states per process and the possible configurations.

We prove that if the number of states per process is less than δ (thus, divides n), the existence of a configuration with exactly one privilege implies the existence of configurations without privilege. In other words, if the number of states per process is too small, the correction implies that deadlock is possible.

We begin this section by some technical definitions (section 3.1), then we give (section 3.2) the optimality theorem (theorem 1) which is proved by a sequence of lemmas.

3.1 Definitions

Definition 11 A **fragment** of a configuration $c = (s_0, \cdots, s_{n-1})$ is a sequence of consecutive process states.

Definition 12 The **length of a fragment** is the number of states in the fragment.

Definition 13 The **size of a fragment** is the number of distinct states in the fragment.

Definition 14 A fragment is **encircled** if its first state is the same than its last state.

We remark that for any fragment f, $length(f) \geq size(f)$ and for any encircled fragment f',

$$length(f') > size(f').$$

Definition 15 A **perfect** fragment is a fragment such that $length(f) = size(f)$.

Definition 16 A **perfect encircled** fragment is an encircled fragment f such that

$$length(f) = size(f) + 1.$$

For example, if $(1,2,2,1,0,2,3)$ is a configuration : (2, 2, 1, 0, 2) is a encircled fragment of size 3, length 5. (0, 2, 3, 1) is a perfect fragment of size and length 4 and (1, 0, 2, 3, 1) is a perfect encircled fragment of size 4 and length 5.

Definition 17 A fragment f is **included** in a fragment f' of a configuration c if :

- f is a fragment of c,
- f is a segment [1] of f'.

Proposition 1 Let f be a fragment of a configuration c of size t and length l. If $t < l$ then there exists a perfect encircled fragment f' included in f.

Proof We consider the longest perfect fragment $f_p = (s_j, \cdots, s_{j+l_p-1})$ included in f. We denote l_p the length (and size) of f_p. The length of f being larger than l_p, s_{j-1} or s_{j+l_p} is equal to a state s_k of f_p thus the fragment, $s_{j-1}, \cdots, s_k$ or $s_k, \cdots, s_{j+l_p}$ is a perfect encircled fragment included in f.

□ **(proposition 1)**

A fragment of length l $(s_i, \cdots, s_{i+l-1})$ is privileged if there exists a couple (s_q, s_{q+1}) $(i \leq q \leq i+l-2)$ in $Priv$.

3.2 Proof of optimality

Theorem 1 The minimum number of states for a self-stabilizing mutual exclusion protocol on unidirectional ring of size n is δ (the smallest integer which does not divide n).

Proof of theorem Let S be a set of possible states. Let R be a set of guarded rules. We consider the set $Priv$ (see definition 4).

Lemma 1 If there exists an encircled fragment f of length l, $l \leq \delta$, which is not privileged in a configuration c, then there exists a configuration c' without privilege.

Proof We denote by $c = (s_0, \cdots, s_{n-1})$ the configuration. Let $f = (s_i, \cdots, s_{i+l-1})$ be an encircled fragment of c of length l equal or less than δ which is not privileged. As f is not privileged, for all q $(i \leq q \leq i+l-2)$, (s_q, s_{q+1}) is not in $Priv$.

From f, we build the configuration $c' = (s'_0, \cdots, s'_{n-1})$ with (see figure 3) :

$$s'_{kl+j} = s_{i+j},\ 0 \leq j \leq l-1 \text{ and } 0 \leq k \leq \frac{n}{l-1} - 1$$

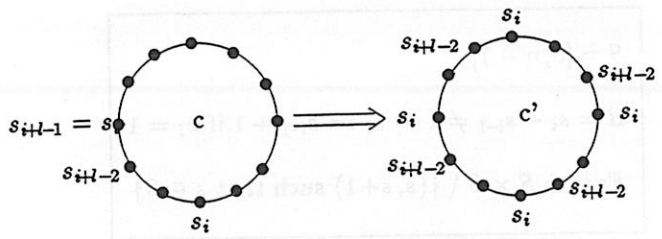

Figure 3: Construction of c'

$\frac{n}{l-1}$ is an integer because $l \leq \delta$ then $l-1$ divides n. It is easy to verify that there is no privilege in the configuration c'.

□ **(lemma 1)**

Lemma 2 If there exists a configuration c with a unique privilege and with less than δ different states, then there exists a not privileged encircled fragment f of size t less than δ.

Proof We denote by c the configuration with a unique privilege and with less than δ different states. Since the number of different states is less than δ, the size of all fragments is less than δ.

Since δ is less than n, a state s of S occurs twice in c. Note k and j the two positions of s in c. There is a unique privilege in c, one of the two encircled fragments $(s_k, \cdots, s_j)$ and $(s_j, \cdots, s_k)$ is privileged and the other is not privileged.

□ **(lemma 2)**

A self-stabilizing mutual exclusion protocol must verify the two assertions :

1. There is no configuration without privilege (no deadlock).
2. There exists at least one configuration with exactly one privilege (correction).

If there is less than δ states per process ($|S| < \delta$), and a configuration with exactly one privilege (assertion 2), lemma 2 ensures that there exists in this configuration a encircled fragment f of size t, $t < \delta$. The proposition 1 ensures that a perfect encircled fragment f' of c is included in f. Since f is not privileged, f' is not privileged.

In this case, lemma 1 ensures that there exists a configuration without privilege.

We conclude that the assertions 1 and 2 cannot be verified simultaneously if the number of states per process is less than δ.

□ **(theorem 1)**

4 A optimum probabilistic protocol

We present here a probabilistic self-stabilizing algorithm for the mutual exclusion specification on a uniform unidirectional oriented n-ring. It uses only δ states per process where δ is the smallest integer which does not divide n ($\delta = 2$ if n is odd).

In our solution, the state of a process is an integer in the range $[O, \delta-1]$. All the computations on states are made modulo δ (for example if the state s is equal to $\delta-1$, the state $s+3$ is equal to 2).

There is a unique guarded rule. The protocol is probabilistic, the rule uses the values of the random bits x_i of each process i :

$$s_i - s_{i-1} \neq 1 \rightarrow s_i := s_{i-1} + 1 \text{ if } x_i = 1,$$

with $Pr(x_i = 0) = r_i$ and $Pr(x_i = 1) = 1 - r_i$ $(0 < r_i < 1)$.

[1] A segment is a subsequence of consecutives states

$S = [0, n-1]$

$R = s_i - s_{i-1} \neq 1 \rightarrow s_i := s_{i-1} + 1$ if $x_i = 1$

$Priv = S \times S \setminus \{(s, s+1)$ such that $s \in S\}$

Figure 4: Definition of our protocol

4.1 Informal discussion

Intuitively, the protocol is very simple. The privileges always circulate around the ring in the same way. If there is a unique privilege, it circulates around the ring and no other privilege is created. If there is more than one privilege, they circulate in the same direction but at different speeds (depending on the distributed daemon choice and on the random bits). When a privilege catch up to another, one of them disappears.

It is possible that the distributed daemon can prevent the overtaking of two privileges (by slowing down one or the other) but the probability that it does it to the infinity is 0.

This probabilistic protocol is very different from the solution of Israeli and Jalfon [IJ90], where the bidirectional hypothesis is crucial : the number of privileges decreases when two privileges collide, but they collide because they change of direction according to random bits.

Our hypothesis is also weaker than the hypothesis of Herman's protocol[Her90], that assumes that the scheduler always chooses all processes in a synchronous way.

4.2 Proof of self-stabilization

Theorem 2 The protocol is a probabilistic self-stabilizing mutual exclusion protocol for unidirectional uniform rings of any size n, $(n > 2)$. It uses the minimal number of states per process.

Proof of theorem

To prove the protocol, we must first show that starting from any legitimate configuration, only legitimate configurations are reachable, that each process is infinitely many times privileged with probability 1 (*correction*), and that from any configuration, a legitimate configuration is reachable with probability 1(*convergence*).

We divide the proof in three phases. In the first one (section 4.2.1), we present some useful properties about the behavior of the protocol. In the second one (section 4.2.2), we give the entire and detailed proof of correction. In the last one (section 4.2.3), we show that the protocol converges with probability 1 to a legitimate configuration. Then theorem 2 is completely proved.

4.2.1 Study of the protocol

After some definitions and notations, we prove in this section two important propositions illustrating the behavior of the protocol. The first one shows that from any given configuration, there is a positive probability to eventualy reach any configuration in which a single privilege has moved. The second one proves that the number of privileges never increases.

Our solution is probabilistic. Then from a configuration c and an adversary choice A of privileged processes, several configurations are reachable.

Let B be a set of processes. We define $\mathcal{T}(c, B)$ as the configuration obtained from c by changing the state of processes in B according to the preceding process states (like in definition 18).

Definition 18

$$\begin{aligned} \mathcal{T} : S^n \times \mathcal{P}art([0, n-1]) &\longrightarrow S^n \\ (c, B) &\longmapsto \mathcal{T}(c, B) \end{aligned}$$

$$c = (s_0, \cdots, s_{n-1}) \Rightarrow \mathcal{T}(c, B) = (s'_0, \cdots, s'_{n-1})$$

$$\text{where } s'_i = \begin{cases} s_{i-1} + 1 & \text{if } i \in B, \\ s_i & \text{otherwise.} \end{cases}$$

In our protocol, to pass from the step k to the step $k+1$, we define c^{k+1} by $c^{k+1} = \mathcal{T}(c^k, B)$ with $B = A^k \cap \{i \in [0, n-1] \,/ x_i^k = 1\}$ and with $Pr(x_i = 0) = r_i$ and $Pr(x_i = 1) = 1 - r_i$ $(0 < r_i < 1)$.

Definition 19 Let $c^0, A^0, \cdots, c^k, A^k$ be the beginning of an execution, Let B be a set of processes.

$$Pr\left(c^{k+1} = \mathcal{T}(c^k, B)\right) = \begin{cases} 0 & \text{if } B \not\subset A^k, \\ \prod_{i \in A^k \setminus B} r_i \times \prod_{i \in B} (1 - r_i) & \text{otherwise.} \end{cases}$$

It is useful to observe the behavior of the protocol step by step. In the next proposition, we study the special case where only one privileged process changes its state.

Proposition 2 Let c^k be a configuration and i be a process, then for $\mathcal{T}(c^k, \{i\})$, one of the two assertions is true :

1. $G\left(\mathcal{T}(c^k, \{i\})\right) = G(c^k) \cup \{i+1\} \setminus \{i\}$, the number of privileges is the same or decreases of 1 from the step k to the step $k+1$.
2. $G\left(\mathcal{T}(c^k, \{i\})\right) = G(c^k) \setminus \{i, i+1\}$, the number of privileges decreases of 2 from the step k to the step $k+1$.

Proof

Let $c^k = (s_0^k, \cdots, s_{n-1}^k)$ and $\mathcal{T}(c^k, \{i\}) = (s'_0, \cdots, s'_{n-1})$. If $x_i^k = 1$ then $s'_i = s_{i-1}^k + 1$. Now two case are possible :

1. $s_{i+1}^k - s_{i-1}^k \neq 2$, then, we have

$$\begin{aligned} s'_{i+1} - s'_i &= s_{i+1}^k - (s_{i-1}^k + 1) \\ &= s_{i+1}^k - s_{i-1}^k - 1 \\ &\neq 1 \end{aligned} \qquad \begin{aligned} s'_i - s'_{i-1} &= s_{i-1}^k + 1 - s_{i-1}^k \\ &= 1 \end{aligned}$$

$$\Rightarrow i+1 \in G\left(\mathcal{T}(c^k, \{i\})\right) \qquad \Rightarrow i \notin G\left(\mathcal{T}(c^k, \{i\})\right)$$

$$G\left(\mathcal{T}(c^k, \{i\})\right) = G(c^k) \cup \{i+1\} \setminus \{i\}.$$

In this case the privilege on i moves to $i+1$, if $i+1$ was privileged then the number of privileges decreases of 1 else the number of privilege is unchanged.

2. $s_{i+1}^k - s_{i-1}^k = 2$, then, we have

$$\begin{aligned} s'_{i+1} - s'_i &= s_{i+1}^k - (s_{i-1}^k + 1) \\ &= s_{i+1}^k - s_{i-1}^k - 1 \\ &= 1 \end{aligned} \qquad \begin{aligned} s'_i - s'_{i-1} &= s_{i-1}^k + 1 - s_{i-1}^k \\ &= 1 \end{aligned}$$

$$\Rightarrow i+1 \notin G\left(\mathcal{T}(c^k, \{i\})\right) \qquad \Rightarrow i \notin G\left(\mathcal{T}(c^k, \{i\})\right)$$

$$G\left(\mathcal{T}(c^k, \{i\})\right) = G(c^k) \setminus \{i, i+1\}.$$

In this case the number of privileges decreases of 2.

□ **(proposition 2)**

Proposition 3 During an execution, the number of privileges never increases.

Proof We consider two successive configurations c^k and c^{k+1} in an execution.

Let $c^k = (s_0^k, \cdots, s_{n-1}^k)$ and $c^{k+1} = (s_0^{k+1}, \cdots, s_{n-1}^{k+1})$. We denote by B^k the subset of $G(c^k)$ such that $c^{k+1} = \mathcal{T}(c^k, B^k)$.

If $|G(c^k)| = n$, $|G(c^{k+1})| > |G(c^k)|$ is impossible.

If $|G(c^k)| < n$, $|G(c^{k+1})| > |G(c^k)|$ is possible only if there exists a process i privileged in the configuration c^{k+1} which was not privileged in the configuration c^k.

We consider i such that $i \notin G(c^k)$ and $i \in G(c^{k+1})$. We first show that $i-1 \in B^k$:

$$\begin{aligned} s_i^{k+1} - s_{i-1}^{k+1} &\neq s_i^k - s_{i-1}^k \\ s_{i-1}^{k+1} &\neq s_{i-1}^k \\ &\Rightarrow i-1 \in B^k \end{aligned}$$

Let us denote by p_i the smallest positive integer such that $i-p_i \notin B^k$. p_i exists, since $i-1 \in B^k$ and $i-N = i \notin B^k$.

We consider the process $i-p_i+1$. It is in B^k (then in $G(c^k)$) since $p_i-1 < p_i$, $i-p_i+1$ is not in $G(c^{k+1})$ (since $s_{i-p_i+1}^{k+1} - s_{i-p_i}^{k+1} = s_{i-p_i}^k + 1 + s_{i-p_i}^k = 1$).

We have proved that for each process which has obtained a new privilege from c^k to c^{k+1}, at least one other process has lost a privilege from c^k to c^{k+1}. Thus, at each step, the number of privileges cannot increase.

□ **(proposition 3)**

Proposition 4 There exists no configuration without privilege.

Proof

It is obvious that for all configurations $c = (s_0, \cdots, s_{n-1})$., we have $\sum_{i=0}^{n-1}(s_i - s_{i-1}) = 0$. Recall that the computations on states are made modulo δ. Assume that all the processes in c are privileged (i.e. for all i in $[0, n-1]$, $s_i - s_{i-1} = 1$). Then $\sum_{i=0}^{n-1}(s_i - s_{i-1}) = n \bmod \delta \neq 0$. This contradiction ends the proof.

□ **(proposition 4)**

4.2.2 Proof of correction

In the proposition 5, we proove that starting from any legitimate configuration, all configuration are legitimate and each process is infinitely many times privileged.

Proposition 5 The protocol described in figure 4 verifies the property of correction defined by :
for all infinite executions $c^0, A^0, \cdots, c^k, A^k, \cdots$,

$$|G(c^0)| = 1 \Rightarrow \left\{ \begin{array}{ll} & \forall k \in N,\ |G(c^k)| = 1, \\ \text{and} & \\ & \forall i \in [0, n-1]\ Pr\left(\exists (k_i^l)_{l \in N} / \forall l \in N\ G(c^{k_i^l}) = \{i\}\right) = 1. \end{array} \right.$$

Proof We begin by showing the first part of correction. If the initial configuration of an execution is legitimate, then all the configurations in this execution are legitimate.

Lemma 3

$$|G(c^0)| = 1 \Rightarrow \forall k \in N,\ |G(c^k)| = 1.$$

Proof This lemma is a direct consequence of propositions 3 and 4. □ **(lemma 3)**

To prove the second part of correction, we first show (lemma 4) that a unique privilege moves from a process to its successor with probability 1. After that, we show (lemma 5) that a unique privilege moves to each process with probability 1. Then we conclude about the second part of the correction : each processor is infinitely many times privileged.

Lemma 4 For all beginnings of execution $c^0, A^0, \cdots, c^k, A^k$ such that $G(c^k) = \{i\}$,

$$Pr\left(\exists j > 0 / G(c^{k+j}) = \{i+1\}\right) = 1.$$

It can be written : $Pr\left(\exists j > 0 / G(c^{k+j}) = \{i+1\} \mid G(c^k) = \{i\}\right) = 1.$

Proof

Lemma 3 ensures that if $G(c^k) = \{i\}$, for all $j \geq 0$ $\left|G(c^{k+j})\right| = 1$ and then $A^{k+j} = G(c^{k+j})$.
Let $G(c^k) = \{i\}$ then $Pr\left(G(c^{k+1}) = \{i+1\}\right) = 1 - r_i$ or $Pr\left(G(c^{k+1}) = \{i\}\right) = r_i$.
It is easy to show that for all $m \geq 0$ $Pr\left(\forall j, (0 \leq j \leq m)\ G(c^{k+j}) = \{i\}\right) = {r_i}^m$.
Since $\lim_{m \to \infty} r_i = 0$, we have $Pr\left(\forall j \in N,\ G(c^{k+j}) = \{i\}\right) = 0$.

