Lecture Notes in Computer Science 2194

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Ajoy K. Datta Ted Herman (Eds.)

Self-Stabilizing Systems

5th International Workshop, WSS 2001
Lisbon, Portugal, October 1-2, 2001
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Ajoy K. Datta
University of Nevada, Department of Computer Science
Box 454019, Las Vegas, NV 89154-4019, USA
E-mail: datta@cs.unlv.edu

Ted Herman
University of Iowa, Department of Computer Science
Iowa City, IA 52242, USA
E-mail: herman@cs.uiowa.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Self-stabilizing systems : 5th international workshop ; proceedings / WSS 2001, Lisbon, Portugal, October 1 - 2, 2001. Ajoy K. Datta ; Ted Herman (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2194)
ISBN 3-540-42653-1

CR Subject Classification (1998): C.2.4, C.2, C.3, F.1, F.2.2, K.6

ISSN 0302-9743
ISBN 3-540-42653-1 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by Christian Grosche, Hamburg
Printed on acid-free paper SPIN 10840630 06/3142 5 4 3 2 1 0

Preface

Physical systems which right themselves after being disturbed evoke our curiosity because we want to understand how such systems are able to react to unexpected stimuli. The mechanisms are all the more fascinating when systems are composed of small, simple units, and the ability of the system to self-stabilize emerges out of its components. Faithful computer simulations of such physical systems exhibit the self-stabilizing property, but in the realm of computing, particularly for distributed systems, we have greater ambition. We imagine that all manner of software, ranging from basic communication protocols to high-level applications, could enjoy self-corrective properties.

Self-stabilizing software offers a unique, non-traditional approach to the crucial problem of transient fault tolerance. Many successful instances of modern fault-tolerant networks are based on principles of self-stabilization. Surprisingly, the most widely accepted technical definition of a self-stabilizing system does not refer to faults: it is the property that the system can be started in any initial state, possibly an "illegal state," and yet the system guarantees to behave properly in finite time. This, and similar definitions, break many traditional approaches to program design, in which the programmer by habit makes assumptions about initial conditions. The composition of self-stabilizing systems, initially seen as a daunting challenge, has been transformed into a manageable task, thanks to an accumulation of discoveries by many investigators. Research on various topics in self-stabilization continues to supply new methods for constructing self-stabilizing systems, determines limits and applicability of the paradigm of self-stabilization, and connects self-stabilization to related areas of fault tolerance and distributed computing.

The Workshop on Self-Stabilizing Systems (WSS) is the main forum for research in the area of self-stabilization. The first workshop was held in Austin (1989), and since 1995, workshops have been held biennially: Las Vegas (1995), Santa Barbara (1997), Austin (1999), and Lisbon (2001). WSS 2001 was thus our first workshop held outside North America, and reflected the strong growth and international participation in the area. We received 27 submitted papers for this workshop, which is a 50% increase from the previous workshops. The program committee selected 14 of the submitted papers, and Sukumar Ghosh presented our invited contribution.

This volume covers many areas within the field and reflects current trends and new directions in self-stabilization. Important applications of distributed computing are topics in several papers (routing, group membership, publish-subscribe systems). Other papers strike a methodological tone, describing tools to construct self-stabilizing systems. Three papers have "agent" in their titles, which is a topic not mentioned in any previous workshop. Several papers investigate non-standard definitions (or weakenings) of self-stabilization. Our field continues to grow and evolve.

WSS 2001 was held October 1-2 in Lisbon immediately preceding DISC 2001, and we thank members of the DISC steering committee, who helped approve the joint WSS-DISC venture: André Schiper, Shmuel Zaks, and Michel Raynal. We are grateful to all the program committee members and reviewers for their hard and timely work. The referees thoroughly read each submitted manuscript and provided extensive feedback to the authors. We also thank the following people from the University of Lisbon for local organization: Luis Rodrigues (Chair), Paulo Veríssimo (Publicity), Filipe Araújo (Treasurer), Alexandre Pinto (Web), and Hugo Miranda (Registration). Finally we thank the following organization for supporting the workshop: Fundação para a Ciência e a Tecnologia, Ministério da Ciência e da Tecnologia, Portugal.

July 2001

Ajoy K. Datta
Ted Herman

Program Committee

Anish Arora, The Ohio State University, USA
Joffroy Beauquier, Université de Paris Sud, France
Ajoy K. Datta, **Program Chair**, University of Nevada Las Vegas, USA
Shlomi Dolev, Ben-Gurion University of the Negev, Israel
Mohamed Gouda, University of Texas at Austin, USA
Ted Herman, **Publications Chair**, University of Iowa, USA
Jaap-Henk Hoepman, University of Twente, The Netherlands
Shing-Tsaan Huang, National Central University, Taiwan
Sandeep Kulkarni, Michigan State University, USA
Toshimitsu Masuzawa, Osaka University, Japan
Franck Petit, Université de Picardie, France
Sébastien Tixeuil, **Publicity Chair**, Université de Paris Sud, France
Vincent Villain, Université de Picardie, France

Referees

A. Arora
J. Beauquier
P. Boldi
S. Cantarell
A. Ciuffoletti
J. Cobb
S. Delaët
M. Demirbas
S. Dolev
F. Gärtner
C. Genolini
S. Ghosh
M. Gouda
T. Hérault
T. Herman
J.-H. Hoepman
R. Howell
M. Gradinariu
C.-T. Huang
S.-T. Huang
C. Johnen
H. Kakugawa
M. H. Karaata
S. Kulkarni
W. Leal
T. Masuzawa
N. Mittal
M. Mizuno
M. Nesterenko
F. Petit
O. Theel
S. Tixeuil
S. Vigna
V. Villain
H. Voelzer
M. Yamashita

Table of Contents

Cooperating Mobile Agents and Stabilization

Sukumar Ghosh*

The University of Iowa
ghosh@cs.uiowa.edu

Abstract. In the execution of distributed algorithms on a network of processes, the actions of the individual processes are scheduled by their local schedulers or demons. The schedulers communicate with their immediate neighbors using shared registers or message passing. This paper examines an alternative approach to the design of distributed algorithms, where mobile agents are allowed to traverse a network, extract state information, and make appropriate modification of the local states to steer the system towards a global goal. The primary emphasis of this paper is system stabilization. Both single-agent and multi-agent protocols are examined, and the advantages and disadvantages of agent-based stabilization are discussed.

1 Introduction

Consider the execution of a program on a network of processes. The processes communicate with one another through shared memory or message passing. Each process has a scheduler (also called a *demon*) that collects information about the states of the its neighboring processes, and schedules its local actions. Two of the well-known execution models rely on (i) central demon and (ii) distributed demons. In the central demon model, processes execute their actions serially, whereas in the distributed demon model, any subset of the set of enabled processes can execute their actions concurrently.

This paper explores an alternative model for computation that uses *mobile agents* in the context of stabilization. A mobile agent [14] is a program that can migrate from one node to another, perform various types of operations at these nodes, and take autonomous routing decisions. In contrast with messages that are passive, an agent is an active entity, that can be compared with a messenger.

Conventional stabilizing distributed systems expect processes to run predefined programs that have been carefully designed to guarantee recovery from all possible bad configurations. However, in an open environment like the Internet where processes are not under the control of a single administration, expecting every process to modify its program for the sake of stabilization is unrealistic, and a more centralized mechanism for network administration is an viable alternative. This paper explores such an alternative mechanism for stabilizing a distributed system, in which processes are capable of accommodating visiting agents.

* This research was supported in part by the National Science Foundation under grant CCR-9901391.

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 1–18, 2001.

We assume that the underlying system to be stabilized uses message passing for interprocess communication. Note that we could as well consider interprocess communication through shared memory, but the choice has been made only for the sake of uniformity - agents propagate across links exactly like messages. The state of the network consists of the states of each of the processes, as well as those the channels. Thus, an action by a process to update its own state, or to send or receive messages also modifies the state of the network. The set of states of the network can be classified into the categories *legal* and *illegal*. A stabilizing system guarantees that regardless of the starting state, the network eventually reaches a legal state, and remains in the legal state thereafter. Convergence and closure are the two cornerstones of stabilizing systems [3].

We further assume that in addition to the ongoing activities in the above network of processes, one or more processes can send out mobile agents that can migrate from one process to another, read the local states of the visited processes, and update these local states whenever necessary. While the individual processes maintain the closure of legal configurations, agents take up the responsibility of steering the system to a legal configuration when they detect a configuration to be illegal. The detection involves taking a total or partial snapshot of the system state. Corrective actions involve the modification of the states of one or more processes visited by the agents.

In the past, Kutten, Korach, and Moran [12] used tokens with identities to solve the leader election problem. Each process initiates a network traversal with a token labeled with its identity. In the paper Distributed Reset [4], processes first elect a leader and then construct a spanning tree with the leader as the root. Subsequently, the sending of the reset waves can be viewed as the sending out *reset agents* by the leader down the network. In the area of electronic commerce, the use of mobile agents has been steadily increasing. In network management, primitive agents mimicking biological entities like ants (see the article on Swarms [17]) traversing a network have been used in solving various problems like shortest-path computation and congestion control. In such systems, the individual agents do not have explicit problem-solving knowledge, but intelligent action emerges from the collective action by ants. These papers provide the general motivation behind exploring how mobile agents can be utilized to stabilize distributed systems. In our systems, an agent is a sequential program with no obvious limit on its intelligence - problems can be solved either by a single agent, or by a group of agents. Furthermore, there is the additional challenge that the initial system configuration can be arbitrary, and the agents themselves may be corrupted in transit.

This paper is about using agents as a tool for stabilization, and not about a formal computational model using agents, which appears in [1]. The paper has six sections. Section 2 provides a general description of the agent model. Section 3 illustrates the construction of a spanning tree using a single reliable agent. Section 4 explains how to implement a reliable agent. Section 5 presents two different solutions to the spanning tree construction problem using multiple agents. Finally Section 6 contains some concluding remarks about agent-stabilizing systems.

2 The Agent Model

An agent consists of the following six components[1]:

1. The identifier *id*, usually the same as the initiator's id. The *id* is unnecessary if there is a single agent, but is essential to distinguish between multiple agents in the same system.
2. The agent program A
3. The briefcase B containing a set of variables
4. The previous process PRE visited by the agent
5. The next process to visit $NEXT$ that is computed after every hop
6. A supervisory program S for bookkeeping purposes.

An agent may be installed into a system from outside, or it may be installed from within the system. Unless stated otherwise, we will consider the agent to be externally installed into a designated node called the *home* or the *initiator.* In a given system, the number of agents can vary, but for our purpose, we will need at least one agent. When the home of the agent is internally designated, it is done through an initial leader election by the component processes.

When multiple agents are required, we can use either a *static* model, or a *dynamic* model. In the static model, either a single home process sends out a fixed number k ($k > 1$) of agents, or k distinct home processes[2] (each authorized to send out agents) are identified before the computation starts. In the dynamic model, an agent is entitled to create an unspecified number of new agents, and destroy them as and when necessary.

A hop by an agent is an atomic step used to move from one process to a neighbor. Each hop of the agent costs one message, and is completed in unit time. We assume that an agent performs the local computation at any process in zero time. The communication links are half-duplex, as a result, two agents traveling in opposite directions along a path consisting of a chain of processes are guaranteed to meet each other at some process. The message complexity is determined by the total number of hops taken by all the agents to restore the system to a legal configuration. The time complexity is determined by the total number of time units needed to return to a legal configuration. In addition, if the agents are internally installed, then to calculate the overall time or message complexity, the overheads of leader election have to be taken into account.

By choosing the agent model where the progress of computation and the restoration of legitimacy are controlled by a handful of mobile agents instead of an army of demons, we are clearly moving towards centralization of authority. We present two motivations behind such a decision.

[1] For convenience, we will use capital letters to represent agent-related variables or programs.

[2] We somehow need to distinguish k processes from the rest, and designate them as initiators.

Motivation 1. At the risk of sounding trivial, consider a stabilizing solution to the problem of maxima finding on a ring of n processes using a single agent[3]. For each process i, designate one successor $neighbor(i)$ to which the agent can move to make a trip round the ring. The goal is to set the local variable max of every process to the largest id in the ring. These id's are positive integers. The home process will initialize the briefcase variables MAX (that records the largest id) and MODE ($\in \{0,1,2\}$) with 0 and 0. The protocol is presented in Fig. 1.

Program for the agent while visiting process i

agent variables MODE, MAX;
process variables max;

if MAX<id ∧ MODE<2 → MAX:=id;
□ MODE=2 → max(i) := MAX;
fi;
NEXT:= neighbor

Program for the home process

Executed whenever the agent reaches home
initially MAX=0, MODE=0

if MOD < 2 → MODE:= MODE + 1
□ MODE=2 → MAX:=0; MODE:=0
fi;
NEXT := neighbor

Fig. 1. A stabilizing solution for maxima finding using a single agent.

It can take at most two roundtrip traversals for the agent to set MAX correctly to the largest id, and one more traversal to write this value into the individual processes. Thus the message complexity is $3(n-1)$. Note that to solve the same problem on the traditional message passing or shared memory models, we will need $O(n^2)$ steps. This is not an isolated example – similar observations can be made about many other solutions too.

Motivation 2. Solutions using a single agent model can often be derived from existing sequential algorithms through a simple adaptation mechanism. Sequential graph algorithms, for example, specify at each step which node will execute the next action. The adaptation by the corresponding agent model involves making the agent move to the position of next action (using a traversal algorithm)

[3] The leader process need not be the same as the process with maximum id. It can, for example, be a process with the smallest id, or may satisfy any other unique criteria.

prior to executing the action. The preference of a suitable sequential algorithm may depend, among other things, on the extent of movement of the agent. Our stabilizing protocols work under the following two constraints:

Non-interference. The normal operation of the system neither depends on, nor is influenced by the presence of agents.

As an exception, only the home process of an agent can test the arrival of that agent (as a part of evaluating its guards), initialize or update the agent's briefcase, and send out the agent to begin its next traversal (as a part of its action). Such an agent is sent out at convenient intervals to initiate a "clean-up phase", otherwise the operation of the system continues as usual. The individual processes are oblivious to the presence of the agent. We disregard any minor slowdown in the execution speed of a process due the sharing of the resources by visiting agents.

Atomicity. At any node, the arrival of an agent triggers the agent program whose execution is atomic. The agent program ends with the departure of the agent from that node, or with a waiting phase (in case the agent has to wait at that node for another agent to arrive), after which the execution of the application program at that node resumes.

The computation at a node alternates between the agent program and the application program. At any node, the visit of a single agent can be represented by the following sequence of events. We denote an atomic event using $\langle\ \rangle$:

agent arrives, $\langle$ agent program executed $\rangle$, agent leaves

When two agents I and J have to meet at a node k to exchange data, the sequence of events will be as follows:

agent I arrives, $\langle$ agent program of I executed $\rangle$, agent I waits at k

Following this, the application program at node k resumes, and continues until the other agent J arrives. When J arrives at node k, the following sequence of events take place:

agent J arrives, $\langle$ data exchange with I occurs $\rangle$, agent I and J leave.

Then the application program at k resumes once again.

When the agent is not externally installed, it is possible to implement the agent model using shared memory or message passing. An outline is as follows: Let the processes elect a leader and designate it as the home of the agent. Put a copy of the agent program A (and the supervisory program S whose role will be explained later) at every process in the network, including the version for the home process at the leader. Also, put a copy of the briefcase variables B of the agent at the home process. Now, let the leader execute the agent program A, and send the updated briefcase variables of the agent program as messages to the next process whose identity is determined by the execution of A. After

receiving this message, the recipient will execute A, and repeat the same steps as its predecessor. This concludes the outline.

Agent-based stabilization can be viewed as a stabilizing extension of a distributed system as proposed in [13]. While [13] emphasized the feasibility of designing stabilizing distributed systems, we argue that mobile agents have some interesting properties that make implementations straightforward.

3 Stabilization Using a Single Agent

3.1 Spanning Tree Construction

Here we first present a stabilizing protocol for the construction of a DFS spanning tree using a single agent. The solution is taken from [9]. In the subsequent sections, we will build on this to present our multi-agent protocols. Let $g = (v, e)$ represent the topology of the undirected graph, where v is the set of nodes, and e is the set of edges. In the single-agent protocol, the *home* of the agent is the root of the spanning tree. We use $p(i)$ to designate the parent of a node i. The additional variables are as follows:

$child(i) \equiv \{j : p(j) = i\}$
$neighbor(i) \equiv \{j : (i, j) \in e\}$

The program of the agent consists of three types of actions: (i) actions that update the local variables of the process that it is visiting, (ii) actions that modify its briefcase variables, and (iii) actions that determine the next process that it will visit. The individual processes are passive.

A key issue in agent-based solution is graph traversal. To distinguish between consecutive rounds of traversal, we introduce a briefcase variable SEQ ($\in \{0,1\}$) that keeps track of the most recent round of traversal. With every process i, define a boolean $f(i)$ that is set to the value of SEQ whenever the process is visited by the agent. SEQ is complemented by the root before the next traversal begins. Thus, the condition $f(i) \not= SEQ$ is meant to represent that the node has not been visited in the present round.

The agent program has three basic rules: DFS1, DFS2, DFS3 and is described[4] in Fig. 2. A proof appears in [9]. This solution disregards the case when an agent is trapped in a cyclic path and fails to return to the root. We will address this issue in the next section that deals with agent failures.

Both the time complexity and the message complexity for stabilization are $O(n^2)$. Once stabilized, the agent needs $2(n-1)$ hops for subsequent traversals.

4 Agent Failure

An agent, like any other process, is subject to failure or corruption. Since the agent is the main focus of control, the failure of an agent is a matter of major concern.

[4] It disregards the details of how a process i maintains its $child(i)$ and their flags.

Program for the agent while visiting node i

agent variables NEXT, PRE, SEQ;
process variables f, child, p;

If the node is already visited, then retreat
if f(i) = SEQ ∧ PRE /∈ child(i) → NEXT:= PRE **fi**
if f(i) /= SEQ →
    Mark the current node as visited and set the parent
    f(i):= SEQ; p(i):= PRE;
    if *Visit an unvisited child*
        (DFS1) ∃ j ∈ child(i):f(j) /= SEQ → NEXT:=j
        When all neighbors have been visited, return to the parent
    □ (DFS2) ∀ j ∈ neighbor(i): f(j)=SEQ → NEXT:= p(i)
        Create a path to a node that is unreachable using DFS1
    □ (DFS3) ∀ j ∈ child(i):f(j)=SEQ ∧ ∃ k ∈ neighbor(i): f(k) /= SEQ →
            NEXT := k
    fi
fi;

Program for the home process

Executed when the agent visits home

if ∃ j ∈ child: f(j) /= SEQ → NEXT:=j
□ ∀ j ∈ neighbor: f(j)=SEQ → SEQ := 1-SEQ;
        NEXT := k : k ∈ child
fi

Fig. 2. Spanning tree construction with a single agent.

In the case of externally installed agents, we cannot take the help of the individual processes to detect or correct a faulty agent, since no process is aware of the presence of the agent. The agent has to either heal itself, or kill itself after it detects that it is faulty. In the latter case, the home process will sense the loss of the agent using a timeout, and regenerate a new agent.

Following traditions, we rule out the corruption of the agent program A or the supervisory program S, but take into account possible corruption of the agent variables, and the impact of it on the entire stabilization mechanism. We will see later that even the agent program A may be allowed to be corrupted as long as the program of the home process remains good.

The agent traverses the network, and periodically visits its home. The home process appropriately updates the briefcase variables before the next traversal begins. To deal with agent failure, we first introduce a *reliable agent.* Divide the agent variables into two classes: *privileged* and *non-privileged.* Call an agent variable privileged, when it can be modified only by its home process - all other variables will be called non-privileged. Examples of privileged variables are: $MODE$

in Fig. 1, *SEQ* in Fig. 2, or the id assigned to an agent by its home process. Then, an agent will be called reliable, when it satisfies the following two criteria:

1. The agent completes its traversal of the network and returns home within a finite number of steps.

2. The values of all the privileged variables of the agent remain unchanged during the traversal.

An agent can be unreliable either due to the corruption of its privileged variables during a traversal, or due to routing problems. Note that, by simply being reliable, an agent cannot stabilize a distributed system, but it leads to the adoption of a two-phased approach. In the first phase, we demonstrate how a reliable agent guarantees convergence and closure. In the second phase, we present methods by which unreliable agents eventually become reliable, and remain reliable thereafter. This part will use some generic remedies, independent of the problem under consideration. The generic remedies are as follows:

Loss of Agent. If the agent is killed, then the initiator discovers this using timeout[5] and generates a new agent with a new sequence number. If the timeout is due to a delayed arrival of the original agent, then the original agent has to be killed by the leader.

To avoid the risk of multiple agents with identical sequence numbers roaming in the system, the probabilistic technique of [11] can be used. It involves the use of a sequence number from a three-valued set $\Sigma = \{0, 1, 2\}$. If the sequence number of the incoming agent matches with the sequence number of the previous outgoing agent, then the initiator randomly chooses the next sequence number from Σ, otherwise the agent is killed. Another approach to guarantee the uniqueness of the agent is to use counter-flushing [16] that shows how a single-token configuration can be restored on a ring in time $3 \cdot R$, where R is the roundtrip traversal time of the agent.

Corruption of the Agent Identifier. An agent is recognized by its home using the agent's id. If the id of the agent is corrupted, then the home process will not be able to recognize it, and the unreliable agent will roam the network forever. *To prevent this, the supervisory program S of the agent counts the number of hops taken by the agent.* As soon as this number exceeds a predefined limit $c \cdot R$, (c is a large constant) the agent kills itself. The same strategy works, if due to routing anomalies the agent is unable to return home.

Corruption of Agent Variables. An unreliable agent with corrupted variables can arbitrarily alter the global state of the system. Note that the corruption

[5] When the agent is internally installed, it is possible to avoid a system-level timeout by using a modified version of the stabilizing mutual exclusion protocol presented by to Dijkstra [7]. In his system, the number of agents is always positive by definition.

of the non-privileged variables is not a matter of concern, because these are expected to assume arbitrary values when the agent interacts with the underlying system. Our only concern is the possible corruption of privileged variables.

To recover from such failures, we need to demonstrate that despite the corruption of the privileged variables, eventually the agent reaches the global state to which it was initialized by its home process. For each agent-stabilizing system, as a part of the correctness proof, we need to prove the following theorem:

Theorem 1. An unreliable agent is eventually substituted by a reliable agent.

We demonstrate such a proof for the maxima finding protocol described in Fig. 1. Assume that the value of the privileged variable $MODE$ has been corrupted. Since the generic remedies guarantee that the agent eventually returns home (or else, is killed and a new agent is generated by its home process), one of the two actions in the program for the home process is executed. If $MODE = 2$, then the second action changes $MODE$ to 0. Otherwise, the execution of the first action increments the value of $MODE$ until it becomes 2. Thereafter, the execution of the second action changes $MODE$ to 0, which is the desired initial state of the reliable agent. □

The maximum time needed for the above recovery is $3.R$. Taking the other remedies into account, the time required to substitute an unreliable agent by a reliable one is $O(R)$, where R is the maximum time needed by the agent to traverse the network.

For internally installed agents, all the above remedies will apply. Additionally, in some cases, processes can provide extra support in failure detection – for example, each process k knows the identity of the leader $leader(k)$, so if the agent's id is corrupted, then it can be detected by any non-faulty process in the system [5], and the agent can be killed. This failure detection mechanism is clearly unreliable, as it has the extra risk that a faulty process k with an incorrect value of $leader(k)$ may suspect a reliable agent to be faulty, and kill it - leaving the recovery to the leader. Any fault detection must be followed by a fresh round of leader election.

5 Multiple Agents

The motivation behind the use of multiple agents is better parallelism that can possibly reduce the time complexity or the message complexity. The number of agents to be deployed to minimize these complexities depends very much on the nature of the problem. There are cases, where a single agent solution is the best, whereas there are others in which multiple agents accelerate the progress. In some cases, the dynamic model may perform better than the static model. We now present various cases illustrating these ideas.

For notational convenience, we represent the agents by the upper case letters $I, J, K, \cdots$ Our model is a synchronous one, where in unit time, every agent takes a step. We assume that every agent visiting a process i leaves its "footprint" by writing its own id to a local variable $f(i)$, which is a set of agent identifiers. Each

agent can find out which other agents visited process i by examining $f(i)$. We will address the issue of bad data in $f(i)$ soon.

From time to time, an agent may meet with another agent to exchange data from their briefcases. On cyclic topologies, there is a risk for deadlock due to the possible scenario where each agent waits indefinitely for another agent to show up. To prevent this, we use the following rule:

Asymmetric Waiting. An agent with a larger id can wait for an agent with a smaller id, but the converse is not true.

An agent that does not wait for another agent, simply continues with its traversal, and postpones its meeting for a finite number of traversals. The following lemma is presented without proof:

Lemma 1. Both deadlock and livelock are impossible.

Fossils in the Briefcase. Fossils are bad entries in one agent's briefcase about other agents or processes that no longer exist. In a legal configuration, an agent only keeps data pertaining to other agents that it meets, or processes that it visits. The supervisory program S of every agent keeps count of the number of hops made by the agent without meeting other agents or without visiting processes that are included in its briefcase. For each such agent or process, when the count exceeds predefined limits (as determined by the application), the corresponding entries are removed from its briefcase.

A related issue is that of bad entries in $f(i)$ created by agents that are now dead, or created due to bad initialization or by transient failures. To deal with this, whenever an agent J visiting a process i discovers another agent K's footprint (J /=K) in $f(i)$, J checks out if K exists or not. This is a required step regardless of whether any data exchange is involved. If agent K shows up within a specific time period, then both of them continue with their individual protocols, otherwise the entry for $f(i)$ is deleted by agent J. Note that to enforce a timeout for the removal of bad entries in $f(i)$, active agents use the local clock of process i – no active participation from the process i is necessary. Fossil management is reminiscent of soft states used in [2] that need to be periodically refreshed to attain permanence, and is an important function of the supervisory program S. This leads to the following lemma:

Lemma 2. An agent eventually removes all entries pertaining to agents that it does not meet, or processes that it does not visit. Also, all entries in $f(i)$ that correspond to agents that never visited i or do not exist now, are eventually removed or substituted by a default value.

As a consequence of Lemma 2, we will ignore fossils in the subsequent sections.

5.1 Maxima Finding

We revisit the maxima finding problem on a ring that was solved earlier by a single reliable agent in $3(n-1)$ time. This time, we solve it on a static model containing k $(1 < k \leq n)$ agents: each has a home at a designated segment of the network. For simplicity, we assume that the ring segments are of identical size $\frac{n}{k}$ (Fig. 3). Each agent computes the maxima in its local segment (using a variation of the protocol of Fig. 1), compares the maxima when it meets a neighboring agent, and appropriately updates the value of max for the processes in their local region.