□ **(lemma 4)**

Lemma 5 For all beginnings of execution $c^0, A^0, \cdots, c^k, A^k$ such that $G(c^k) = \{i\}$, and for all integers l, we have

$$Pr\left(\exists j \geq 0 / G(c^{k+j}) = \{i+l\}\right) = 1.$$

or equivalently : $Pr\left(\exists j \geq 0 / G(c^{k+j}) = \{i+l\} \mid G(c^k) = \{i\}\right) = 1.$

Proof The proof is obtained by induction on l.
For $l = 1$, the lemme 4 gives the result.
Suppose that for all $m \leq l$, $Pr\left(\exists j > 0 / G(c^{k+j}) = \{i+m\} \mid G(c^k) = \{i\}\right) = 1$, then :

$$\begin{aligned} & Pr\left(\exists j > 0 / G(c^{k+j}) = \{i+m\} \mid G(c^k) = \{i\}\right) \\ = \ & Pr\left(\exists j_1 > 0, / G(c^{k+j_1}) = \{i+m+1\} \mid G(c^k) = \{i\}\right) \\ & \times Pr\left(\exists j_2 > 0, G(c^{k+j_1+j_2}) = \{i+m+1\} \mid G(c^{k+j_1}) = \{i+m\}\right) \\ = \ & 1. \end{aligned}$$

□ **(lemma 5)**

□ **(proposition 5)**

4.2.3 Proof of convergence

Proposition 6 The protocol described in figure 4 verifies the property of convergence defined by : for all initial configuration c^0 and all choice function, (i.e. for all infinite trees as figure 2),

$$Pr\left(\exists k > 0 / c^k \in Leg\right) = 1.$$

Proof

We use a sequence of technical lemmas to prove that, from any unlegitimate initial configuration of any execution, there is a positive probability to move to a legitimate configuration after a fixed number of step.

We first prove that from any unlegitimate configuration of any execution, there is a positive probability to move to a configuration where only one process changes its state.

Lemma 6 Let T be an infinite tree rooted at c^0, for all $k \geq 0$, for all beginnings of execution (or branches) $c^0, A^0, \cdots, c^k$ in T, for all $i \in G(c^k)$,there exists J_i such that

$$Pr\left(c^{k+J_i} = \mathcal{T}(c^k, \{i\})\right) > 0.$$

Proof Consider the infinite branch of T :

$$c^0, A^0, \cdots, c^k, A^k, c^{k+1}, A^{k+1}, \cdots, c^{k+j}, A^{k+j}, \cdots \text{ such that } \forall j > 0,\ c^{k+j} = c^k.$$

As for all infinite branches of T, this one corresponds to an execution, then it must verify the fairness property (see definition 10). Thus, for all $i \in G(c^k)$, there exists $K_i \geq 0$ such that $i \in A^{k+K_i}$.

Now, by definition of the probability (see definition 19) we have

$$Pr\left(c^{k+K_i+1} = \mathcal{T}(c^{k+K_i+1}, \{i\})\right) > 0.$$

Since $c^{k+K_i+1} = c^k$, we set $J_i = K_i + 1$ and, consequently, $Pr\left(c^{k+J_i} = \mathcal{T}(c^k, \{i\}\right) > 0$.

□ **(lemma 6)**

We prove that for any unlegitimate configuration in which two consecutive processes are privileged, and for any execution, there is a positive probability to move to a configuration with less privileges.

Lemma 7 Let T be an infinite tree rooted at c^0.

For all $k \geq 0$, for all $c^0, A^0, \cdots, c^k, A^k$ in T. If $D_{min}(c^k) = 1$ then there exists J such that

$$Pr\left(\left|G(c^{k+J})\right| < \left|G(c^k)\right|\right) > 0.$$

Proof If $D_{min}(c^k) = 1$ then there exists i such that $\{i, i+1\} \subset G(c^k)$. Note $J = J_i$ where J_i is given by the lemma 6. We have $Pr\left(c^{k+J} = \mathcal{T}(c^k, \{i\})\right) > 0$. Now, by proposition 2, we observe that, $G\left(\mathcal{T}(c^k, \{i\})\right) \subset G\left(c^k\right) \setminus \{i\}$.

Thus, we have proved $\left|G\left(\mathcal{T}(c^k, \{i\})\right)\right| < \left|G\left(c^k\right)\right|$,

and consequently, $Pr\left(\left|G(c^{k+J})\right| < \left|G(c^k)\right|\right) > 0$.

□ **(lemma 7)**

In the next lemma, we prove that, from any unlegitimate configuration of any execution such that no consecutive processes are simultaneously privileged, there is a positive probability to move to a configuration in which the distance between two privileges is smaller.

Lemma 8 Let T be an infinite tree rooted at c^0. For all $k \geq 0$, for all $c^0, A^0, \cdots, c^k, A^k$ in T, if $D_{min}(c^k) > 1$ then there exists J such that

$$Pr\left(D_{min}(c^{k+J}) < D_{min}G(c^k)\right) > 0.$$

Proof If $D_{min}(c^k) > 1$ then there exists i such that $\{i, i+D_{min}(c^k)\} \subset G(c^k)$ Note $J = J_i$ where J_i is given by the lemma 6. We have $Pr\left(c^{k+J_i} = \mathcal{T}(c^k, \{i\})\right) > 0$. Now, by proposition 2, we observe that, $G\left(\mathcal{T}(c^k, \{i\})\right) = G\left(c^k\right) \cup \{i+1\} \setminus \{i\}$.

Thus, we have proved $D_{min}(\mathcal{T}(c^k, \{i\}) = i + D_{min}(c^k) - (i+1)$,

and consequently, $Pr\left(D_{min}(c^{k+J}) < D_{min}(c^k)\right) > 0$.

□ **(lemma 8)**

In the next lemma we will generalize the lemma 7 to the case of an arbitrary unlegitimate configuration of any execution (i.e. without restriction on $D_{min}(c^k)$) .

Lemma 9 Let T be an infinite tree rooted at c^0.
For all $k \geq 0$, for all $c^0, A^0, \cdots, c^k, A^k$ in T, $|G(c^k)| > 1$ there exists M such that

$$Pr\left(|G(c^{k+M})| < |G(c^k)|\right) > 0.$$

Proof The proof is obtained by induction on $D_{min}(c^k)$.
If $D_{min}(c^k) = 1$ the lemma 7 applies.
Suppose that for all $D_{min}(c^k) \leq d$, there exists M such that

$$Pr\left(|G(c^{k+M})| < |G(c^k)|\right) > 0.$$

We must show that for $D_{min}(c^k) = d+1$, there exists M such that

$$Pr\left(|G(c^{k+M})| < |G(c^k)|\right) > 0.$$

Assume $D_{min}(c^k) = d+1$, then the lemma 8 applies and there exists J such that

$$Pr\left(D_{min}(c^{k+J}) < D_{min}(c^k)\right) > 0.$$

Using the induction hypothesis, there exists M_d such that

$$Pr\left(|G(c^{k+J+M_d})| < |G(c^{k+J})|\right) > 0.$$

But by proposition 3, $G(c^{k+J}) \leq G(c^k)$, and we have proved,

$$Pr\left(|G(c^{k+J+M_d}| < |G(c^k)|\right) > 0.$$

This ends the proof using $M = J + M_d$.

□ **(lemma 9)**

The lemma 10 is the consequence of all the previous lemmas. It shows that from any unlegitimate configuration of any execution, the probability to move to a legitimate configuration is positive.

Lemma 10 Let T be an infinite tree rooted at c^0.
For all $k \geq 0$, for all $c^0, A^0, \cdots, c^k, A^k$ in T, such that $|G(c^k)| > 1$, there exists M with

$$Pr\left(|G(c^{k+M})| = 1\right) > 0.$$

Proof The proof is obtained by induction on $|G(c^k)|$.
If $|G(c^k)| = 2$, the lemma 9 applies.
Suppose that for all $|G(c^k)| \leq g$, there exists M such that $Pr\left(|G(c^{k+M})| = 1\right) > 0$. We must show that for $|G(c^k)| = g+1$, there exists M such that $Pr\left(|G(c^{k+M})| = 1\right) > 0$:
For $|G(c^k)| = g+1 > 1$, then the lemma 9 implies that there exists M_1 such that

$$Pr\left(|G(c^{k+M_1})| < |G(c^k)|\right) > 0.$$

By the induction hypothesis, there exists M_2 such that $Pr\left(|G(c^{k+M_1+M_2})| = 1\right) > 0$.
This ends the proof using $M\ (= M_1 + M_2)$.

□ **(lemma 10)**

Corollary 1 Let T be an infinite tree rooted at c^0
For all $k \geq 0$, for all $c^0, A^0, \cdots, c^k, A^k$ in T, $|G(c^k)| > 1$ there exists M such that

$$Pr\left(|G(c^{k+M})| > 1\right) < 1.$$

Lemma 11 Let T be an infinite tree rooted at c^0. For all $k \geq 0$, there exists M^k_{max} such that for all $c^0, A^0, \cdots, c^k, A^k$ in T,

$$Pr\left(\left|G(c^k + M^k_{max})\right| > 1\right) < 1.$$

Proof For $k > 0$ and for all $c^0, A^0, \cdots, c^k, A^k$ in T, we have proved by corollary 1, that there exists $M_{c^0,\cdots,c^k}$ such that

$$Pr(\left|G(c^{k+M_{c^0,\cdots,c^k}})\right| = 1) > 0.$$

Since there is a finite number of possible configurations in S, there is also, for any value of k, a finite number of possible sets of configurations $\{c^0, \cdots, c^k\} \in S^{k+1}$. Therefore , we can define :

$$M^k_{max} = (\max_{c^0,\cdots,c^k \in S^{k+1}} M_{c^0,\cdots,c^k}) < \infty.$$

Using the property (equivalent to proposition 3)

$$k \geq k' \Rightarrow Pr\left(\left|G(c^k)\right| > 1\right) \geq Pr\left(\left|G(c^{k'})\right| > 1\right),$$

we have prove that there exists M_{max} such that

$$\forall c^0, A^0, \cdots, c^k, A^k \in S^{k+1}, Pr\left(\left|G(c^k + M^k_{max})\right| > 1\right) < 1.$$

□ **(lemma 11)**

Lemma 12 Let T be an infinite tree starting at c^0. There exists M_{max} such that for all $k > 0$, for all $c^0, A^0, \cdots, c^k, A^k$ in T,

$$Pr\left(\left|G(c^k + M_{max})\right| > 1\right) < 1.$$

Proof We set $M_{max} = \max_{k=0}^{D} M^k_{max}$ where D is given by definition 9.

By construction, the lemma holds for all $k \leq D$. Assume $k > D$, the determination of c^{k+1} depends only of the D preceding configurations. Therefore, we will define D configurations as follow $d^0, \cdots, d^D$ equal to $c^{k-D-1}, \cdots, c^{k-1}$ such that,

$$M_{c^0,\cdots,c^k} = M_{d^0,\cdots,d^D}.$$

Hence, we have

$$Pr\left(\left|G(c^k + M_{max})\right| = 1\right) > Pr\left(\left|G(c^k + M_{d^0,\cdots,d^D})\right| = 1\right).$$

□ **(lemma 12)**

The proof of proposition 6 is now straigthforward.

Setting $Max = \max(M_{max}, D)$, we have $Pr\left(\forall j > 0 \left|G(c^j)\right| > 1\right) \leq Pr\left(\forall k > 0 \left|G(c^{kMax})\right| > 1\right)$

But, $Pr\left(\left|G(c^{kMax})\right| > 1\right) = Pr\left(\left|G(c^{kMax})\right| > 1 \mid \left|G(c^{(k-1)Max})\right| > 1\right)$

Since the evolution after the $kMax$ iterations is independant of the configurations before $(k-1)Max$. Therefore by induction on k, we have $Pr\left(\left|G(c^{kMax})\right| > 1\right) = Pr\left(\left|G(c^{Max})\right| > 1\right)$. And

$$\begin{aligned} Pr\left(\forall k > 0 \left|G(c^{kMax})\right| > 1\right) &= \prod_{k=0}^{\infty} Pr\left(\left|G(c^{Max})\right| > 1\right) \\ &= 0. \end{aligned}$$

□ **(proposition 6)**

We have proved the correction (proposition 5) and the convergence (proposition 6). The absence of deadlock is proved in proposition 4. Then we have proved that the protocol defined in figure 4 is a probabilistic mutual exclusion self-stabilizing protocol.

□ **(theorem 2)**

5 conclusion and perspectives

In this paper, we have presented an optimal self-stabilizing mutual exclusion protocol on unidirectional uniform rings, which is optimal in the number of states per process. This protocol is proved under the hypothesis of a fair distributed daemon with bounded memory.

A key issue is to relax this hypothesis for the distributed daemon. Note that, in the proof of convergence, the hypothesis of bounded memory is just used in the last lemma (lemma 12). The goal is to bound each factor of the infinite product of the proposition 6 (which is smaller than 1), in order to ensure that this infinite product, which is the probability of no convergence of the protocol, is equal to 0.

But it is not necessary, for an infinite product to be null, to bound each of its terms. In fact, it is sufficient to bound an infinity of them. We think that this weaker condition can be obtained from a weaker hypothesis about the distributed daemon. Roughly speaking, this hypothesis expresses that, during any infinite execution, the number of times where the distributed daemon tries to move a privilege without trying to move another privilege cannot tend speedly to the infinity. Moreover we think that this hypothesis is the weakest one allowing a solution to the problem. At the present time, we are working at the technical proof of this result.

References

[BD94] Joffroy Beauquier and Sylvie Delaët. Probabilistic self-stabilizing mutual exclusion in uniform rings. In *Proceeding* 13^th *Principles of Distributed Computing*, page 378, august 1994.

[BP89] James E. Burns and Jan Pachl. Uniform self-stabilizing rings. *Transactions of programming languages and systems*, 11(2):330–344, April 1989.

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

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

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

[IJ90] Amos Israeli and marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. *Proceeding* 9^th *Principles of distributed Computing*, pages 119–129, 1990.

Paper Number 16

Self-Stabilizing Ring Orientation Protocols

Ming-Shin Tsai and Shing-Tsaan Huang

Self-Stabilizing Ring Orientation Protocols[1]

Ming-Shin Tsai
Shing-Tsaan Huang

Department of Computer Science
National Tsing Hua University
HsinChu, Taiwan 30043
Republic of China

Abstract

This paper proposes two deterministic self-stabilizing orientation protocols for uniform rings. The first one works with a central demon for rings of arbitrary size; the second one works with a distributed demon but for rings of odd size only. They improve the previous results by Israëli and Jalfon [IJ90, IJ93].

Index Terms—Ring orientation, deterministic protocols, self-stabilization.

---

[1] This research is supported by National Science Council of the Republic of China under the contract NSC83-0408-E-007-028.

1. Introduction

This paper considers the self-stabilizing orientation problem of a ring with uniform processors. All the processors in the ring are identical; i.e., they are equally programmed and do not possess unique identifiers. Each processor in the ring maintains an orientation variable which is a pointer pointing to one of its two neighbors. The orientations of the processors may be inconsistent in the sense that some are clockwise, and the other(s) counter-clockwise. A ring is oriented if the orientations of all processors are consistent. This problem was first studied by Attiya et al [ASW88] and Syrotiuk and Pachl [SP88].

The protocols proposed in this paper are different from the previous ones [ASW88, SP88] because the proposed ones are self-stabilizing protocols. The term self-stabilization was first introduced by Dijkstra [Dijkstra74]. In a self-stabilizing system, each processor makes its decision based only on the states of its neighbors and itself. Global coordination is not required, and each processor can start from any possible initial state. The design of a self-stabilizing protocol must ensure that the system will reach a legitimate state within a finite number of moves, regardless of the initial state of the system and the execution sequence of the processors. So a self-stabilizing protocol has the ability to recover from unexpected perturbations without outside intervention.

Dijkstra originally proposed three token circulation protocols on a ring [Dijkstra74]. These protocols require the existence of a special processor in the ring, and each processor must be able to distinguish its left neighbor from its right neighbor. Later works [BP89, Huang93] have shown that it is possible to have token circulation protocols on uniform ring if the number of processors is prime; however, these protocols still require the processors to have consistent orientations. This raised the question whether a token circulation protocol exists on uniform un-oriented rings. This question was answered by Israeli and Jalfon [IJ90, IJ93]. Their solution is to superimpose a token circulation protocol for uniform oriented rings on top of a ring orientation protocol for uniform rings.

In their articles, Israeli and Jalfon concluded that if a ring has an even number of processors then it has no self-stabilizing orientation protocol under the central demon model. A central demon is a global scheduler which arbitrarily activates one of the processors. The activated processor reads the states of all its neighbors and then moves to a new state if necessary. In other words, the execution of the processors is serialized. The central demon model is widely used in most papers on self-stabilization [Dijkstra74, BP89, Huang93].