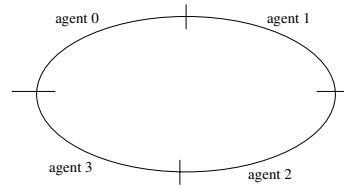


Fig. 3. Maxima finding with multiple agents.

Each agent starts with zero knowledge about the value of the maxima MAX. Thereafter, each of the k agents communicates with their neighboring agents to compute the maximum value MAX, after which this value is written into the processes (i.e $max(i) := MAX$) belonging to their respective segments. The proposed protocol is a modification of the protocol of Fig. 1. The variable $MODE$ is now used to keep track of the number of times an agent meets a neighboring agent. Initially, $MODE = 0$. When $MODE$ becomes 2, each reliable agent must have correctly read the largest id in its segment into its MAX. In addition, each agent obtains the largest id in its neighbors' segments after two meetings with each neighboring agent. There are k agents, and the farthest agent is $(k-1)/2$ segments away. This leads to the following lemma:

Lemma 3. To find out the maxima, each of the k agents must communicate with its neighboring agents at least $(k-1)$ times.

Once $MODE$ equals k, it has to move past its home at least twice to write the value of MAX to all the processes in its segment. The protocol is shown in Fig. 4.

Theorem 2. The protocol of Fig. 4 stabilizes the system to a legal configuration in which for every process i, $max(i)$ is equal to the largest id in the entire system.

Program for agent I when the number of agents > 1

agent variables MODE, MAX;
process variables max;

if MAX<id $\wedge$ MODE<k $\rightarrow$ MAX(I):=id
When agent I meets agent J
$\Box$ MAX(I)<MAX(J)$\wedge$ MODE< k $\rightarrow$ MAX(I):=MAX(J);
MODE := MODE + 1
$\Box$ MODE $\geq$ k $\rightarrow$ max(i) := MAX;
fi;
NEXT:= neighbor

Program for the home process

Executed whenever the agent reaches home
initially MAX=0, MODE=0

if MODE = k $\rightarrow$ MODE:= MODE + 1
$\Box$ MODE = k+1 $\rightarrow$ MAX:=0; MODE:=0
fi;
NEXT := neighbor

Fig. 4. Stabilizing protocol for maxima finding using multiple agents.

Proof. We first assume that the agents are reliable. Each agent has a variable $MODE \in \{0..k+1\}$. Each agent computes its MAX as $MODE$ increases from 0 to $k-1$, and writes this value into $max(i)$ for each process i in its local segment, after which MAX is reset to 0. Any bad initial value of MAX is thus removed from the system in a finite number of steps.

Consider the agent in a segment where the process with the maximum id resides. In this segment, the value of MAX is correctly set to the the largest id after $MODE$ increases from 0 to 2, and this value is maintained until $MAX = k+1$. Since the agents are not simultaneously initialized by the home process, the value of $MODE$ for a neighboring segment may be arbitrary. Regardless of this, as soon as $MODE$ is incremented from 0 to 4 in a neighboring segment, the value of MAX in that segment equals the largest id in the entire system.

Using inductive arguments we can show that the agent in the farthest segment will set its MAX to the largest id in the system after its $MODE$ reaches $2 \cdot (k-1)/2 = k-1$. Since for each segment the agent begins updating max when $MODE$ reaches k, each segment correctly updates max for all the processes belonging to its segment.

The proof for the closure of the legal configuration is trivial.

Finally, if the agents are unreliable, then eventually both MODE and MAX are reinitialized to $0, 0$ when they visit their homes and $MODE$ reaches $k+1$. Thus the agents eventually become reliable. $\Box$

The message complexity which is determined by the maximum number of hops taken by *all* the agents is $O(n \cdot k)$. Paradoxically, in this case, multiple agents increase the message complexity. The time complexity however remains unchanged at $O(n)$ as in the single agent case. The lesson is that much of the work done by these agents is unproductive, and this form of parallelism does not lead to faster stabilization.

5.2 Spanning Tree Construction

We consider a connected graph $g = (v, e)$ where the root is the home of the agent. A single-agent protocol for stabilizing DFS spanning tree generation is presented in Section 3. We now employ multiple agents to generate a spanning tree (not necessarily DFS) of g, with the hope for reducing the message or time complexity.

The Static Model. Our static model uses a fixed number k of agents ($1 \leq k \leq n$). The proposed protocol is an adaptation of Chen-Yu-Huang protocol [6] for spanning tree generation. The home of the agent with the smallest id is designated the root of the spanning tree - we call it the *root agent.* The spanning tree generation has two layers: In the first layer, the agents work independently and continue to build disjoint subtrees of the spanning tree, until they meet other agents. In the second layer, the agents meet other agents to build appropriate bridges among the different subtrees – this results in a single spanning tree of the entire graph.

For the first layer, we will use protocol of Fig. 2 with the only modification that an agent does not distinguish between a node being visited by itself, or by another agent. We will only elaborate on the second layer, where a pair of agents K, L meet to make a decision about the bridge between them during an *unplanned meeting* at some process x. We will designate a bridge by the briefcase variable BB. During the meeting, one of two agents (say K) that is yet to define its bridge, sets its briefcase variable BB to (L, x). Thereafter, node x will have two parents p_K and p_L from the two subtrees generated by K and L (see node j in Fig. 5). When node x does not have a parent in the subtree defined by agent K, $p_K = \phi$.

The maximum number of parents for any node is $min(\delta, k)$ where δ is the degree of the node, and k is the number of agents. By definition, the root agent does not have a bridge (we use $BB = \perp, \perp$ to represent this).

In addition to BB, we add another non-negative integer variable $Y (0 \leq Y \leq k)$ to the briefcase of every agent. By definition, $Y = 0$ for the root agent. Furthermore, during a meeting between two agents K, L, when K sets up its bridge to L, x, it also sets $Y(K)$ to $Y(L) + 1$. Thus, $Y(K)$ denotes "how many subtrees away" the subtree of K is from the root segment. In a consistent configuration, for every agent, $Y < k$. Therefore, if $Y = k$ for any agent, then the bridge for that subtree is invalidated.

Fig. 6 describes the protocol for building a bridge between two subtrees. The home processes initialize each BB to $\perp, \perp$ once, but like other variables, these

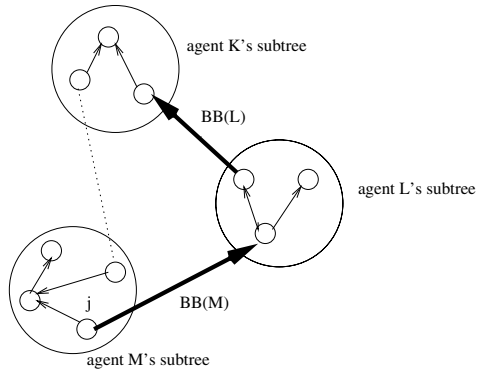


Fig. 5. The spanning tree viewed as a graph with the nodes as subtrees and the edges as bridges.

are also subject to corruption. The description of this protocol does not include the fossil removal actions.

Program for agent K while meeting agent L at node i

agent variables BB, NEXT, PRE;
process variables p;
initially $BB = \perp, \perp$;

do $BB(K) = L, i \wedge BB(L) = K, i \wedge K < L \rightarrow BB(K) := \perp, \perp$;
□ $BB(K) = \perp, \perp \wedge BB(L) \neq K, i \wedge K \neq$ *root agent* $\wedge\, Y(L) \neq k \rightarrow$
$BB(K) := L, i;\ Y(K) := Y(L) + 1$
□ $BB(K) = L, i; \wedge BB(L) \neq K, i \wedge Y(L) \neq k \wedge Y(K) \neq Y(L) + 1 \rightarrow$
$Y(K) := Y(L) + 1$
□ $BB(K) \neq L, i \wedge BB(L) = K, i \wedge p_K(i) \neq PRE \rightarrow p_K(i) := PRE$
□ $BB(K) = L, i \wedge Y(L) = k \wedge Y(K) \neq k \rightarrow Y(K) := k$;
□ $Y(K) = k \wedge BB(K) \neq L, i \wedge Y(L) < k - 1 \rightarrow$
$BB(K) := L, i;\ Y(K) := Y(L) + 1$
□ $BB(L) \neq K, i \wedge p_K(i) \neq \phi \rightarrow p_K(i) := \phi$
od;
$NEXT := PRE$

Fig. 6. Program for building a bridge between adjacent subtrees.

Theorem 3. For a given graph, if each agent independently generates disjoint subtrees, then the protocol in Fig. 6 stabilizes to a spanning tree that consists of all the tree edges of the individual subtrees.

Proof Outline. As a consequence of the fossil removal mechanism, for every agent K, eventually $BB = \perp, \perp$ or L, i, where i is a process visited by both L and K. By definition, each subtree has *exactly one bridge* BB linking it with another subtree. Draw a graph g', in which the nodes are the subtrees (excluding the bridges) of g, and the edges are the bridges linking these subtrees. Using the arguments in [6], we can show that g' will eventually be connected and acyclic. Therefore the set of edges (connecting a node with its parents) generated by the protocol of Fig. 6 define a spanning tree. □

Note that any existing spanning tree configuration is closed under the actions of the protocol.

To estimate the complexities, assume that each subtree is of equal size $\frac{n}{k}$. Let $M(s_K)$ be the number of messages required by a single agent K to build a subtree of size s_K starting from an *arbitrary initial state*. From [9], $M(s_K) = O(s_K^2)$ Also, once the subtree is stabilized, the number of messages required to traverse the subtree is $2 \cdot (s_K - 1)$. Since we assume $s_K = \frac{n}{k}$, the number of messages needed to build the k subtrees is $k \cdot M(\frac{n}{k})$. To estimate the number of messages needed to detect a cycle in the graph g' using the condition $Y \geq k$, consider a cycle $s_0 s_1 \cdots s_t s_0$ $(t \leq k)$ in g', where each node is a subree. To correctly compute Y, each agent has to read the value of Y from the agent in its predecessor segment. This can take upto $1+2+3+\cdots+(t-1) = \frac{t(t-1)}{2}$ traversals of the subtrees. Since the maximum value of t is k, for correctly detecting cycles in g', at most $\frac{k(k-1)}{2} \cdot \frac{2n}{k}$ will be required. Also, each time a cycle is broken, the number of disjoint subtrees in g' is reduced by one (see [6]), so this step can be repeated no more than $(k-1)$ times. Therefore the maximum number of messages needed for the construction of a spanning tree using a set of cooperating reliable agents will not exceed

$$k \cdot O(\tfrac{n^2}{k^2}) + (k \cdot \tfrac{k(k-1)}{2} \cdot \tfrac{2n}{k}) = O(\tfrac{n^2}{k} + n \cdot k^2)$$

To estimate the worst-case message complexity, we also need to take into account the overhead of fossil removal. This is determined by the number of hops taken by the agents to traverse the subtrees of size $\frac{n}{k}$, which is $O((\frac{n}{k}) \cdot k) = O(n)$. Note that this does not increase the order of the message complexity any further. The interesting result, at least with this particular protocol is that as the number of agents increases, the message complexity first decreases, and then increases. The minimum message complexity is $O(n^{\frac{5}{3}})$ when $k = O(n^{\frac{1}{3}})$.

To estimate the time complexity, assume that each of the k agents simultaneously builds subtrees of size $\frac{n}{k}$ in time $O(\frac{n^2}{k^2})$. The time required by the k agents to correctly establish their Y values is $k \cdot O(\frac{n}{k}) = O(n)$. At this time, the condition $Y \geq k$ can be correctly detected. The resulting actions reduce the number of disjoint subtrees by 1, so these action can be repeated at most $(k-1)$ times. The time complexity is thus $O(\frac{n^2}{k^2} + n \cdot k)$. The overhead of fossil removal (which is $O(\frac{n}{k})$) does not increase the time complexity any further. Therefore, the smallest value of the time complexity is $O(n^{\frac{4}{3}})$ when $k = O(n^{\frac{1}{3}})$.

The Dynamic Model. Finally, we examine the dynamic model of multiple agents, where an agent can create multiple agents as and when necessary, dele-

gate subtasks to these agents, and kill them when they complete their subtasks. There is a strong similarity between this approach and that used in many wave algorithms [15]. The key ideas are:

1. When an agent reaches a "fork" (defined as a node of degree δ, $\delta > 2$) it creates $\delta - 1$ child agents, one for each remaining edge, for traversing the rest of the graph.
2. When an agent visits a node that (i) has a degree $\delta = 1$, or (ii) has already been visited by another agent, it retreats.
3. When the children of an agent return to their parent, the parent retrieves the required data from the child agents, and then kills them.

Agents spawn children with a limited lifespan and a limited agenda. When an agent spawns child agents at a node i, that node plays the role of the home for its children. The parent installs its own programs A and S into the child agents[6]. The children of an agent I bear the identification tag $I : J$, and the parent agent assigns distinct values of J to each child. Each child follows the single agent protocol, until it returns to its parent agent.

For each process i, we recognize at most one agent K that reaches that node first. Accordingly, $f(i)$ is set to (K, SEQ) where SEQ is a boolean flag representing the sequence number of the most recent visit by K. When a process has not been visited by any agent, $f(i) = \perp, \perp$. For fossil management, any agent K, after reaching a node i that claims to have been visited by another agent L ($L \neq K$), waits for L to show up. If indeed L shows up, then both K and L continue with their protocols, otherwise agent K resets $f(i)$ to $\perp, \perp$. The program for each agent is shown in Fig. 7 that assumes all agents to be valid and reliable.

Program for agent K while visiting node i

agent variables NEXT, PRE, SEQ;
process variables f, child, p;

if (f(i) = L, - ∨ f(i) = K, SEQ) ∧ PRE /∈child(i)→ NEXT:= PRE **fi**;
if f(i) = K, 1- SEQ →
    Mark the current node as visited and set the parent
    f(i):= K, SEQ; p(i) := PRE;
    if $\delta > 2 \rightarrow \forall$ j ∈ neighbor(i)\p(i): create a child agent with NEXT:=j
    □ all child agents are back → kill the child agents; NEXT:= p(i)
    fi;
fi

Fig. 7. The spanning tree protocol in the dynamic model of multiple agents.

We now examine the additional types of agent failures in the dynamic model of multiple agents. Each child agent eventually returns to its parent whose iden-

[6] There is one limitation – we expect that at least S will be correctly installed.

tity can be derived by stripping the last component of its own id. The death of the parent agent converts its child agents into orphans. Using the supervisory program S, each child agent increments a counter with every hop that it takes, until it meets its parent agent. As a result, an orphan unable to locate its parent eventually finds the value of this counter exceeding the maximum possible size of the system. At this point, the orphan kills itself.

Due to the overhead of fossil management, the message complexity of the above protocol is not better than that of the single-agent protocol. However, the time complexity is $O(h^2)$ where h the height of the spanning tree with the home as the root node. This result is not surprising, but is encouraging for dense graphs.

6 Concluding Remarks

This paper demonstrates that agents can be a viable tool for implementing stabilizing distributed systems. It also demonstrates the power of a single focus of control.

The agent model can tolerate the corruption of the agent program A, if the home process correctly installs this every time the agent visits home. Note that the supervisory program S must be incorruptible, since it has the crucial responsibility of failure detection.

It is possible to solve a stabilization problem by first devising an agent-based protocol, and then implementing the agent(s) using the method outlined in Section 4. The complexity of the overall solution is determined by the sum of (i) the complexity of the agent-based protocol, and (ii) the complexity of stabilizing the agents which themselves could possibly be unreliable and (iii) the complexity of a stabilizing protocol for leader election. We claim that in many cases, the overall complexity is comparable to that of a traditional message-based solution. When the agent fails less frequently than the underlying distributed system, there is a potential to amortize the overhead due to the second component, which further reduces the stabilization time.

The other noteworthy observation is the issue of productive and unproductive parallelism. The observation that the time complexity of single-agent protocols is sometimes comparable to that achieved by distributed demons or alternators [10] reveals that uncontrolled parallelism does not necessarily lead to faster stabilization. Multiple agents, installed at appropriate processes, or spawned at appropriate times have the potential of achieving productive parallelism, and therefore faster stabilization. The multi-agent protocols are examples of divide-and-conquer strategies in stabilization that needs further exploration.

The biggest advantage of agent stabilization seems to be the ease of derivation of single-agent protocols from known sequential algorithms for graphs, conversion of those into multi-agent protocols using from the known techniques for parallelization. On the negative side, agent-based protocols are not silent. The time delay between the occurrence of a failure and the next visit of the agent to address that failure may be a matter of concern if there are too few agents, or if the network is very large.

Acknowledgment

The author thanks Ted Herman for many constructive criticisms.

References

1. Araragi T, Attie P, Keidar I, Kogure K, Luchanugo V, Lynch N, Mano K. On formal modeling agent computations. NASA Workshop on Formal Approaches to Agent-based Systems (2000).
2. Wang YM, Russell W, and Arora A. A toolkit for building dependable and extensible home networking applications. Fourth USENIX Windows Systems Symposium USENIX-WIN 2000 (2000)
3. Arora A, Gouda MG. Closure and convergence: A foundation of fault-tolerant computing. IEEE Transactions on Software Engineering 19 (1993) pp. 1015-1027.
4. Arora A, Gouda MG. Distributed reset. IEEE Transactions on Computers 43 (1994) pp. 1026-1038, 1994.
5. Chandra TD, Toueg S. Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43:2 (1996) pp. 225-267.
6. Chen NS, Yu HP, Huang ST. A self-stabilizing algorithm for constructing a spanning tree. Information Processing Letters 39 (1991) pp.147-151.
7. Dijkstra EW. Self stabilizing systems in spite of distributed control. Communications of the ACM 17 (1974) pp. 643-644.
8. Dorigo M, Maniezzo V, and Colorni A. The Ant system: Optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics 26:1 (1996) pp. 1-13.
9. Ghosh S. Agents, distributed algorithms, and stabilization. In Computing and Combinatorics (COCOON 2000), Springer-Verlag LNCS1858 (2000), pp. 242-251.
10. Gouda MG, Haddix F. The alternator. Proceedings of the Third Workshop on Self-Stabilizing Systems. (published in association with ICDCS'99. The 19th IEEE International Conference on Distributed Computing Systems), pages 48-53, 1999.
11. Herman T. Self-stabilization: randomness to reduce space. Distributed Computing 6 (1992) pp. 95-98.
12. Korach E, Kutten S, and Moran S. A modular technique for the design of efficient leader finding algorithms. ACM Transactions on Programming Languages and Systems 12 (1990), pp. 84-101.
13. Katz S, Perry K. Self-stabilizing extensions for message-passing systems. Proceedings of 9th Annual Symposium on Principles of Distributed Computing (1990) pp. 91-101.
14. Gray R, Kotz D, Nog S, Rus D, Cybenko G. Mobile agents for mobile computing. Proceedings of the Second Aizu International Symposium on Parallel Algorithms/Architectures Synthesis, Fukushima, Japan (1997).
15. Tel G. *Introduction to Distributed Algorithms.* Cambridge University Press (1994).
16. Varghese G. Self-stabilization by counter flushing. SIAM Journal on Computing, 30(2):486-510, 2000.
17. White T, and Pagurek B. Towards multi-swarm problem solving in networks. Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS'98), July 1998, pp. 333-340.

Cross-Over Composition - Enforcement of Fairness under Unfair Adversary

Joffroy Beauquier, Maria Gradinariu, and Colette Johnen

Laboratoire de Recherche en Informatique, UMR CNRS 8623
Université de Paris Sud, F91405 Orsay cedex, France
{jb,mariag,colette}@lri.fr

Abstract. We study a special type of self-stabilizing algorithms composition : the cross-over composition ($A \diamond B$). The cross-over composition is the generalization of the algorithm compiler idea introduced in [3]. The cross-over composition could be seen as a black box with two entries and one exit. The composition goal is to improve the qualities of the first algorithm A, using as medium the second algorithm B. Informally, the obtained algorithm is A after the transfer of B's properties.
Here, we provide a complete analysis of the composition, when the algorithms (A and B) are deterministic and/or probabilistic algorithms. Moreover, we show that the cross-over composition is a powerful tool in order to enforce a scheduler to have a fair behavior regarding to A.

1 Introduction

The idea of composing self-stabilizing algorithms in order to improve their adaptability was introduced by Gouda and Herman in [8]. In their approach an algorithm is composed by a number of k layers such that the layer i, $1 < i \leq k$ depends on the variables which stabilize due to the actions of the layers from 1 to $i-1$. The proof of convergence of the composed algorithm follows by induction.

In the same paper, the authors present another type of composition which uses a selection predicate. The two modules which enter in the composition do not inter-communicate, but they are allowed to modify the same output variables. At a given time, the selection predicate is true only for one module (module that is allowed to modify the output variables) while the other module is waiting the flipping of predicate value.

Another type of independent module composition was defined by Varghese in [12]. The entities interact by means of their outputs. The obtained algorithm is the composition of the modules.

A special form of composition was defined by Dolev and Herman in [5]. The goal of this composition is to accelerate the self-stabilization of an algorithm P. P will pick up the result of the fastest self-stabilizing algorithm of (S_i), $i \in I$ in order to perform its own task. This technique needs some fair scheduler.

The cross-over composition $A \diamond B$ goal is to improve the qualities of the algorithm A, using as medium the second algorithm B. Informally, the obtained algorithm

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 19–34, 2001.

is the algorithm A after the transfer of B's computation properties. The main use of the cross-over composition is the transformation of a self-stabilizing algorithm A under weak scheduler (central, alternating, k-fair, fair, ...) into an algorithm $(A \diamond B)$ which maintains the self-stabilization property under any unfair scheduler.

Moreover, we guarantee that the composed algorithm will satisfy the conjunction of the properties of the algorithms as in the Varghese composition. We show that all liveness and safety properties of A and B are also conveyed by $A \diamond B$ iff B is fair when A and B are deterministic and/or probabilistic algorithms.

In Sec. 2, the model and self-stabilization definitions for deterministic and probabilistic algorithm are given. The cross-over definition is presented in Sec. 3. In Sect. 4, we study the propagation of the self-stabilization property. We explain how to use the cross-over composition to transform any self-stabilizing algorithm under some specific scheduler into an algorithm that converges under any unfair scheduler in Sec. 5.

2 Model

Distributed Systems. A distributed system can be modeled by a transition system. A transition system is a three-tuple $S = (\mathcal{C}, \mathcal{T}, \mathcal{I})$ where $\mathcal{C}$ is the collection of all configurations, $\mathcal{I}$ is a subset of $\mathcal{C}$ called the set of initial configurations, and $\mathcal{T}$ is a function from $\mathcal{C}$ to the set of $\mathcal{C}$ subsets. A $\mathcal{C}$ subset of $\mathcal{T}(c)$ is called a c transition. An element of a c transition t, is called an output of t. In a probabilistic system, there is a probabilistic law defined on the output of a transition; in a deterministic system, each transition has only one output. In Fig. 2, we can see the C00 transition called CH00 that has four outputs: C11, C12, C21 and C22.

The abstract model defined above is a mathematical representation of the reality. In fact, the distributed system is the collection of processors that communicate only with theirs processor neighbors to execute a distributed algorithm.

A *computation* of a distributed system DS is a sequence of computation steps. A *maximal* computation is a sequence such that it is either infinite, or with a deadlock terminal configuration. The computations set of a distributed system DS is denoted by $\mathcal{E}_{DS}$. A maximal computation e is *fair* if and only if any processor performs infinity often an action. A fair computation e is *k-fair* if and only if between two actions of a processor, any other processor performs at most k actions. A maximal computation e is *k-bounded* if and only if along e, till a processor p is enabled to perform an action, another processor can perform at most k actions. A k-fair computation is k-bounded; but the converse is not true. When the distributed algorithm prevents the fairness because some processors are no more enabled, the k-bounded property guarantees the fairness between "enabled" processor. On a network of 4 processors ($p1$, $p2$, $p3$, $p4$), the following computation is not 1-fair $(p1's\ action, p3's\ action)^*$ but it is 1-bounded if along this computation $p2$ and $p4$ are never enabled to perform any action.

Scheduler. In this model, a *scheduler* is a *predicate* over the system computations. In a computation, a transition (c_i, c_{i+1}) occurs due to the execution of a nonempty subset of the enabled processors in the configuration c_i. In every computation step, this subset is chosen by the scheduler. At a computation step, a *central scheduler* chooses an enabled processor to execute its action; A *distributed unfair scheduler* chooses any nonempty subset of the enabled processors at each computation step. A *k-bounded scheduler* produces only k-bounded computations: it ensures the k-fairness between processors that are enabled to perform an action. An *alternating scheduler* produces only alternating computations: between two actions of a processor p each p's neighbor performs one and only one action.

An algorithm under a scheduler D is *fair* (resp. k-fair) if any computation of the algorithm under D is fair (resp. k-fair). When the property of fairness (resp. k-fairness) is verified by an algorithm under any scheduler then the algorithm is simply called fair (resp. k-fair).

Built on previous works on probabilistic automata (see [11,13,10]), [4] presented a framework for proving self-stabilization of probabilistic distributed systems based on the notion of strategy. A strategy is the set of computations that can be obtained under a specific scheduler choice. At the initial configuration, the scheduler "chooses" one set of enabled processors (it chooses a transition). For each output of the selected transition, the scheduler chooses a second transition, and so on. The formal strategy definition is based on the tree of computations. Let c be a configuration. A *TS-tree* rooted in c, $\mathcal{T}ree(c)$, is the tree-representation of all computations beginning in c. Let nd be a node in $\mathcal{T}ree(c)$ (i.e. a configuration), a *branch* rooted in nd is the set of all $\mathcal{T}ree(c)$ computations starting in nd with a computation step of the same nd transition. The *degree* of nd is the number of branches rooted in nd. A *sub-TS-tree of degree* 1 rooted in c is a restriction of $\mathcal{T}ree(c)$ such that the degree of any $\mathcal{T}ree(c)$'s node (configuration) is at most 1. Figure 2 contains a strategy rooted in C00. A strategy may have a non-countable number of infinite computations. A strategy is defined as follows:

Definition 1 (Strategy). *Let DS be a distributed system, let D be a scheduler and let c be a configuration. We call a strategy of DS under D rooted in c a sub-TS-tree of degree 1 of $\mathcal{T}ree(c)$ such that any computation of the sub-tree satisfies the scheduler D.*

Let st be a strategy of the distributed system DS, an *st-cone* $\mathcal{C}_h$ is the set of all possible st-computations with the same prefix h (for more details see [10]). The last configuration of h is denoted $last(h)$.

We have equipped a strategy with a probabilistic space (see [4] for more details). The measure of an st-cone $\mathcal{C}_h$ is the measure of h (i.e., the product of the probability of every computation step occurring in h). An st-cone $\mathcal{C}_{h'}$ is called a *sub-cone* of $\mathcal{C}_h$ if and only if h is a prefix of h'. Let st be the strategy of Fig. 2; let h be the prefix $(C00, ch00, C12)(C12, ch12, C56)$; in st, the probability of $\mathcal{C}_h$ is $p_A^2 \cdot (1 - p_B)^2$.

Deterministic Self-Stabilization. In order to define self-stabilization for a distributed system, we use two types of predicates: the legitimate predicate (defined on the system configurations and denoted by $\mathcal{L}$) and the problem specification (defined on the system computations and denoted by $\mathcal{PS}$). To prove the self-stabilization to $\mathcal{SP}$, one has to prove that all computations reach a legitimate configuration and that from a legitimate configuration any computation satisfies the predicate $\mathcal{SP}$. For instance, the leadership problem specification is "there is one leader in the network, called p, and p stays the only leader forever". In this case, a legitimate configuration would be a configuration where there is one and only one leader. The correctness proof consists in ensuring that the system does not diverge from a legitimate configuration: once p is the only leader, no other processor becomes leader and p keeps its leadership.

Let $\mathcal{X}$ be a set and $Pred$ be a predicate defined on $\mathcal{X}$. The notation $x \vdash Pred$ means that the element x of $\mathcal{X}$ satisfies the predicate $Pred$.

Definition 2 (Deterministic Self-Stabilization). *Let DS be a distributed system. DS is self-stabilizing for a specification $\mathcal{PS}$ if and only if the following two properties hold:*

- convergence — *all computations of DS reach a configuration that satisfies the legitimate predicate denoted $\mathcal{L}$. Formally, $\forall e \in \mathcal{E}_{DS} :: e = ((c_0, c_1)(c_1, c_2)\ldots) : \exists n \geq 1, c_n \vdash \mathcal{L}$;*
- correctness — *all computations starting in configurations satisfying the legitimate predicate satisfy the problem specification $\mathcal{PS}$. Formally, $\forall e \in \mathcal{E}_{DS} :: e = ((c_0, c_1)\ (c_1, c_2)\ \ldots) : c_0 \vdash \mathcal{L} \Rightarrow e \vdash \mathcal{PS}$.*

Probabilistic Self-Stabilization. Let DS be a distributed system. A predicate P is closed for the computations of DS if and only if when P holds in a configuration c, P also holds in any configuration reachable from c.

Notation 1. *Let DS be a distributed system, D be a scheduler and st be a strategy of DS under D. Let CP be the set of all system configurations satisfying a closed predicate P (formally $\forall c \in CP, c \vdash P$). The set of st-computations that reach configurations of CP is denoted by $\mathcal{EP}$ and its probability by $Pr_{st}(\mathcal{EP})$.*

Definition 3 (Probabilistic Stabilization). *A distributed system DS is* self-stabilizing *under a scheduler D for a specification $\mathcal{PS}$ if and only if there exists a closed legitimate predicate $\mathcal{L}$ defined on configurations such that in any strategy st of DS under D, the following conditions hold:*

- convergence — *The probability of the set of st-computations, that reach a configuration satisfying $\mathcal{L}$ is 1. Formally, $\forall st, Pr_{st}(\mathcal{EL}) = 1$.*
- correctness — *Any computation starting in a configuration satisfying $\mathcal{L}$ satisfies the specification $\mathcal{PS}$.*

3 Cross-Over Composition

3.1 Definitions

In the sequel, we define the cross-over composition $A \diamond B$. The cross-over composition could be seen as a black box with two independent entries (two algorithms that do not share any variable) and one exit. The composition goal is to improve the qualities of the first Algorithm A, using as medium the second Algorithm B. The two algorithms which enter in the composition have different parts. A, referred in the following as the weak algorithm is the target of the transformation. B referred as the strong algorithm is the transformation medium which transfers its properties to the weak algorithm.

The actions of A are synchronized with the actions of B: when an A action is performed then a B action is performed too. Thus, the computations of the composite algorithm under any scheduler have the same properties as the computations of B in term of fairness.

- when a processor p performs an action of A it performs simultaneously an action of B (both action guards were satisfied on p);
- a processor p may perform an action of B without performing an action of A (in this case all action guards of A are disabled on p).

The strong algorithm B acts as a computation filter for the weak algorithm A: A will only deal with computations that can be obtained by B under the current scheduler: D. The obtained algorithm $A \diamond B$ has the properties of B under D and the properties of A under a scheduler that produces "B's computations".

Definition 4.
Let A be an algorithm with n actions as follows:
$\forall i \in \{1, \ldots, n\} \quad < guard\ a_i > \quad \Rightarrow \quad < action\ a_i >$
Let B be an algorithm with m actions as follows:
$\forall j \in \{1, \ldots, m\} \quad < guard\ b_i > \quad \Rightarrow \quad < action\ b_j >$
Assume that A and B do not share any variable. The cross-over composition $A \diamond B$ is the algorithm with the $m.(n+1)$ following actions:
$\forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\}$
$< guard\ a_i > \wedge < guard\ b_j > \quad \Rightarrow \quad < action\ a_i >; < action\ b_j >$
$\forall j \in \{1, \ldots, m\}$
$\neg < guard\ a_1 > \wedge \ldots \wedge \neg < guard\ a_n > \wedge < guard\ b_j > \quad \Rightarrow \quad < action\ b_j >$

Example 1. Let r be a unidirectional ring of n processors. Algorithm B has one integer variable v and one action: on any processor p (lp being the left neighbor of p), $v_p \leq v_{lp} \Rightarrow v_p := v_{lp} + 1$. Any computation of B has a suffix that is alternating. Algorithm A has two variables $Currentlist$ and $BackupList$ (list of processor ids) and one action (without guard): $\Rightarrow$ p copies the content of $CurrentList$ of its left neighbor into its own $CurrentList$; then it concatenates its own id at the end of the list. If p id appears two times in this list, p copies the segment between its ids into the $backupList$. If in p's $CurrentList$, p's id

appears three times then p empties its $CurrentList$. The cross-over composition $A \diamond B$ has three variables: one integer and two lists of processors id; and the following action:

b1. $v_p \leq v_{lp} \quad \Rightarrow \quad v_p := v_{lp} + 1$; p copies the content of $CurrentList$ of its left neighbor into its own $CurrentList$; then ... .

As the computations of $A \diamond B$ have an alternating suffix; one may prove that the algorithm stabilizes: every $BackupList$ list will contain the ordered list of processors id on the ring.

A probabilistic self-stabilizing leader election algorithm on anonymous ring is presented in [3]. This algorithm is the cross-over composition of three algorithms $(L \diamond RTC) \diamond DTC$: DTC ensures that the computations are k-fair, RTC provides a token circulation on the ring if the computations are k-fair, and L manages the leadership under the assumption that a token circulates in the ring.

Observation 1. *One may notice that the algorithm $A2 \diamond (A1 \diamond B)$ is not the algorithm $(A2 \diamond A1) \diamond B$. To prove that one may study the cross-over composition of three simple algorithms (i.e. each algorithm has one action).*

3.2 Deterministic Properties Propagation

In the following, we study the propagation of deterministic algorithm properties on the obtained algorithm.

Lemma 1 (Propagation of Properties on Computations). *Let $A \diamond B$ be the cross-over composition between the algorithms A and B. Let P be a predicate on B's computations. If any maximal computation of B under the scheduler D satisfies the predicate P then any maximal computation of $A \diamond B$ under D satisfies P.*

Proof. Assume that there is at least one maximal computation of $A \diamond B$, e, under D which does not satisfy the predicate P. The projection of e on B is unique and maximal. Let e_B be this projection. Since e does not satisfy the predicate P then e_B does not either which contradicts the Lemma hypothesis.

Corollary 1 (Propagation of Fairness). *Let $A \diamond B$ be the cross-over composition between the algorithms A and B. If Algorithm B is fair under D then $A \diamond B$ is a fair algorithm under D.*

Corollary 2 (Propagation of k-Fairness). *Let $A \diamond B$ be the cross-over composition between the algorithms A and B. If B is k-fair under a scheduler D then $A \diamond B$ is k-fair under D.*

The proof of the following Lemma is similar to the proof of the Lemma 1.

Lemma 2 (Propagation of Convergence Properties). *Let $A \diamond B$ be the cross-over composition between the algorithms A and B. Let P be a predicate on the B's configurations. If any maximal computation of B under the scheduler D reaches a configuration which satisfies the predicate P then any maximal computation of $A \diamond B$ under D reaches a configuration which satisfies P.*

Corollary 3 (Propagation of Liveness). *Let $A \diamond B$ be the cross-over composition between the algorithms A and B. If B is without deadlock under the scheduler D then $A \diamond B$ is without deadlock under D.*

Lemma 3 (Maximality of the Weak Projection). *Let $A \diamond B$ be the cross-over composition between A and B. If B is fair under the scheduler D then the projection on A of any maximal computation of $A \diamond B$ under D is maximal.*

Proof. Let e be a maximal computation of $A \diamond B$. Let e_A be the projection of e on A. Assume that e_A is not maximal. Hence e_A is finite and its last configuration is not a deadlock. Let e be $e = e_1 e_2$ where the projection of e_1 on A is e_A and the projection of e_2 is a maximal computation which does not contain any action of A. Let c be the last configuration of e_1. The projection of this configuration on A is not a deadlock, then there is at least one processor, p, which satisfies a guard of A in c. Using the fairness of B and Corollary 1 we prove that p executes an action of B in e_2. According to the definition of the cross-over composition, during the computation of p's first action in e_2, it executes an action of A. There is a contradiction with the assumption that no action of A is executed in e_2.

3.3 Propagation on a Simple Probabilistic Cross-Over Composition

In this section, we study the properties of a strategy of $A \diamond B$ when X ($= A$ or B) is a probabilistic algorithm and the other one is a deterministic algorithm. Figure 1 displays an example of such a cross-over composition.

Lemma 4. *Let X be a probabilistic algorithm, let Y be a deterministic one, and let $Z = X \diamond Y$ and $W = Y \diamond X$ be their cross-over composition. Let st be a strategy of Z or W under the scheduler D. Let st_X be the projection of st on the algorithm X. st_X is a strategy of X under D.*

Proof Outline: Let c be the initial configuration of st, and c_X its projection on X. Let $\mathcal{TS}(c_X)$ be the tree representation of X computations beginning at c_X. Let st' be the sub-tree of $\mathcal{TS}(c_X)$ that contains all computations that are st projections. st' is a strategy: all computation steps beginning at n (a node) in st', belong to the same transition: nd; and all computation steps of nd transition are in st'.

Theorem 1. *Let X be a probabilistic algorithm, let Y be a deterministic one, and let $Z = X \diamond Y$ and $W = Y \diamond X$ be their cross-over composition. Let st be a strategy of Z or W under the scheduler D. Let st_X be the projection of st on X. Let PC_X be a predicate over the $X's$ configurations, then $Pr_{st}(\mathcal{EPC}_{\mathcal{X}}) = Pr_{st_X}(\mathcal{EPC}_{\mathcal{X}})$. Let PE_X be a predicate over the $X's$ computations, then $Pr_{st}\{e \in st \mid e \vdash PE_X\} = Pr_{st_X}\{e' \in st_X \mid e' \vdash PE_X\}$.*

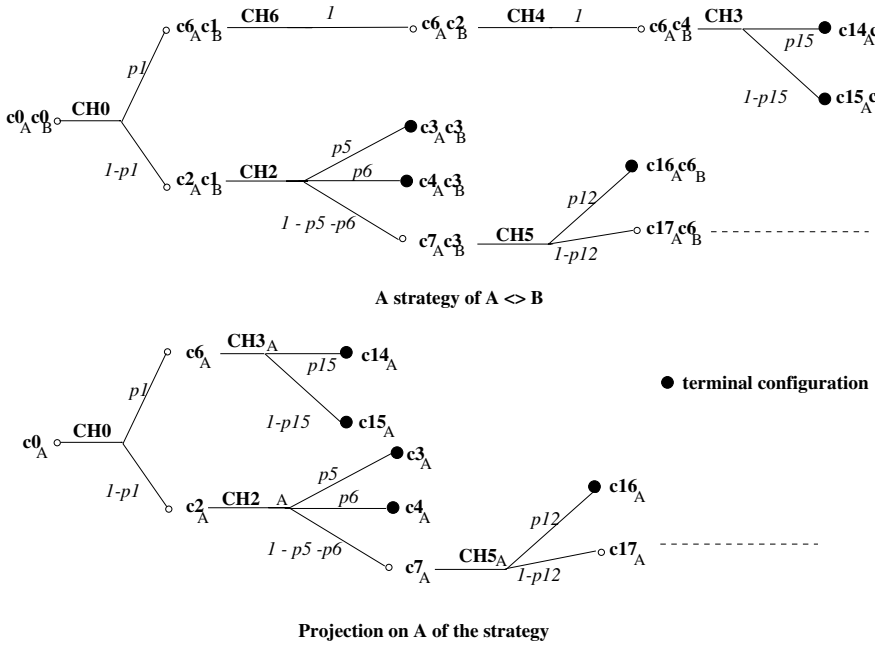


Fig. 1. The projection of a strategy of A ⋄ B on A (B being a deterministic algorithm).

Proof Outline: Let $\mathcal{C}_h$ be a cone of st. The projection of $\mathcal{C}_h$ on X is a cone of st_X: $\mathcal{C}_{h'}$ where h' is the projection of h on X.

3.4 Propagation on a Double Probabilistic Cross-Over Composition

In this section, we study the properties of a strategy of $A \diamond B$ when A and B are probabilistic algorithms. The projection of a strategy on an algorithm is not a strategy (Fig. 3 is the projection on B of the strategy of Fig. 2). The projection is decomposed in strategies.

Definition 5 (Derived Strategies). *Let st be a strategy of $A \diamond B$ and let st_{projX} be the projection tree obtained after the projection of the computations in st on X ($X = A$ or $X = B$). A derived strategy of st_{projX} is a subtree of st_{projX} whose degree equals 1.*

Observation 2. *Let st be a strategy of $A \diamond B$. Let st_{projX} be the projection of st on X ($X = B$ or A). For the sake of simplicity, we assume that $X = B$. Let st_B be a strategy derived from st_{projB}. Each cone of computations in st_B, $\mathcal{C}_{h|B}$, is the projection of a cone of st, $\mathcal{C}_h$, which probability is given by the probability of $\mathcal{C}_{h|B}$ in st_B multiplied by δ^{hB}. The weight of the cone of history $h|B$ in st_B (denoted by δ^{hB}) is the probability of the A computation steps executed in the history of $\mathcal{C}_h$. Hence, $Pr_{st}(\mathcal{C}_h) = \delta^{hB} \cdot Pr_{st_B}(\mathcal{C}_{h|B})$.*

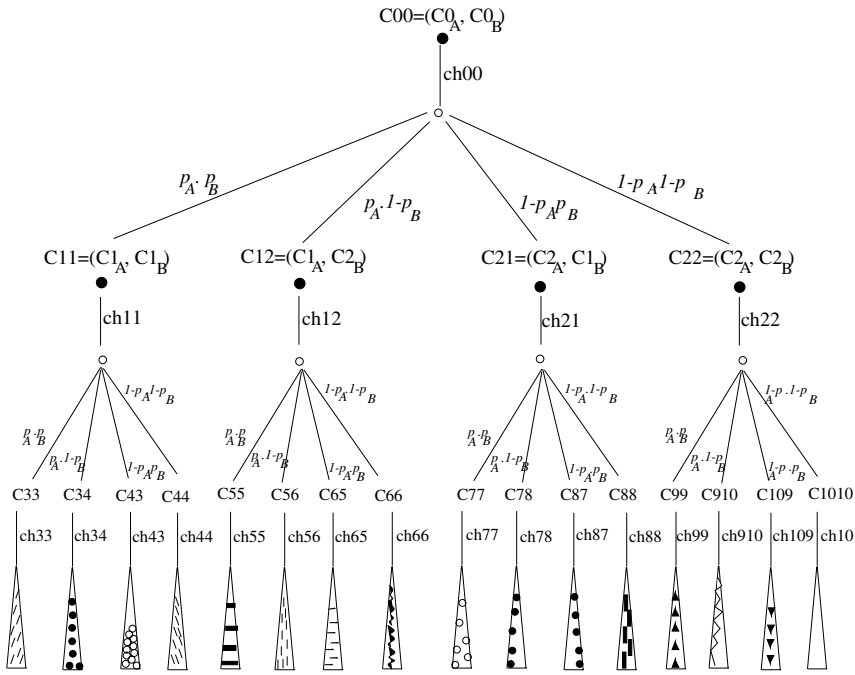


Fig. 2. The beginning of the strategy st of $A \diamond B$.

Definition 6 (Projection Strategies on X). *We call a projection strategy st_X a derived strategy of st_{projX} such that all cones of st_X having the same length have the same weight. We note $\delta_n^{st_X}$ the weight of n-length cones of st_X.*

For instance, Fig. 4 and Fig. 5 contain 4 projection strategies on B of the strategy *st* (Fig. 2). Each strategy has a different δ_2 value.

Observation 3. *Let N be an integer. Let st be a strategy of $A \diamond B$. Let st_{projX} be the projection of st on X ($X = B$ or A). For the sake of simplicity, we assume that $X = B$. Let M^B be the set of projection strategies of st_{projB}. There is a finite subset of M^B, denoted M_N^B such that $\sum_{st_B \in M_N^B} \delta_N^{st_B} = 1$. Such a subset of M is called a N-length B picture of st.*
Each N-length cone of st has one and only one projection on B in M_N^B.
There are several subsets of M_B that are "N-length B pictures of st".

The 4 strategies of Fig. 4 and Fig. 5 constitute a M_2^B set. We have $\sum_{i=1}^{i=4} \delta_2^{st_i} = 1$

We show that if any strategy of an algorithm for which the set of computations reaching a configuration satisfying a predicate P has the probability 1, then in any strategy of the composed algorithm, this set has the probability 1.

Lemma 5 (Probabilistic Propagation). *Let $\mathcal{L}$ be a predicate over the X's states ($X = A$ or $X = B$). Let st be a strategy of $A \diamond B$ under a scheduler*

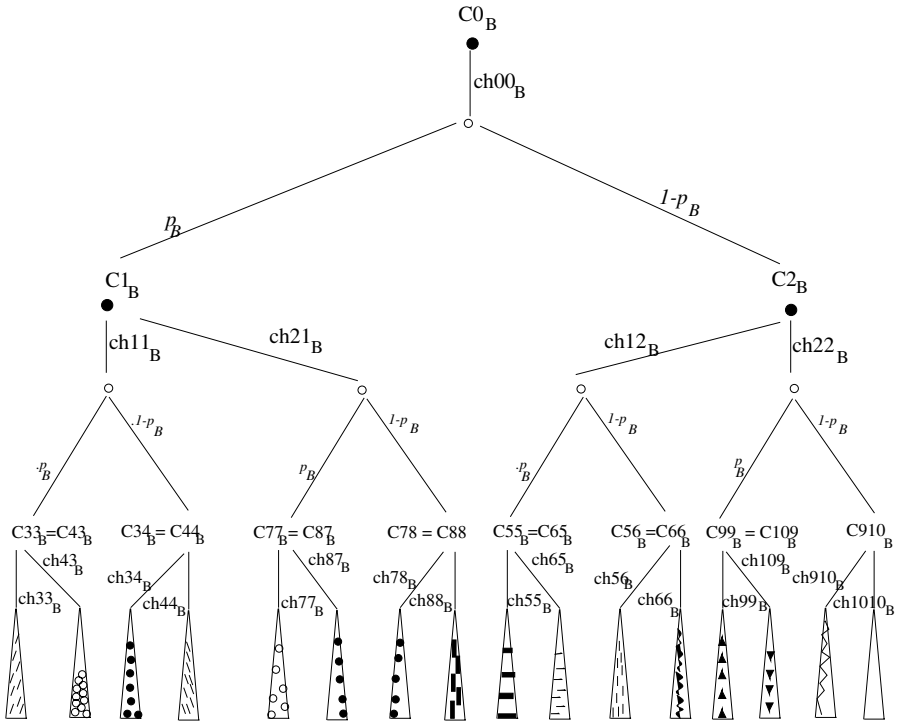


Fig. 3. The beginning of the projection of the strategy st on B.

D. If for every strategy st_X, being a projection strategy of st on X, we have $P_{st_X}(\mathcal{EL}) = 1$ then $P_{st}(\mathcal{EL}) = 1$.

Proof. Let *st* be a strategy of $A \diamond B$. Let st_{projX} be the projection of *st* on X ($X = B$ or A). We assume that $X = B$. Let M_B be the set of projection strategies of st_{projB}.

We denote by $\mathcal{EL}_N$ the union of the cones of a given strategy that have the following properties: (i) their history length is N and (ii) they have reached a legitimate configuration.

Let ϵ be a real inferior to 1. By hypothesis, there is an integer N such that on any strategy st_B of M_N^B (a N-length B picture of *st*) we have $Pr_{st_B}(\mathcal{EL}_N) \geq 1 - \epsilon$.
$Pr_{st}(\mathcal{EL}_N) = \sum_{st_i \in M_N^B} [Pr_{st_i}(\mathcal{EL}_N) \cdot \delta_N^{st_i}] \geq (1 - \epsilon) \cdot \sum_{st_i \in M_N^B} \delta_N^{st_i} \geq 1 - \epsilon$. In *st*, the set of computations reaching legitimate configurations in $N' \leq N$ steps has a probability greater than $1 - \epsilon$.

Therefore, for any sequence $\epsilon_1 > \epsilon_2 > \epsilon_3 ...$ there is a sequence $N_1 \leq N_2 \leq N_3 ...$ such that $Pr_{st}(\mathcal{EL}_{N_i}) \geq 1 - \epsilon_i$. Then, $\lim_{n \to \infty} P_{st}(\mathcal{EL}_n) = P_{st}$(computations reaching a legitimate configuration) $= 1$.

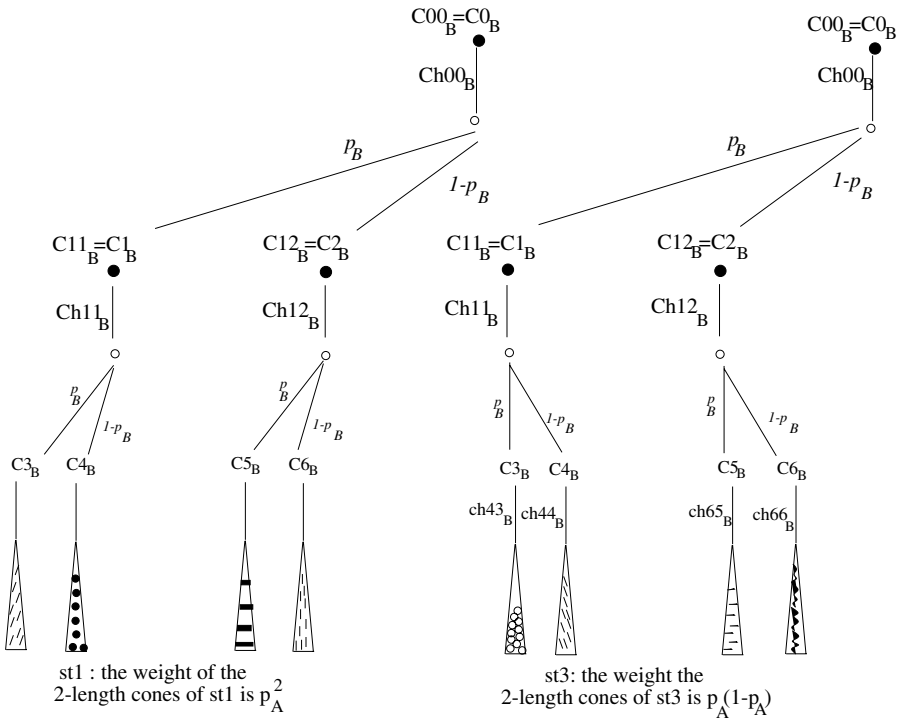


Fig. 4. 2-length beginning of projection strategies on B.

4 Cross-Over Composition and Self-Stabilization

In the sequel, we study the propagation of the self-stabilization property from an algorithm to the resulting algorithm of a cross-over composition. The propagation with self-stabilization (in the deterministic case) is a direct consequence of Lemma 1 and 2 when the strong algorithm (B) is the propagation initiator.

Lemma 6. [Self-Stabilization Propagation to the Deterministic Algorithm $A \diamond B$ from Algorithm B] *Let $A \diamond B$ be the cross-over composition between the deterministic algorithms A and B. If Algorithm B self-stabilizes for the specification SP under the scheduler D then $A \diamond B$ is self-stabilizing for SP under D.*

Proof. The proof is a direct consequence of the Lemma 1 and 2. In order to prove the convergence we apply Lemma 1 for the property which characterizes the legitimate configurations. The correctness proof results from Lemma 2 applied for the specification SP.

In order to ensure the liveness of the weak algorithm, the strong algorithm must be fair.

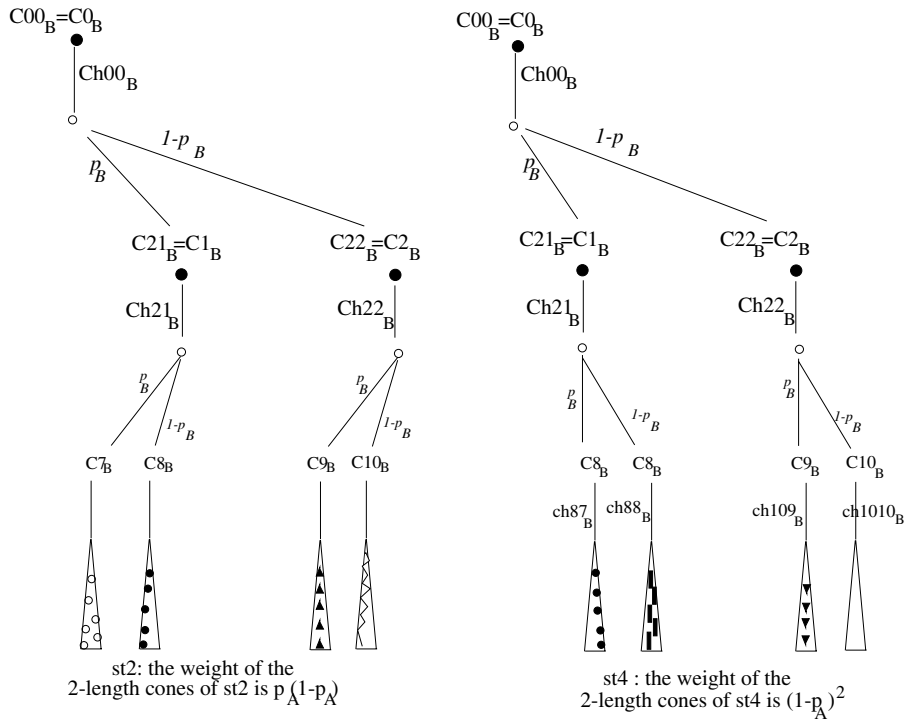


Fig. 5. 2-length beginning of projection strategies on B.