There is another scheduler commonly discussed in the literature: the distributed demon. The distributed demon can activate any subset of the processors at a time. All the activated processors simultaneously read all the states of their neighbors, then move to their new states at the same time. The distributed demon model is more realistic than the central demon model in distributed systems because

more than one processor may be activated at the same time; however, it is more difficult to design self-stabilizing protocols under this model. Israeli and Jalfon also concluded that if a ring has an odd number of processors then it has no self-stabilizing orientation protocol under the distributed model.

Both conclusions above are based on the *assumption* that the orientation of a processor is simply a state which looks the same to both its neighbors. The paper of Attiya et al [ASW88] also had a similar conclusion for non self-stabilizing ring orientation protocols. Here in this paper, the assumption is relaxed by introducing an orientation variable which is a pointer. A processor can determine if the orientation of a neighbor is pointing to itself by inspecting the orientation variable of that neighbor. On the other hand, in the model of Israeli and Jalfon a processor does not know if the orientation of a neighbor is pointing to itself or not. This paper then proposes two deterministic orientation protocols. The first one is for rings of arbitrary size and works with a central demon; the second one works with a distributed demon but for rings of odd size only.

The rest of this paper is organized as follows. In the next section, we first review the conclusions of Israeli and Jalfon. We then show that it is possible to relax their assumption. In Section 3, we present a self-stabilizing ring orientation protocol for rings of arbitrary size under the central demon model. Its correctness is proven in Section 4. In Section 5 we modify the first protocol so that it works with a distributed demon but for odd rings only. Finally in Section 6 we give some concluding remarks.

2. Theoretical background

In [IJ93], each processor can internally distinguish between a *first* neighbor and a *second* neighbor. Deciding which neighbor is first and which one is second is done by the hardware. The orientation of a processor is either 1 or 2 to correspond to its first or second neighbor, respectively. To simplify the discussion the processors are arbitrarily numbered clockwise, as shown in Figure 2.1. Such a numbering is for discussion only because the processors do not know their identities in a uniform ring.

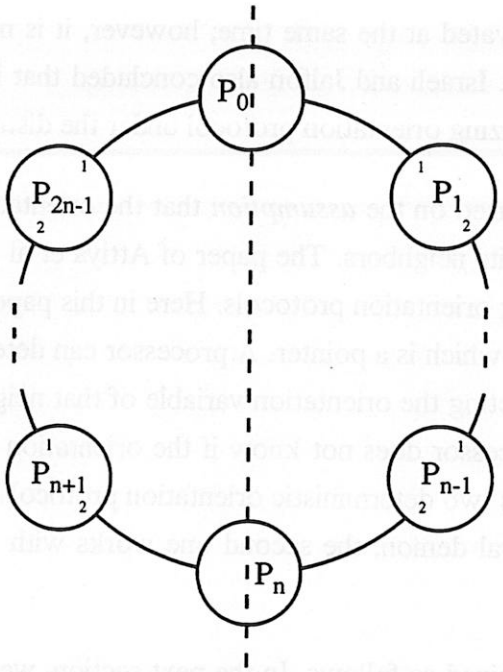

Figure 2.1 Symmetric neighbor ordering of an even ring

We use Figure 2.1 to explain why Israeli and Jalfon concluded that there is no self-stabilizing orientation protocol under the central demon model if the ring has an even number of processors. In Figure 2.1, the system has $2n$ processors. The processors are connected as follows: Processor i-1 is the first neighbor of processor i for $1 \le i \le n$-1, and processor i-1 is the second neighbor of processor i for n+1 $\le i \le 2n$-1. There is no requirement on the neighbor ordering of processor 0 and processor n. Let c_i denote the orientation of processor i. Now let the system has an initial configuration such that:

$$c_1 = c_{2n-1},\ c_2 = c_{2n-2},\ \ldots,\ c_{n-1} = c_{n+1}$$

In other words, the system has a symmetry axis as shown in Figure 2.1. It is easy to see that such a configuration is not oriented because for any i, $1 \le i \le n$-1, processor i and processor $2n$-i have opposite orientations. Besides, the state of the first (second) neighbor of processor i is the same as the state of the first (second) neighbor of processor $2n$-i. If the central demon activates processor i followed by processor $2n$-i, then both will enter the same state, and the system still has the same symmetry axis. So the system may never reach a legitimate state if the central demon always activates processor i followed by processor $2n$-i for $1 \le i \le n$-1. Using a similar argument Israeli and Jalfon also showed that there is no self-stabilizing orientation protocol for odd rings under the distributed demon model.

In this paper, we assume that the orientation of a processor is a pointer instead of a number. For example, in Figure 2.1, if processor 1 is the first neighbor of processor 0 and the orientation of processor 0 is 1, then the orientation of processor 0 seen by both processor 1 and processor $2n$-1 is 1 in the

protocol of [IJ93]. On the other hand, in our protocols processor 1 regards that the orientation of processor 0 is *pointing to* itself, while processor $2n$-1 regards that as *opposite to* itself. So the symmetry axes of the systems in both Figure 2.1 and Figure 2.2 are no longer preserved.

Although it is possible to have self-stabilizing orientation protocols for rings of arbitrary size under the central demon model and for odd rings under the distributed demon model once the assumption of [IJ93] is relaxed, it is still impossible to have orientation protocols for even rings under the distributed demon model. This is shown in Figure 2.2.

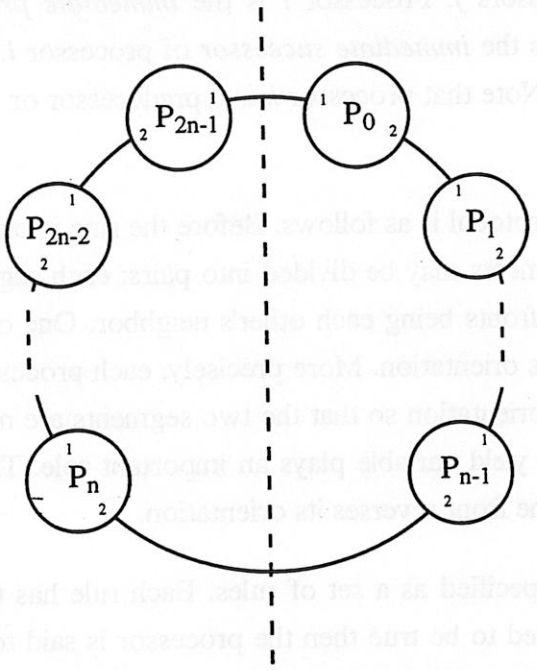

Figure 2.2 Symmetric neighbor ordering of an even ring with a distributed demon

The system in Figure 2.2 has an symmetry axis if the initial configuration is that processor 0 and processor $2n$-1 have opposite orientations, processor 1 and processor $2n$-2 have opposite orientations, and so on. If the distributed demon always activates processor 0 and processor $2n$-1 together, and processor n-1 and processor n together, and processor i followed by processor $2n$-i-1 for $1 \le i \le n$-2, then the symmetry axis is preserved and the system may never reach a legitimate state. This concludes that there is no self-stabilizing orientation protocols for even rings with a distributed demon.

3. The first protocol

In the first protocol, each processor maintains an orientation variable which points to one of its two neighbors. Besides, each processor also has a yield variable, whose value is either y or n for yield or not yield. To simplify our discussion, we use the notation ${}_i(\rightarrow, Y)_j$ to indicate that the orientation of

processor i is pointing to processor j and the yield variable of processor i has the value Y, where Y may be y or n. We also use $Y(i)$ to denote the yield variable of processor i when necessary. Both the orientation variables and the yield variables are subject to arbitrary initialization.

For an arbitrary initialization of the orientation variables, the ring can be divided into segments. Each segment consists of a maximum number of neighboring processors that have the same orientation. Let processors $i, j, \ldots k$ with $(\leftarrow)_i(\rightarrow)_j(\rightarrow)\ldots.(\rightarrow)(\leftarrow)$ be a segment S. Processor i is called the *root* of S, and processor k the *front* of S. Processors j is a *predecessor* of processor k, denoted as $j \gg k$; conversely, processor k is a *successor* of processors j. Processor i is the *immediate predecessor* of processor j, denoted as $i \rightarrow j$, while processor j is the *immediate successor* of processor i. We use $S.i$ to denote the segment that processor i belongs to. Note that processor k is a predecessor or a successor of processor i implies $k \in S.i$.

The idea adopted in the first protocol is as follows. Before the ring is oriented, there must exist an even number of segments. Those segments may be divided into pairs; each segment pair consists of two neighboring segments with their two fronts being each other's neighbor. One of the two fronts of a pair must yield to the other by reversing its orientation. More precisely, each processor of one (and only one) of the two segments must reverse its orientation so that the two segments are merged into one. To avoid alternative yielding of two fronts, the yield variable plays an important role. The intention to yield by a front must propagate upward before the front reverses its orientation.

A self-stabilizing protocol is specified as a set of rules. Each rule has two parts: a *guard* and a *move*. If the guard of a rule is evaluated to be true then the processor is said to have a *privilege* of that rule. A privileged processor may then make the corresponding move, which brings the processor to a new state. The first protocol consists of the following four rules for each processor i:

R1: If $_i(\rightarrow, n)(\leftarrow, n)$ then $_i(\rightarrow, y)(\leftarrow, n)$;
R2: If $_i(\rightarrow, n)(\rightarrow, y)$ then $_i(\rightarrow, y)(\rightarrow, y)$;
R3: If $(\rightarrow, y)_i(\rightarrow, y)(\leftarrow,)$ then $(\rightarrow, y)_i(\leftarrow, n)(\leftarrow,)$;
R0: If $(\leftarrow,)_i(\rightarrow, y)(\leftarrow,)$ then $(\leftarrow,)_i(\leftarrow, n)(\leftarrow,)$;

Rl is used to resolve the confrontation with processor i expressing its intention to yield. R2 is to propagate the intention of yielding. R3 and R0 is used to reverse the orientation. The numbering of the rules may look strange; such a numbering is to fit the definition of the bounded function defined in the next section.

We use an example to show how the protocol works. In Figure 3.1 there are two segments (2, 1 and 3, 4, 0). A possible execution sequence is shown below. The first column shows the system state

after the move. The second and the third columns are the processor selected by the central demon and the rule it applies.

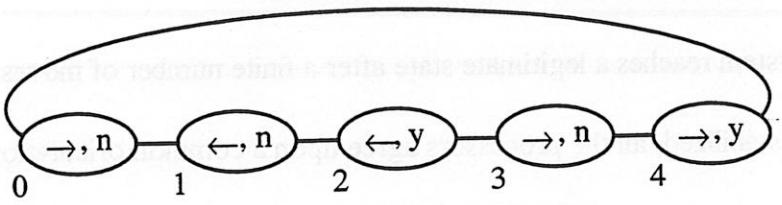

Figure 3.1 Initial state of a uniform ring with five processors.

$0(\rightarrow, n)1(\leftarrow, n)2(\leftarrow, y)3(\rightarrow, n)4(\rightarrow, y)$		
$0(\rightarrow, y)1(\leftarrow, n)2(\leftarrow, y)3(\rightarrow, n)4(\rightarrow, y)$	processor 0	R1
$0(\leftarrow, n)1(\leftarrow, n)2(\leftarrow, y)3(\rightarrow, n)4(\rightarrow, y)$	processor 0	R3
$0(\leftarrow, n)1(\leftarrow, n)2(\leftarrow, y)3(\rightarrow, y)4(\rightarrow, y)$	processor 3	R2
$0(\leftarrow, n)1(\leftarrow, n)2(\leftarrow, y)3(\rightarrow, y)4(\leftarrow, n)$	processor 4	R3
$0(\leftarrow, n)1(\leftarrow, n)2(\leftarrow, y)3(\leftarrow, n)4(\leftarrow, n)$	processor 3	R0
$0(\leftarrow, n)1(\leftarrow, n)2(\leftarrow, y)3(\leftarrow, y)4(\leftarrow, n)$	processor 3	R2
$0(\leftarrow, n)1(\leftarrow, n)2(\leftarrow, y)3(\leftarrow, y)4(\leftarrow, y)$	processor 4	R2
$0(\leftarrow, y)1(\leftarrow, n)2(\leftarrow, y)3(\leftarrow, y)4(\leftarrow, y)$	processor 0	R2
$0(\leftarrow, y)1(\leftarrow, y)2(\leftarrow, y)3(\leftarrow, y)4(\leftarrow, y)$	processor 1	R2

In the above example, both processor 0 and processor 1 have privileges of R1 in the initial configuration; however, they cannot apply R1 concurrently because a central demon is assumed. After one of them applies R1 the other processor no longer has a privilege of R1. Similarly, in the following configuration

$$0(\rightarrow, y)1(\leftarrow, y)2(\leftarrow, y)3(\rightarrow, n)4(\rightarrow, y),$$

both processor 0 and processor 1 have privileges of R3. Suppose that processor 0 is chosen by the central demon. After processor 0 applies R3 the system configuration becomes

$$0(\leftarrow, n)1(\leftarrow, y)2(\leftarrow, y)3(\rightarrow, n)4(\rightarrow, y).$$

Processor 1 no longer has a privilege of R3 in this configuration. So the first protocol prevents both fronts of a confrontation yielding to each other simultaneously under the central demon model.

4. Correctness of the first protocol

We now show that the first protocol is self-stabilizing with a central demon. A protocol is self-stabilizing if it satisfies the following requirement:

Liveness: The system can always make a move in an illegitimate state.

Safety: The system remains in a legitimate state once it does.

No livelock: The system reaches a legitimate state after a finite number of moves

When the system is stabilized, all the processors agree upon a common orientation; i.e.,

$$(\forall i : : {}_i(\rightarrow,)(\rightarrow,)).$$

Based on the above predicate, it is easy to see that the first protocol satisfies the liveness and safety requirements in Lemma 4.1 and Lemma 4.2.

Lemma 4.1: The system can always make a move in an illegitimate state.

Proof: If the system is not oriented, then there is at least one confrontation. If it has the form ${}_i(\rightarrow, n)_j(\leftarrow, n)$ then processor i or processor j can apply R1. Otherwise it must have the form ${}_i(\rightarrow, y)_j(\leftarrow,),$. Let us look at the other neighbor k of i, if ${}_k(\rightarrow, n)_i(\rightarrow, y)_j(\leftarrow,)$ then processor k can apply R2; if ${}_k(\rightarrow, y)_i(\rightarrow, y)_j(\leftarrow,)$ then processor i can apply R3; if ${}_k(\leftarrow,)_i(\rightarrow, y)_j(\leftarrow,)$ then i can apply R0. ❑

Lemma 4.2: The system remains in a legitimate state once it does.

Proof: This is obvious because the only rule which may be applied if all the processors have the same orientation is R2, which does not change the orientation variables. ❑

To show that a self-stabilizing protocol satisfies the no livelock requirement with a central demon, most of the proposed self-stabilizing algorithms used the *variant function* approach reported in [Kessels88]. The central idea is to find a bounded function F of the system state, then show that F decreases or increases monotonically toward the bound after each move. For the first protocol, the bounded function F is defined as $F \equiv (F_0, F_1, F_2, F_3)$, where

F_0 is the number of segment pairs;

F_1 is the number of the confrontations of the form $(\rightarrow, n)(\leftarrow, n)$;

F_2 is the number of processors i such that $Y(i) = n$ and $(\exists k : i \gg k : Y(k) = y)$;

F_3 is the number of processors i such that $Y(i) = y$ and $(\forall k : i \gg k : Y(k) = y)$.

The comparison of F is by lexicographic order. Intuitively, F_1 represents the number of processors that may apply R1, F_2 the number of processors that may apply R2, and F_3 the number of processors that may apply R3.

Lemma 4.3: F decreases monotonically each time after a rule is applied.

Proof: We shall examine the rules one by one. First, note that the rules create neither a new segment pair nor a new confrontation of the form $(\rightarrow, n)(\leftarrow, n)$; i.e., no rule can make F_0 or F_1 increase.

It is easy to see that each time a processor i applies R0, the number of segment pair decreases by one: three segments of the form ...$\leftarrow \rightarrow \leftarrow$... merge into ...$\leftarrow \leftarrow \leftarrow$... That is, any application of R0 makes F decrease because F_0 decreases by one.

Each time a processor i applies R1, F_1 decreases by one with F_0 remaining unchanged. Thus, any application of R1 makes F decrease.

Each time a processor i applies R2, F_2 decreases by one with F_0, F_1 being unchanged. Hence, the application of R2 makes F decrease.

Finally, each time a processor i applies R3, F_3 decreases by one. It is also obvious that F_0, F_1, and F_2 remains unchanged. Therefore, the application of R3 also makes F decrease. ❑

Since F is bounded, by Lemma 4.3 the system must reach a legitimate state after a finite number of moves. This concludes that the first protocol is deterministic and self-stabilizing with a central demon.

5. The second protocol and its correctness

The first protocol does not work with a distributed demon because two fronts of a confrontation may yield to the other simultaneously. For example, the following system may never stabilize if the same execution sequence is repeated forever. The notation (i, Ri; j, Rj; ...) denotes that processors i, j, ... are activated at the same time with processor i applying Ri, processor j applying Rj, etc.