Lemma 7. [**Self-Stabilization Propagation to the Deterministic Algorithm** $A \diamond B$ **from Algorithm** A] *Let $A \diamond B$ be the cross-over composition between the deterministic algorithms A and B (B is a fair algorithm). If A is self-stabilizing for the specification SP under the scheduler D then $A \diamond B$ stabilizes for the specification SP under D.*

Proof. Let e be a maximal computation of $A \diamond B$.

- convergence of Algorithm $A \diamond B$. Let P be the predicate which characterizes the legitimate configurations of A and let e_A be the projection of e on A. According to Lemma 3, e_A is a maximal computation of A and e_A reaches a legitimate configuration (A is self-stabilizing).
- correctness of Algorithm $A \diamond B$. Let e be a computation of $A \diamond B$ which starts in a configuration satisfying P. Let e_A be its projection on A. The computation e_A is maximal and starts in a configuration which satisfies P. A is self-stabilizing then e_A satisfies the specification SP, hence e satisfies also the specification SP.

Lemma 8. [**Self-Stabilization Propagation to the Probabilistic Algorithm** $A \diamond B$ **from Algorithm** B] *Let $A \diamond B$ be the probabilistic cross-over*

composition between the algorithms A and B. If the algorithm B self-stabilizes for the specification SP under the scheduler D then $A \diamond B$ is a probabilistic self-stabilizing algorithm for SP under D.

Proof. Let us study the propagation in the two possible cases: B is a deterministic algorithm or is a probabilistic one. The idea of the proof is to analyze an arbitrary strategy st of $A \diamond B$ under D.

- B is deterministic. The projection on B of every computation of the strategy st is maximal. Every computation of st reaches a legitimate configuration; and then, it satisfies the specification SP. $A \diamond B$ is self-stabilizing for SP under D.
- B is probabilistic. Let st be a strategy of $A \diamond B$. Let L be the legitimate predicate associated with SP. According to Lemma 5 or to Theorem 1. $P_{st}(\mathcal{E}L) = 1$. Moreover, according to Lemma 1, all computations of st that reach L have a suffix that satisfies SP.

The self-stabilization propagation from the weak algorithm is possible only if the strong algorithm is fair.

Lemma 9. [Self-Stabilization Propagation to the Probabilistic Algorithm $A \diamond B$ from Algorithm A] *Let $A \diamond B$ be the probabilistic cross-over composition between the algorithms A and B. If Algorithm A self-stabilizes for the specification SP under the scheduler D and Algorithm B is a fair algorithm under D then $A \diamond B$ is a probabilistic self-stabilizing algorithm for SP under D.*

Theorem 2. [Self-Stabilization Propagation from A and B] *Let $A \diamond B$ be the probabilistic cross-over composition between the algorithms A and B. If Algorithm A self-stabilizes for the specification SP under the scheduler D and Algorithm B is self-stabilizing for the specification SR and is fair under D then $A \diamond B$ is a probabilistic self-stabilizing algorithm for $SP \wedge SR$ under D.*

Note that in both cases — deterministic and probabilistic — the strong algorithm will propagate the self-stabilization property to the result of composition without any restriction, while the propagation initiated by the weaker one can be realized if and only if the strong algorithm is fair.

5 Application: Scheduler Transformation

In this section, we present the main application of the cross-over application, the scheduler transformation. We show how to use the cross-over composition to transform any self-stabilizing algorithm under some specific scheduler into an algorithm that converges under any unfair scheduler.

Definition 7 (Fragment of Owner p). *Let e be a computation of a distributed system and let p be a processor such that p executes its actions more than one time in e. A fragment of e of owner p, f_{pp} is a fragment of e such that:*

- *f_{pp} starts and finishes with a configuration where p executes an action;*
- *along f_{pp}, p executes exactly two actions (during the first and the last step of f_{pp}).*

Lemma 10 (From k-Fairness to the k-Bound Property). *Let us consider the cross-over composition $A \diamond B$. Let e be a computation of $A \diamond B$ under an arbitrary scheduler. If B is k-fair then the projection of e on A is k-bounded.*

Proof. Suppose that e_A is not k-bounded. Hence, there is a fragment f_A of e_A such that a processor q performs $k+1$ actions during f_A and such that another processor p performs no action and it is always enabled along f_A. f_A is the projection of a fragment of e called f. According to the definition of $A \diamond B$, f has the following properties (i) p performs no action in f (ii) q performs at least $k+1$ actions in f. f is part of a fragment owned by p called f_{pp} such that q performs at least $k+1$ actions in f_{pp}. f_{pp} does not exist because $A \diamond B$ is k-fair (Corollary 2).

The following theorem gives a tool for transforming an algorithm A that self-stabilizes under a k-bounded scheduler into an algorithm A' that self-stabilizes under an unfair scheduler. Let B be an algorithm whose computations are k-fair, the transformed is $A' = A \diamond B$.

Theorem 3. *Let $A \diamond B$ be the cross-over composition between A and B. A is a self-stabilizing algorithm for the specification SP under a k-bounded scheduler. B is a k-fair algorithm. The algorithm $A \diamond B$ is self-stabilizing for the specification SP under an unfair scheduler.*

Proof. Let e be a maximal computation of $A \diamond B$ and let e_A be its projection on the weak module. Since B is k-fair, according to Lemma 10, e_A is a k-bounded computation.
Correctness Proof: Let $\mathcal{L}$ be the legitimate predicate associated with SP. Once e_A has reached a legitimate configuration (a configuration that satisfies the predicate $\mathcal{L}$), it satisfies the specification SP.
convergence proof:

- *A is a deterministic algorithm*: e_A reaches a legitimate configuration.
- *A is a probabilistic algorithm.* Let st be a strategy of $A \diamond B$ under a distributed unfair scheduler. Let st_A be a projection strategy of st on A. According to Lemma 10 any execution of st_A is k-bounded. According to the hypothesis, $P_{st_A}(\mathcal{EL}) = 1$. According to Lemma 5 or to Theorem 1, $P_{st}(\mathcal{EL}) = 1$.

Note that the transformation depends directly on the properties of the strong algorithm of a cross-over composition. The main question is "are there algorithms able to satisfy the k-bound property under any unfair scheduler ?" The answer is positive and in the following we show some examples:

Protocol	Topology	Network type	Scheduler transf.
[7]	general networks, bidir.	with id	central to unfair
[1]	general networks, bidir.	with id	central to unfair
[1]	general networks, bidir.	with id	X_1-bounded to unfair
[3]	rings, unidir.	anonymous	X_1-bounded to unfair
[6]	rings, unidir.	anonymous	alternating to central
[2]	general networks, unidir.	anonymous	X_2-bounded to unfair

$X_1 = n - 1$; and $X_2 = n.MaxOut^{Diam}$ where $MaxOut$ is the maximal network out-degree and $Diam$ is the network diameter.

The protocols [1] and [7] are working in the id-based networks. In the case of anonymous networks an algorithm which ensures the transformation of a central scheduler to a distributed scheduler could be the algorithm of [1] executed on top of an algorithm which ensures an unique local naming (neighbor processors do not have the same id; but distant processors may have the same id).

6 Conclusion

We have presented a transformation technique to transform self-stabilizing algorithms under weak scheduler (k-bounded, central, *alternating*, ...) into algorithms which maintain the self-stabilizing property under unfair and distributed schedulers.

The key of this transformation is the cross-over composition $A \diamond B$: roughly speaking, the obtained computations are the computations of A under a scheduler that provides the B's computations. The cross-over composition is a powerful tool to obtain only specific computations under any unfair scheduler. Indeed, if all B's computations have "the D properties" then A only needs to be a self stabilizing algorithm for the specification SP under the weak scheduler D to ensure that $A \diamond B$ is a self stabilizing algorithm, for the specification SP under any unfair scheduler.

References

1. Beauquier J., Datta A., Gradinariu M., and Magniette F.: Self-stabilizing Local Mutual Exclusion and Daemon Refinement. *DISC 2000, 14th International Symposium on Distributed Computing*, LNCS:1914 (2000) 223-237
2. Beauquier J., Durand-Lose J., Gradinariu M., Johnen C.: Token based self-stabilizing uniform algorithms. *tech. Rep. no. 1250, LRI, Université Paris-Sud; to appear in The Chicago Journal of Theoretical Computer Science* (2000)
3. Beauquier J., Gradinariu M., Johnen C.: Memory Space Requirements for Self-stabilizing Leader Election Protocols. *PODC'99, 18th Annual ACM Symposium on Principles of Distributed Computing* (1999) 199-208
4. Beauquier J., Gradinariu M., Johnen C.: Randomized self-stabilizing optimal leader election under arbitrary scheduler on rings. Tech. Rep. no. 1225, LRI, Université Paris-Sud (1999)

5. Dolev S., Herman T.: Parallel composition of stabilizing algorithms. *WSS'99, fourth Workshop on Self-Stabilizing Systems* (1999) 25-32
6. Fich F., Johnen C., A space optimal deterministic, self-stabilizing, leader election algorithm for unidirectional Rings. *DISC 2001, 15th International Symposium on Distributed Computing* (2001)
7. Gouda M., Haddix F.: The alternator. *WSS'99, Fourth Workshop on Self-Stabilizing Systems* (1999) 48-53
8. Gouda M., Herman T.: Adaptive programming. *IEEE Transactions on Software Engineering* **17** (1991) 911-921
9. Kakugawa, H., Yamashita, M.: Uniform and Self-stabilizing Token Rings Allowing Unfair Daemon. *IEEE Transactions on Parallel and Distributed Systems* **8** (1997) 154-162
10. Segala R.: *Modeling and Verification of Randomized Distributed Real-time Systems.* PhD thesis, MIT, Department of Electrical Engineering and Computer Science (1995)
11. Segala R., Lynch N.: Probabilistic simulations for probabilistic processes. *CONCUR'94, 5th International Conference on Concurrency Theory*, LNCS:836, (1994)
12. Varghese G.: Compositional proofs of self-stabilizing protocols. *WSS'97, Third Workshop on Self-stabilizing Systems* (1997) 80-94
13. Wu S. H., Smolka S. A., Stark E. W.: Composition and behaviors of probabilistic I/O automata. *CONCUR'94, 5th International Conference on Concurrency Theory*, LNCS:836, (1994) 513-528

Easy Stabilization with an Agent

Joffroy Beauquier[1], Thomas Hérault[1], and Elad Schiller[2]

[1] Laboratoire de Recherche en Informatique
Bâtiment 490, Université Paris-Sud, 91 405 Orsay, France
{jb,herault}@lri.fr
[2] Department of Computer Science, Ben-Gurion University of the Negev
Beer-Sheva 84105, Israel
schiller@cs.bgu.ac.il

Abstract. The paper presents a technique for achieving stabilization in distributed systems. This technique, called agent-stabilization, uses an external tool, the agent, that can be considered as a special message created by a lower layer. Basically, an agent performs a traversal of the network and if necessary, modifies the local states of the nodes, yielding stabilization.

1 Introduction

Fault tolerance and robustness are important properties of distributed systems. Frequently, the correctness of a distributed system is demonstrated by limiting the set of possible failures. The correctness is established by assuming a predefined initial state and considering every possible execution that involves the assumed set of failures.

The requirements of some distributed systems may include complex failures, such as a corruption of memory and communication channels. The self-stabilization methodology, introduced by Dijkstra [3], deals with systems that can be corrupted by an unknown set of failures. The system self-stabilizing property is established by assuming that any initial state is possible, and considering only executions without transient failures. Despite a transient failure, in a finite time, a self-stabilizing system will recover a correct behavior.

Self-stabilizing systems are generally hard to design, and still harder to prove [4]. In this paper, we are proposing to investigate a specific technique for dealing with the corruption problem. The basic idea appears in a paper of Ghosh [5] and consists of the use of mobile agents for failure tolerance. A mobile agent can be considered as an external tool used for making an application self-stabilizing. The notion that we propose here is different from Ghosh's notion. A mobile agent can be considered (in an abstract way, independently of its implementation) as a message of a special type, that is created by a lower layer (the outside). We will assume that some code corruption is possible and a mobile agent (or more simply an agent) can carry two types of data: a code, that can be installed at some nodes, and some information (referred to as contained in the

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 35–50, 2001.

briefcase), that can be read or modified by the processors. Code and briefcase are specific to the application controlled by the agent.

The outside can create a single agent and install it on an arbitrary node (the initiator). As long as the agent has not been destroyed, no other agent can be created. An agent has a finite lifetime. It can be destroyed either by its initiator, or by any site if it is supposed to have traveled too long. After an agent has been destroyed, the outside will eventually create a new one (with possibly another initiator). Once created, an agent moves from a processor to a neighbor of this processor, following a non-corruptible rule. More precisely, each node has permanently in its ROM some code, which is the only non-corruptible information and does not depend on any particular application. Moreover, it is simple and compact.

When an agent reaches a node (by creation or by transmission), the code carried by the agent is installed. The code is in two parts, agent rules and application (or algorithm) rules. Agent rules are the only applicable rules when an agent is present, and, on the contrary, application rules are the only applicable rule when it is not. Agent code execution is always finite and ends by a call to the agent circulation code, responsible for transmitting the agent to another site. The ability of the agent to overwrite a corrupted application code (or simply to patch the code in a less severe fault model), motivates the inclusion of code in the agent. Because it is supposed to travel, an agent (code and briefcase) can be corrupted.

In our agent model, there are two crucial properties, non-interference and stabilization. The non-interference property expresses that if no failure (corruption) occurred after the creation of the previous agent, after the destruction of this agent, the controlled application must not behave differently whether or not a new agent is present in the network. The property involves that the application cannot use the agents for solving its problem and that the agents cannot perform resets in a predetermined global configuration. The stabilization property states that if a failure occurred, after a finite number of agent creations, the application behaves accordingly to its specification.

Agent-stabilization has some advantages over classic self-stabilization. With self-stabilization, even if the stabilization time can be computed, it says nothing about the effective time needed for stabilization, which strongly depends on the communication delays and, consequently on the global traffic in the network. At the contrary, if agents are implemented in a way that guarantees small communication delays (as high priority messages for instance), an efficient bound on agent-stabilization time can be given, because such a bound depends only on the time of the network traversal by the fast agent, plus the time between two successive agent creations, that can be tailored to the actual need. A second advantage is that agent-stabilization is more powerful than self-stabilization. Any self-stabilizing solution is agent-stabilizing, with agents doing nothing, but some impossibility results from self-stabilization theory can be bypassed with agents, as we will show in the sequel.

2 The Specification

The system is a set P of n communicating entities that we call *processors*, and a relation $neighbors \subset P \times P$. To each processor p in P is associated some internal variables, an algorithm and a communication register r_p (r_p can have a composite type). Each processor $p \in P$ has a read and write access to its register r_p and a read access to its neighbor's communication registers. The local state of a processor is the set of the values of the internal variables and of its register. A vector whose components are the local states of every processor of the system is the configuration of the system.

The algorithm is defined as a set of guarded rules, of the form $Guard_i \longrightarrow Action_i$ where $Guard_i$ is a predicate on the local variables, the communication register and the communication registers of the processor's neighbors, and $Action_i$ is a list of instructions which could modify the communication register and/or the local variables of the processor. A rule is said to be enabled if its guard is true and disabled if its guard is false.

The system can evolve from a configuration to another by applying at some processor p an atomic step. For defining the computation rules of a processor, we need now to look more precisely at what is held in the communication register. It holds two parts : an algorithm-specific part including all information that the processor has to communicate to its neighbors and an agent-specific part. The agent-specific part of the communication register is composed of 6 fields : Agent Code, Algorithm Code, Briefcase, Next, Prev and Present.

The fields Agent Code and Algorithm Code hold two finite sets of guarded rules, depending on the algorithm ; the Briefcase is a field of constant size depending on the algorithm ; the fields Next and Prev are either $\perp$ or pointers to the neighbors of the processor (they point respectively to the previous processor which had the agent and to the next which will have it), the field Present is a boolean and indicates whether or not the agent is present at the processor.

An atomic step for a processor p is the execution by p of one of the following rules :

Agent step: if $Present_p$ is true, select and compute one of the enabled rules held by the Agent Code part of the register, if $Next_p$ is not $\perp$, install the agent in $Next_p$ from p ; set $Present_p$ to false.

Algorithm step: if $Present_p$ is false, select and compute one of the enabled rules held by the Algorithm Code part of the register ;

Agent installation: spontaneously install the agent in p from $\perp$;

Installing an agent in p from q is performing the following algorithm :

1. Set $Present_p$ to true;
2. Set $Prev_p$ to q;
3. If q is $\perp$, instantiate Algorithm Code with a fresh copy in Agent Code;
4. If q is not $\perp$, copy the values of $Briefcase_q$, $AlgorithmCode_q$ and $AgentCode_q$ from q to p.

An execution with initial configuration C is an alternate sequence of configurations and atomic steps, each configuration following an atomic step being obtained by the computation of the atomic step from the previous configuration, C being the first configuration of the sequence. We will consider here only fair executions. A fair execution is an execution such that if a guard of a processor P is continuously enabled, this processor will eventually perform an atomic step.

System executions satisfy two axioms that we describe here :

Uniqueness of creation. The atomic step *Agent Installation* can be performed only on a configuration where for all processor p, $Present_p$ is false ;

Liveness of creation. In each execution there is infinitely many configurations in which an agent is present;

For the sake of clarity, we define a special alternate sequence of configurations and atomic steps, which is not an execution with respect to the two axioms, but that we call an agent-free execution and which is used to define the agent-stabilizing notion below. An agent-free execution is an alternated sequence of configurations and atomic steps, such that the initial configuration of the sequence satisfies $\neg Present_p$ for all p and only algorithm steps are computed in this sequence. The set of agent-free executions starting at L (where L is a set of configurations) is the collection of agent-free executions such that the initial configuration is in L. We define also the agent-free projection of an execution : it is the alternate sequence of configurations and atomic steps, where the agent-part of the registers is masked and agent-steps are ignored.

Finally, we define the type of failures that we will consider by defining initial configurations. There is no constraint on an initial configuration, except the fact that agent circulation rules must be correct (non corruptible).

Let $\mathcal{S}$ be a specification (that is a predicate on executions). A system is agent-stabilizing for $\mathcal{S}$ iff there exists a subset $\mathcal{L}$ of the set of configurations satisfying :

Convergence. For every configuration C, for every execution starting at C, the system reaches a configuration L in $\mathcal{L}$;

Correctness. For very configuration L in $\mathcal{L}$, every execution with L as initial configuration satisfies the specification $\mathcal{S}$;

Independence. For very configuration L in $\mathcal{L}$, every agent-free execution with L as initial configuration satisfies the specification $\mathcal{S}$;

Non-interference. For every configuration L in $\mathcal{L}$, the agent-free projection of every execution with L as initial configuration satisfies the specification $\mathcal{S}$;

Finite time to live. Every execution has the property that, after a bounded number of agent-steps, an agent is always removed from the system.

Convergence and correctness parts of this definition meet the traditional definition of a self-stabilizing algorithm. The independence part ensures that the algorithm will not rely on the agent to give a service : e.g. an agent-stabilizing system yielding a token circulation cannot define the token as the agent. Thus, an agent is indeed a tool to gain convergence, not to ensure correctness. Moreover,

the non-interference property enforces the independence property by ensuring that an agent cannot perturb the behavior of a fault-free system. Finally, the finite time to live property ensures that an agent will not stay in the system forever and that an agent is a sporadic tool, used only from time to time.

3 General Properties of Agent-Stabilizing Systems

The study of the convergence property of an agent-stabilizing system is as difficult as the convergence property of a self-stabilizing system, because executions can start from any initial configuration. Moreover, the traditional fault model of self-stabilization does not include the corruption of the code. Here, we accept some code corruption (algorithm code corruption), making the study still harder.

We will prove in this part that without loss of generality, for proving the convergence property of an agent-stabilizing system, the set of initial configurations can be restricted to those in which code is correct and there is exactly one agent, just "correctly" installed at some processor p.

Theorem 1. *Assume the finite time to live property. Then, for proving convergence of an agent-stabilizing system, it is sufficient to consider as initial configurations, those following the installation of a unique agent.*

Sketch of Proof. Let C be an arbitrary configuration with n processors and p agents at processors $P_1, \ldots, P_p$ (thus, $Present_i$ is true for $i \in \{P_j, 1 \leq j \leq p)$. Let $\mathcal{E} = C\, s_1\, C_1\, \ldots$ be an arbitrary execution with initial configuration C. $\mathcal{E}$ is fair, thus there are infinitely many agent steps in the execution $\mathcal{E}$. The property "finite time to live" is assumed, thus the execution reaches necessarily a configuration in which an agent is removed from the system, leading to a configuration C', with one agent less than in C.

The axiom "uniqueness of creation" involve that while there is at least one agent in the system (thus one processor verifying "$Present_p$ is true"), the agent installation step cannot be performed on any other processor in the system. By recursion on the number of agents in the system, the system reaches a configuration C^p where there is no agents in the system (potentially, $C^p = C$ if there was no agent in the initial configuration).

The execution $\mathcal{E}$ also satisfies the axiom "liveness of creation" ; there is a finite number of configurations between C and C^p, thus at least one agent will be present in a configuration A after C^p. The only atomic step which could install an agent is the "Agent installation" step, thus there is always a configuration A reachable from C where a single agent is just installed in the system. □

4 Agent Circulation Rules

One of the main goals of the agent is to traverse the network. We will present here three algorithms for achieving a complete graph traversal. Two of them (depth first circulation and switch algorithm) hold for arbitrary graphs, whereas

the third is an anonymous ring traversal. We will also show that, in any agent-system using one of these algorithms as agent circulation rules, it is easy to implement the finite time to live property.

Depth First Circulation.
The algorithm assumes that every processor of the system is tagged with a color black or white. Assume that agent has just been created. The briefcase carries the direction of the traversal (Up or Down). The processor visited by the agent for the first time (with a Down Briefcase) flips its color, chooses as a parent the previous processor, chooses a neighbor tagged with a color different from its new color and sends the agent to this neighbor, with the briefcase Down. When the agent reaches a dead end (i.e. is sent down to a processor with no neighbors with the right color), it is sent back by this processor with Up in the Briefcase. When a processor receives an agent with Up in the Briefcase, if it is a dead end, it sends the agent Up, if not, it selects a new neighbor to color and sends the agent down.

Moreover, the briefcase holds a *hop* counter, namely the number of steps since the installation. This counter is incremented each time the agent moves. If the agent makes more than $4|E|$ steps (a bound greater than 2 times the number of edges of the depth first spanning tree), the agent is destroyed (next processor is $\perp$). More formally, in figure 1 the description of the traversal algorithm is given. For this, we define :

$$\theta(\Gamma_i, c) = \begin{cases} \perp \text{ if } \forall p \in \Gamma_i, color_p \not= c \\ p \text{ s.t. } color_p = c \text{ and } p \in \Gamma_i \text{ else} \end{cases} \text{ and also } \begin{array}{l} \overline{black} = white \\ \overline{white} = black \end{array}$$

We claim that this algorithm produces a complete traversal of the system even in the case of transient failures because, as we will show in theorem 3, at each execution of rule 1, the number of processors of the same color strictly increases, until reaching the total number of processors in the system.

Theorem 2. *Every fair agent-system using the depth first circulation algorithm of figure 1 as agent circulation algorithm satisfies the finite time to live property.*

The proof is straightforward and uses the fairness of the system for showing that every agent increments its *hop* counter until the bound is reached.

Remark that the bound is large enough to allow a complete depth first traversal of the system : it is 2 times the number of edges, and in a depth first traversal, each edge of the spanning tree is used 2 times (one time for going down to the leaves, one time for going up to the root). We will now prove that this algorithm produces a complete graph traversal when there is a single agent in the system.

Let $p, q \in P$. We say that $p \sim q$ iff $color_p = color_q$ and $(p, q) \in neighbours$. In the sequel, the term "connected components" refers to the connected components of the graph of the relation $\sim$.

Lemma 1. *Let C be a configuration with m connected components $(M_1, \ldots M_m)$ and let p be a processor with an agent just correctly installed. Let M_j be the connected component of p.*

Local: $\langle parent_i \in \Gamma_i \bigcup \{\bot\} \quad — \quad color_i \in \{black, white\}\rangle$
Briefcase: $\langle Briefcase_i \in (\{down, up\}, hop), 1 \leq hop \leq 2.|E|\rangle$
Constant rule: At each rule, we add the following statement:
if $hop > 2|E|$ then $Next_i \leftarrow \bot$.

– Agent rules :

1. $Prev_i = \bot \vee \big((Briefcase_i = (down, hop)) \wedge (\theta(\Gamma_i, color_{Prev_i}) \not=)\big) \longrightarrow$
$\begin{cases} parent_i \leftarrow Prev_i \\ Next_i \leftarrow \theta(\Gamma_i, color_i) \\ color_i \leftarrow \overline{color_i} \\ \text{if } Prev_i = \bot, hop = 0 \\ Briefcase_i \leftarrow (down, hop + 1) \end{cases}$
(* going down *)
2. $(Prev_i \not=) \wedge (Briefcase_i = (down, hop)) \wedge (\theta(\Gamma_i, color_{Prev_i}) = \bot) \longrightarrow$
$\begin{cases} parent_i \leftarrow Prev_i \\ Next_i \leftarrow Prev_i \\ Briefcase_i \leftarrow (up, hop + 1) \\ color_i \leftarrow \overline{color_i} \end{cases}$
(* reaching a dead-end *)
3. $(Prev_i \not=) \wedge (Briefcase_i = (up, hop)) \wedge (\theta(\Gamma_i, \overline{color_i}) \not=) \longrightarrow$
$\begin{cases} Next_i \leftarrow \theta(\Gamma_i, \overline{color_i}) \\ Briefcase_i \leftarrow (down, hop + 1) \end{cases}$
(* done one branch, go down for others *)
4. $(Prev_i \not=) \wedge (Briefcase_i = (up, hop)) \wedge (\theta(\Gamma_i, \overline{color_i}) = \bot) \longrightarrow$
$\begin{cases} Next_i \leftarrow parent_i \\ Briefcase_i \leftarrow (up, hop + 1) \end{cases}$
(* completely done one node, going up*)

Fig. 1. Depth first agent traversal algorithm for anonymous networks.

In every execution $\mathcal{E} = C\, s_1 \ldots C' \ldots$ there is a configuration such that the agent is in p, has visited all processors in M_j and every processor in M_j has flipped its color.

Theorem 3. *Consider the same configuration C, as in lemma 1.*

In every execution $\mathcal{E} = C\, s_1 \ldots C' \ldots$ there is a configuration such that there is no agent, one agent has visited all processors of the system and all processors in C' have the same color.

Sketch of Proof. Lemma 1 states that from an initial configuration C, the system reaches a configuration C_1, with all processors of M_j (the connected component of p) visited and having flipped their color. If there is only one connected component in C, then the theorem is true (rule 4 set $Next$ to $\bot$ in p and when p makes an atomic step, it removes the agent). If there are $m > 1$ connected components in C, then there are $1 \leq n < m$ connected components in C_1 (because all processors of M_j flipped their color, thus merged with the neighbor connected

component). Thus, lemma 1 applies again in a configuration C'_1 with the agent removed from p (rule 4) and installed in a processor p' (liveness of the agent).

Recursion on the number of connected components in C, proves the theorem. □

The Switch Algorithm. The switch algorithm is another anonymous graph traversal algorithm. The algorithm presented in [12] is defined for semi-uniform Eulerian networks, but as we will show now, it works for uniform Eulerian networks with the agent-stabilizing system assumptions. This algorithm is an improvement of the depth first circulation because here circulation is decided purely locally, without communicating with the neighbors of the processor which holds the agent. Moreover, in the depth first token circulation, the agent performs a complete graph traversal only after all connected components merged into one. Here, the algorithm converges with only one agent installation. This has a price : an agent circulation round takes $2 \times n \cdot |E|$ agent steps, even after stabilization, while an agent circulation round with the depth first circulation takes only $2|E'|$ agent steps (E' being the set of covering edges of the depth first tree).

The idea of the algorithm is the following. Suppose that each node has ordered its (outgoing) edges and let $PointsTo$ be a pointer on some of these edges. When the token is at some node, it is passed to the neighbor pointed by $PointsTo$, then $PointsTo$ is set to the next edge in the list (which is managed circularly).

In [12], it is proved that, starting with a unique token, this algorithm eventually performs an Eulerian traversal of the network.

We propose here a slight modification of this algorithm so that it performs a complete traversal of an anonymous Eulerian agent-system. The initiator (with $Prev = \perp$) will set the Briefcase to 0 and every time the agent performs an agent step, Briefcase will be incremented by 1. If a processor holds the agent and if Briefcase is greater than $2 \times n \cdot |E|$ (where E is the set of edges of the system and n is the number of processors in the system), then this processor sets $Next$ to $\perp$, destroying the agent. The agent rules are presented more formally in figure 2. We will now prove that this circulation algorithm satisfies the finite time to live property.

Theorem 4. *Every fair agent-system using the switch circulation algorithm of figure 2 as an agent circulation algorithm satisfies the Finite Time to Live property.*

The proof is straightforward and uses the fairness of the system for showing that the Briefcase counter is incremented until reaching the bound.

Theorem 5. *The switch circulation algorithm is a circulation algorithm, which performs a complete graph traversal from a configuration in which an agent installation follows a configuration without agents.*

We use the convergence property of the switch, proven in [12], and the convergence time, which is $n \cdot |E|$ to state that after $n \cdot |E|$ agent steps, every edge, thus every node of the system has been visited by the agent.

Local: $\langle PointsTo_i \in \Gamma_i$
Briefcase: $\langle 0 \leq Briefcase \leq 2 \times n \cdot |E| + 1 \rangle$
Operator: $Inc(PointsTo_i)$ increments $PointsTo_i$ by 1 modulo $Degree_i$.

– Agent rules :

1. $Prev = \perp \longrightarrow \begin{cases} Inc(PointsTo_i) \\ Next_i = PointsTo_i \\ Briefcase_i = 0 \end{cases}$
2. $(Prev \neq \perp) \wedge (Briefcase < 2 \times n \cdot |E|) \longrightarrow \begin{cases} Inc(PointsTo_i) \\ Next_i = PointsTo_i \\ Briefcase_i = Briefcase_i + 1 \end{cases}$
3. $(Prev \neq \perp) \wedge (Briefcase \geq 2 \times n \cdot |E|) \longrightarrow \{ Next_i = \perp$

Fig. 2. Switch circulation algorithm

Anonymous Oriented Ring Circulation.
In such a topology circulation is trivial : the agent rule consists of sending the agent to the successor and counting in the Briefcase the number of processors already seen. If this number is greater than or equal to the number of processors in the system, the agent is destroyed. With this algorithm, the finite time to live property is obvious, as is the correctness property of the agent circulation : starting from a configuration with only one agent, just installed at some processor p, this agent visits all the processors in the system. Furthermore, every new agent is eventually destroyed by its initiator.

5 Examples Illustrating the Power of Agents

We will first present general transformations which, given a self-stabilizing solution for semi-uniform networks or for networks with distinct identifiers, automatically transform them into agent-stabilizing solutions for the same problem, but for completely anonymous networks (uniform networks). The basic idea is that a generic agent algorithm, with an empty set of algorithm rules, either distinguishes a particular processor or gives distinct identifiers in an anonymous networks. By adding the rules of the self-stabilizing solution as algorithm rules one gets an agent stabilizing solution for anonymous network.

Let us briefly describe the ideas of the transformation from uniform to semi-uniform. Note that an algorithm that always declares the processor in which the agent is created as the leader, does not satisfies the non-interference property. Thus, the agent must perform a complete traversal (using the circulation rules) for checking whether or not there is only one distinguished processor. If several distinguished processors (resulting from a corruption) are found, all but the first are canceled and if no one is found, the initiator becomes distinguished.

Note that there is no algorithm rule in this solution, then it can be combined with any self-stabilizing solution for semi-uniform networks. The main advantage of automatic transformations is that there are automatic. But they very seldom yield the most efficient solution. In the second part, we will give another illustration of the power of agents, by giving efficient specific solutions to some classical problems.

5.1 General Transformers

Leader Election within a Uniform Network. This transformer allows a self-stabilizing system designed for semi-uniform networks to work for uniform networks. This is achieved by solving the Leader election problem.

Starting at the initiator, the agent performs a complete graph traversal using the algorithm in figure 1. We proved that this algorithm eventually performs a complete graph traversal, and it is obvious that after the first traversal, no processor has $prev|_p = \perp$ Then we can ensure that there is no processor with $parent_p = \perp$ after the first agent traversal, by adding $parent_p = Prev$ to the action part of rule 4 (the rule which is used when every neighbor of this node has been visited).

The algorithm is simple : the agent carries in the briefcase the information "did I meet a leader yet" (boolean value). If this information is true when the agent meets a leader, the leader is destroyed, if the agent meets the initiator (which is the only processor which has $parent_i = \perp$ for every agent installation after the first complete traversal) and this information is false, this processor becomes the leader.

Lemma 2. *Consider a configuration C of Depth first agent traversal, in which all processors have the same $color_i$ and an agent has just been installed at processor p. The agent performs a complete graph traversal and rule 4 is executed exactly once by each processor before the agent is removed from the system.*

Theorem 6. *The leader election algorithm is agent-stabilizing.*

Sketch of Proof. The agent has a finite time to live, as it was proven in theorem 2. It is then legitimate to use the result of theorem 1 and to consider for the convergence property a configuration with one agent in the system. This agent will eventually traverse every node and come back to the initiator (theorem 3). According to algorithm rules, there will remain one and only one leader in the system, thus convergence is verified. Consider a legitimate configuration. The algorithm rules are empty (thus the agent-system satisfies independence), and when an agent is installed, it will perform a complete graph traversal, visiting every node and executing rule 4 at each node exactly once (lemma 2 applies, the configuration is legitimate, thus every node is of the same color). Thus, it will visit the leader and note it in the briefcase. Then, the initiator will not be designated as the leader and the correctness property is satisfied. Moreover, the leader stays the leader, thus non-interference property is also verified. □

Network Naming. The naming problem is to assign to each processor of the network a unique identity. We consider that there is a set of identifiers (at least as large as the network size), and that identities should be chosen within this set. The agent carries in the briefcase a boolean tag for each element of this set. When the agent reaches a processor p, if p uses an identifier already used, its identifier is set to the first available identifier, and this identifier is tagged in the briefcase ; if p uses an identifier which is not yet used, this identifier is tagged in the briefcase. The agent performs a complete graph traversal according to figure 1 algorithm and actions are executed when the guard of rule 4 is true.

Theorem 7. *The network naming algorithm is agent-stabilizing.*

Sketch of Proof. As it was said before, the Finite time to live property is inherited from the agent circulation algorithm. Like for leader election, independence is obvious since the algorithm has no rule. For proving the correctness property, we have to show that the set of legitimate configurations is closed. Let us define legitimate configurations as configurations such that $\forall p, q \in P, tag_p \ / = tag_q$. In a token circulation, rule 4 of depth first traversal applies at most once (theorem 2). Thus, the agent will never meet a processor with a tag equal to an already used tag. Thus, the rule will never be applied and the configuration will not change with respect to the tags. Let us consider an arbitrary configuration C' with an agent just installed at processor p. If this configuration is not legitimate, then there exists a processor p and a processor q tagged by the same name. Theorem 3 states that in every execution with initial configuration C', there is a configuration C'' reached after the agent visited every node. When this agent has been installed, $Briefcase$ was set to nil ; it first reached either p or q. If it reached p, tag_p is an element of $Briefcase$, and when it reaches q, tag_q is set to a no yet used tag. □

5.2 Local Mutual Exclusion within a Uniform Network

With the previous algorithm, we can easily transform every self-stabilizing algorithm designed for networks with identities to an agent-stabilizing solution for anonymous networks. Note that we assume that the agent can know the different Ids already seen in the network (briefcase must have $n \log(n)$ bits). Obviously, in specific cases, better solutions can be found. We propose now a solution for solving Local Mutual Exclusion within a uniform anonymous network. Local Mutual Exclusion is a generalization of the dining philosophers problem [7], and can be defined as "having a notion of privilege on processors, two neighbors cannot be privileged at the same time".

The solution is based on the self-stabilizing local mutual exclusion algorithm proposed in [1]. Every processor has a variable b in range $[0; n^2 - 1]$. Assuming that neighbors have different b values, then a cyclic comparison modulo n^2 of the b values yields an acyclic orientation of edges (from greater to lesser). If we consider that every sink (i.e. a node with only ingoing edges) is privileged, then two neighbors never can be privileged simultaneously. To pass the privilege, b is

set to the maximum of the neighbors value plus one (modulo n^2), and in [1] it is shown that this rule provides a cyclic selection of every processor in the system.

It is also shown that this solution is based upon the fact that a processor has a value different from the values of its neighbors, and identifiers are used to implement that. An agent can be easily used in an anonymous network, for creating asymmetry between a processor and its neighbors.

- ◇ Algorithm rules
  1. $b < min^{[n^2]}_{q \in \Gamma_p}(b|_q) \longrightarrow b \leftarrow max^{[n^2]}_{q \in \Gamma_p}(b|_q + 1)$
- ◇ Agent rules
  1. Every rules of figure 1, composed with
  2. $\exists q \in \Gamma_p$ s.t. $b|_q = b \longrightarrow b \leftarrow max^{[n^2]}_{q \in \Gamma_p}(b|_q + 1)$

Fig. 3. Anonymous uniform Local mutual exclusion agent-stabilizing algorithm.

Theorem 8. *The local mutual exclusion algorithm of figure 3 is agent-stabilizing.*

Sketch of Proof. This algorithm uses the depth first circulation algorithm, thus the Finite Time to live property is satisfied. Then, according to theorem 1, we can consider for proving convergence a configuration with a single agent just installed at some node. Let C be such a configuration. Either, $\forall p \in P, \forall q \in \Gamma_p, b|_p \not= b|_q$ and it was proven in [1] that the algorithm converges or $\exists p \in P, q \in \Gamma_p, b|_p = b|_q$. In this case, the agent will eventually visit p and q (theorem 3), either p first or q first. If the first visited processor is p (resp. q), agent rule 2 will be enabled, thus the system will reach a configuration in which $\exists p \in P, q \in \Gamma_p, b|_p = b|_q$ is false. That will remain false in the sequel of the execution. Then convergence is proven.

Starting from a legitimate configuration, the agent does not modify the b bit of any processor (a legitimate configuration satisfies $\forall p \in P, \forall q \in \Gamma_p, b|_p \not= b|_q$). Correctness of the algorithm has been proven in [1]. Thus every execution from a legitimate initial configuration is correct. In such an execution, the agent does not modify the behavior of the algorithm, then non interference is satisfied. Finally, the independence property is also satisfied since every agent-free execution with a legitimate initial configuration is correct. □

5.3 Token Circulation on a Bidirectional Uniform Oriented Ring

As it was shown in [5] a simple non self-stabilizing algorithm for achieving token circulation on a ring with a leader can be made easily agent-stabilizing, assuming that the agent is always installed at the leader. We will prove that this assumption can be removed. Furthermore, we propose a token circulation algorithm on a bidirectional uniform oriented ring which is agent-stabilizing.

Predicates:
$sensitive(p) \equiv b|_{p+1} = b|_p \ / \ \not{b}|_{p-1}$
$HasToken(p) \equiv b|_{p-1} = 1 \wedge b|_p = 0$,
$Context(p) = Tokens$ if $HasToken(p)$, $NoTokens$ else.

◇ **Algorithm rules**
(a) $sensitive(p) \wedge b|_p = 1 \longrightarrow b|_p = 0$
(b) $sensitive(p) \wedge b|_p = 0 \longrightarrow b|_p = 1$

◇ **Patch 1 Rules**
$color|_p \ / \ \not{S}een \wedge sensitive(p) \wedge b|_p = 1 \longrightarrow b|_p = 0$
$color|_p \ / \ \not{S}een \wedge sensitive(p) \wedge b|_p = 0 \longrightarrow b|_p = 1, color|_p = Seen$

◇ **Patch 2 Rules** $\emptyset$

◇ **Agent rules** (Rules of ring circulation, in conjunction with) :
1. $Prev = \perp \longrightarrow$
   $color|_p = NotSeen, Briefcase.context = Context(p)$, install Patch 1.
2. $Prev \ / \not\perp \wedge Briefcase.hop = n \longrightarrow$
   if $Briefcase.context = NoTokens$ and $color|_p = NotSeen$,
   create a new token
   elsif $Briefcase.context = Tokens$ and $color|_p = Seen$,
   install Patch 2, $Briefcase.hop = 0$
   else install Algorithm.
3. $Prev \ / \not\perp \wedge Briefcase.context = Tokens \longrightarrow$
   if $HasToken(p)$ destroy the token
4. $Prev \ / \not\perp \wedge Briefcase.context = NoTokens \longrightarrow$
   $Briefcase.context = Context(p)$

Fig. 4. Anonymous uniform token circulation agent-stabilizing algorithm

The idea is the following : Let b be a boolean variable stored at each node. Let us say that processor p has the token in configuration C if in C, $b|_p = 0$ and $b|_{p-1} = 1$ (the ring is oriented). Let us say that a processor is sensitive if it has the same b value as its successor and a b value different from its predecessor (then we allow the token to be or not to be sensitive). The algorithm performs the following rules : If a processor p is sensitive, it is allowed to flip its bit. Let us denote a configuration by a word, like 01...10, where a digit represents the value of the b bit of a processor. A legitimate configuration is of the form 1^p0^{n-p}.

This algorithm is not self-stabilizing. The solution that we propose here to stabilize uses agents. The first idea is simple : an agent traverses the ring in the same direction as the tokens, carrying in the briefcase the information "did I see a token until now". If so, every other token can be destroyed by setting the b bit to 1. When the agent comes back to the Initiator, if there is at least one token in the system (Briefcase = Tokens), then the agent simply disappears, if not a token is created at the Initiator (flipping the bit) and then the agent disappears.

The circulation algorithm used here is the oriented anonymous ring circulation. It is detected that the agent has performed a complete traversal by checking if $Briefcase$ has the value n when the agent is back to the Initiator. This so-

lution works only if tokens do not move. Then the next problem to solve is to ensure that the agent will meet the supplementary tokens, if any. For that, the initiator receives a patch from the agent : instead of letting every token pass through him while the agent is inside the ring, it allows only one token to pass and freezes the others.

The last problem is: if the agent saw one or more tokens while another token passed through the initiator, when the agent will come back to the initiator there will be two tokens in the system. But this situation can be detected by the initiator. One solution could be not to let the agent overtake a token, another not to let a token pass through the initiator. But both would fail with the non-interference requirement. A solution matching the requirements is to give to the initiator the responsibility to destroy the supplementary token when it reaches it. Note that, if a bounded convergence time is mandatory (related to the speed of the agent), it could be better to let the initiator freeze one token while the agent performs a second round for deleting the other. The solution is presented more formally in figure 4.

Theorem 9. *The agent-algorithm of figure 4 is agent-stabilizing*

Sketch of Proof. Let us first prove the independence property : a legitimate configuration is defined by 1^p0^{n-p}. In such a configuration, if p is greater than 1, the system can choose between algorithm rules a and b and reaches either the configuration $1^{p+1}0^{n-p-1}$ (the token moves one step backward) or the configuration $1^{p-1}0^{n-p+1}$ (the token does not move). If p is 1, then the system is only allowed to reach the configuration 110^{n-2} (the token moves backward), and if p is $n-1$ the system is only allowed to reach the configuration $1^{n-2}00$. In every reachable configuration the number of tokens is one and every processor of the system eventually has the token. So, starting from a legitimate configuration if there are no agent steps, the behavior is correct.

We prove then the non-interference property : if there is exactly one token in the system, either it will go through p and the agent will never meet this token, $Briefcase.context$ is $NoTokens$, but the agent is silently destroyed, or the agent meets the token, $Briefcase.context$ is $Tokens$ when the agent reaches back p and the agent is silently destroyed : the configuration remains legitimate. Then, starting from a legitimate configuration, the agent has no effect on the behavior of the system (the token is neither moved nor blocked). This proves also the correctness property of the algorithm.

The finite time to live property is a consequence of the circulation algorithm (an agent performs at most 2 rounds of the ring). For the convergence property, we consider an initial configuration with one agent installed at processor p. If there are no tokens in the system, when the agent reaches back to p, $Briefcase.context$ is $NoTokens$, thus rule 3 applies and the following configuration is legitimate ; if there are more than one token in the system, either the agent meets the supplementary tokens and then they are destroyed, or these tokens are frozen (they cannot move to the next processor) by the first patched algorithm rule. At most one token could pass through the Initiator during the

first round. If the agent has met a token and a token has pass through the initiator, the agent will make another round, destroying one token and no token could pass through the initiator within this round. Thus, the agent has to meet the tokens, and every supplementary token is destroyed. Thus, the configuration following the configuration after the agent came back to the initiator for the second time if necessary is legitimate. □

6 Conclusions

In this paper, we have formally defined the notion of agent (in the context of stabilizing systems), and agent-stabilizing systems. We proposed a set of axioms for having a powerful tool for dealing with failure tolerance. We illustrated the use and the power of this tool on two static problems, which provide a general transformation scheme and two dynamic classical problems.

The agent is a special message that circulates in the system, checks its consistency and mends it if faults are detected. It is created by a lower layer which was not described here, but whose properties are well defined (uniqueness and liveness of creation). We also defined a set of constraints (properties), that the system must satisfy to be considered as agent-stabilizing. In addition to the classical correctness and convergence properties, we added independence and non-interference notions. These notions ensure that the system will not rely on the agent in the (normal) case without failure, and that the presence of the agent in the normal case is not disturbing the system.

We provided a general transformation for self-stabilizing algorithms designed for non-anonymous networks into agent-stabilizing algorithms for anonymous networks. Finally, We have shown that the agent can also be used to transform non-stabilizing algorithms, like the token circulation on anonymous ring with two states per processor, into stabilizing ones.

References

1. J.Beauquier, A.K.Datta, M.Gradinariu, and F.Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In DISC 2000 Distributed Computing 14th International Symposium, Springer-Verlag LNCS:1914, 2000.
2. J.Beauquier, C.Genolini, and S. Kutten. Optimal reactive k-stabilization: the case of mutual exclusion. In PODC'99 Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, pages 209-218, 1999.
3. E.W.Dijkstra. Self stabilizing systems in spite of distributed control. Communications of the Association of the Computing Machinery, 17:643-644, 1974.
4. M.G.Gouda. The triumph and tribulation of system stabilization. In WDAG'95 Distributed Algorithms 9th International Workshop Proceedings, Springer-Verlag LNCS:972, pages 1-18, 1995.
5. S.Ghosh. Agents, distributed algorithms, and stabilization. In Computing and Combinatorics (COCOON 2000), Springer-Verlag LNCS:1858, pages 242-251, 2000.

6. S.Ghosh and A.Gupta. An exercise in fault-containment: self-stabilizing leader election. Information Processing Letters, 59:281-288, 1996.
7. S.T.Huang. The Fuzzy Philosophers. In Parallel and Distributed Processing (IPDPS Workshops 2000), Springer-Verlag LNCS:1800, pages 130-136, 2000.
8. S.Kwek. On a Simple Depth-First Search Strategy for Exploring Unknown Graphs. In WADS97 Proc. 5th Worksh. Algorithms and Data Structures, Springer-Verlag LNCS:1272, 1997.
9. D.Kotz, R.Gray, D.Rus, S.Nog and G.Cybenko, Mobile Agents for Mobile Computing. In Technical Report PCS-TR96-285, May 1996, Computer Science Department, Dartmouth College
10. M.Schneider. Self-stabilization. ACM Computing Surveys, 25:45-67, 1993.
11. G.Tel. Introduction to distributed algorithms. Cambridge University Press, 1994.
12. S.Tixeuil, Auto-stabilisation Efficace. Thèse de doctorat, LRI, January 2000

Stabilization of Routing in Directed Networks

Jorge A. Cobb[1] and Mohamed G. Gouda[2]

[1] Department of Computer Science (EC 31)
The University of Texas at Dallas, Richardson, TX 75083-0688
jcobb@utdallas.edu
[2] Department of Computer Sciences, The University of Texas at Austin
Austin, TX 78712-1188
gouda@cs.utexas.edu

Abstract. Routing messages in a network is often based on the assumption that each link, and so each path, in the network is bidirectional. The two directions of a path are employed in routing messages as follows. One direction is used by the nodes in the path to forward messages to their destination at the end of the path, and the other direction is used by the destination to inform the nodes in the path that this path does lead to the destination. Clearly, routing messages is more difficult in directed networks where links are unidirectional. (Examples of such networks are mobile ad-hoc networks and satellite networks.) In this paper, we present the first stabilizing protocol for routing messages in directed networks. We keep our presentation manageable by dividing it into three (relatively simple) steps. In the first step, we develop an arbitrary directed network where each node broadcasts to every reachable node in the network. In the second step, we enhance the network such that each node broadcasts its shortest distance to the destination. In the third step, we enhance the network further such that each node can determine its best neighbor for reaching the destination.

1 Introduction

The routing of messages from a source node to a destination node is a fundamental problem in computing networks. In general, routing protocols are divided into two broad categories [13]: link-state protocols and distance-vector protocols. In link-state protocols, each node broadcasts a list of its neighbors to all nodes in the network. Each node then builds in its memory the topology of the network, and computes the shortest path to each destination. A total of $O(n^2)$ storage is required to store this topology. Examples of link-state protocols include [7,14]. In distance-vector protocols, each node forwards to all neighboring nodes a vector with its distance to each destination. In this way, only $O(n)$ storage is required. Examples of distance-vector protocols include [1, 5, 6, 8, 12].

There is an implicit assumption in these protocols. It is assumed that each link, and therefore each path, is bidirectional. The forward path from a source node to a destination node is discovered by sending routing messages along this path, and the backward path is used to inform the source of the existence of

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 51–66, 2001.

the forward path. However, new technologies are producing directed networks, that is, networks with unidirectional links. An example is networks with satellite links, since these links allow only unidirectional traffic [3, 4]. Another example is mobile wireless networks, in particular, ad-hoc networks [9, 10]. In these networks, nodes communicate with each other via radio links. These links may be unidirectional for several reasons. For example, there might be a disparity in the transmission power of two neighboring nodes. Thus, only one node may be able to receive messages from the other. In addition, if there is more interference at the vicinity of one node, then the node with the higher interference is unable to receive messages from its neighboring node.

Routing protocols that assume bidirectional links will fail in a directed network [9,10]. That is, the shortest path to a destination will not be found when this path contains unidirectional links. To remedy this shortcoming, new routing protocols have been developed that take unidirectional links into account [3,4,9,10]. Because routing protocols are essential in computing networks, it is desirable for them to be stabilizing. A protocol is said to be stabilizing iff it converges to a normal operating state starting from any arbitrary state [2,11]. Although stabilizing routing protocols exist for undirected networks, the protocols in [3, 4, 9, 10] for directed networks have not been shown to be stabilizing. In [3,4], "tunnels" need to be configured to go around single unidirectional links. Furthermore, this technique is not applicable to networks with an arbitrary number of unidirectional links. In [9,10], the stabilization of the protocol is not addressed (unbounded sequence numbers are used, which may require an unbounded stabilization time.)

In this paper, we present the first stabilizing protocol for routing messages in directed networks. We keep our presentation manageable by dividing it into three (relatively simple) steps. In the first step, we develop an arbitrary directed network where each node broadcasts to every reachable node in the network. In the second step, we enhance the network such that each node broadcasts its shortest distance to the destination. In the third step, we enhance the network further such that each node can determine its best neighbor for reaching the destination. In our network, each node is a process, and for simplicity, processes communicate with each other via shared memory. A message passing implementation is briefly discussed in Section 8. In addition, we assume that the cost of each link in the network is one. Thus, the shortest path (i.e., with the least number of links) is found to each destination. It is straightforward to modify our network to work with arbitrary positive costs assigned to each link.

2 Directed Networks

We consider a network of communicating processes that can be represented by a directed graph. In this directed graph, each node represents a distinct process in the network, and each directed edge from a process u to a process v represents a possible flow of information from u to v. Specifically, a directed edge from a process u to a process v indicates that each action of v can read the variables of both u and v, but can write only the variables of v. Thus, the existence of

a directed path from a process u to a process v indicates that information can flow from u to v (via the intermediate processes in the directed path), and the lack of a directed path from a process u to a process v indicates that information cannot flow from u to v.

If there is a directed edge from a process u to a process v, then u is called a *backward neighbor* of v and v is called a *forward neighbor* of u. In every process u, two constant sets, $B.u$ and $F.u$, are declared as follows.

const $B.u$: set of identifiers of all backward neighbors of u
$F.u$: set of identifiers of all forward neighbors of u

Without loss of generality, we assume that for each process u in a network, both $B.u$ and $F.u$ are non-empty. Each process in the network has a unique identifier in the range $0\,..\,n-1$, where n is the number of processes in the network. Process 0 is called the *network root*.