$_0(\rightarrow, y)_1(\rightarrow, y)_2(\leftarrow, y)_3(\leftarrow, y)$	(1, R3; 2, R3)
$_0(\rightarrow, y)_1(\leftarrow, n)_2(\rightarrow, n)_3(\leftarrow, y)$	(0, R0; 3, R0)
$_0(\leftarrow, n)_1(\leftarrow, n)_2(\rightarrow, n)_3(\rightarrow, n)$	(0, R1; 3, R1)
$_0(\leftarrow, y)_1(\leftarrow, n)_2(\rightarrow, n)_3(\rightarrow, y)$	(1, R2; 2, R2)
$_0(\rightarrow, y)_1(\rightarrow, y)_2(\leftarrow, y)_3(\leftarrow, y)$	

. . . . . .

The above problem can be prevented if the size N of the ring is odd. The idea is to introduce a third variable, the label variable, in each processor. The notation $_i(\rightarrow, Y)_j$ is modified to $_i(\rightarrow, Y, L)_j$ to incorporate the label variable L. The label of a root processor has the value 0; The labels of the other processors of a segment have the value 1 or 0 alternatively along the segment. We use $L(i)$ to denote the label variable of processor i when necessary. When a confrontation of the form $_i(\rightarrow, n, 0)_j(\leftarrow, n, 1)$ or

$_i(\rightarrow, y, 0)_j(\leftarrow, y, 1)$ exists, only processor i; i.e., the processor with the label 0, is allowed to yield. This prevents processors i and j yielding to each other simultaneously.

The second protocol consists of the following seven rules for each processor i:

R0: If $(\leftarrow, ,)_i(\rightarrow, y,)(\leftarrow, n,)$ then $(\leftarrow, ,)_i(\leftarrow, n,)(\leftarrow, n,)$;

R1.1: If $_i(\rightarrow, n, 0)(\leftarrow, n, 1)$ then $_i(\rightarrow, y, 0)(\leftarrow, n, 1)$;

R1.2: If $(\rightarrow, y, 0)_i(\leftarrow, y, 1)$ then $(\rightarrow, y, 0)_i(\leftarrow, n, 1)$;

R2: If $_i(\rightarrow, n,)(\rightarrow, y,)$ then $_i(\rightarrow, y,)(\rightarrow, y,)$;

R3: If $(\rightarrow, y,)_i(\rightarrow, y,)(\leftarrow, n,)$ then $(\rightarrow, y,)_i(\leftarrow, n,)(\leftarrow, n,)$;

R4: If $_i(\leftarrow, , 1)(\rightarrow, ,)$ then $_i(\leftarrow, , 0)(\rightarrow, ,)$;

R5: If $(\rightarrow, , L)_i(\rightarrow, , L)$ then $(\rightarrow, , L)_i(\rightarrow, , \bar{L})$;

To simplify our discussion, the rules are separated into two sub protocols: the L protocol and the M protocol. The L protocol maintains the labels of the processors and consists of the rules R4 and R5. R4 set the label of the root of a segment to 0. R5 updates the label of a processor i which is not a root. Intuitively, the label of a front i should have the value 0 if the size of $S.i$ is odd, otherwise it should be 1. For a ring of odd size, since the number of segments is even before the ring is oriented, the number of segments with odd size must be odd. In other words, there exists at least one segment pair such that one segment is of odd size while the other is of even size. So the system can always make a move before it is oriented.

The M protocol consists of the rules R0 to R3 and is almost identical to the first protocol except that at most one front of each confrontation having the form $(\rightarrow, n,)(\leftarrow, n,)$ or $(\rightarrow, y,)(\leftarrow, y,)$ can reverse its orientation by R1.1 and R1.2. Before a front reverses its orientation by applying R3 or R0, the other front of the confrontation must not have the intention to yield. This prevent simultaneous yielding of the two fronts. R2 is the same as in the first protocol.

In the following we shall show that the second protocol works with a distributed demon if N is odd. We first prove the liveness and safety properties of the second protocol.

Lemma 5.1: The system can always make a move in an illegitimate state.

Proof: The proof is similar to Lemma 4.1. If the processors do not have the same orientation then there is at least one segment pair such that one segment is of odd size while the other is of even size. R4 or R5 can be applied if any processor of this segment pair does not have correct label. If all processors have correct labels then the confrontation must be one of the following forms: $_i(\rightarrow, n, 0)_j(\leftarrow, n, 1)$, $_i(\rightarrow, y, 0)_j(\leftarrow, y, 1)$, or $_i(\rightarrow, y,)_j(\leftarrow, n,)$. In the first and second cases R1.1 and R1.2 can be applied, respectively. In the third case let k be the other neighbor of i. If $_k(\rightarrow, n,)_i(\rightarrow, y,)_j(\leftarrow, n,)$ then k can

apply R2. If $_k(\rightarrow, y,)_i(\rightarrow, y,)_j(\leftarrow, n,)$ then i can apply R3. If $_k(\leftarrow, ,)_i(\rightarrow, y)_j(\leftarrow, n,)$ then i can apply R0. ❑

Lemma 5.2: The system remains in a legitimate state once it does.

Proof: This is obvious because the only rules which may be applied if all the processors have the same orientation are R2 and R5, which do not change the orientation variables. ❑

Before showing that the second protocol stabilizes within a finite number of moves with a distributed demon, we first show that it does so with a central demon. Again we use the variant function approach. For the second protocol, the bounded function G is defined as $G \equiv (G_0, G_1, G_2, G_3, G_4, G_5)$, where

G_0 is the number of segment pairs;

G_1 is the number of the confrontations of the form $(\rightarrow, n,)(\leftarrow, n,)$ or $(\rightarrow, y,)(\leftarrow, y,)$;

G_2 is the number of processors i such that $Y(i) = n$ and $(\exists k : i \gg k : Y(k) = y)$;

G_3 is the number of processors i such that $Y(i) = y$ and $(\forall k : i \gg k : Y(k) = y)$;

G_4 is the number of root processors i such that $L(i) = 1$;

G_5 is defined as $p_0 + p_1 + \ldots + p_{n-1}$, where p_i is one plus the number of successors of processor i if $(\rightarrow, , L)_i(\rightarrow, , L)$, or 0 otherwise. In other words, $p_i \neq 0$ implies that processor i has a privilege of R5. We let $G_5 = n$ if the ring has only one segment. For example, in the following configuration:

$$_0(\leftarrow, n, 0)_1(\leftarrow, n, 1)_2(\leftarrow, y, 0)_3(\leftarrow, n, 0)_4(\rightarrow, y, 0),$$

$p_0, p_1, \ldots, p_4$ have the values 0, 0, 3, 0, 0, respectively. Intuitively, if a processor i has a privilege of R5, then all the successors of processor i and processor i itself may have to apply R5 to change their labels. In the example above the label of processor 3 is changed to 1 if it applies R5. After that processor 2 has to apply R5 to change its label to 0, then processor 1 also has to apply R5 to change its label to 1. In the worst case processor 3 and all its successors have to apply R5. So p_i indicates the number of processors that may need to apply R5 in the worst case if processor i has a privilege of R5. Note that, if processor j is a successor of processor i and both processor i and processor j have privileges of R5, then $p_i > p_j$.

For the other bounded functions, G_0, G_2, G_3 are identical to F_0, F_2, F_3 of the first protocol. G_1 is a slight modification of F_1. G_4 represents the number of processors that may apply R4.

Lemma 5.3: Before the ring is oriented, G decreases monotonically each time when a rule of the second protocol is applied.

Proof: We consider the L protocol and the M protocol separately. G_0 decreases by 1 each time a processor applies R0; it does not increase each time a processor applies any other rule of the M protocol. G_1 decreases by 1 each time a processor applies R1.1 or R1.2; it does not increase each time a processor applies R2 or R3. G_2 decreases by 1 each time a processor applies R2; it does not increase each time a processor applies R3. G_3 decreases by 1 each time a processor applies R3. So G decreases each time a processor applies any rule of the M protocol.

The L protocol does not modify the orientation variables or the yield variables, so G_0 to G_3 does not increase each time a processor applies R4 or R5. G_4 decreases by 1 each time a processor applies R4; it remains unchanged each time a processor applies R5. Let j be the immediate successor of i and let $p_i = p$, then after i applies R5 p_i becomes 0. p_j may increase from 0 to p - 1,; however, G_5 still decreases. So G decreases each time a processor applies R4 or R5. ❑

Next we show that the second protocol works with a distributed demon. We use the same bounded function G and show that G decreases monotonically even if more than one processor makes a move concurrently in each computation step. Note that, if none of these processors is a neighbor of one another, then executing them concurrently has the same effect as executing them one by one. So we only need to consider the case that these processors are the neighbors of one another.

Lemma 5.4: G decreases monotonically in each step if more than one processor applies R0 concurrently.

Proof: Let processors i, j be two neighboring processors which apply R0 concurrently, then they must have the configuration $(\rightarrow, n,)_i(\leftarrow, y,)_j(\rightarrow, y,)(\leftarrow, n,)$, so the other neighbors of i and j cannot apply R0 at the same time because their yield variables have the value n. After i, j apply R0 concurrently we have $(\rightarrow, n,)_i(\rightarrow, n,)_j(\leftarrow, n,)(\leftarrow, n,)$; i.e., the number of segment pairs decreases by 1, so G decreases. ❑

Lemma 5.5: G decreases monotonically in each step if no processor applies R0 and at least one processor applies R1.1 or R1.2.

Proof: For a confrontation of the form $_i(\rightarrow, n, 0)_j(\leftarrow, n, 1)$ only processor i can apply R1.1. Conversely, for a confrontation of the form $_i(\rightarrow, y, 0)_j(\leftarrow, y, 1)$ only processor j can apply R1.2. In both cases the other neighbors of i and j cannot apply R1.1 or R1.2. In other words, no neighboring processors can apply R1.1 and/or R1.2 simultaneously. So from Lemma 5.3 G decreases monotonically in each step if at least one processor applies R1.1 or R1.2. ❑

Lemma 5.6: G decreases monotonically in each step if no processor applies R0, R1.1 or R1.2 and at least one processor applies R2 or R3.

Proof: Let i be a processor applying R2, then none of the neighbors of i can apply R2 at the same time. Similarly, if i applies R3 then neither neighbor of i can apply R3 simultaneously. So the correctness follows from Lemma 4.3. ❑

Lemma 5.7: G decreases monotonically in each step if only rules R4 and/or R5 are concurrently applied.

Proof: G_4 decreases if at least one processor applies R4 because the number of root processors i such that $L(i) = 1$ decreases by 1 after R4 is applied. Let $p_i = p$ and $i, j, ..., k$ apply R5 concurrently; i.e., we have $(\rightarrow, , L)_i(\rightarrow, , L)_j \ldots {}_k(\rightarrow, , L)$ before the move. After the move we have $(\rightarrow, , L)_i(\rightarrow, , 1 - L)_j \ldots {}_k(\rightarrow, , 1 - L)$. So p_i decreases from p to 0 and $p_j, ..., p_k$ remain unchanged. Let l be the immediate successor of k. p_l may increase from 0 to q; however, $q < p$ because l is a successor of i, so G_5 decreases. ❑

Based on Lemma 5.4 to Lemma 5.7, G decreases monotonically even if more than one processor makes a move concurrently in each computation step. So we can conclude that the second protocol is correct with a distributed demon if N is odd.

6. Conclusion

We have presented two self-stabilizing protocols for orienting uniform rings. By relaxing the assumption that the orientation of a processor is simply a state which looks the same to both its neighbors, we can have a deterministic orientation protocol for rings of arbitrary size with a central demon, and a protocol with a distributed demon but for rings of odd size only.

The orientation variables in this paper may be implemented in the following way. Each processor records its orientation internally as 1 or 2 to correspond to its first or second neighbor. Let processor j be a neighbor of processor i. Processor i needs to know whether processor j is its first neighbor or its second neighbor. This can be done by the hardware easily. Now suppose that processor j is the first neighbor of processor i. If the orientation of processor i is internally recorded as 1 then processor i informs processor j that ${}_i(\rightarrow, ,)_j$ whenever processor j wants to read the state of processor i, otherwise processor i sends ${}_i(\leftarrow, ,)_j$ to j.

Israeli and Jalfon have proposed a randomized ring orientation protocol with a distributed demon for rings of arbitrary size [IJ93]. We can also have such a randomized ring orientation protocol by removing the label variables from the M protocol and replacing the rules R1.1 and R1.2 with the following rules:

R1.1: If $_i(\rightarrow, n)(\leftarrow, n)$ then $_i(\rightarrow, y)(\leftarrow, n)$;

R1.2: If $_i(\rightarrow, y)(\leftarrow, y)$ then $_i(\rightarrow, Random(y, n))(\leftarrow, y)$;

The function $Random(y, n)$ is a function which randomly returns the value y or n. That is, we can use randomization to break the symmetry if all confrontations have the form $(\rightarrow, y)(\leftarrow, y)$.

The problem of orienting odd rings was also discussed by Hoepman [Hoepman94]. His solution is a modified version of the randomized protocol of Israeli and Jalfon. Both protocols assumed the link-register model and consisted of two phases. The first phase broke symmetry between every pair of neighboring processors by directing the edge connecting them. The second phase oriented the ring whose edges are directed. The first phase of the protocol of Israeli and Jalfon is a randomized protocol, while that of Hoepman is a deterministic one. On the other hand, there is no clear boundary between the M protocol and the L protocol in Section 5. In fact, the L protocol never deadlocks even when the ring is oriented. Another difference is our protocol uses less space. The protocol of Hoepman used five binary variables and one three-state variable for each link. Our protocol uses only three binary variables for each node.

Acknowledgment

The authors would like to thank the anonymous referees for their invaluable suggestions and pointing out the presence of [Hoepman94].

References

[ASW88] Attiya, H., Snir, M., and Warmuth, M.K. "Computing on an Anonymous Ring," *J. ACM 35*, 4 (Oct. 1988), pp. 845-875.

[BP89] Burns, J.E. and Pachl, J. "Uniform Self-Stabilizing Rings," *ACM Trans. Program. Lang. Syst. 11*, 2 (Apr. 1989), pp. 330-344.

[Dijkstra74] Dijkstra, E.W. "Self-Stabilizing Systems in Spite of Distributed Control," *Commun. ACM 17*, 11 (Nov. 1974), pp. 643-644.

[Huang93] Huang, S.T. "Leader Election in Uniform Rings," *ACM Trans. Program. Lang. Syst. 15*, 3 (July 1993), pp. 563-573.

[Hoepman94] Hoepman, J.H. "Uniform Deterministic Self-Stabilizing Ring-Orientation on Odd-Length Rings," *Proc. 8th Workshop on Distrib. Algorithms* (1994), pp. 265-279..

[IJ90] Israeli, A. and Jalfon, M. "Self-Stabilizing Ring Orientation," *Proc. 4th Workshop on Distrib. Algorithms* (1990), pp. 1-15.

[IJ93] Israeli, A. and Jalfon, M. "Uniform Self-Stabilizing Ring Orientation," *Information and Computation 104* (1993), pp. 175-196.

[Kessels88] Kessels, J.L.W. "An Exercise in Proving Self-Stabilization with a Variant Function," *Inf. Process. Lett. 29* (Sep. 1988), pp. 39-42.

[SP88] Syrotiuk, V. and Pachl, J. "A Distributed Ring Orientation Algorithm," *Proc. 2nd Workshop on Distrib. Algorithms* (1988), also in *Lecture Notes on Computer Science No. 312*, pp. 332-336, Springer-Verlag.

Paper Number 17

An Optimal Self-Stabilizing Algorithm for Mutual Exclusion on Bidirectional Non Uniform Rings

Joffory Beauquier and Oliver Debas

An Optimal Self-stabilizing Algorithm for Mutual Exclusion on Bi-directional non Uniform Rings *

Joffroy Beauquier (jb@lri.fr) and Olivier Debas (debas@lri.fr)
LRI - Université de Paris-SUD

Abstract

In 1974, Dijkstra introduced the self-stabilization problem and presented a self-stabilizing algorithm achieving mutual exclusion on a bi-directional non uniform ring. The complexity study of this algorithm has never been made. In this paper, we reveal the lower bound for the worst case stabilization time for any algorithm which runs under the same hypothesis as Dijkstra's algorithm. The lower bound is $\Omega(n^2)$ where n is the number of processes in the ring. Furthermore, we present a new algorithm which runs under the same hypothesis as Dijkstra's algorithm. Since our algorithm has a $O(n^2)$ worst case stabilization time, the lower bound is reached.

Key Words Mutual Exclusion, Ring, Worst Case Stabilization Time, Optimality.

1 Introduction

Starting-point of research on self-stabilization took place in 1974, when Dijkstra [DIJ74] proposed the concept of self-stabilization and presented three algorithms achieving self-stabilizing mutual exclusion on non uniform bi-directional rings (a process can communicate with its two neighbors and all processes do not execute the same program). These three algorithms differ by the number of states used by each process.

With respect to Dijkstra's first algorithm, Chang, Gonnet and Rotem [CGR87] proved that its expected stabilization time is $\Theta(n^2)$. Ghosh [GHO93] proposed a nearby algorithm of Dijkstra's second algorithm. Worst case stabilization time of these two algorithms is $\Theta(n^2)$.

In [DIJ86], Dijkstra presented a proof of correctness for his third algorithm. Kessel [KES88] proposed an elegant proof based on variant functions for the same algorithm. Tchuente [TCH81] analyzed the three Dijkstra's algorithms. In particular, he pointed out that the third algorithm has a $\Omega(n^2)$ worst case stabilization time. Surprisingly, no exact result on worst case stabilization time has been published. The reason for this is perhaps that Dijkstra's algorithm does not monotonically converge towards a stabilized state. Some punctual bursts can momentarily lead it far from its goal.

In this paper, we propose a nearby algorithm of Dijkstra's third algorithm that does not have this above drawback and we present an analysis of its $O(n^2)$ worst case stabilization time. Moreover we are interested in the algorithms which run under Dijkstra's hypothesis. These hypothesis are the following. The network topology is a non uniform bi-directional ring. This n sized ring includes $n-2$ uniform processes and two immediate neighbored distinguished processes. All processes are three states machines. Communications

*This work was supported by the MEP projet.

between processes operate by direct reading of the neighbors' state. Each process is provided with rules. A process is said to own a privilege if one of its rule guards is true. We prove that for any algorithm running under Dijkstra's hypothesis, $\Omega(n^2)$ is the best that can be done and then that our solution is optimal for the stabilization time.

Dijsktra expressed his algorithm with states. One can remark that in his algorithm, the rules involve only the difference of two immediate neighbors' states. For a given orientation of the ring, we call *gaps* these differences. The consequence is that Dijsktra's algorithm can be expressed with *gaps*. In the rest of the paper, we will express algorithms with the notion of *gap*.

This paper is organized as follows. We begin with some definitions and notations in section 2. In section 3, we present our algorithm. Section 4 contains the analysis of worst case self-stabilization time. The lower bound of worst case self-stabilization time of algorithms which run under the same hypothesis as Dijkstra's algorithm is produced in section 5. Finally, in section 6, we discuss about our work.

2 Definitions and Notations

Definition 1 System. *A system is a pair $S = (C, \rightarrow)$ in which C is a set of configurations and $\rightarrow$ is a binary relation on C. An execution of S is a maximal sequence $E = (\gamma_0, \gamma_1, \gamma_2 \cdots)$ such that $\forall i \geq 0, \gamma_i \rightarrow \gamma_{i+1}$.*

Burns and Pachl [BP89] adopted a formal definition of self-stabilizing mutual exclusion which is given below and which will be used in the sequel of the paper.

Definition 2 Self-stabilizing Mutual exclusion - [BP89]. *A system $S = (C, \rightarrow)$ is self-stabilizing if and only if there is a subset $L \subseteq C$, called the legitimate configurations of S, such that the following conditions are satisfied.*

1. *For every configuration l in C, there exists a configuration l' in C such that $l \rightarrow l'$ (No deadlock).*
2. *For every configuration l in L, every configuration l' such that $l \rightarrow l'$ is in L (Closure of L).*
3. *Every infinite computation of S contains a configuration in L (No livelock).*
4. *For every configuration l in L, exactly one process has the privilege (Mutual exclusion).*
5. *For every process, every infinite computation consisting of configurations in L contains an infinite number of steps completed by this process (Fairness).*

Definition 3 Stabilization Time (or delay) of a system. *Stabilization time of a self-stabilizing system is the minimal number of steps which must be executed by the set of processes in order to reach a legitimate configuration.*

Notation 1 SSME. *We call $SSME$ algorithm an algorithm achieving Self-Stabilizing Mutual Exclusion under the Dijkstra's hypothesis which respects the Burns and Pachl's definition.*

Notation 2 . *We note $\mathcal{G}$ the set $\{0, 1, 2\}$.*

3 The Algorithm

Our algorithm achieves self-stabilizing mutual exclusion that guarantees the existence of a unique privilege after a finite number of actions. This algorithm applies to non uniform bi-directional rings with any size n: two adjacent processes *Bottom* and *Top* are distinguished. Other processes than *Bottom* and *Top* are called *Middle*. Communications between processes are bi-directional and operate by direct reading of the neighbors' state. The application of rules is scheduled by a *central daemon*: All processes test in parallel their own guard, then the *central daemon* selects a single process which has a true guard and which can execute a step.

A ring is represented by a chain oriented from *Top* to *Bottom* and including the state of each process which is an integer of $\mathcal{G}$. A process whose rule guard is satisfied is said to hold a *privilege*. A process modifies its state in accordance with the state L of the previous process (left) and/or the state R of the following process (right). Since states belong to the set $\mathcal{G}$, all computations are made modulo 3.

Here are the rules of our algorithm written with the state of processes in the same way as Dijsktra [DIJ74]:

For *Top* : If $L = S + 1 \wedge S = R + 2$ then $S := S + 2$

For *Middle* **processes** : If $L = S + 1 \vee S = R + 2$ then $S := S + 1$

For *Bottom* : If $S = R$ then $S := S + 1$ If $S = R + 2 \wedge L \neq S$ then $S := S + 2$

Another totally equivalent way to express this algorithm is to use the notion of *gap*. In the sequel of the paper, we will use this kind of expression. Let us label the processes from 0 (*Top*) to $n-1$ (*Bottom*). We define the gap between two processes i and $i+1 \bmod n$ as the difference between their respective states: $\forall i \in [0, n-1]$, $g(i) = \text{state}(i) - \text{state}((i+1) \bmod n) \pmod 3$.

By definition, gaps take their value in $\mathcal{G}$. Here are the rules written with gaps:

Top	$1\,2 \rightarrow 2\,1$	(T)	
Middle	$1\,x \rightarrow 0\,(x+1)$	$(M1)$	$1\,0 \rightarrow 0\,1$
		$(M2)$	$1\,1 \rightarrow 0\,2$
		$(M3)$	$1\,2 \rightarrow 0\,0$
	$x\,2 \rightarrow (x+2)\,0$	$(M4)$	$0\,2 \rightarrow 2\,0$
		$(M5)$	$2\,2 \rightarrow 1\,0$
Bottom	$x\,0 \rightarrow (x+2)\,1$	$(B1)$	$0\,0 \rightarrow 2\,1$
		$(B2)$	$1\,0 \rightarrow 0\,1$
		$(B3)$	$2\,0 \rightarrow 1\,1$
	$x\,2 \rightarrow (x+1)\,1$ if $x \neq 0$	$(B4)$	$1\,2 \rightarrow 2\,1$
		$(B5)$	$2\,2 \rightarrow 0\,1$

For each process, the rules are expressed with the "previous" gap and the "following" gap.

The behavior of the algorithm can be described as follows. 2-value gaps move anti-clockwise up to *Top* where they "turn into" a 1-value gap. On the opposite, 1-value gaps move clockwise down to *Bottom* where they "turn into" a 2-value gap. Other gaps, 0-value gaps, constitute *substratum* in which 1 and 2-value gaps move. The number of non null value gaps decrease until only one non null value gap circulates in the ring (excepted $g(n-1)$): then mutual exclusion is achieved.

Now, we prove a lemma whose result will be continually used in the sequel of the paper.

Lemma 1 *The sum of gaps over the ring is null.*

Proof: Let us define state(i) to refer to the state of process i. The sum of gaps is

$$\sum_{i=0}^{n-1} g(i) = (\sum_{i=0}^{n-2} \text{state}(i) - \text{state}(i+1)) + \text{state}(n-1) - \text{state}(0) = 0.$$

□

We do not present a complete proof of the correctness of our algorithm due to lack of space. We only show the lemmas used in the proof and the main ideas for proving each lemma. Let us define a configuration as being the sequence $(g(0), \cdots, g(n-1))$ of n gaps over the ring.

The legitimate configurations set L is defined as the set of configurations in which only one rule is applicable. When the system is in a legitimate configuration, the process which applies the only applicable rule holds the *privilege.*

Lemma 2 *For every configuration, there is at least one applicable rule.*

Proof: One only has to consider the conditions for all rule guards being false and to show there is a contradiction with the necessity that the sum of all the gaps is null. □

Lemma 3 *When the system is in a legitimate configuration, every application of a rule leads to a legitimate configuration.*

Proof: Since only one privilege exists in a legitimate configuration, we can examine all possible cases according to the process which holds the privilege.

If *Top* process holds the privilege, then $g(n-1) = 1$ and $g(0) = 2$. Other gaps are necessarily null. Applying (the only one possible) *Top* rule modifies $g(n-1)$ into 2 and $g(0)$ into 1. Then, the process number 1 holds the privilege. Therefore, the reached configuration is legitimate.

If *Bottom* process holds the privilege, the system is in one of the following configurations:
$c_1 = 00\cdots012$, $c_2 = 20\cdots010$, $c_3 = 00\cdots000$.
Applying *Bottom* rule leads for the first and third case to a configuration in which only the process number $n-2$ holds the privilege and for the second case to a configuration in which only *Top* process holds the privilege.

In the same way, for *Middle* processes, the study of all rules shows that only one process will hold the privilege after the application of a rule. □

Lemma 4 *System reaches a legitimate configuration after a finite number of rule applications.*

Proof: By theorem proved in section 4, the worst case stabilization time is $O(n^2)$. Therefore, the present lemma is a corollary of this theorem. □

Lemma 5 *When the system is in a legitimate configuration, mutual exclusion is achieved.*

Proof: Mutual exclusion takes places when only one process holds the privilege. The set L of legitimate configurations has been defined to obtain this property. □

Lemma 6 *Every infinite computation in L contains an infinite number of steps by each process.*

Proof: In L, after a finite number of rule applications, only the rules (T), $(M1)$, $(B4)$ and $(M4)$ will be applied in a cyclical way in the order of writing these rules. The application of (T) gives the privilege to process number 1. For all $i \in [1, n-2]$, the application of $(M1)$ gives the privilege to process number $i+1$. The application of $(B4)$ gives the privilege to process number $n-2$. Finally, for all $i \in [1, n-2]$, the application of $(M4)$ gives the privilege to process number $i-1$.
Therefore, after a finite number of rule applications, in every sequence of n successive applications, the n processes have held exactly once the privilege. □

4 Analysis of the worst case stabilization time

Now, we present the analysis of the first result of the paper. The following theorem states this result.

Theorem 1 *The worst case stabilization time of our algorithm is $O(n^2)$.*

Before proving the theorem, we establish a few results, presented in the following lemmas.

Lemma 7 *In every execution, rules $(B1)$, $(B2)$ and $(B3)$ are applied at the most once.*

Proof: Application of rules $(B1)$, $(B2)$ and $(B3)$ requires $g(n-1) = 0$. No rule, amongst those which take action on $g(n-1)$ (rules *Top* and *Bottom*), puts $g(n-1)$ to 0. □

Lemma 8 *Between two successive applications of the rule Top, the rule Bottom is applied exactly once.*

Proof: Suppose that the rule *Top* is applied. After that, $g(n-1) = 2$. Rules $(B1)$, $(B2)$ and $(B3)$ can not be applied because $g(n-1) \neq 0$.
Only application of rule $(B4)$ or rule $(B5)$ modifies the value of gap $g(n-1)$ from 2 to 1 in order to allow a further application of rule *Top*. On the other hand, only rule *Top* allows to modify $g(n-1)$ from 1 to 2. Therefore, the rule *Bottom* is applied exactly once before the next application of rule *Top*. □

Lemma 9 *When the system is not yet in a legitimate configuration, between two successive applications of the rule Top, the number of non null value gaps decreases by at least one.*

Proof: Because the system is not stabilized, there is an applicable rule together with the rule *Top*. Since $g(n-1) = 1$, this rule cannot be *Bottom*, then it is a rule *Middle*. So there is a non null value gap $g(i)$ with $i \in [1, n-2]$. After application of rule *Top*, the reached configuration is such that $g(0) = 1$. In order to allow a next application of *Top*, this gap $g(0)$ must be a 2-value gap. Only rules $(M2)$ or $(M3)$ perform this and then rule $(M4)$. Application of rule $(M2)$ decreases of one unity the number of non null value gaps. For the rule $(M3)$, decrease is two. Rule $(M4)$ does not modify the number of non null value gaps. Therefore, the number of non null value gaps decreases at least by one between two successive applications of the rule *Top*. □

Proof: Now we prove the result of the theorem. We construct a function E which associates an integer with each configuration. We determine the maximal and the minimal values of this function and the variation of the value of E after the application of each rule. Then, we will be able to deduce an upper bound to the number of application of each rule.

Let be a configuration $c = (g(0), \cdots g(n-1))$. For all gaps $g(i), i \in [0, n-1]$, we associate the value $e(i) = u(i) + p(i)$ as follows:

if $g(i)=0$ then $u(i)=0$ $p(i)=0$.
if $g(i)=1$ then $u(i)=n+3$ $p(i)=n-2-i$.
if $g(i)=2$ then $u(i)=n+3$ $p(i)=i$ for $i<n-1$, $p(n-1)=-1$.
The value of $E(c)$ is $\sum_{i=0}^{n-1} e(i)$.

One remarks that for all $i \in [0, n-1]$: $e(i) \geq 0$ and $e(i)=0$ if and only if $g(i)=0$. Thus, the minimal value of the function E is $E_{min}=0$. When $E_{min}=0$, the configuration is legitimate. The upper bound $Sup(E)$ of E is obtained when:
$\forall i \in [0, \lceil (n-1)/2-1 \rceil]$, $g(i)=1$,
$\forall i \in [\lceil (n-1)/2-1 \rceil + 1, n-2]$, $g(i)=2$,
$g(n-1)=1$ or 2 indifferently.

Let us remark that the value $Sup(E)$ is not always reached according to the value of n since the sum of all gaps must be null. Let us define $half = \lceil (n-1)/2-1 \rceil$. The following computation gives $Sup(E) = Sup(\sum_{i=0}^{n-1} e(i))$.

$Sup(E) = Sup(\sum_{i=0}^{n-1}(u(i)+p(i)))$.
$Sup(E) = n(n+3) + \sum_{i=0}^{half}(n-2-i) + \sum_{i=half+1}^{n-2}(i) + (-1)$.
Now, we use the relation $(n-1)/2-1 \leq half \leq n/2-1$, then:
$Sup(E) \leq n(n+3) - 1 + (n-2)n/2 - (n-1)/2 \cdot ((n-1)/2-1) + (n-2)(n-1)/2$.
Thus,
$Sup(E) \leq 7n^2/4 + 3n/2 - 3/4$.

Let us compute for each rule, the variation $\delta(E)$ of E when a rule is applied to proceed from configuration $c = (g(0) \cdots g(i)\, g(i+1) \cdots g(n-1))$ to configuration $c' = (g(0) \cdots g'(i)\, g'(i+1) \cdots g(n-1))$. Different values follow:

Rule R	$\delta(E)$
(T)	$n-2$
$(B1)$	$3(n-1)$
$(B2)$	-1
$(B3)$	4
$(B4)$	$n-2$
$(B5)$	$-(2n+1)$
$(M1)$	-1
$(M2)$	$-3(n-1-i)$
$(M3)$	$-(3n+5)$
$(M4)$	-1
$(M5)$	$-3(i+2)$

Since the function E is bounded, if the rules for which $\delta(E) > 0$ are applicable a finite number of times, then the rules for which $\delta(E) < 0$ are also applicable a finite number of times.

According to lemma 9, the rule *Top* can be applied at the most n times before the system enters a legitimate configuration. According to lemma 8, between two successive applications of *Top*, the rules $(B4)$ or $(B5)$ are applied exactly once. According to lemma 7, rules $(B1)$, $(B2)$ or $(B3)$ are applied once at the most. Thus, the rule *Bottom* can be applied at the most $n+2$ times before the system goes in a legitimate configuration.
Consequently, in any execution before the system enters a legitimate configuration, the application of rules *Top* and *Bottom* increases the value of E at the most $n(n-2)+3(n-1)(n+2)$. The lower variation $\delta(E)$

for the rule *Middle* is -1. Hence, rule *Middle* can be applied at the most $(Sup(E) + n(n-2) + 3(n-1)(n+2) - E_{min})$ times. Thus, the polynomial $23n^2/4 + 5n/2 - 27/4$ is an upper bound for the maximum number of applications of rules before the system enters a legitimate configuration. Therefore, worst case stabilization time is $O(n^2)$. □

5 Optimality

Let us call gap algorithm, an algorithm that can be expressed with gaps. As noticed above, Dijkstra's and our algorithm are gap algorithms. We recall the hypothesis on algorithms in which we are interested. The network topology is a non uniform bi-directional ring. This n sized ring includes $n-2$ uniform processes and two immediate neighbored distinguished processes. All processes are three states machines. Communications between processes operate by direct reading of the neighbors' state. Each process is provided with rules. A process is said to own a privilege if one of its rule guards is true.

Under Dijkstra's third algorithm hypothesis, Tchuente [TCH81] has shown that the number of 3 states per process is minimal. In this section, we will prove that any SSME gap algorithm with 3 states per process (in short $SSME - G3$) has a worst case stabilization time that is $\Omega(n^2)$. Under the above hypothesis, the rules *Middle* appearing in an algorithm are in three categories, because the sum of the gaps in the left part of a rule has to be equal (mod 3) to the sum of the gaps in the right part of the rule and the gaps can have three different values. Our purpose is to deal with rules *Middle*, we do not consider rules *Bottom* and *Top*.