3 Routing Trees

Information needs to flow from every process in a directed network to the network root. To achieve this goal, every process u maintains the identifier of one of its forward neighbors, the one closest to the network root, in a variable named $next.u$. When the network reaches a stable state, the values of the $next.u$ variables define a directed rooted tree where all the directed paths lead to the network root. This tree is called a *routing tree*.

Also, every process u maintains the length of (i.e., the number of edges in) the shortest directed path from u to the root in a variable called $dist.u$. When the network reaches a stable state, the value of each $dist.u$ variable defines the length of the directed path from u to the root in the routing tree.

In fact, not every process in the network can be in the routing tree for two reasons. First, if the network has no directed path from a process u to the root, then information cannot flow from u to the root, and u cannot be in the routing tree. Second, if the network has no directed path from any backward neighbor of the root to a process u, then u cannot be informed of whether there is a directed path from u to the root. In this case, no information flow will be attempted from u to the root, and u cannot be in the routing tree.

From this discussion, u is in the routing tree of a network iff there is a backward neighbor v of the network root such that the network has a directed path from u to v and a directed path from v to u. The first path (from u to v) can be used as a route for the information flow from u to the root, and the second path (from v to u) can be used to inform u about the existence of the first path. When the network reaches a stable state, if a process u is in the routing tree, then the value of variable $dist.u$ is in the range $0\,..\,n-1$; otherwise, the value of variable $dist.u$ is n.

As an example, consider the directed network in Fig. 1. There is no directed path from processes 3 and 7 to process 0 (the root); thus, processes 3 and 7 cannot be in the routing tree. Also, there is no directed path from a backward

neighbor of process 0 (i.e., from process 2) to processes 1 and 5; thus, processes 1 and 5 cannot be in the routing tree. For each of the other processes (i.e. processes 2, 4, and 6), there is a backward neighbor of process 0 such that there are a directed path from the process to the backward neighbor and a directed path from the backward neighbor to the process. Thus, each of the processes 2, 4, and 6 is in the routing tree.

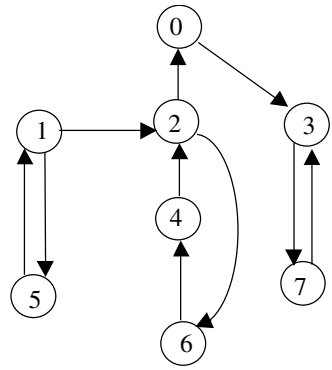


Fig. 1. A directed network.

The routing tree for this network is shown in Fig. 2. In this figure, the values of the two variables *next.u* and *dist.u* are written beside every process *u*. Also, the network edges that belong to the routing tree are shown as solid lines, whereas the other edges are shown as dashed lines.

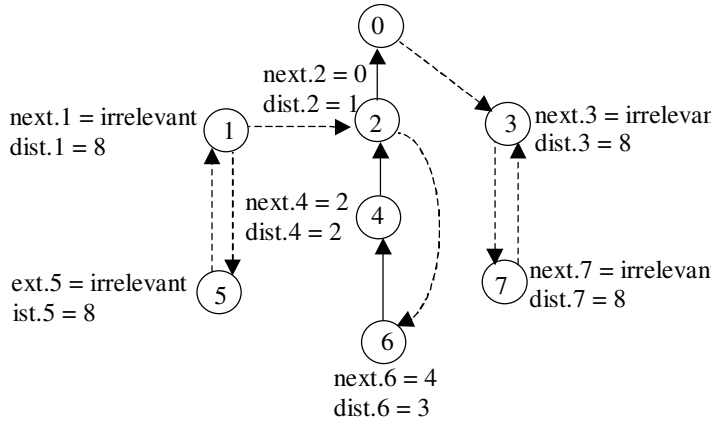


Fig. 2. The routing tree of the network in Fig. 1

Next, we discuss how to make the processes in a directed network maintain a routing tree. For simplicity, we divide our discussion into three steps. In the first step, we discuss how to make each process u broadcast a local value $x.u$ to every other process (that can be reached via a directed path from u) in the directed network. In the second step, we enhance the network process such that each value $x.u$, which is broadcasted by process u, is the shortest distance from u to the root. When the modified network reaches a stable state, each process u knows the network distance vector that stores, for every process v in the network, the shortest distance from v to the network root. In the third step, each process u computes from its network distance vector the two values $next.u$ and $dist.u$ needed for maintaining the routing tree. These three steps are discussed in more detail in Sections 5, 6 and 7.

4 Network Notation

Before presenting our networks of processes, we first give a short overview of the notation that we use in specifying our processes. For simplicity, our processes are specified using a shared memory notation. In particular, each process is specified by a set of constants, a set of variables, a set of parameters, and a set of actions. A process is specified as follows.

process <process name>
const
    <constant name> : <type>,
    ...
    <constant name> : <type>
var
    <variable name> : <type>,
    ...
    <variable name> : <type>
par
    <parameter name> : <type>,
    ...
    <parameter name> : <type>
begin
    <action>
[]
    ...
[]
    <action>
end

The constants declared in a process can be read, but not written, by the actions of that process. The variables declared in a process can be read by the actions of that process and the actions of forward neighbors of that process. The

variables declared in a process can be written only by the actions of that process. Parameters are discussed below.

Every action in a process is of the form <guard> → <body>. The <guard> is a boolean expression over the constants, variables, and parameters declared in the process, and also over the variables declared in the backward neighbors of that process. The <body> is a sequence of assignment statements that update the variables of the process.

Each parameter declared in a process is used as a shorthand to write a set of actions as one action. For example, if we have the following parameter definition,

par g : 1 . . 3

then the following action

$x = g \quad \rightarrow \quad x := x + g$

is a shorthand notation for the following three actions.

$x = 1 \quad \rightarrow \quad x := x + 1$
[]
$x = 2 \quad \rightarrow \quad x := x + 2$
[]
$x = 3 \quad \rightarrow \quad x := x + 3$

An execution step of a network consists of evaluating the guards of all the actions of all processes, choosing one action whose guard evaluates to true, and executing the body of this action. An execution of a network consists of a sequence of execution steps, which either never ends, or ends in a state where the guards of all the actions evaluate to false. We assume all executions of a network to be weakly fair, that is, an action whose guard is continuously true is eventually executed.

5 Directed Broadcast

In this section, we discuss a directed network where each process u computes an array $X.u$ that has n elements, where n is the number of processes in the network. When the network reaches a stable state, the v^{th} element $X[v].u$ in array $X.u$ has the value $x.v$ that is local to process v.

In this network, each process u maintains, along with each element $X[v].u$, two corresponding elements:

$b[v].u$ = identifier of a backward neighbor of u from which u has read the latest value of $X[v].u$

$d[v].u$ = length of (i.e., number of edges in) the directed path along which the value $x.v$ (in v) is transmitted to $X[v].u$ (in u)

Note that if the value of $d[v].u$ ever becomes n, then process u recognizes that it has not yet found a directed path from v to u and that the current value

of $X[v].u$ is probably incorrect. Thus, if the network has no directed path from process v to process u, then the value of $d[v].u$ stabilizes to n and the value of $X[v].u$ stabilizes to a probably incorrect value.

Each process u in the directed broadcast network is defined next. In this definition, we use the expression $a \oplus b$ to mean $\min(a + b, n)$.

process $u : 0 \,..\, n-1$
const
  $B.u$: set of identifiers of all backward neighbors of u
  $F.u$: set of identifiers of all forward neighbors of u
  $x.u$: $0 \,..\, n$ {local constant in u}
var
  $X.u$: **array** $[0 \,..\, n-1]$ **of** $0 \,..\, n$,
  $b.u$: **array** $[0 \,..\, n-1]$ **of** $B.u$,
  $d.u$: **array** $[0 \,..\, n-1]$ **of** $0 \,..\, n$
par
  v : $0 \,..\, n-1$, { any process in the network }
  w : $B.u$ { any backward neighbor of u }
begin
  $X[u].u \neq x.u \ \vee \ d[u].u \neq 0 \ \rightarrow$
    $X[u].u := x.u;$
    $d[u].u := 0$
[]
  $v \neq u \ \wedge \ b[v].u = w \ \wedge \ (X[v].u \neq X[v].w \ \vee \ d[v].u \neq d[v].w \oplus 1) \ \rightarrow$
    $X[v].u := X[v].w;$
    $d[v].u := d[v].w \oplus 1$
[]
  $v \neq u \ \wedge \ d[v].w \oplus 1 < d[v].u \ \rightarrow$
    $X[v].u := X[v].w;$
    $b[v].u := w;$
    $d[v].u := d[v].w \oplus 1$
end

Process u has three actions. In the first action, u ensures that $X[u].u$ equals its local constant $x.u$ and $d[u].u$ is zero. In the second action, u recognizes that it has read the latest value of $X[v].u$ from a backward neighbor w and so ensures that $X[v].u$ equals $X[v].w$ and $d[v].u$ equals $d[v].w \oplus 1$. In the third action, u recognizes there is a shorter directed path from v to u along its backward neighbor w. In this case, u assigns to $X[v].u$ the value of $X[v].w$, assigns to $b[v].u$ the value of w, and assigns to $d[v].u$ the value $d[v].w \oplus 1$.

This network maintains for every process u, a stabilizing, rooted, shortest-path spanning tree $T.u$. The root of $T.u$ is process u itself, and $T.u$ contains every process that is reachable from u via a directed path. The value of the constant $x.u$ flows from u over $T.u$ to every process in $T.u$.

6 Distance Vectors

In this section, we modify the network in the previous section such that the value of each element $X[v].u$ in each process u, when the network reaches a stable state, is the shortest distance (i.e., the smallest number of edges in a directed path) from process v to the network root. The modification is slight. The second and third actions in every process remain the same as before. Only the first action of each process u is modified to become as follows.

$$\begin{array}{l} X[u].u \neq f(u, F.u, X.u) \ \vee \ d[u].u \neq 0 \ \rightarrow \\ \qquad\qquad X[u].u := f(u, F.u, X.u); \\ \qquad\qquad d[u].u := 0 \end{array}$$

Above, $f(u, F.u, X.u)$ computes the shortest distance from u to the network root, and is defined as follows.

$$\begin{array}{l} f(u, F.u, X.u) = 0 \ \text{ if } u = 0, \\ \qquad\qquad\qquad\quad 1 \ \text{ if } u \neq 0 \wedge \ 0 \in F.u, \\ \qquad\qquad\qquad\quad (\text{min over } v,\ v \in F.u \wedge d[v].u < n, \text{ of } X[v].u) \oplus 1 \ \text{ otherwise} \end{array}$$

In the appendix, we present a proof of the stabilization of this network.

7 Maintaining a Routing Tree

To make the network in the previous section maintain a routing tree (as defined in Section 3), each process u, $u \neq 0$, is modified as follows. First, the following two variables $next.u$ and $dist.u$ are added to process u.

$$\begin{array}{lll} \textbf{var} & next.u & : \ F.u, \\ & dist.u & : \ 0\,..\,n \end{array}$$

Second, the following action is added to process u.

$$\begin{array}{l} next.u \neq h(F.u, X.u) \ \vee \ dist.u \neq X[u].u \quad \rightarrow \\ \qquad\qquad next.u := h(F.u, X.u); \\ \qquad\qquad dist.u := X[u].u \end{array}$$

where

$$\begin{array}{l} h(F.u, X.u) = \text{the smallest identifier } w \text{ in } F.u \text{ such that} \\ \qquad\qquad X[u].u = X[w].u \oplus 1 \end{array}$$

8 Message Passing Implementation

In order to simplify our presentation, processes in our network communicate with their neighbors using shared memory. In this section, we discuss a message passing implementation of our network.

We first address the construction of the shortest spanning trees discussed in Section 5. In our shared memory model, each process reads array d from each of its backward neighbors. To implement this, each process places the contents of its d array into a message, and periodically sends this message to all its forward neighbors. We will refer to this message as the spanning-tree message. Since array d has n entries, the size of the spanning-tree message is $O(n)$.

We next address the broadcast of the distance to the root (i.e. $X[u].u$) by every process u. To implement this, each process u places its distance to the root and its process identifier into a message, which we refer to as the broadcast message. This message is periodically sent to all forward neighbors of u. When a process v receives a broadcast message whose process identifier is u, this message is forwarded to all the forward neighbors of v, provided the message was received from neighbor $b[u].v$. In this way, the broadcast message is forwarded only along the spanning tree $T.u$, and the propagation of this message is cycle free. Note that the size of the broadcast message is $O(1)$.

The above two messages, namely the spanning-tree and broadcast message, are all that is required to implement the process network in a message passing model. We next address the storage requirements of each process.

Each process is required to store its d array, whose size is $O(n)$. With respect to the distances to the root (i.e., array X), note that when a process computes its distance to the root, it only requires the distance to the root of each of its forward neighbors. Therefore, in a message passing implementation, the distance to the root broadcasted by any other process is simply forwarded as soon as it is received. Thus, only the distances of the forward neighbors need to be stored.

Furthermore, with a more careful implementation, only the distance to the root of the next hop neighbor ($next.u$) needs to be stored. If process u receives a broadcast from a forward neighbor indicating a smaller distance to the root than that of neighbor $next.u$, then $next.u$ is updated to this neighbor, and the new distance is recorded. Thus, we require $O(1)$ storage for the distance to the root.

In our network, we considered only a single process (the root process) as the destination. To allow any process to be a destination, each process needs to maintain the distance to each destination. Thus, instead of $O(1)$ storage for the distance to the root mentioned above, we require $O(n)$ storage, i.e., $O(1)$ storage for each of the n possible destinations. Note, however, that array d remains as before, since our original network allows every process to perform a broadcast. Since array d requires $O(n)$ storage, the storage remains $O(n)$. In addition, the broadcast message would include the distance to each destination, and thus would be of size $O(n)$. The spanning-tree message remains $O(n)$. Therefore, since vectors of size n are sent to each neighbor, this network falls into the category of distance-vector routing networks.

One final issue remains to be addressed, and that is the detection of channel failure. Thus far, we have assumed that if a process v is in the forward neighbor set $F.u$ of a process u, then the channel from u to v is in working order. This is possible in networks where the channel is implemented by a lower layer, and the

status of this channel is monitored by the lower layer (e.g., the channel could be an ATM circuit, and the status of the channel is maintained by the ATM layer). If the lower layer at u detects that the channel from u to v has failed, then v is removed from $F.u$. However, if the lower layer does not provide the capability of monitoring the status of the channel, process u can monitor the status as follows. When process v sends a broadcast message, in addition to including its distance to each destination, it also includes a list of its backward neighbors from whom it has recently received messages. When process u receives a broadcast from v, it checks if u is in the list of backward neighbors of v. If so, then the channel from u to v is in working order.

Note that if the broadcast message includes the list of backward neighbors, we may choose to only include this list in the message, and not include the distance to each destination. In this case, each process would have to collect the list of neighbors from each process in the network, build in its memory a graph representing the network topology, and choose its next hop neighbor using Dijkstra's [13] shortest path algorithm. In this case, the storage required would increase to $O(n^2)$, and the network would fall into the category of link-state routing networks.

Finally, note that if a list of neighbors is required in the broadcast message, either because we have a link-state network or we require to detect the status of the channel in a distance-vector network, then the requirements for a process u to be in the routing tree are different than those presented in Section 3. In this case, a path must exist from u to the root and a path must also exist from the root to u.

9 Concluding Remarks

In this paper, we presented a network of processes that constructs a routing tree to a given destination, even though the network is directed, i.e., communication between neighboring processes may be unidirectional. We presented our network in three steps. First, we presented a network that allows each process to broadcast a value to all other processes. Next, we presented a network where each process can compute its distance to the destination, and broadcast this distance to all processes. Finally, we presented a network where a routing tree is constructed by having each process choose its parent in the routing tree in accordance to its distance to the destination.

Since an undirected network is a special case of a directed network, the network of processes we presented in Section 7 will correctly build a routing tree in an undirected network. However, it will not do so in the most efficient way, since processes are tailored towards a directed network. In future work, we will investigate networks of processes whose behavior will vary depending on the number of processes that have unidirectional communication with their neighbors. That is, processes will adapt to the "level" of unidirectional communication in the network, and adapt their behavior accordingly to improve performance.

Routing in directed networks is a fertile area of research, and much is yet to be done. Existing approaches assume bidirectional communication between neighbors, and thus will fail or exhibit different behaviors in directed networks. In addition, other distributed algorithms, in addition to routing, may be affected by directed networks. Two reasons for this may be routing asymmetry, i.e., for two nodes u and v, the path from u to v is not necessarily the same as the path from v to u, and one-way reachability, that is, there is a path from u to v but there is no path from v to u.

References

1. Cobb J., Waris M.: Propagated Timestamps: A Scheme for the Stabilization of Maximum-Flow Routing Protocols. In: Proceedings of the Third Workshop on Self-Stabilizing Systems (1997) pp. 185-200
2. Dolev S.: Self-Stabilization. MIT press, Cambridge, MA (2000)
3. Duros E., Dabbous W.: Supporting Unidirectional Links in the Internet. In: Proceedings of the First International Workshop on Satellite-Based Information Services (1996)
4. Duros E., Dabbous W., Izumiyama H., Fujii N., and Zhang Y.: A Link Layer Tunneling Mechanism for Unidirectional Links. Internet Request for Comments (RFC) 3077 (2001)
5. Garcia-Luna-Aceves, J.J.: Loop-Free Routing Using Diffusing Computations. IEEE/ACM Transactions on Networking. Vol 1 No. 1 (Feb. 1993)
6. Hedrick C.: Routing Information Protocol. Internet Request for Comments (RFC) 1058 (1988)
7. Moy J: OSPF Version 2. Internet Request for Comments (RFC) 1247 (1991)
8. Perkins C., Bhagwat P.: Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In: Proceedings of the ACM SIGCOMM Conference on Communication Architectures, Protocols and Applications (1994)
9. Prakash R.: Unidirectional Links Prove Costly in Wireless Ad Hoc Networks. In: Proceedings of the ACM International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL M for Mobility) (1999)
10. Prakash R., Singhal M.: Impact of Unidirectional Links in Wireless Ad Hoc Networks. DIMACS Series in Discrete Mathematics and Computer Science, Vol. 52 (2000)
11. Schneider M.: Self-Stabilization. In: ACM Computing Surveys Vol. 25 No. 1 (1983)
12. Schneider M., Gouda M.: Stabilization of Maximal Metric Trees. In: Proceedings of the International Conference on Distributed Computing Systems, Workshop on Self-Stabilizing Systems (1999)
13. Tanenbaum A.: Computer Networks (3rd edition). Prentice Hall (1996)
14. Vutukury S., Garcia-Luna-Aceves J.J.: A Simple Approximation to Minimum Delay Routing. In: Proceedings of the ACM SIGCOMM Conference on Communication Architectures, Protocols and Applications (1999)

Appendix: Proof of Stabilization

Let $P_{min}(v, u)$ be a shortest path from v to u. Let $G(v)$ be the graph obtained from all edges of the form $(b[v].u, u)$ for every process u, $u \neq v$. Let $P_G(v, u)$ be

the path from v to u in $G(v)$[1]. If no such path exists, then $P_G(v,u)$ is the empty path. Let *btree* denote the following predicate.

$$
\begin{array}{l}
(\forall\, u :: d[u].u = 0) \wedge \\
(\forall\, v,u : v \neq u \;\wedge\; P_{min}(v,u) = \emptyset : d[v].u = n) \wedge \\
(\forall\, v,u : v \neq u \;\wedge\; P_{min}(v,u) \neq \emptyset : \\
\qquad\qquad P_G(v,u) \neq \emptyset \;\wedge\; |P_G(v,u)| = |P_{min}(v,u)| = d[v].u)
\end{array}
$$

Lemma 1.

1. *btree is stable*
2. *true converges to btree*

Proof. We focus only on arrays $d.u$ and $b.u$ of each process u, since only these variables are involved in *btree*. We first show the stability of *btree*.

Consider the first action. This action affects only $d[u].u$. From *btree*, $d[u].u = 0$ before the action. Thus, executing the action does not change $d[u].u$.

Consider the second action. This action affects only $d[v].u$. We have two cases.

1. Consider first $d[v].w < n$. In this case, from *btree*, $P_G(v,w) \neq \emptyset$, and $d[v].w = |P_G(v,w)| = |P_{min}(v,w)|$. Since $P_G(v,w) \neq \emptyset$ and $b[v].u = w$, then $P_G(v,u) = P_G(v,w);(w,u)$, and thus $P_{min}(v,u) \neq \emptyset$. This, along with *btree*, implies that $d[v].u = |P_{min}(v,u)| = |P_G(v,u)| = |P_G(v,w)| + 1 = d[v].w + 1$. Also, since $|P_{min}(v,u)| < n$, then $d[v].w + 1 < n$, and hence, $d[v].w + 1 = d[v].w \oplus 1$. Thus, $d[v].u$ is not changed when it is assigned $d[v].w \oplus 1$.
2. Consider instead $d[v].w = n$. In this case, from *btree*, $P_{min}(v,w) = \emptyset$. Hence, $P_G(v,w) = \emptyset$. From $b[v].u = w$, we have $P_G(v,u) = \emptyset$. From *btree*, if $P_{min}(v,u) \neq \emptyset$, then $P_G(v,u) \neq \emptyset$. Thus, we must have $P_{min}(v,u) = \emptyset$. Again, from *btree*, $d[v].u = n$, and thus $d[v].u$ does not change when it is assigned $d[v].w \oplus 1$.

Consider now the third action. Again, we have two cases.

1. Consider first $d[v].u < n$. From *btree*, $d[v].u = |P_{min}(v,u)| = |P_G(v,u)|$. From the action's guard, $d[v].w \oplus 1 < d[v].u$, which implies $d[v].w < n$, and from *btree*, $d[v].w = |P_G(v,w)| = |P_{min}(v,w)|$. Note, however, that since w is a backward neighbor of u, $d[v].w \oplus 1 < d[v].u$ implies that $|P_G(v,w);(w,u)| < |P_G(v,u)|$, which is impossible, since $P_G(v,u)$ is the shortest path from v to u. Thus, the guard must be false if *btree* holds.
2. Consider now $d[v].u = n$. From *btree*, there is no path from v to u. However, if $d[v].w \oplus 1 < d[v].u$, then $d[v].w < n$, and *btree* implies there is a path from v to w, and thus there is also a path from v to u. Thus, again, the guard must be false if *btree* holds.

[1] Note that at most only one such path may exist, since u has only one incoming edge in $G(v)$, and v has no incoming edge.

Since no action falsifies *btree*, *btree* is stable. We now show that true converges to *btree*. First, note that the first action of u assigns zero to $d[u].u$. No other action modifies $d[u].u$, thus, for all u, true converges to $d[u].u = 0$, and $d[u].u = 0$ is stable.

Next, define $btree(v, i)$, where $1 \leq i < n$, as follows:

$$(\forall\, u : u \neq v \;\wedge\; (P_{min}(v,u) = \emptyset \;\vee\; |P_{min}(v,u)| > i) : d[v].u > i) \;\wedge$$
$$(\forall\, u : u \neq v \;\wedge\; P_{min}(v,u) \neq \emptyset \;\wedge\; |P_{min}(v,u)| \leq i : d[v].u = |P_{min}(v,u)| = |P_G(v,u)|)$$

We show that eventually, for all i, $btree(v, i)$ holds and continues to hold. We show this by induction.

As a base case, consider $i = 1$. Consider any process u, $u \neq v$. The second and third actions assign at least 1 to $d[v].u$. Thus, we can assume we reach a state where $d[v].u \geq 1$ holds and continues to hold for all processes u, $u \neq v$.

Consider a process u which is *not* a forward neighbor of v. Thus, for any backward neighbor w of u, $d[v].w \geq 1$, and thus $d[v].u \geq 2$ after the second or third action. Thus, $d[v].u \geq 2$ will hold and continue to hold, as desired in $btree(v, 1)$.

Consider now a process u which *is* a forward neighbor of v. We have two cases.

1. Assume u executes its second or third action with parameter $w = v$. Then, since $b[v].v = 0$, we obtain $b[v].u = v \wedge d[v].u = 1$, thus, $d[v].u = |P_{min}(v,u)| = |P_g(v,u)|$ as desired by $btree(v, 1)$. Note that this continues to hold for the following reason. First, the second action of u does not change $b[v].u$, and it assigns $d[v].v \oplus 1 = 1$ to $d[v].u$. The third action of u is not enabled, since for any backward neighbor w of u (including $w = v$), $d[v].w \oplus 1 \geq 1 = d[v].u$.
2. Assume u executes its second or third action with parameter $w \neq v$. Since $d[v].w \geq 1$, it results in $b[v].u = w \wedge d[v].u > 1$. Since $d[v].u > 1$, then the third action of u is enabled with $w = v$, and must be eventually executed, after which $b[v].u = v \wedge d[v].u = 1$, as shown above. Also, this continues to hold as shown above.

For the induction hypothesis, we assume $1 < i < n$, and $btree(v, i-1)$ holds.

Consider first any process u, where $u \neq v \wedge P_{min}(v,u) = \emptyset$. For any backward neighbor w of u, $P_{min}(v,w) = \emptyset$, and from $btree(v, i-1)$, $d[v].w \geq i-1$. Hence, when process u executes its second or third action, $d[v].u \geq i$, as desired. Since $d[v].w \geq i-1$ will continue to hold, then so will $d[v].u \geq i$.

Consider next any process u, $u \neq v$, where $P_{min} \neq \emptyset \wedge |P_{min}(v,u)| > i > i-1$. In this case, any backward neighbor w of u must have $P_{min}(v,u) = \emptyset \vee |P_{min}(v,w)| \geq i > i-1$, and from $btree(v, i-1)$, $d[v].w \geq i$, and this continues to hold. When u executes its second or third action, $d[v].u > i$ will result. Thus, $d[v].u > i$ will hold and continue to hold.

Consider now a process u with $P_{min} \neq \emptyset \wedge |P_{min}(v,u)| = i$. This implies that for all backward neighbors w of u, $P_{min}(v,w) = \emptyset \vee |P_{min}(v,w)| \geq i-1$.

From $btree(v, i-1)$, $d[v].w \geq i-1$. In addition, u must have a backward neighbor y such that $P_{min}(v,y) \neq \emptyset \wedge |P_{min}(v,y)| = i-1$. From $btree(v, i-1)$, $i-1 = d[v].y = |P_G(v,y)| = |P_{min}(v,y)|$ holds and will continue to hold. We have two cases.