$ab \rightarrow cc$	6 rules
$ab \rightarrow ba$	6 rules
$cc \rightarrow ab$	6 rules
for three different values a, b, c in $\mathcal{G}$ such that $a+b = c+c$ (mod 3).	

For instance, $ab \rightarrow cc$ represents all the following rules: $01 \rightarrow 22$, $10 \rightarrow 22$, $02 \rightarrow 11$, $20 \rightarrow 11$, $12 \rightarrow 00$ and $21 \rightarrow 00$. Whenever we consider a particular rule, we note it with greek letters: for instance we note $\alpha\beta \rightarrow \gamma\gamma$.

The proof of optimality is twofold. Firstly, we will prove that any algorithm which does not include at least one rule *Middle* which belongs to the second category $ab \rightarrow ba$ is either not $SSME - G3$ or has a worst case stabilization time which is $\Omega(n^2)$ (proposition 1). For doing that, we study different well chosen cases in which there is no rule *Middle* of the second category. In the majority of these cases, we show that the assumption of convergence, fairness or closure is falsified and for the rest, we show that the worst case stabilization time is $\Omega(n^2)$. Secondly, we consider a $SSME - G3$ algorithm that includes a rule *Middle* of the category $ab \rightarrow ba$ and we present an execution whose length is $\Theta(n^2)$ and which leaves the system in an illegitimate configuration (proposition 2).

We define *NO_INVERS* as being the predicate: "the algorithm does not include any rule *Middle* of the second category or any couple of rules *Middle* whose successive activation has the same effect that a rule of the second category". Suppose that this predicate is true. Now, we examine the two following cases:

- There exist α, β and γ in $\mathcal{G}$ such that the rules $\gamma\gamma \rightarrow \alpha\beta$ and $\alpha\beta \rightarrow \gamma\gamma$ appear in the algorithm.
- There exist α, β and γ in $\mathcal{G}$ such that the rules $\gamma\gamma \rightarrow \alpha\beta$ and $\beta\alpha \rightarrow \gamma\gamma$ appear in the algorithm.

In the first case, one can activate the two rules successively n^3 times. This assures that the stabilization time is $\Omega(n^2)$. Remark that such an algorithm cannot be $SSME - G3$ under an unfair daemon since the convergence cannot be achieved. The second case is forbidden since the predicate is supposed to be true.

The previous result leads us to assume that:

Assumption 1

$$\forall \gamma \in \mathcal{G}, \gamma\gamma \rightarrow \alpha\beta \Longleftrightarrow \neg(\alpha\beta \rightarrow \gamma\gamma \vee \beta\alpha \rightarrow \gamma\gamma).$$

In the following lemmas, the predicate NO_INVERS is assumed to be true and the assumption 1 is done. These lemmas are used to prove the proposition 1 which is the first step of our proof.

The previous assumption states that under the hypothesis NO_INVERS the rules of the first category and the rules of the third category are mutually exclusive in that a rule $\gamma\gamma \rightarrow \alpha\beta$ forbids the two rules $\alpha\beta \rightarrow \gamma\gamma$ and $\beta\alpha \rightarrow \gamma\gamma$. This fact leads to distinguish four cases. Each of these cases are defined by the number (0-3) of rules of the third category which are included in the algorithms. Since the algorithm is deterministic, it cannot include more than three rules of the third category. One can remark that the smaller the number of third category rules is, the more complex the case is. The following lemmas deal with each of these cases.

Lemma 10 *Under the hypothesis NO_INVERS, every algorithm which includes all the rules $cc \rightarrow ab$ (where c takes all values in $\mathcal{G}$) is not $SSME - G3$.*

Proof: According to the assumption 1, there is no rule $ab \rightarrow cc$. All possible privileges of *Middle* processes are given by the couples cc where c takes all the three values in $\mathcal{G}$. Let us take a ring size n such that $n \geq 8$.
If $n - 1 = 0 \pmod 2$, let us consider the following configurations where α, β and γ have any value in $\mathcal{G}$ such that $\alpha \neq \beta \neq \gamma$:

$$g_0\beta g_2 g_3 \alpha\beta \cdots \alpha\beta g_{n-3} g_{n-2} g_{n-1}.$$

g_0 is valued with α, $g_{n-3}g_{n-2}$ is valued with $\alpha\beta$ or $\gamma\alpha$, g_2g_3 is valued with the couples $\alpha\beta$, $\alpha\gamma$, $\gamma\beta$. In these configurations, no *Middle* process holds a privilege. Thus, one of the *Top* and *Bottom* processes must hold a privilege. Let us remark that g_0 and g_{n-2} can take every value in $\mathcal{G}$. Furthermore, the sum of the values of the couple g_2g_3 can take every value in $\mathcal{G}$ independently of the value of g_0 and g_{n-2}. This assures that g_{n-1} can also take all the values in $\mathcal{G}$ independently of the value of g_0 and g_{n-2}. Then, for any value of g_0, g_{n-2}, g_{n-1}, one of the *Top* and *Bottom* processes must hold a privilege. Thus, the convergence cannot be achieved (livelock is certain).
If $n - 1 = 1 \pmod 2$, let us consider the following configurations:

$$g_0\beta g_2 g_3 \alpha\beta \cdots \alpha\beta g_{n-2} g_{n-1}.$$

g_0 is valued with α, g_{n-2} is valued with α or γ, g_2g_3 is valued with the couples $\alpha\beta$, $\alpha\gamma$, $\gamma\beta$. In the same way than in the previous case, for all values of g_0, g_{n-2}, g_{n-1}, one of the *Top* and *Bottom* processes must hold a privilege. □

Lemma 11 *Under the hypothesis NO_INVERS, every algorithm which includes two rules $cc \rightarrow ab$ (where c takes two values in $\mathcal{G}$) is not $SSME - G3$.*

Proof: According to the assumption 1, only one rule of the category $ab \rightarrow cc$ is possible. Thus, there exist two unique different values α and β in $\mathcal{G}$ such that all possible privileges of *Middle* processes are given by the couples $\alpha\alpha$, $\beta\beta$, $\alpha\beta$, $\beta\alpha$.
Let us consider for $n \geq 4$, the following configurations where γ has a value different from α and β:

$$g_0\gamma\gamma\cdots\gamma g_{n-2}g_{n-1}.$$

g_0 and g_{n-2} take any value in $\mathcal{G}$. For every value of these gaps (which means in all further configurations), all the *Middle* processes never hold a privilege. Therefore, the fairness cannot be achieved. □

Lemma 12 *Under the hypothesis NO_INVERS, every algorithm which includes only one rule $cc \rightarrow ab$ (where c takes one value in $\mathcal{G}$) is not $SSME - G3$.*

Proof: Once again according to the assumption 1, there exist three unique different values α, β and γ in $\mathcal{G}$ such that all possible privileges of *Middle* processes are given by the couples $\gamma\gamma$, $\alpha\gamma$, $\gamma\alpha$, $\beta\gamma$, $\gamma\beta$. All possible *Middle* rules are the following:

- $\gamma\gamma \rightarrow \alpha\beta$
- $\gamma\gamma \rightarrow \beta\alpha$
- $\alpha\gamma \rightarrow \beta\beta$
- $\gamma\alpha \rightarrow \beta\beta$
- $\beta\gamma \rightarrow \alpha\alpha$
- $\gamma\beta \rightarrow \alpha\alpha$

Let us consider for $n \geq 5$, the following configurations:

$$g_0\alpha\alpha\cdots\alpha\alpha g_{n-2}g_{n-1}.$$

g_0 and g_{n-2} take any value in $\mathcal{G}$. All the *Middle* processes number i, $1 < i < n-2$, do not hold a privilege. These processes can hold a privilege iff one of the processes number 1 or $n-2$ transmits to them the privilege. Consider the process number 1. Whatever the value of g_0 may be, g_1 moves to α or β. It does not give a privilege to *Middle* processes number i, $1 < i < n-2$. Thus, only the rule $\gamma\beta \rightarrow \alpha\alpha$ can modify the value of g_1, the new value is α. Therefore, the value of g_1 is always in $\{\alpha, \beta\}$ and the value of gaps g_i, $1 < i < n-2$, is always equal to α. The same argument can be given about the gap g_{n-3} whose value is always in $\{\alpha, \beta\}$. In conclusion, whatever the action of other processes may be, all the *Middle* processes number i, $1 < i < n-2$, never hold a privilege. Therefore, the fairness cannot be achieved. □

Lemma 13 *Under the hypothesis NO_INVERS, every algorithm which includes no rule $cc \rightarrow ab$ and is $SSME - G3$ has a $\Omega(n^2)$ worst case stabilization time.*

Proof: All possible privileges of *Middle* processes are given by the couples ab where a and b take any two different values in $\mathcal{G}$. Let us consider that there are two different values α and β in $\mathcal{G}$ such that the algorithm does not include the rules $\alpha\beta \rightarrow \gamma\gamma$ and $\beta\alpha \rightarrow \gamma\gamma$. Hence, all possible privileges of *Middle* processes are given by the couples $\alpha\gamma$, $\gamma\alpha$, $\beta\gamma$ and $\gamma\beta$. In the same way that in the lemma 12, we can prove that the fairness cannot be achieved by considering the following configuration:

$$g_0\alpha\alpha\cdots\alpha\alpha g_{n-2}g_{n-1}.$$

This result leads us to state that a $SSME - G3$ algorithm must include at least three rules. Supposing that the algorithm includes three rules, there are only two cases:

1. There exist α, β and γ in $\mathcal{G}$ such that the three rules are $\alpha\beta \rightarrow \gamma\gamma$, $\beta\gamma \rightarrow \alpha\alpha$ and $\gamma\alpha \rightarrow \beta\beta$
2. There exist α, β and γ in $\mathcal{G}$ such that the three rules are $\alpha\beta \rightarrow \gamma\gamma$, $\beta\gamma \rightarrow \alpha\alpha$ and $\alpha\gamma \rightarrow \beta\beta$

In the first case, let us take the following initial configuration in which no *Middle* process holds a privilege:

$$g_0 g_1 \alpha g_3 g_4 \gamma \cdots \gamma\beta g_{n-3} g_{n-2} g_{n-1}.$$

The couple $g_0 g_1$ can take the values $\alpha\alpha$, $\beta\beta$ and $\gamma\beta$. The couple $g_{n-3}g_{n-2}$ can take the values $\alpha\alpha$, $\beta\beta$ and $\alpha\gamma$. Finally, the couple $g_3 g_4$ can take the values $\alpha\alpha$, $\alpha\gamma$ and $\gamma\gamma$. Since the sum of the couple $g_3 g_4$ can take any value in $\mathcal{G}$, the gap g_{n-1} can also take any value in $\mathcal{G}$. In consequence, whatever the value of the gaps g_0, g_{n-2}, g_{n-1} may be, one of the *Top* and *Bottom* processes must hold a privilege. Therefore, the convergence cannot be achieved.

In the second case, let us take the following initial configuration in which no *Middle* process hold a privilege:

$$\alpha\alpha \cdots \alpha g_{n-1}.$$

In order to prevent deadlock, one or two of the *Top* and *Bottom* processes must hold a privilege. Let us remark that if *Top* holds a privilege, this privilege cannot be transmitted to the *Middle* process number 1 since there is no couple $\delta\alpha$ with $\delta \in \mathcal{G}$ which gives a privilege to a *Middle* process. Consequently, we activate *Top* until the privilege it holds disappears or is transmitted to *Bottom*. Like that, the system reaches a legitimate configuration. Again, we make *Bottom*, and if necessary *Top*, active until the privilege is transmitted to the *Middle* process number $n-2$. This must be possible, otherwise the fairness cannot be achieved. The system reaches the following legitimate configuration:

$$g_0 \alpha \cdots \alpha g_{n-2} g_{n-1}.$$

Since the *Middle* process number $n-2$ holds the privilege, $g_{n-2} = \beta$ or γ. In the two cases, the following sequences show that a second privilege is created (each privilege is represented by an overlined couple of gaps).

$$\begin{array}{ll} g_0\alpha\cdots\alpha\alpha\overline{\alpha\beta}g_{n-1} & g_0\alpha\cdots\alpha\alpha\overline{\alpha\gamma}g_{n-1} \\ g_0\alpha\cdots\alpha\overline{\alpha\gamma}\gamma g_{n-1} & g_0\alpha\cdots\alpha\alpha\overline{\alpha\beta}\beta g_{n-1} \\ g_0\alpha\cdots\overline{\alpha\beta\beta\gamma}g_{n-1} & g_0\alpha\cdots\alpha\overline{\alpha\gamma}\gamma\beta g_{n-1} \\ & g_0\alpha\cdots\overline{\alpha\beta\beta\gamma}\beta g_{n-1} \end{array}$$

Therefore there exists no $SSME - G3$ with only three rules under the hypothesis of the lemma.

Since there is no $SSME - G3$ algorithm with less than four rules, let us consider the algorithms which include at least four rules. Without loss of generality, amongst all possible privileges, the three following arrangements are possible:

- $\alpha\beta$, $\beta\gamma$, $\gamma\alpha$ and $\beta\alpha$.
- $\alpha\beta$, $\gamma\beta$, $\gamma\alpha$ and $\beta\alpha$.
- $\alpha\beta$, $\beta\gamma$, $\alpha\gamma$ and $\beta\alpha$.

To be convinced, one just have to consider a three nodes directed graph (α, β, γ) and to put four arcs between nodes which constitute a privilege.

$\diamond$ In the first case, we are going to show that there is a sequence of actions whose length is $\Theta(n^2)$ and which leaves the system in an illegitimate configuration. For $n \geq 18$, let us take the following initial configuration:

$$\underbrace{\gamma\beta\beta\cdots\gamma\beta\beta\gamma\beta\langle\beta}_{3\lfloor\frac{n}{6}\rfloor}\underbrace{\gamma\rangle\gamma\cdots\gamma}_{n-2-3\lfloor\frac{n}{6}\rfloor}\alpha g_{n-1}.$$

The couple between angles is the next one to be moved. Since more than one *Middle* process hold a privilege, the configuration is not legitimate. The application of the rule $\beta\gamma \rightarrow \alpha\alpha$ leads to the following configuration which contains the same number of privileges:

$$\underbrace{\gamma\beta\beta\cdots\gamma\beta\beta\gamma\langle\beta\alpha\rangle}\underbrace{\alpha\gamma\cdots\gamma}\alpha g_{n-1}.$$

Then, the application of the rules $\beta\alpha \rightarrow \gamma\gamma$ and then $\gamma\alpha \rightarrow \beta\beta$ leads to the following illegitimate configuration:

$$\underbrace{\gamma\beta\beta\cdots\gamma\beta\beta\gamma\gamma\beta}\underbrace{\langle\beta\gamma\rangle\gamma\cdots\gamma}\alpha g_{n-1}.$$

After a total of $3(n-2-3\lfloor\frac{n}{6}\rfloor)$ rule applications, the system reaches the following configuration which still contains the same number of privileges:

$$\underbrace{\gamma\beta\beta\cdots\gamma\beta\langle\beta\gamma\rangle\gamma\gamma}\underbrace{\gamma\cdots\gamma\gamma\gamma\beta}\alpha g_{n-1}.$$

We can repeat $\lfloor\frac{n}{6}\rfloor$ times the previous sequence of rule applications. The system reaches the following configuration which still contains more than one privilege:

$$\underbrace{\gamma\cdots\gamma}\underbrace{\gamma\beta\beta\cdots\gamma\beta\beta}\alpha g_{n-1}.$$

In total, there were $\lfloor\frac{n}{6}\rfloor(n-2-3\lfloor\frac{n}{6}\rfloor)$ rule applications and the system never reached a legitimate configuration. Therefore, if the algorithm is $SSME-G3$, the worst case stabilization time is $\Omega(n^2)$.

$\diamond$ In the third case, we take the same initial configuration as in the first case:

$$\underbrace{\gamma\beta\beta\cdots\gamma\beta\beta\gamma\beta\langle\beta}_{3\lfloor\frac{n}{6}\rfloor}\underbrace{\gamma\rangle\gamma\cdots\gamma}_{n-2-3\lfloor\frac{n}{6}\rfloor}\alpha g_{n-1}.$$

Let us apply the rule $\beta\gamma \rightarrow \alpha\alpha$ and then the rule $\beta\alpha \rightarrow \gamma\gamma$. After, the application of rule $\alpha\gamma \rightarrow \beta\beta$ leads to the following configuration:

$$\underbrace{\gamma\beta\beta\cdots\gamma\beta\beta\gamma\gamma\gamma}\underbrace{\beta\langle\beta\gamma\rangle\gamma\cdots\gamma}\alpha g_{n-1}.$$

The fragment $\gamma\beta\beta$ has moved of one more position to the right than in the first case. Thus, the length of the whole sequence is $\lfloor\frac{n}{6}\rfloor\lfloor\frac{(n-2-3\lfloor\frac{n}{6}\rfloor)}{2}\rfloor$. Therefore, we obtain the same result than in the first case.

$\diamond$ The second case is equivalent to the third case if one take the following initial configuration

$$\underbrace{\gamma\cdots\gamma\langle\gamma}_{n-2-3\lfloor\frac{n}{6}\rfloor}\underbrace{\beta\rangle\beta\gamma\beta\beta\gamma\cdots\beta\beta\gamma}_{3\lfloor\frac{n}{6}\rfloor}\alpha g_{n-1}$$

and if one activates the rules $\gamma\beta \rightarrow \alpha\alpha$, $\alpha\beta \rightarrow \gamma\gamma$ and then $\gamma\alpha \rightarrow \beta\beta$. □

We have studied all possible algorithms which do not include a rule $ab \rightarrow ba$. For all these algorithms, either at least one property that must be owned by a $SSME-G3$ algorithm is not satisfied, or there is a sequence of actions which leaves the system in an illegitimate configuration and whose length is $\Omega(n^2)$. This result is stated in the following proposition.

Proposition 1 *Every $SSME - G3$ algorithm which respects the hypothesis NO_INVERS has a worst case stabilization time which is $\Omega(n^2)$.*

Now, for all algorithms which do not respect the hypothesis NO_INVERS, we present a sequence of rule applications whose length is $\Omega(n^2)$ and which leaves the system in an illegitimate configuration.

Proposition 2 *Every $SSME - G3$ algorithm which does not respect the hypothesis NO_INVERS has a worst case stabilization time which is $\Omega(n^2)$.*

Proof: According to the hypothesis NO_INVERS, the $SSME - G3$ algorithm includes a rule $\alpha\beta \rightarrow \beta\alpha$, $\alpha \neq \beta$, $\alpha, \beta \in \mathcal{G}$ or a couple of rules whose successive activation has the effect that the previous rule. In the second case, there exist α, β and γ in $\mathcal{G}$ ($\alpha \neq \beta \neq \gamma$) such that the algorithm includes the two following rules: $\alpha\beta \rightarrow \gamma\gamma$ and $\gamma\gamma \rightarrow \beta\alpha$. The following execution can be made in the first case or the second case (without considering the intermediate configurations resulting of the activation of the first rule $\alpha\beta \rightarrow \gamma\gamma$). For $n \geq 8$, we consider the following initial configuration:

$$\underbrace{\alpha\alpha\cdots\alpha\alpha}_{n-1-2\lfloor\frac{n}{4}\rfloor}\underbrace{\langle\alpha\beta\rangle\alpha\beta\cdots\alpha\beta}_{2\lfloor\frac{n}{4}\rfloor}g_{n-1}.$$

The couple between angles is the next one to be moved. Since more than one *Middle* process hold a privilege, the configuration is not legitimate. The application of the rule $\alpha\beta \rightarrow \beta\alpha$ leads to the following configuration which contains the same number of privileges:

$$\underbrace{\alpha\alpha\cdots\alpha\langle\alpha}\underbrace{\beta\rangle\alpha\alpha\beta\cdots\alpha\beta}g_{n-1}.$$

After a total of $n-1-2\lfloor\frac{n}{4}\rfloor$ applications of the rule, the system reaches the following configuration which still contains the same number of privileges:

$$\underbrace{\alpha\beta\alpha\cdots\alpha\alpha}\underbrace{\alpha\alpha\langle\alpha\beta\rangle\cdots\alpha\beta}g_{n-1}.$$

We can repeat $\lfloor\frac{n}{4}\rfloor$ times the previous sequence of rule applications. The system reaches the following configuration which still contains more than one privilege:

$$\underbrace{\alpha\beta\cdots\alpha\beta}\underbrace{\alpha\cdots\alpha}g_{n-1}.$$

In total, there were $\lfloor\frac{n}{4}\rfloor(n-1-2\lfloor\frac{n}{4}\rfloor)$ rule applications and the system never reached a legitimate configuration. Therefore, the worst case stabilization time is $\Omega(n^2)$. □

Now we can state the following theorem which is the main result of our paper.

Theorem 2 *The lower bound of worst case stabilization time of $SSME - G3$ algorithms is $\Omega(n^2)$.*

Proof: The proof of the theorem is given by the results of the propositions 1 and 2. □

The algorithm we described in the section 3 is $SSME - G3$. Then we can apply the result of the theorem 2 and claim that this algorithm is optimal.

6 Conclusion and further researches

In this this paper, we used the notion of gaps. Using gaps allowed a better understanding and intuition of the global running of the system. With using gaps, we proved that a lower bound of the worst case stabilization time is $\Omega(n^2)$. Furthermore, we presented a nearby algorithm of Dijkstra's algorithm which reaches this bound.

The difference between the rules expressed with gaps and the rules expressed with states is significant when the rule guards include a test on the very process state. Therefore, any $SSME$ algorithm whose rules are expressed with states can be written with rules expressed with gaps and a test on the process state. With respect to algorithms running under the same hypothesis as Dijkstra's third algorithm, we conjecture that, for algorithms whose rules are expressed with states, the worst case stabilization time is still $\Omega(n^2)$.

Acknowledgements. We would like to thank the referees for remarks which helped us to correct and to simplify the analysis.

References

[BP89] "Uniform Self-Stabilizing Rings" J.E. Burns and J. Pachl. ACM TPLS, vol 11, n 2, 4/89 p. 330-344 (1989).

[CGR87] "On the costs of self-stabilization" E.J.H. Chang, G.H. Gonnet and D. Rotem Information Processing Letters 24, p. 311-316 (1987).

[DIJ74] "Self-stabilizing systems in spite of distributed control." E.W. Dijkstra. Comm ACM 17 p. 643-644 (1974).

[DIJ86] "A belated proof of self-stabilization" E.W. Dijkstra. Distributed Computing 1,1 p.5-6 (1986).

[GHO93] "An Alternative Solution to a Problem on Self-Stabilization" S. Ghosh. ACM TPLS, vol 15, n 4, 09/93, p. 735-742 (1993).

[KES88] "An exercise in proving self-stabilization with a variant fonction." J.L.W. Kessels. Information Processing Letters, 29 , p. 39-42 (1988).

[TCH81] "Sur l'auto-stabilisation dans un reseau d'ordinateurs" M. Tchuente. R.A.I.R.O. Informatique theorique, vol 15, n 1, p. 47-66 (1981).

Paper Number 18

A Highly Safe Self-Stabilizing Mutual Exclusion Algorithm

I-Ling Yen and Farokh B. Bastani

A Highly Safe Self-Stabilizing Mutual Exclusion Algorithm

I-Ling Yen
Department of Computer Science
A-714 Wells Hall
Michigan State University
East Lansing, MI 48824-1027

Email: yen@cps.msu.edu
Phone: (517)353-3542

Farokh B. Bastani
Department of Computer Science
University of Houston
Houston, TX 77204-3475

Email: FBastani@uh.edu
Phone: (713)743-3354

Abstract

Conventional self-stabilizing algorithms cannot be safely used for safety-critical systems. This is due to the vulnerable period that is present in self-stabilizing systems after a failure occurs but before the system stabilizes. In this paper, we consider a highly safe self-stabilizing system where the vulnerability problem is tackled. The design principles we use to achieve this goal include sobriety test and processor specialization. Sobriety test is used to prevent the system from performing incorrect action when the system state may be faulty. Specialization disables individual processors from making faulty moves.

We have developed a self-stabilizing mutual exclusion algorithm that guarantees mutual exclusion with a very high probability even in the presence of failures. It also satisfies all the properties that are required for a self-stabilizing algorithm.

1 Introduction

The rapid advances in technology will soon facilitate high quality services that are provided by thousands of computers and devices linked together via high performance optical communication networks. An effective method is required to coordinate these computer systems and special purpose devices to enhance the performance as well as the safety, reliability, and availability of these services. Due to the large number of devices in the system, it is essential to use decentralized coordination to minimize the overhead and make the system more robust. The method of self-stabilizing systems, first proposed by Dijkstra [14], is an ideal approach for solving such large-scale control problems. In a self-stabilizing system, the system achieves global control by communicating with a limited number of neighboring processors and by making localized decisions. Also, starting from any initial state, legitimate or illegitimate, the system is guaranteed to self-stabilize, i.e., converge to a legitimate state, within a finite time. Further, due to the localized control, the code is generally simple and elegant and, thus, yields programs that are easy to verify.

However, the self-stabilization method generally suffers from one well-known side-effect, namely, the possibility of vulnerable periods when the system is in an illegitimate state due to certain transient failures. In these cases, a self-stabilizing system may cause incorrect operation of the system which is not acceptable for safety-critical systems. As a specific example, consider an energy distribution system that consists of some generators and a large number of electrical devices. With uncoordinated device activations, it is possible to have wide swings in energy usage. To avoid having to prepare for the peak usage by maintaining a high reserve capacity which may have negative economical and ecological impacts, it is possible to coordinate the active periods of some devices to reduce the variance in energy demand. Due to the large number of devices involved, the self-stabilizing method is an obvious candidate for coordinating the activation of the devices. However, a vulnerable period after some transient failures can cause the system to become uncoordinated and, hence, create transient instabilities in the energy distribution network which may result in the failure of safety-critical devices, such as security systems and medical devices, leading to the loss of life and property.

In this paper, we develop an approach for effectively dealing with the vulnerability of self-stabilizing systems. The approach we take is to design the system so that only a very limited number of states in the set of possible system states is legitimate. Thus, the moment the

system steps into an illegitimate state, the errors can be detected with a high probability through local checks and the detecting processor can move to a safe state. We propose two general principles for designing self-stabilizing systems that have no vulnerable periods (with a high probability). Also, a detailed algorithm for mutual exclusion is presented that guarantees mutual exclusion with an arbitrarily high probability even with failures.

In the next section, we review self-stabilization and describe our model. Section 3 presents a safe self-stabilizing mutual exclusion algorithm. Section 4 states the conclusion and outlines some future research directions.

2 Background and System Model

The concept of self-stabilizing systems was first proposed by Dijkstra [14]. In physics, an object is said to be in a stable state if the forces acting on it tend to restore it to its original state whenever it is perturbed from its position by a temporary external force. Similarly, if a computer system (regardless of its initial state) can eventually converge to and remain in a legitimate state then it is said to be a self-stabilizing system. When transient failures occur and bring the system into an illegitimate state, a self-stabilizing system will reach a legitimate state after a finite number of transitions.

Dijskra illustrated the concept of self-stabilizing system with a cyclic relaxation algorithm [13] that allowed points on a unit circle to distribute themselves evenly along the circumference irrespective of the starting state. He also developed three self-stabilizing protocols for assuring mutual exclusion in a ring network where processes could only communicate with nearest neighbors [14]. These preliminary algorithms required nondeterministic execution and special processes ("top" and "bottom") that behaved differently from the other processes. Kruijer extended self-stabilization to tree networks [22] and LeLann applied it to develop a self-stabilizing token passing protocol [25]. In the keynote address at the 3rd PODC, Lamport bemoaned the fact that self-stabilization had not been pursued actively in the ensuing years [24]. Subsequently, Brown, Gouda, and Wu developed a self-stabilizing algorithm that does not require nondeterministic control and, hence, is viable for implementation using delay sensitive circuits [7], and Burns and Pachl developed an algorithm for a prime number of processes that dispenses with the need for special processes [9]. There has also been some research on general methods of achieving self-stabilizing systems [3] [4] [5] [21] and on incorporating self-stabilization in various applications [1] [2] [8] [11] [15] [16] [18] [19] [29]. Some

recent work has also dealt with self-stabilizing algorithms that can tolerate permanent failures — a specific example is the dynamical system of Dolev, Israeli, and Moran [15]. An interesting self-stabilizing algorithm that tolerates permanent malicious processor failures [23] in a modified ring network was developed by Bianchini and Buskens [6] [10]. Also, Line and Ghosh have developed a method for incorporating crash tolerance within self-stabilizing algorithms using program composition [26]. Recently, there has been a rapidly growing awareness and interest in self-stabilizing systems. For example, Iyer, Trivedi, and Goldberg state that fault tolerance may be viewed as the ability of a system to withstand a destabilizing shock [20]. Also, Debest exemplifies the potential of the self-stabilization approach for designing systems that are well adapted to rapidly changing environments [12].

The major attraction of self-stabilization is the perceived elegance of eliminating clumsy mechanisms for detecting illegal states and initiating recovery actions. It also does not require the system to be initialized, a task which is difficult to coordinate in distributed systems. Further, the decentralized control, where communication is required only between neighboring processors, greatly reduces the potential overhead for global control.

A major problem that prevents the self-stabilizing approach from being used for critical applications is its vulnerability. Essentially, the system may go through a vulnerable period after failures that bring the system into an illegitimate state. During this period, the system behavior may not fully satisfy the requirements. Frequently, it is not possible to deal with the vulnerability in self-stabilizing systems due to the decentralization. When the violation of the global requirement does not directly imply violation of the local requirement, then the system can be in a state in which no processor can determine by itself whether the system state is legitimate and, hence, the system becomes vulnerable. However, algorithms can be designed such that the system will enter a small set of states (that can make the system vulnerable) with a very small probability. In this case, it is important to identify the source of failures such that algorithms can be designed accordingly with appropriate levels of complexity to tackle the problem. Two failure models can be considered, namely, the random failure model and the malicious failure model. In the random failure model, a transient failure can bring the system into any illegitimate state with equal probability. In the malicious failure model, some failed processors may maliciously try to violate the system legitimacy and cause critical damages. It is much harder to tackle the vulnerability problem in the presence of malicious failures. Malicious processors can deliberately create state values that violate the global constraint but

satisfy the local correctness checks.

We classify self-stabilizing systems into three categories: (1) Conventional self-stabilizing systems. In this case, the system does not care about temporary anomalous behaviors. (2) Fail-safe self-stabilizing systems. In this case, the system self stabilizes and does not have any vulnerable period even during the unstable period. However, the failure should not be malicious; instead, it should be random failures. (3) Malicious-failure-proof self-stabilizing systems. In this case, the system is designed such that it does not suffer from the vulnerability problem even when the system is in an illegitimate state due to some malicious system behaviors. In all three cases, the system state can be brought into any arbitrary state with uniform probability due to any type of failures other than malicious ones. Thus, it is helpful to identify the specific requirements of the system and accordingly design different self-stabilizing algorithms to economically handle the vulnerability problem.

In this paper, we develop a self-stabilizing mutual exclusion algorithm that copes with the vulnerability problem. We consider fail-safe self-stabilizing systems. Specifically, the system we consider consists of a group of processors (or intelligent devices) interconnected via a ring network where each processor has only a one-way communication channel with the processor immediately to its right. Let P_i, $0 \leq i \leq N-1$ denote the N processors in the system. We assume that only the simplex communication is allowed from processor P_i to $P_{(i+1) \bmod N}$. The goal is to provide mutually exclusive access to some shared resource for the N processors. The required properties of the algorithm include (1) decentralized control where only the states of the nearest neighbors, $P_{(i+1) \bmod N}$ for processor P_i, can be examined for the global control; and (2) fully asynchronous execution of all processors without requiring any centralized daemon process. The algorithm we develop guarantees that, with a high probability, the system is safe from failures.

3 Mutual Exclusion Algorithm

In this section, we present a self-stabilizing mutual exclusion algorithm. The algorithm is free of the vulnerability problem when the random failure model is considered. Our approach uses two new design principles.

1. *Sobriety test.* The system must be designed so that the set of legitimate states is a very small fraction of the set of all possible states. This will allow nonfaulty units to detect

illegitimate states and go to a home state that is known to be safe. This also allows rapid restabilization of the system to a stable operating mode.

2. *Specialization.* To prevent a group of faulty units from collectively subverting the safety of the system, the privileged processors in the system are specialized into two (or more) categories such that no processor by itself has sufficient information or capability to damage the system. To operate properly, two or more processors must cooperate with each other, either by sharing information or pooling their capabilities to effect changes in the system state. Thus, this approach guarantees that the failure of a processor will not result in any unsafe operation.

In our algorithm, the sobriety test is implemented by using two methods: expanding the set of possible states and using public key cryptography [27]. In the mutual exclusion algorithm, for a given state of a given processor, there is only one value that can be the new state value if the system is to continue to be legitimate. On the other hand, if all the integer values can be the possible state of any processor then the ratio of possible legitimate states to possible illegitimate state is very small. Thus, with random failures, the probability that the state values of the current and next states represent a legitimate transition is very small. Also, due to the use of public key cryptography, the probability of a nonrandomly generated value that can be the next state value is also very small. The specialization approach can be realized by using different private keys for the top and bottom processors. A new state value is not generated solely by the top processor or any other single processor, but by the bottom and top processors together. Each one is specialized by its own private key which is only known by itself. Thus, the failure of a processor will not make the system vulnerable.

Let P_{N-1} denote the "top" processor (as defined by Dijkstra), P_0 denote the "bottom" processor, and P_i, $0 < i < N-1$, denote the "other" processors. The top processor has knowledge of a private encryption key K_T while the bottom processor has knowledge of a private encryption key K_B. The public keys (for decryption), D_T and D_B, are known to all the processors. All the processors have a state variable which is an integer – in the code for processor P_i, this is referred to as S while R refers to the state of the processor to its right, i.e. $P_{(i+1) \bmod N}$. The top processor has an additional state variable, n, which is a natural number. Also, we use CR to indicate that the processor is in the critical region. The self-stabilizing algorithm without a vulnerable period is given in the following.