1. Assume the second or third action of u is executed where $w = y$. Then, after the action is executed, $b[v].u = y \wedge d[v].u = i$, and hence, $P_G(v,u) = PG(v,y);(y,u) \wedge |P_G(v,u)| = i = |P_{min}(v,u)|$ as desired for $btree(v,i)$. Furthermore, executing the second action does not change these values, and note that the third action cannot execute, since all backward neighbors w of u must have $d[v].w \geq i-1$, and thus, $d[v].w \oplus 1 \geq d[v].u$. Hence $i = d[v].u = |P_{min}(v,u)| = |P_G(v,u)|$ will continue to hold forever.
2. Assume the second or third action of u is executed for some backward neighbor w, where $w \neq y$. In this case, $P_{min}(v,w) = \emptyset \vee |P_{min}(v,w)| = i$. From $btree(v, i-1)$, $d[v].w \geq i$. Then, after the action, $d[v].u \geq i+1$. Note that in this case, the third action is enabled with $w = y$, and it will eventually be executed. As argued above, $i = d[v].u = |P_{min}(v,u)| = |P_G(v,u)|$ will then hold and continue to hold.

This finishes the induction step, and thus the induction proof.

Note that $btree = (\forall v, i : 1 \leq i < n : btree(v,i)) \wedge (\forall u :: d[u].u = 0)$. Thus, from the above, eventually we reach a state where $btree$ holds and continues to hold.

Corollary 1. *For all u, v, and y, where $u \neq v$,*

$$btree \wedge b[v].u = y \text{ is stable}$$

Proof. Only the third action affects $b[v].u$, so we focus on this action.

Assume first that $P_{min}(v,u) = \emptyset$. Thus, $P_{min}(v,w) = \emptyset$. From $btree$, $d[v].u = n \wedge d[v].w = n$. Thus, the third action is not enabled.

Assume next that $P_{min}(v,u) \neq \emptyset$. From $btree$, $d[v].u = |P_{min}(v,u)| < n$. For any backward neighbor w of u, if the guard $d[v].w \oplus 1 < d[v].u$ is true, then this implies $d[v].w < n-1$, and from $btree$, $d[v].w = |P_{min}(v,w)|$. Combining the above, $|P_{min}(v,w)| \oplus 1 < |P_{min}(v,u)|$. This is not possible since $P_{min}(v,u)$ is a minimum path. Thus, the third action is not enabled.

Lemma 2.

1. *Consider a computation in which $btree \wedge X[v].v \leq k$ holds for some k and continues to hold. Let $u \neq v$ and $P_G(v,u) \neq \emptyset$. Then, for any y, where $y \neq v$ and y is a process in $P_G(v,u)$, $X[v].y \leq k$ will hold and continue to hold.*
2. *Consider a computation in which $btree \wedge X[v].v \geq k$ holds for some k and continues to hold. Let $u \neq v$ and $P_G(v,u) \neq \emptyset$. Then, for any y, where $y \neq v$ and y is a process in $P_G(v,u)$, $X[v].y \geq k$ will hold and continue to hold.*

Proof. We consider only part 2 above. The proof for part 1 is similar.

From Lemma 1 and Corollary 1, *btree* continues to hold, and $P_G(v,u)$ does not change. Also, only the second action modifies $X[v].u$, so we focus on this action.

The proof is by induction on the length of $P_G(v,u)$. Assume that $P_G(v,u) = (v,u)$, i.e., $b[v].u = v$. If $X[v].u \not= X[v].v$, then the second action of u is enabled with $v = w$. When this action executes, $X[v].u = X[v].v \geq k$. If this action becomes disabled before being executed, then from its guard it must also be that $X[v].u = X[v].v \geq k$. Thus, $X[v].u \geq k$ will hold and continue to hold.

For the induction step, let $w = b[v].u$, and let $|P_G(v,u)| = i, i > 1$. We assume the lemma holds for all paths in $G(v)$ of length $i-1$, in particular, $P_G(v,w)$. Thus, eventually, $X[v].w \geq k$ holds, and it continues to hold. A similar argument as the one above, except that process u assigns $X[v].w$ to $X[v].u$, shows that $X[v].u \geq k$ will hold and continue to hold. This concludes the proof.

Let S_i, where $0 \leq i < n$, be a set of processes defined as follows.

$$
\begin{aligned}
S_0 &= \{\ 0\ \} \\
S_1 &= \{\ u \mid u \text{ is a backwards neighbor of process } 0\} \cup S_0 \\
S_i &= \{\ u \mid \text{there is a path from a backwards neighbor } v \\
&\qquad \text{of the root to } u, \text{ and a path of length at most } i \\
&\qquad \text{from } u \text{ to the root via } v\ \} \cup S_1,\ 1 < i < n
\end{aligned}
$$

Note that a process u is in the routing tree iff $u \in S_{n-1}$.

Lemma 3. *For every i, $1 < i < n$,*

$$u \in S_i \Leftrightarrow (u \in S_1\ \vee\ (\exists\, v : v \in F.u : v \in S_{i-1}\ \wedge\ P_{min}(v,u)\ \not= \emptyset))$$

Proof. Consider first the following implication.

$$u \in S_i \Leftarrow (u \in S_1\ \vee\ (\exists\, v : v \in F.u : v \in S_{i-1}\ \wedge\ P_{min}(v,u)\ \not= \emptyset))$$

If u is in S_1 then, from the definition of S_i, u is in S_i for any i, $i > 1$. Instead, assume u is not in S_1. Thus, assume there exists a v satisfying the quantification. We are given that $v \in S_{i-1}$. Note that $i - 1 > 0$, since otherwise, v would be the root, and u would be in S_1.

Assume first that $i - 1 = 1$. In this case, v is a backward neighbor of the root, and from $P_{min}(v,u)\ \not= \emptyset$, there is a path from v to u. Thus, u is in S_i. Assume instead that $i - 1 > 1$. Then there is a path of length $i - 1$ from v to the root via a backward neighbor w of the root, and a path from w to v. Since $v \in F.u$ and $P_{min}(v,u)\ \not= \emptyset$, there is a path of length i from u to the root via w (and via v), and also a path from w to u (via v). Thus, u is in S_i.

Consider now the other implication.

$$u \in S_i \Rightarrow (u \in S_1\ \vee\ (\exists\, v : v \in F.u : v \in S_{i-1}\ \wedge\ P_{min}(v,u)\ \not= \emptyset))$$

If u is in S_1, then we are done. Assume instead that u is in S_i, but u is not in S_1. Then, since u belongs to S_i, there is a path from u to the root via a backward

neighbor w of the root, and the length of this path is i. Let v be the next hop from u to w along this path. Thus, there is a path of length $i-1$ from v to the root via w. Also, since there is a path from w to u, then there is a path from v to u. Hence, $P_{min}(v,u) \neq \emptyset$ and v is in S_{i-1}.

Combining both implications we obtain the desired result.

Theorem 1. *For every i, $0 \leq i < n$, the distance vector network stabilizes to the following predicate.*

$$(\forall\, u :: u \in S_i \Leftrightarrow X[u].u \leq i)$$

Proof. $X[u].u$ is changed only in the first action, so we focus only on this action. Also, from Lemma 1, the network stabilizes to *btree*. Thus, consider a computation starting from a state in which *btree* holds.

Consider first $i = 0$. The only process in S_0 is the root process 0. The definition of function f ensures that $X[0].0$ is assigned 0 regardless of the network state. Thus, $X[0].0 = 0$ is stable. For any process u, $u \neq 0$ f assigns at least 1 to $X[u].u$. Thus, $u \in S_0 \Leftrightarrow X[u].u \leq 0$ will hold and continue to hold.

Consider next $i = 1$. Let $u \in S_1$. By the definition of S_1, the root is a forward neighbor of u. From the definition f, $X[u].u$ will be assigned 1 regardless of the network state. Thus, $X[u].u = 1$ will hold and continue to hold. Consider now any process u, where $u \notin S_1$. Thus, for every forward neighbor v of u, $v \neq 0$, i.e., $v \notin S_0$, and from above, $X[v].v \geq 1$ will hold and continue to hold. If there is a path from v to u then, from Lemma 2 part 2, eventually $X[v].u \geq 1$ holds and continues to hold. If there is no path from v to u, then from *btree*, $d[v].u = n$ holds and continues to hold. Thus, from the definition of f, $X[u].u$ will always be assigned at least 2, i.e., $X[u].u > 1$ will hold and continue to hold.

The remainder of the proof is by induction over i, where $1 \leq i < n$. We show that for each i, the network stabilizes to $(\forall\, u :: u \in S_i \Leftrightarrow X[u].u \leq i)$. The base case, $i = 1$, was shown above. Thus, consider $1 < i < n$, and assume we have a computation where we have reached a state where $(\forall\, u :: u \in S_{i-1} \Leftrightarrow X[u].u \leq i-1)$ holds and continues to hold.

Consider a process u, $u \in S_i$, and $u \notin S_{i-1}$. From the definition of S_i, and from Lemma 3, a forward neighbor v belongs to S_{i-1} and there is a path from v to u. From the induction hypothesis, $X[v].v \leq i-1$, and from *btree*, $d[v].u < n$. Furthermore, from Lemma 2 part 1, eventually $X[v].u \leq i-1$ holds and continues to hold. Thus, from the definition of f, $X[u].u$ is assigned a value at most i, and this continues to hold.

Consider now a process u, $u \notin S_i$. From Lemma 3, $u \notin S_1$, and for all forward neighbors v of u, $v \notin S_{i-1} \ \vee \ P_{min}(v,u) = \emptyset$. If $v \notin S_{i-1}$, then from the induction hypothesis, $X[v].v > i-1$, i.e., $X[v].v \geq i$, holds and continues to hold, and from Lemma 2 part 2, eventually $X[v].u \geq i$ holds and continues to hold. If $P_{min}(v,u) = \emptyset$, then from *btree*, $d[v].u = n$. Hence, from f, eventually $X[u].u > i$ holds and continues to hold.

Thus, by induction, for all $i, 0 \leq i < n$, the network stabilizes to $(\forall\, u :: u \in S_i \Leftrightarrow X[u].u \leq i)$.

Dijkstra's Self-Stabilizing Algorithm in Unsupportive Environments

Shlomi Dolev[1⋆] and Ted Herman[2⋆⋆]

[1] Department of Computer Science, Ben-Gurion University — Israel
dolev@cs.bgu.ac.il
[2] Department of Computer Science, University of Iowa
herman@cs.uiowa.edu

Abstract. The first self-stabilizing algorithm published by Dijkstra in 1973 assumed the existence of a central daemon, that activates one processor at time to change state as a function of its own state and the state of a neighbor. Subsequent research has reconsidered this algorithm without the assumption of a central daemon, and under different forms of communication, such as the model of link registers. In all of these investigations, one common feature is the atomicity of communication, whether by shared variables or read/write registers. This paper weakens the atomicity assumptions for the communication model, proposing versions of Dijkstra's algorithm that tolerate various weaker forms of atomicity, including cases of regular and safe registers. The paper also presents an implementation of Dijkstra's algorithm based on registers that have probabilistically correct behavior, which requires a notion of weak stabilization, where Markov chains are used to evaluate the probability to be in a safe configuration.

1 Introduction

The self-stabilization concept is not tied to particular system settings. Our work considers several new system settings and demonstrates the applicability of the self-stabilization paradigm to these systems. In particular, we investigate systems with regular and safe registers and present modifications of Dijkstra's first self-stabilizing algorithm [5] that stabilizes in these systems.

Dijkstra's presentation of self-stabilization [5, 6, 7] relies on communication by reading neighbor states and updating one machine's state in one atomic operation. The well-known first algorithm of [5] represents a machine state by a counter value. Subsequently, this fundamental algorithm has been adapted to link register [8] and message-passing [23] models. These adaptations are straightforward, essentially changing only the domain of the counter values and taking care to compare communication variables to the local state variables in the correct manner. The processing of communication variables, be they message buffers

⋆ Dolev's work was supported by BGU seed grant.
⋆⋆ Herman's work is sponsored by NSF award CAREER 97-9953 and DARPA contract F33615-01-C-1901.

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 67–81, 2001.

or communication registers, is atomic in these adaptations of [5]. On the other hand, only few papers [17, 20, 14, 3] address non-atomic communication operations in the context of self-stabilization. Lamport initially demonstrated that interprocess communication without explicit synchronization is possible [15], and formalizations of less-than-atomic communication were subsequently developed in [21, 16]. The register hierarchy and register constructions of [16] inspired an active research area. The register hierarchy (safe, regular, and atomic registers) has many motivations, including implementation cost for shared register operations. Another view of weaker forms of registers (safe or regular, when compared to atomic) is that they are possible "failure modes" atomic registers. Thus, if we can adapt algorithms such as [5] to accommodate weaker communication assumptions, the result will be an algorithm that not only recovers from transient faults but also deals with certain types of functional errors — hence the title of our paper, the so-called "unsupportive environments." The idea of combining self-stabilization with other forms of fault-tolerance has previously been studied [1, 12, 9, 10, 2].

We summarize our modifications to [5] as follows. The solution for the regular registers case uses a special label in between writes of counter values. In the case of safe registers we prove impossibility results, for the cases in which neighboring processors use a single safe register to communicate between themselves — where the register is/isn't divided to multiple fields. In the positive side, we define a composite safe register that, roughly speaking, ensures reads return at most one corrupted field and design an algorithm for that case. Subsequently, we consider a stronger model where processors can read the value written in their output registers (therefore avoiding extra writes for refreshes). We present two algorithms for the above case, one that uses unary encoding and another that is based on Gray code.

Then we introduce *randomized registers* that, roughly speaking, return the "correct value" with probability p. It is impossible to ensure closure in such a system, since all reads may return incorrect values. We introduce the notion of *weak self-stabilization* for such systems. We use Markov chains to compute the ratio between the number of safe configurations and unsafe configurations in an infinite execution.

Markov chains associate each state (system configuration) with a probability to be in that state during an infinite execution. The fixed probability of the state is a "stabilizing" value. It is clear that the probability is either zero or one in the first configuration. Given the probability of transitions between configurations, one can compute the stable probability in an infinite execution, which is typically greater than zero and less than one. We found the definition of weak stabilization and the use of Markov chains to be an interesting and promising way for extending the applicability of the self-stabilizing concept.

The remainder of the paper is organized as follows. The next section reviews Dijkstra's algorithm and discusses the problem of adapting it to different register models. Section 3 describes a solution for regular registers. Then in Section 4 we present impossibility results and algorithms for different settings of systems that use safe registers. Randomized registers and the use of Markov chains are

presented in Section 5. Some concluding remarks are given in Section 6. Detailed proofs are omitted from this extended abstract.

2 Adapting Dijkstra's Algorithm

For the remainder of the paper, we let DijAlg refer to the first self-stabilizing algorithm of [5]. This algorithm is expressed by guarded assignment statements and based on a ring of n machines that communicate unidirectionally with atomic (central demon) execution semantics. Generally, conversion of self-stabilizing algorithms from one model to another can be difficult [13]. However the specific case of DijAlg is not difficult to adapt to register models.

For the register models, each "machine" of DijAlg is replaced by a processor, which has an internal state represented by variables and a program counter. Each processor p_i, for $0 \leq i < n$, can read from one or more input registers and write to one or more output registers (these communication registers are often called *link* registers). The unidirectional nature of the ring implies that the set of output registers for p_i can only be written by p_i and that the input registers of p_{i+1} (processor subscripting implicitly uses mod n arithmetic) are precisely the output registers of p_i. One variant of the register model allows p_i to write its output registers, but prohibits p_i from reading its output registers. In the literature on registers, this type of link register is called 1W1R, because the register has exactly one writer p_i and exactly one reader p_{i+1}. We also consider the case of 1W2R link registers, which allows both p_i and p_{i+1} to read the output register of p_i.

The control structure of each processor consists of an infinite iteration of a fixed list of steps that read input registers, perform local calculations, update processor variables, and write to output registers. We call each such iteration of a processor a *cycle*.

The processor-register model of a system does not specify algorithms using the guarded assignment notation of [5]. Instead, processors are non-terminating automata that continuously take steps in any execution, where an execution is a fair interleaving of processor steps [8]. This means that the direct association of "privilege" with "enabled guard" in [5] does not apply to the processor-register model. Instead, we use the idea of a token, introduced by [4], since each processor is always enabled to take steps.

We do not present a formal statement of what it means to adapt DijAlg to other models in this extended abstract. Informally, an adaptation of DijAlg satisfies the following constraints for each processor: except for the program counter and temporary variables used to read communication registers or used for local calculations, there is one *counter* variable that represents the state of a machine of DijAlg; processor p_0 plays the role of the exceptional machine of DijAlg by incrementing its counter (modulo some value K) when the values it reads from its input registers represent its current counter value, and otherwise its counter value does not change; and processor p_i, $i \neq 0$, plays the role of an unexceptional machine of DijAlg, changing its counter only if it reads input registers that represent a value different from its current counter, and then p_i assigns its

counter to be equal to the representation from the input registers. A "token" is thus equivalent to the conditions that enable a processor to change its counter.

In subsequent sections we investigate adaptations of DijAlg using non-atomic registers. Observe that, since the nature of DijAlg's legitimate behavior is single-token circulation in the ring (mutual exclusion), it follows that transferring a token from one processor to the next is essentially atomic — the token only moves in one direction and cannot reverse course. Thus the challenge of using non-atomic registers is to simulate this atomic behavior. Section 5 weakens this atomicity for a model of registers that are only probabilistically correct.

3 Regular Registers

Before we introduce our results for the case of regular registers let us present "folklore" results concerning read/write registers. Here and for the remainder of the paper, we refer to processors by $p_1 \cdots p_n$ rather than $p_0 \cdots p_{n-1}$.

Read/Write Atomicity: It is known that $n-1$ labels are sufficient for the convergence of DijAlg assuming a central daemon, where n is the number of processors in the ring. We next prove that $n-2$ labels are not sufficient.

Lower Bound: Consider the case of $n-2$ states in a system of $n=5$ processors. Thus there are three possible processor states, which we label {0,1,2}. To prove impossibility we demonstrate a non-converging sequence of transitions (the key to constructing the sequence is to maintain all three types of labels in each system state, which violates the key assumption for the proof of convergence).

$$\{0,0,2,1,0\} \stackrel{p_1}{\rightarrow} \{1,0,2,1,0\} \stackrel{p_5}{\rightarrow} \{1,0,2,1,1\}$$

$$\stackrel{p_4}{\rightarrow} \{1,0,2,2,1\} \stackrel{p_3}{\rightarrow} \{1,0,0,2,1\} \stackrel{p_2}{\rightarrow} \{1,1,0,2,1\}$$

We now present a reduction (see [8]) of a ring with $2n$ processors that is activated by a central daemon to a ring with n processors that assumes read write atomicity. We conclude that at least $2n-1$ states are required.

Each processor p_j has an internal variable in which p_j stores the value p_j reads from p_{j-1}. Each read is a copy to an internal variable and each write is a copy of internal variable to a register. Thus, we have in fact a ring of $2n$ processors in a system with a central daemon. Hence, $2n-1$ states are required and are sufficient.

Regular Registers: We now turn to the design of an algorithm for the case of regular registers. Informally, a regular register has the property that a read operation concurrent with a write operation can return either the "old" or "new" value. More formally, to define a regular register r we need to define the possible values that a read operation from r returns. Let x^0 be the value of the last write operation to r that ends prior to the beginning of the read operation (let x^0 be the initial value of r if no such write exists).

A read operation from a regular register r that is not executed concurrently with a write operation to r returns x^0. A read operation from a regular register r that is executed concurrently with a write of a value x^1 returns either x^0 or x^1. Note that more generally, a read concurrent with a sequence of write operations of the values $x^1, x^2, \cdots, x^m$ to r could return any x^k, $0 \leq k \leq m$, however once a read returns x^k for $k > 0$, no subsequent read by the same reader will return x^j for $j < k - 1$ (for two successive read operations, there is at most one write operation concurrent with both).

s_1	s_2	s_3	
0	0	0	p_1 starts to write 1
1	0	0	p_1 still writing 1, and p_2 reads 1
1	1	0	p_1 still writing 1, and p_2 writes 1
1	1	0	p_1 still writing 1, p_2 writes 1, and p_3 reads 1
1	1	0	p_1 still writing 1, p_2 reads 0
1	0	1	p_2 writes 0, and p_3 writes 1
1	0	1	p_1 reads 1, p_2 reads 1, and p_3 reads 0

Fig. 1. Straightforward regular register implementation fails.

A naive implementation of DijAlg using regular registers may result in the execution presented in Figure 1. The figure shows states of a three processor system. The system starts in a safe configuration in which all the values (in the registers and the internal variables) are 0 and we have reached a configuration in which all the processors may simultaneously change a state.

To overcome the above difficulty we introduce a new value $\perp$ that is written before any change of a value of a register (the domain of register values thus includes all counter values and the new value $\perp$). The algorithm for the case of regular registers appears in Figure 2. In the figure, IR_i is the input register for p_i (thus IR_i is the output register of p_{i-1}). Variable x_i contains the counter from DijAlg, and variable t_i is introduced to emphasize the fine-grained atomicity of the model (one step reads a register, and the value it returns is tested in another step). If processor p_i reads a value $\perp$ from an input register, it ignores the value (see line 10 of the protocol).

A safe configuration is a configuration in which all the registers have the same value, say x, and every read operation that has already started will return x. For simplicity we assume there are $2n + 1$ states. Therefore, it is clear that a state is missing in the initial configuration, say the state y. Hence, when p_1 writes y, p_1 does not change its state before reading it from p_n. p_n can read y only when p_{n-1} has the state y. Any read operation of p_{n-1} that starts following the write operation that assigns y to p_{n-1} may return either $\perp$ or y, which is effectively y (see lines 3 to 6 and 10 to 13 of the code).

```
1  p_1:            do forever
2                      read t_1 := IR_1
3                      if t_1 = x_1 then
4                              x_1 := (x_1 + 1) mod (2n + 1)
5                              write IR_2 := ⊥
6                              write IR_2 := x_1
7                      else write IR_2 := x_1
8  p_i (i /= 1): do forever
9                      read t_i := IR_i
10                     if x_i /≠ t_i  ∧  t_i /≠ then
11                         x_i := t_i
12                         write IR_{i+1} := ⊥
13                         write IR_{i+1} := x_i
14                     else write IR_{i+1} := x_i
```

Fig. 2. DijAlg for Regular Registers.

4 Safe Registers

Safe registers have the weakest properties of any in Lamport's hierarchy. A read concurrent with a write to a safe register can return any value in the register's domain, even if the value being written is already equal to what the register contains. There are two cases to consider for the model of safe registers. If a processor is unable to read the register(s) that it writes, we can show that DijAlg cannot be implemented. We initially consider the model of a single link register for each processor under the restriction that a writer is unable to read its output registers, that is, the model of 1W1R registers.

The construction of a regular 1W1R boolean register from a safe boolean 1W1R register given in [16] is simple: the writer skips actually writing to the register if the new value is the same as the value already in the register. This is possible because the writer keeps an internal variable equal to the current value of the register. However, this technique is not self-stabilizing because the initial value of the writer's internal variable may not be correct.

Lemma 1. *DijAlg cannot be adapted by using only a single 1W1R safe register between p_i and p_{i+1}.*

Proof. Processor p_i ($i \neq 1$) that copies from the output register of p_{i-1} must continually rewrite its output register for p_{i+1} — otherwise there can be a deadlock where the value written by p_i is different from the value p_i reads from p_{i-1}. Similarly, p_1 must repeatedly write, otherwise there can be a deadlock where the value written by p_1 is the same as the value p_1 reads from p_{n-1}. Therefore, processors continually write into their output registers. Since all processors repeatedly write their output registers, we can construct an execution where reads are concurrent with writes and obtain arbitrary values. This construction can be used to show that the protocol does not converge (and also that it is not stable).

Multiple Fields Safe Register: Lemma 1 can be generalized, and we sketch the argument here. We consider the case of multiple safe registers per processor, but where processors cannot read the registers they write. Suppose each processor p_i has m safe registers to write, which p_{i+1} reads, and also p_i reads m safe registers written by p_{i-1}. If a protocol allows a state in which a processor does not write any of its registers so long as its state does not change, then we may construct a deadlock because the local state of the processor differs from the encoding of values contained in its output registers. Therefore, in any implementation of the protocol, we can construct an execution fragment so that any chosen processor p_i writes at least some of its registers t times, for arbitrary $t > 0$, and during the same execution fragment, p_{i-1} takes no steps. Moreover, if p_i does not write to all m registers, then the registers it does not write can have arbitrary values inherited from the initial state. Therefore, p_{i+1} can read any value from p_i, since at each step of p_{i+1} reading one of the m registers written by p_i, we can construct an execution in which p_i is concurrently writing to the same safe register. Because p_{i+1} can read any value, it is possible that for $i \not= n$ that p_{i+1} reads a value equal to its own current value, which for DijAlg, means that p_{i+1} will maintain its current value rather than changing it; for the case $i = n$, there is an execution where each time p_1 reads its input registers, the value read differs from its own value, and again p_1 makes no change to its current value. These situations can repeat indefinitely with no processor entering the critical section.

Composite Safe Register: Next we sketch a solution in which fields of the registers can be written and the entire register is read *at once*. We call such a register *composite safe register*. A read from a composite safe register may return an arbitrary value for at most one of the register fields, a field in which a write is executed concurrently to the read[1]. We note that there is a natural extension of our algorithm in which at most k fields of a register may return an arbitrary value.

Each bit of the label value is stored in three 1-bit safe registers (three fields). This will ensure that a read during a refresh operation will return the value of the register. Assume that the value 101 is stored in nine 1-bit safe registers as 111000111. Assume further that a processor refreshes the value written in these registers each time writing in one of the 1-bit safe registers. A read operation returns the value of the entire composite safe register in which at most one bit is wrong. The Hamming distance ensures that the original value of the label bit can be determined.

To allow a value change we add a three bits guard value. Hence, the composite safe register has three bits that function as a guard value and $3 \times 2(n + 1)$ bits for the label.

A processor p_i, $i \not= 1$, that reads a new value from p_{i-1} first sets the guard value to 0 (writing 000 in the guard bits), and then changes the value of the label. p_i writes 111 to its guard bits once p_i finishes updating the label.

[1] This assumption reflects reality in system in which a read operation is much faster than a write operation.

A processor p_i that reads a guard value 0 does not use the value read. When p_i reads a guard value 1 it examines the value it read.

The correctness proof starts in convincing ourselves that after the first time a processor p_i refreshes (or writes a new value in) its register any read operation from its register (that returns a value) results in the last value written to this register. p_1 eventually writes a non existing label, this label cleans the system. More details are omitted from this extended abstract.