Top processor:

if

$decrypt(D_B, R) = S \rightarrow CR$; $S := encrypt(K_T, R)$; [] -- stable active

$decrypt(D_T, S) = R \rightarrow$ **skip**; [] -- stable inactive

$encrypt(K_T, n) = S \rightarrow$ **skip**; [] -- unstable inactive

otherwise $\rightarrow n := n + 1$; $S := encrypt(K_T, n)$; -- unstable active

end if

Other processors:

if

$decrypt(D_B, decrypt(D_T, R)) = S \rightarrow CR$; $S := R$; [] -- stable active

$R = S \rightarrow$ **skip**; [] -- stable inactive

otherwise $\rightarrow S := R$; -- unstable active

end if

Bottom processor:

if

$decrypt(D_T, R) = S \rightarrow CR$; $S := encrypt(K_B, R)$; [] -- stable active

$R = decrypt(D_B, S) \rightarrow$ **skip**; [] -- stable inactive

otherwise $\rightarrow S := encrypt(K_B, R)$; -- unstable active

end if

Functions *decrypt* and *encrypt* perform the public key encryption and decryption operations [27]. Instead of having an *increase-by-one* function for the top processor to generate new state values as in Dijkstra's algorithm, we use the *encrypt* function. Thus, the legitimacy of the new state propagated from the right processor can be easily verified by decryption using the public keys. Consequently, the faulty state of a processor can be easily detected when referenced by its left processor, and, hence, will not allow incorrect access to a critical section. To prevent a faulty top processor from arbitrarily generating a stream of tokens, we let the bottom processor also encrypt the state using its private encryption key K_B. A new value generated by the top processor will not be effective for entering the critical section unless it has traversed the entire ring and has been encrypted by the bottom processor.

Assume that decoding of the public key cryptosystem is not possible other than by random trials. It can be formally proved that, with a high probability, the algorithm guarantees that no two units will enter the critical section at the same time in spite of nonmalicious processor failures or arbitrary state transitions. Let p denote the probability that an encoded value (using a private key) can be decoded with a random selection of a number. Consider a 32-bit value. We have $p = 1/2^{32}$. Let $\overline{S}$ denote the system state, where

$$\overline{S} = \{S_0 S_1 \cdots S_{N-2} S_{N-1} n\},$$

and S_i, $0 \leq i < N$, is the local state of processor P_i. We divide the legitimate states of the system into *top privileged states*, TPS, where the top processor holds the privilege; *bottom privileged states*, BPS, where the bottom processor holds the privilege; and *other legitimate states* OLS, where a processor other than the top and bottom processors holds the privilege. Each of these sets can be expressed as follows.

$$\mathbf{TPS} = \{\overline{S} | S_{N-1} = decrypt(D_B, S_0) \wedge \forall i : 0 < i < N-1 :: S_i = S_{N-1}\},$$

$$\mathbf{BPS} = \{\overline{S} | S_0 = decrypt(D_T, S_{N-1}) \wedge \forall i : 0 < i < N-1 :: S_i = S_{N-1}\},$$

$$\begin{aligned}\mathbf{OLS} = \{\overline{S} | S_{N-1} &= encrypt(K_T, S_0) \wedge \\ &\exists j : 0 < j < N-1 :: \\ &\quad \forall i : j < i < N-1 :: S_i = S_{N-1} \wedge \forall i : 0 < i \leq j :: S_i = decrypt(D_B, S_0)\}.\end{aligned}$$

These sets of states are referred to in the following correctness proofs. We will first show that from any state, the system will converge to a legitimate state, TPS, if there are no further failures. We then show that from a legitimate state, the system will always move to a legitimate state after a legitimate transition. Finally, we will show that the system guarantees mutual exclusion irrespective of random failures.

Theorem 1. A state in TPS will be reached from any arbitrary initial state.

Proof. The proof follows the ladder structure of Gouda and Multari [18] with the following intermediate steps:

R_0: $\{true\}$

R_1: $\{S_{N-1} = encrypt(K_T, n) \vee S_{N-1} = encrypt(K_T, S_0)\}$

R_2: $\{(S_{N-1} = encrypt(K_T, n) \vee S_{N-1} = encrypt(K_T, S_0))$
$\wedge \forall i : 0 < i < N-1 :: S_i = S_{N-1}\}$

R_3: $\{S_{N-1} = decrypt(D_B, S_0) \wedge \forall i : 0 < i < N-1 :: S_i = S_{N-1}\}$

If the condition given in R_1 is not satisfied, then the top processor will make a move and R_1 will be satisfied. Otherwise, R_1 is already true. Thus, from any arbitrary state in R_0, the system will obviously move to R_1.

Assume that the system state $\overline{S}$ is in R_1 and $(S_1 S_2 \cdots S_{N-1})$ contains k segments of values, $v_1, ..., v_k$, $1 < k \leq N$ (if $k = 1$ then R_2 is true). If the current state of the top processor satisfies $encrypt(K_T, n) = S_{N-1}$ then the top processor will make a move (stable move) only if the specific condition "$\exists i : 1 \leq i < k :: v_i = S_{N-1}$" is satisfied. If the current state of the top processor satisfies $S_{N-1} = encrypt(K_T, S_0)$ then the top processor will make a move as

long as S_0 changes. To make a stable move, the system has to satisfy $v_1 = S_{N-1}$. Otherwise, any other value of v_1 will make the top processor make a move (unstable move) and lead the system into a state that satisfies $encrypt(K_T, n) = S_{N-1}$ and, hence, the top processor will not make a move unless the specific condition $v_i = S_{N-1}$, for some i, is true. Essentially, a stable move can be preceded by at most one unstable move for the top processor.

If the top processor does not make a move before R_2 is satisfied, then no new value is generated and the number of segments in $\overline{S}$ will be reduced since other processors will make their moves and propagate the segment values to S_1 which will be eliminated by the top processor after being encrypted by P_0. Consequently, $\forall i : 0 < i < N-1 :: S_i = S_{i+1}$ will be satisfied. That is, R_2 will be satisfied. Thus, we need to consider the case in which the top processor moves. However, we can consider only the stable moves (and double the number of moves to take into account the possible unstable moves). As we can see from the discussion above, the top processor will not make a stable move before R_2 is satisfied if $\forall i : 1 \leq i < k :: v_i \neq S_{N-1}$. Since the current S_{N-1} should not have propagated to v_i, for $1 \leq i < k$, the probability of having $v_i = S_{N-1}$ is p. Hence, the probability that the top processor will not make a stable move is $(1-p)^{k-1} < (1-p)^N$ and so the probability that the top processor will make $2l$ moves (l stable moves) is less than $(1-(1-p)^N)^l$. When l becomes large, $(1-(1-p)^N)^l \to 0$. Thus, the top processor will eventually not make any moves till $\forall i : 0 < i < N-1 :: S_i = S_{i+1}$ holds. In other words, the system will converge from R_1 to R_2 eventually.

When R_2 holds, the top processor will not make a move. Neither will the other processors except for the bottom processor P_0. Since P_0 holds the privilege, it will make a move and, hence, $R_3 = TPS$ will be satisfied. □

Theorem 1 shows the convergence property of the system where the system is guaranteed to stabilize to a legitimate state irrespective of the current state. For a 32-bit state value, we have $p = 1/2^{32}$. Assume that the system consists of 64 processors. The probability that the system will converge to a legitimate state in one iteration (i.e., $O(N)$ time) is $1-(1-(1-p)^N)^1 = (1-p)^N \approx 1 - 1.5 \times 10^{-8}$. Note that converging from R_0 to R_1 and R_2 to R_3 only require single steps.

In Theorem 2, the closure property of the algorithm is proved.

Theorem 2. If the system is in a legitimate state, then each move following the given algorithm will guarantee that the system remains in a legitimate state if there is no failure.

Proof. If the system is in a legitimate state, then the system state s is in $TPS \vee BPS \vee OLS$. If s is in TPS, then only the top processor will make a move (assuming that there is no failure). The top processor will enter the critical section and set $S_{N-1} = encrypt(K_T, S_0)$. Thus, the new system state is in OLS with $j = N-2$. If s is in BPS then the bottom processor will make the move by executing $S_0 := encrypt(K_B, S_1)$. Thus, the new system state will be in TPS. Otherwise, the system will be in OLS. Let processor j be the one that holds the privilege, where $1 < j < N-1$. The new system state will still remain in OLS but with j decreased by one after setting $S_j := S_{j+1}$. If processor P_1 makes the move, then the system state will satisfy BPS. In conclusion, if the system state is legitimate, then after a legitimate transition, the system will continue to remain in a legitimate state. □

The fairness property of the algorithm is obvious. We will omit the formal proof. Essentially, when the system is in legitimate states, the privilege moves circularly from the top processor down to the bottom processor iteratively. Thus, the system satisfies all the properties required for conventional self-stabilizing systems. Finally, we need to show that the system is free of vulnerable period (with a high probability). In Theorems 3 and 4, we show that our algorithm guarantees the mutual exclusion property with a high probability.

Theorem 3. If the system is in a legitimate state, then the mutual exclusion property is guaranteed.

Proof. As with the conventional self-stabilizing mutual exclusion algorithms, this algorithm only allow one privileged processor when the system is in legitimate state. Thus, the mutual exclusion property is guaranteed. □

Theorem 4. If the system is in an illegitimate state, then the probability that mutual exclusion property is satisfied is $(1-p)^{N(N-1)/2}$.

Proof. Each processor has to pass its sobriety test to enter the critical section. However, passing all local checks does not imply global correctness. The system can be in an illegitimate state and have more than one processor entering the critical section since more than one local condition for entering the critical section is satisfied. However, the probability that this occurs is very small. Essentially, at least one processor has to be in a faulty state and, either currently or after several moves, some local sobriety test becomes true illegally.

Let PL_i denote the local condition of processor P_i that allows P_i to enter the critical section. If processor $P_{(k+1) \bmod N}$, $0 \leq k \leq N-1$, is in a faulty state and causes PL_k to become true, then P_k will enter the critical section illegally. It is also possible that both S_k and S_{k+1}

are faulty and PL_k is true. Further, it is possible that for all i, $k < i \leq j$, S_i is faulty, and after S_j has been propagated to S_{k+1}, PL_k becomes true (this can include the case when S_k is also faulty). In all these cases, given any index k, S_k can be of any value (faulty or not) and as long as there exists an index j such that S_j is faulty and S_j can propagate to $S_{(k+1) \bmod N}$ and cause PL_k to be true, then the mutual exclusion condition may be violated. Consequently, as long as there exists no pair such that the above condition is true, then the mutual exclusion can be guaranteed. Given an arbitrary pair of processors P_j and P_k, the probability that S_j can cause PL_k to be true illegally is p, which is the probability that for a given value, a randomly selected value happens to be the decrypted value of that given value. Thus, the probability that no pair of processors has the matching state values is $(1-p)^{N(N-1)/2}$. Hence, the probability that mutual exclusion property is violated is $1-(1-p)^{N(N-1)/2}$. □

Consider $N = 64$. The probability of violation of mutual exclusion using this algorithm is approximately 4.8×10^{-7}. To increase the reliability, we can increase the number of bits in the state variable. Thus, the system can essentially guarantees mutual exclusion with an arbitrarily high probability. With the increasing number of bits in the state variable, the system will also have an increased probability for convergence after a failure in one iteration.

4 Conclusion

The motivation for this paper was whether it is possible to use self-stabilization for safety-critical systems. Toward this goal, we have developed a highly safe self-stabilizing mutual exclusion algorithm that guarantees that, with a high probability, the mutual exclusion property is satisfied even in unstable states. We used two rules that appear to have general applicability, namely, the sobriety test and processor specialization. Even in the presence of faulty states, our algorithm guarantees mutual exclusion with a probability of 5×10^{-7} when a 32-bit state value is used. Our approach can be extended to provide a general method of tolerating failures in self-stabilizing algorithms for a class of fixed-point computations.

Currently, we are also investigating a mutual exclusion algorithm that is general-failure-proof self-stabilizing. Algorithms with this property will incur a higher cost but can tolerate Byzantine general type of failures.

References

[1] Y. Afek and G. Brown, "Self-stabilization of the alternating-bit protocol," *Proc. 8th Symp.*

Reliable Distributed Systems, pp. 80-83, 1989.

[2] A. Arora, M. Gouda, and T. Herman, "Composite routing protocols," *Proc. 2nd IEEE Symp. on Parallel and Distributed Systems*, Dallas, 1990, pp. 288-292.

[3] A. Arora and M.G. Gouda, "Distributed reset," *Proc. 10th Conf. Foundations of Softw. Tech. and Theo. Comp. Sc.*, LNCS 472, 1990, pp. 316-331, Springer-Verlag.

[4] B. Awerbuch and G. Varghese, "Distributed program checking: A paradigm for building self-stabilizing distributed protocols," *Proc. FOCS'92*, pp. 258-267.

[5] B. Awerbuch, B. Patt-Shamir, and G. Varghese, "Self-stabilization by local checking and correction," *Proc. FOCS'92*, pp. 268-277.

[6] R.P. Bianchini, Jr., Private Communication, Jan. 1995.

[7] G.M. Brown, M.G. Gouda, and C.-L. Wu, "Token systems that self-stabilize," *IEEE Transactions on Computers*, Vol. 38, No. 6, June 1989, pp. 845-852.

[8] J.C. Browne, A. Emerson, M. Gouda, D. Miranker, A. Mok, and L. Rosier, "Bounded-time fault tolerant rule-based systems," *Telematics and Informatics*, Vol. 7, Nos. 3/4, 1990, pp. 441-454.

[9] J.E. Burns and J. Pachl, "Uniform self-stabilizing rings," *ACM Trans. on Prog. Langs. & Sys.*, Vol. 11, No. 2, April 1989, pp. 330-344.

[10] R.W. Buskens, *Practical On-Line Diagnosis in Distributed Systems*, Ph.D. Dissertation, Carnegie Mellon University, Oct. 1994.

[11] A.M.K. Cheng, "Self-stabilizing real-time rule-based systems," *Proc. SRDS'92*, pp. 172-179.

[12] X.A. Debest, "Remark about self-stabilizing systems," *Comm. of the ACM*, Vol. 38, No. 2, Feb. 1995, pp. 115-117.

[13] E.W. Dijkstra, EWD 306 "The solution to a cyclic relaxation problem," 1973. Reprinted in *Selected Writings on Computing: A Personal Perspective,* Springer-Verlag, pp. 34-35, 1982.

[14] E.W. Dijkstra, "Self-stablizing systems in spite of distributed control," *Comm. ACM*, Vol. 17, No. 11, Nov. 1974, pp. 643-644.

[15] S. Dolev, A. Israeli, and S. Moran, "Self-stabilization of dynamic systems assuming only read-write atomicity," *Proc. 9th ACM Symp. Principles of Distributed Computing,* 1990, pp. 103–117.

[16] S. Ghosh, "Stabilizing Petri Nets," *3rd IEEE Symp. on Parallel and Distributed Systems,* Dec. 1991.

[17] M.G. Gouda and M. Evangelist, "Convergence/response tradeoffs in concurrent systems," *Proc. 2nd IEEE Symp. Parallel and Distributed Systems*, Dec. 1990, pp. 288-292.

[18] M.G. Gouda and N.J. Multari, "Stabilizing communication protocols," *IEEE Trans. on Computers*, Vol. 40, No. 4, April 1991, pp. 448-458.

[19] A. Israeli and M. Jalfon, "Token management schemes and random walks yield self-stabilizing mutual exclusion," *Proc. 9th ACM Symp. Principles of Distributed Computing,* 1990, pp. 119-130.

[20] R.K. Iyer, K.S. Trivedi, and J. Goldberg, "Introduction to special issue on fault-tolerant computing," *IEEE Trans. on Computers*, Vol. 44, No. 2, Feb. 1995, pp. 165-169.

[21] S. Katz and K. Perry, "Self-stabilizing extensions for message-passing systems," *Proc. 9th ACM Symp. Principles of Distributed Computing*, 1990.

[22] H.S.M. Kruijer, "Self-stabilization (in spite of distributed control) in tree-structured systems," *Information Processing Letters*, Vol. 8, No. 2, Feb. 1979, pp. 91-95.

[23] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. on Programming Languages and Systems*, Vol. 4, No. 3, July 1982, pp. 382-401.

[24] L. Lamport, "Solved problems, unsolved problems and non-problems in concurrency," *Operating Systems Review,* Vol. 19, No. 4, Oct. 1985, pp. 34-44.

[25] G. LeLann, "An analysis of different approaches to distributed computing," *Proc. 1st Int. Conf. Dist. Proc. Sys.,* Huntsville, AL, Oct. 1979, pp. 222-232.

[26] J.C. Line and S. Ghosh, "A methodology for constructing stabilizing crash-tolerant applications," *Proc. 13th Symp. on Reliable Distributed Systems*, Data Point, CA, Oct. 1994, pp. 12-21.

[27] R.L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public cryptosystems," *Comm. of the ACM*, Vol. 21, No. 2, Feb. 1978, pp. 120-126.

[28] M . Schneider, "Self-stabilization," *ACM Comp. Surveys*, Vol. 25, No. 1, Mar. 1993, pp. 45-67.

[29] Y. Zhao and F.B. Bastani "A self-adjusting algorithm for Byzantine Agreement," *Distributed Computing,* Vol. 5, 1992, pp. 219-226.