Safe Registers with Reads instead of Refreshes: Given the above impossibility results, we examine settings where a processor can read the contents of the registers in which it writes. Consider $2n+1$ single bit, safe, 1W2R registers rather than a single register per processor. Each processor maintains a counter with domain $[1, 2n+1]$ for DijAlg. Unary encoding represents this counter: for a counter value k, the proper encoding is to write all registers 0 except for the register with index k, which has value 1. Figure 3 gives an adaptation of DijAlg for this 1W2R model. Lines 2–7 are concerned with correcting the values of registers to agree with internal counter values, but such writing is only done where needed. This correction is for the case of incorrect values in illegitimate initial configurations. However, for convenience, we let lines 2–7 also do the work of transmitting a new counter value (since lines 2–7 are repeated after lines 8–13 in the execution of each processor).

```
1   p1:            do forever
2                      do k := 1 to 2n + 1
3                          read t1 := IR2[k]
4                          if k /= x1   ∧   t1 = 1
5                              write IR2[k] := 0
6                          if k = x1    ∧   t1 = 0
7                              write IR2[k] := 1
8                      do s := 0, j := 0, k := 1 to 2n + 1
9                          read t1 := IR1[k]
10                         if t1 = 1
11                             s := s + 1; j := k
12                     if s = 1    ∧   j = x1
13                         x1 := 1 + x1 mod (2n + 1)

1   pi (i /= 1): do forever
2-11                   (similar to code for p1)
12                     if s = 1    ∧   j /= xi
13                         xi := j
```

Fig. 3. DijAlg adaptation for Safe Registers.

A legitimate configuration for this protocol is that each register vector represents the processor's last counter value (it differs only when a processor updates its counter) and counters correspond to DijAlg.

Lemma 2. *Figure 3 is a self-stabilizing adaptation of DijAlg.*

Proof. There are two proof obligations, stability (closure) from legitimate configurations and convergence from arbitrary configurations to legitimate ones.

Closure. It is straightforward to verify that in any processor cycle from a legitimate configuration, a processor writes to at most two registers as it changes the counter value. Thus when the neighbor reads these registers, at most two reads can have incorrect values due to concurrent writing. If both have correct values, the token passes correctly (a subsequent read by the process can still obtain an incorrect value, but only by getting 0 for all reads, which causes no harm). If both have incorrect values, then the reader observes no change in counter values. If just one returns an incorrect value, then the reader observes parity of zero, which is harmless. This reasoning shows that the protocol is stable.

Convergence. The remaining task is to verify that the protocol guarantees to reach a legitimate configuration in any execution. Suppose all processors have completed at least one cycle of statements 1-13. In the subsequent execution, a processor only writes a register if that register requires change to agree with the processor's counter. Note that by standard arguments, no deadlock is possible in this system and that p_1 increments its counter infinitely many times in an execution. It is still possible that one processor can read more than two incorrect values due to concurrent writes (consider an initial state with many counter values; as these values are propagated to some p_i, it could be that p_{i+1} happens to read many registers concurrent with p_i writing to them). Since the counter range is $[1, 2n + 1]$ and there are n processors, it follows that at least one counter value t is not present in the system. By the arguments given for the proof of closure, no processor incorrectly reads input registers to get the value t in such a configuration. Because p_1 increments x_1 infinitely, we can suppose $x_1 = t$ but no other processor or register encoding equals t, and by standard arguments (and the propagation of values observed in the proof of closure), a legitimate configuration eventually is reached.

A standard construction of 1W2R registers from 1W1R registers is just to allocate an added new output register and arrange for the writer to ensure the outputs are duplicates. The reader may wonder if such a construction contradicts Lemma 1 and its generalization. There are two interesting cases to examine. First, we reject any protocol where p_{i+1} has a new output register that p_i reads, since such an adaptation of DijAlg would violate the unidirectional model constraint. Second, adding a new 1W1R register written by p_i and read only by p_i would be equivalent to an internal variable, which we have already considered in the proof of Lemma 1. Thus, under our constraints on adaptation of DijAlg, the models of 1W1R and 1W2R registers differ in capability.

The protocol of Figure 3 uses an expensive encoding of counter values, requiring $2n+1$ separate registers. The argument for closure shows that changing a counter and transmitting it is effectively an atomic transfer of the value — once the new value is observed, then any subsequent read of the registers either returns the new value or some invalid value (where the sum of bits does not equal 1), which is ignored. Note that this technique *is not* an implementation of an atomic register from safe registers; it is specific to the implementation of DijAlg.

Can we do better than using $2n+1$ registers? The following protocol uses the Gray code representation [11] of the counter, plus a extra bit for parity. The number of registers per processor is $m+1$ where $m = \lceil \lg(2n+1) \rceil$.

```
1   p1:          do forever
2                    do k := 1 to m
3                         read t1 := IR2
4                         if t1 /= Graycode(x1)[k]
5                              write IR2[k] := Graycode(x1)[k]
6                    read t1 := IR2[m + 1]
7                    if t1 /= parity(Graycode(x1))
8                         write IR2[m + 1] := parity(Graycode(x1))
9                    do k := 1 to m
10                        read g1[k] := IR1[k]
11                   read t1 := IR1[m + 1]
12                   if t1 = parity(g1)   ∧   Graycode(x1) = g1
13                        x1 := (x1 + 1) mod (2n + 1)

1   pi (i /= 1): do forever
2-11                 (similar to code for p1)
12                   if ti = parity(gi)   ∧   Graycode(xi) /= gi
13                        xi := gi
```

Fig. 4. DijAlg using Gray Code.

Lemma 3. *Figure 4 is a self-stabilizing adaptation of DijAlg.*

Proof. The closure argument is the same as given in the proof of Lemma 2, inspecting each of the four cases of reading overlapping with writing of the two bits that change when a processor changes its counter and writes the one new Gray code bit and the parity bit. In each case, the neighbor processor either reads the old value, or ignores the values it reads (because parity is incorrect), or obtains the new counter value. The change from old to new counter value is essentially atomic.

Proof of convergence requires new arguments. Consider some configuration of an execution prior to which each processor has completed at least two cycles of statements 1-13 in Figure 4, so that output registers agree with counter values

(unless the processor has read a new value and updated its counter). Observe that thereafter, if processor p_i successively reads two different Gray code values from its input registers, each with correct parity, then p_{i-1} concurrently wrote at least once to its output registers. Moreover, if p_i successively reads k different Gray code values with correct parity, then p_{i-1} wrote at least $k-1$ times a new counter value and read at least $k-1$ times from its own input registers (in turn, written by p_{i-2}). A consequence of these observations is that if p_1 successively reads k different counter values with correct parity, then p_{n-k} wrote at least one new counter value in the same period. In particular, if p_1 successively reads $n+2$ different counter values, then we may assert that p_2 read p_1's output registers and wrote a new counter in the same period. By the standard argument refuting deadlock, processor p_1 increments its counter infinitely often in any execution. Therefore we can consider an execution suffix starting with $x_1 = 0$. In the reflected Gray code [11], the high-order bit starting from $x_1 = 0$ does not change until the counter has incremented 2^m times. Therefore, until p_1 has incremented x_1 at least 2^m times, any read by p_2 obtains a value with zero in the high-order bit. The observations above imply that, before x_1 changes at the high-order bit, each processor has copied some counter value that originated via p_1 — such counter values may be inaccurate due to reads overlapping writes or more than one write (bit change) for one scan of a set of registers, however the value for the high-order bit stabilizes to zero in this execution fragment. In a configuration where no counter or register set has 1 in the high-order bit, the event of p_1 changing the high-order bit creates a unique occurrence of 1 in that position. Since p_1 does not again change its counter until observing the same value from p_n, convergence is guaranteed.

5 Randomized State Reads and Weak Stabilization

Consider a system with a *probabilistic central daemon*, in any given configuration the daemon activates each of the processors with equal probability. A system is *weakly stabilizing* if, in any execution, the probability that the system remains in any set of illegitimate configurations is zero. This definition implies that a weakly stabilizing system has the property that its state is infinitely often legitimate. In addition, one can sum up the probabilities for being in a legitimate state and use this value to compare algorithms.

To apply the definition of weak stabilization, we model register behavior probabilistically: a processor that makes a transition may "read" an incorrect value and therefore make an errant transition. When the daemon selects a processor, we consider the transition by that processor to be one full cycle of reading its input and writing its output register atomically. We use Markov chains to analyze the stationary probabilities, i.e., to determine the probability that the system will be in a legitimate configuration. See [18] for a description of Markov chains.

We continue describing our approach using a system of three processors and two states. A read of a neighboring state returns with probability p the correct value, and with probability $1-p$ the complement of the correct value. There

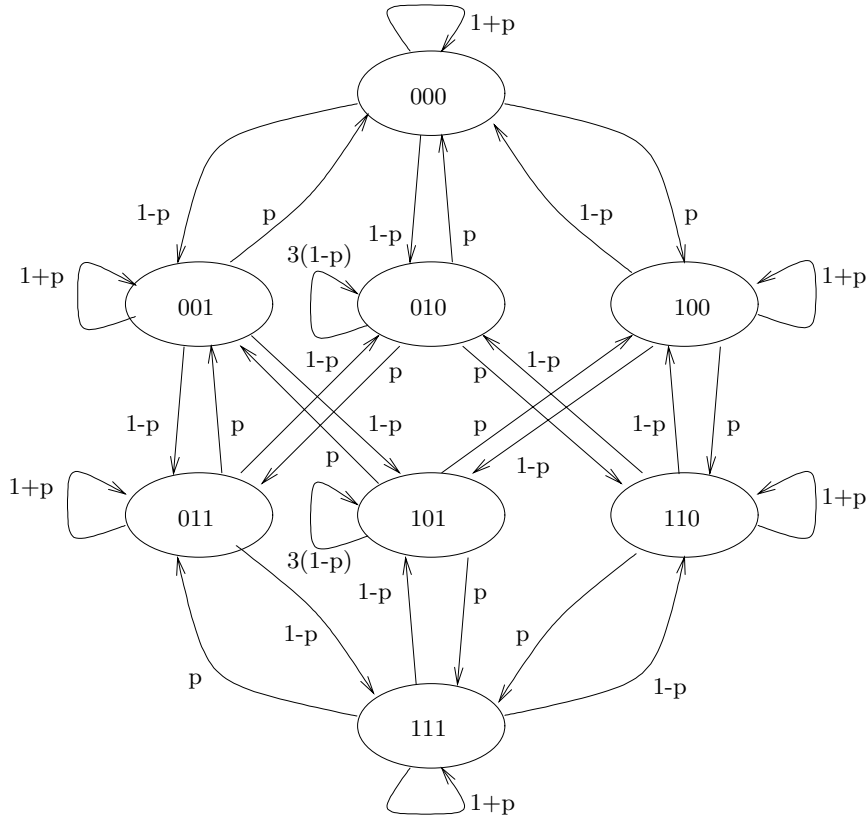


Fig. 5. Transition Probabilities (factorized by 3).

are eight states for this system; Fig. 5 shows the transition diagram for this system. We explain the figure by considering the configuration 111. With probability 1/3, the daemon activates p_1, and p_1 reads with probability $1 - p$ the incorrect value 0, which is ignored by DijAlg; thus the joint probability for this transition from 111 to 111 is $(1 - p)/3$. Two other transitions from 111 to 111 are also possible: either p_1 or p_2 could read correctly with probability p, thus the probability for each of these transitions is $p/3$. The sum of all cases for this transition is therefore $(1-p)/3+2p/3 = (1+p)/3$. To simplify the presentation, all transition arcs have an implicit factor of 3, so the 111 to 111 arc in Fig. 5 is labeled $(1 + p)$. Similar case analysis derives probabilities for the other arcs shown in the figure. Each configuration has four outgoing arrows, one arrow for each state change of a processor, and one for staying in the same state.

We now choose specific values for p and compute powers of the transition probability matrix $\mathcal{P}$, such that the matrix in power i and $i + 1$ are equal ($\mathcal{P}^i = \mathcal{P}^{i+1}$). Each entry $\mathcal{P}_{jk}$ of the 8×8 transition probability matrix P con-

tains the probability for a transition from configuration j to configuration k. The equilibrium probability of being in a legal configuration (i.e., not in the configurations 010 or 101) is then derived from $\mathcal{P}^i$. The following table shows different values for p (1, 3/4, 1/2, 1/4) and the corresponding equilibrium vector $\mathcal{E}^T$; two figures display the transition matrix $\mathcal{P}$ for the cases of $p = 1$ and $p = 3/4$.

Matrix	p	Equilibrium Vector
Fig 6	1	[1/6,1/6,0,1/6,1/6,0,1/6,1/6]
Fig 7	3/4	[7/48,7/48,1/16,7/48,7/48,1/16,7/48,7/48]
	1/2	[1/8,1/8,1/8,1/8,1/8,1/8,1/8,1/8]
	1/4	[1/12,1/12,1/4,1/12,1/12,1/4,1/12,1/12]

The vectors show that the equilibrium probability for illegitimate configurations is zero for the deterministic case, then increasing as p reduces. Clearly, we can investigate the behavior of other systems with a range of probabilities, using the same approach. The results can assist us in comparing different system designs.

Lemma 4. *The adaptation of DijAlg is weakly stabilizing when register reads are correct with probability $p > 0$.*

2/3	0	0	0	1/3	0	0	0
1/3	2/3	0	0	0	0	0	0
1/3	0	0	1/3	0	0	1/3	0
0	1/3	0	2/3	0	0	0	0
0	0	0	0	2/3	0	1/3	0
0	1/3	0	0	1/3	0	0	1/3
0	0	0	0	0	0	2/3	1/3
0	0	0	1/3	0	0	0	2/3

Fig. 6. Transition Matrix for $p = 1$.

6 Conclusion

This paper presents a number of results related to weakening the model of atomic link registers for the unidirectional ring model associated with DijAlg. Our results are both positive (the constructions for regular and safe registers) and negative (the impossibility for 1W1R safe registers). Some similar experiences with the difficulties of self-stabilizing register constructions are reported in [14], however the problem adapting DijAlg has additional constraints and also advantages: the constraint of unidirectional communication rules out certain techniques, but the

7/4	1/4	1/4	0	3/4	0	0	0
3/4	7/4	0	1/4	0	1/4	0	0
3/4	0	3/4	3/4	0	0	3/4	0
0	3/4	1/4	7/4	0	0	0	1/4
1/4	0	0	0	7/4	1/4	3/4	0
0	3/4	0	0	3/4	3/4	0	3/4
0	0	1/4	0	1/4	0	7/4	3/4
0	0	0	3/4	0	1/4	1/4	7/4

Fig. 7. Transition Matrix for $p = 3/4$ factorized by 3.

ring topology does provide sufficient feedback (eventually information flows from p_i back to p_{i-1}) to make constructions possible.

The introduction of probabilistic registers motivates a weakened form of self-stabilization. (Another model of probabilistically correct registers appears in [19].) Our model of probabilistically correct registers could be overly pessimistic: it could be interesting to refine the model so that reads are always correct if there is no concurrent write, but only correct with probability p when a read overlaps a write. The weakly stabilizing protocol does not guarantee closure, since there is always the possibility of a read returning an incorrect value. The usefulness of this kind of weakening of stability is typically associated with control theory, e.g., in [22] the goal is to find systems such that all trajectories visit the "good" states infinitely often. Though rarely formalized in the literature of self-stabilization, this reasoning is implicit in many papers: each new fault perturbs the system, and provided the execution is fault-free for long enough, the system returns to a legitimate state. Our probabilistic model quantifies both fault and scheduler probabilities to characterize executions with a Markov model.

Acknowledgment

We are grateful to the reviewers, whose comments and suggestions helped us to improve the presentation of this paper.

References

1. A. Arora and M.G. Gouda, "Closure and convergence: a foundation of fault-tolerant computing," *IEEE Transactions on Software Engineering,* vol. 19(11), 1993, pp. 1015–1027.
2. A. Arora and S.S. Kulkarni, "Component based design of multitolerance," *IEEE Transactions on Software Engineering,* vol. 24(1), 1998, pp. 63–78.
3. U. Abraham, S. Dolev, T. Herman, and I. Koll, "Self-stabilizing l-exclusion," *Proceedings of third workshop on self-stabilizing systems,* pp. 48–63, 1997.
4. G.M. Brown, M.G. Gouda, and C.L. Wu, "Token systems that self-stabilize," *IEEE Transactions on Computers,* vol. 38, 1989, pp. 845–852.

5. E.W. Dijkstra, EWD391 Self-stabilization in spite of distributed control. In *Selected Writings on Computing: A Personal Perspective,* pages 41–46, Springer-Verlag, 1982. EWD391's original date is 1973.
6. E.W. Dijkstra, "Self stabilizing systems in spite of distributed control," *Communication of the ACM*, vol. 17, 1974, pp. 643–644.
7. E. W. Dijkstra, "A belated proof of self-stabilization," *Distributed Computing*, 1:5–6, 1986.
8. S. Dolev, *Self-stabilization*, MIT Press.
9. S. Dolev and J.L. Welch, "Wait-Free Clock Synchronization," *Proc. of the 12th Annual ACM Symp. on Principles of Distributed Computing*, pp. 97-108, 1993.
10. S. Dolev and J.L. Welch, "Self-stabilizing clock synchronization in the presence of byzantine faults," *Proceedings of the Second Workshop on Self-Stabilizing Systems,* pp. 9.1-9.12, 1995.
11. M. Gardner, "The Binary Gray Code," in *Knotted Doughnuts,* Freeman & Company, New York, 1986.
12. A.S. Gopal and K.J. Perry, "Unifying self-stabilization and fault-tolerance," *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing (PODC'93)*, pp. 195–206, 1993.
13. M.G. Gouda, R.R. Howell, and L.E. Rosier, "The instability of self-stabilization," *Acta Informatica,* vol. 27(8), 1990, pp. 697–724.
14. J.H. Hoepman, M. Papatriantafilou, and P. Tsigas, "Self-Stabilization in Wait-Free Shared Memory Objects," *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG'95)*, Lecture Notes in Computer Science Vol. 972, pages 273-287, Springer-Verlag, September 1995.
15. L. Lamport, "Concurrent reading and writing," *Communication of the ACM*, vol. 20(11), 1977, pp. 806–811.
16. L. Lamport, "On interprocess communication, parts 1 and 2," *Distributed Computing*, vol. 1(1), 1986, pp. 77-101.
17. L. Lamport, "The mutual exclusion problem. Part II: Statement and solutions," *Journal of the ACM*, vol. 33(2), 1986, pp. 327–348.
18. D.G. Luenberger, *Introduction to Dynamic Systems, Theory, Models & Applications*, John Wiley & Sons.
19. H. Lee and J.L. Welch, "Applications of probabilistic quorums to iterative algorithms," in *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 2001)*, pp. 21-28, 2001.
20. E.A. Lycklama and V. Hadzilacos, "A First-Come-First-Served Mutual-Exclusion Algorithm with Small Communication Variables," *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, 1991, pp. 558-576.
21. J. Misra, "Axioms for memory access in asynchronous hardware systems," *ACM Transactions on Programming Languages and Systems*, vol. 8(1), 1986, pp. 142-153.
22. C.M. Özveren, A. S. Willsky, and P.J. Antsaklis, "Stability and stabilizability of discrete event dynamic systems," *Journal of the ACM*, Vol. 38, No. 3, 1991, pp. 730-752.
23. G. Varghese, "Self-stabilization by counter flushing," *SIAM Journal on Computing,* vol. 30(2), 2000, pp. 486–510.

Communication Adaptive Self-Stabilizing Group Membership Service

(Extended Abstract)

Shlomi Dolev[1⋆] and Elad Schiller[1]

Department of Computer Science, Ben-Gurion University — Israel
{dolev,schiller}@cs.bgu.ac.il

Abstract. This paper presents the first algorithm for implementing self-stabilizing group communication services in an asynchronous system. Our algorithm converges rapidly to a legal behavior and is *communication adaptive*. Namely, the communication volume is high when the system recovers from the occurrence of faults and is low once a legal state is reached. The communication adaptability is achieved by a new technique that combines transient fault detectors.

1 Introduction

Group communication services are becoming widely accepted as useful building blocks for the construction of fault-tolerant distributed systems and communication networks. Designing robust distributed systems is one of the most important goals in the research of distributed computing. One way to simplify the design and programming process of a distributed system is to design a useful set of programming high-level primitives that forms a robust set of tools. A group communication system enables processors that share a collective interest, to identify themselves as a single logical communication endpoint. Each such endpoint is called a *group*, and is uniquely identified. A processor may become a *member* of or departure a group, by issuing a join/leave request to the *group membership service*. The membership service reports membership changes to the members, by delivering *view*s. A *view* of a group includes a set of members and a unique identifier. A processor may send a message to a group, using a group *multicast service*.

A very important property in the implementation of the primitives of group communication services is its fault tolerance and robustness. It is assumed that processors leave and join the group either voluntarily or due to crashes or recoveries. The distributed algorithms that implement these services assume a particular set of possible failures, such as crash failures, link failures or messages loss. The implementing algorithms should provide the specified services in spite of the occurrence of these failures. The correctness of the implementing algorithms is proved by assuming a predefined initial state and considering every possible execution that involves the assumed possible set of failures.

⋆ Dolev's work was supported by BGU seed grant.

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 82–97, 2001.

The abstraction that limits the possible set of failures is convenient for establishing correctness of the algorithm, but it can at the same time be too restrictive. Group communication service is a long-lived on-line task and hence it is very hard to predict in advance the exact set of faults that may occur. Therefore, it may be the case that due to the occurrence of an unexpected fault, the system reaches a state that is not attainable from the initial state by the predefined transitions (including the predefined fault occurrences). Self-stabilizing systems [11, 14] can be started in any arbitrary state and exhibit the desired behavior after a convergence period. Thus, self-stabilizing group communication services can automatically recover following the occurrence of such unexpected faults.

Related Work: The Isis system [8] is the first implementation of group communication services that triggered researchers and developers to further examine such services. Cristian [9] formalized a definition of group communication for synchronous systems. Group communication services were implemented with different guarantees for reliability and message delivery order. For example, Isis [8], Transis [17], Totem [23], Newtop [19], Relacs [7], Horus [25] and Ensemble [26]. None of the above implementation is self-stabilizing. A specification that guarantees performance once the system stabilizes to satisfy certain properties is presented in [20]. This is a consequence of existing impossibility results for requirements that hold in all possible executions e.g., [10]. Still it is assumed in [20] that the system is started in a certain global state, and the transitions are from a predefined set of transitions — thus the specification and algorithm presented in [20] are not designed for self-stabilizing systems.

A different approach (part of which is randomized) is used in [27]. Every processor periodically transmits a list of the processors that it can directly communicate with. A processor is consider "up" and connected as long as it can successfully transmit a "fresh" time-stamp; otherwise it will be eventually discarded from the system. The algorithm presented in [27] may be a base for a self-stabilizing algorithm, if for example, each processor has access to a local pulse generator, such that the maximum drift between the pulse generators is negligible.

Congress [2] is an elegant protocol for registration of membership information at (hierarchically organized) servers. Hierarchy of servers improves scalability. Users send a message to servers with join or leave requests. The servers maintain the membership information. The design fits wide area network using virtual links to define neighboring relation.

Moshe [21] is a group membership service implementation, that considers an abstract network service (such as Congress [2]). The network service monitors the up and connected members of every group and delivers multicast messages to the members of a group. The common cases of membership changes (joins/leaves) are considered in order to achieve scalability. The group membership algorithm of Moshe uses unbounded counters.

A self-stabilizing group membership service for synchronous systems is considered in [3]. A common periodic signal initiates a broadcast of local topology of every processor. Every processor uses the local topologies in order to compute

the connected component to which it belongs. Unbounded signal numbers are used and changes in the group are discovered following a common signal.
Our Contribution: this paper presents the first algorithm for implementing a self-stabilizing group membership service in asynchronous systems. We assume that processors eventually know the set of non-crashed processors with which they can communicate directly. We show that once every processor knows the correct set, the membership task is achieved within the order of the diameter of the communication graph. Moreover, the activity of the processors is according to the required group membership service. Our algorithm converges rapidly to a legal behavior and is *communication adaptive*. Namely, the communication volume is high when the system recovers from the occurrence of faults and is low once a legal state is reached. The communication adaptability is achieved by a new technique that combines transient fault detectors. Furthermore, randomized techniques can be used to dramatically reduce the communication complexity of the deterministic transient failure detector.

Our group membership service can be extended to implement different levels of broadcast services, such as single-source FIFO; totally ordered; and causally ordered.

The rest of the paper is organized as follows. The system settings appear in Section 2. Our algorithms for implementing a self-stabilizing group membership service appear in Section 3. Concluding remarks are in Section 4. The proofs are omitted from this extended abstract, more details can be found in [18].

2 The System

The *distributed system* consists of a set of P communicating entities. We call each entity a *processor*, and assume that $1 \leq |P| = n \leq N$, where N is an upper bound on the number of processors. The processors may represent a network of real physical CPU, or correspond to an abstract entity like a process or thread in a timesharing system. Processors are connected by communication links through which they communicate by exchanging messages. $neighbors_i$ is the set of processors that processor p_i can directly communicate with. The communication link may represent a (real physical) communication channel device attached to the processor, a virtual link, or any inter-process communication facility (e.g., UDP, or TCP connections).

It is convenient to represent a distributed system by a communication graph $G = (V, E)$, where each node represents a processor and each edge represents a communication link. Let $p_i, p_j \in P$, $p_j \in neighbors_i$ iff $(p_i, p_j) \in E$.

The system is asynchronous. We assume however that processors eventually identify the crashed/non-crashed status of their attached links and neighbors. We sometime use the term *time-out* in the code of the processors for a repeated action of the processors. In fact, a zero time-out period will result in the desired behavior as well. The time-out period may only reduce the number of messages sent when processors have access to a time device.

A state machine models each processor. The communication links are modeled by two anti-directed FIFO queues. We use a (randomized) self-stabilizing

data-link algorithm on every link [1]. The existence of the self-stabilizing data-link algorithm ensures that when a message is sent it arrives to its destination before the next message is sent. Thus, input communication buffers or communication registers (when buffers contain at most one message and when the content of an arriving message replaces the previous content of the buffer) can be assumed whenever it is convenient instead of message passing.

The system configuration is a vector of the states of the processors and the values of the queues (the messages in the queues).

A *communication operation* is an operation in which a message is sent or a message is received. We also allow a processor to send the same message to every one of its neighbors in a single communication operation. A step of a processor consists of internal computations that are followed by a single communication operation. A system *execution* is an alternating sequence of configurations and (atomic) steps.

Processors may crash and recover during the execution. The neighbors of a crashed processor eventually identify the fact that it is crashed.

The program of a processor used here consists of a do-forever loop that includes communication step with every neighboring processor. Let R be an execution and let A be a connected component of the system such that no processor in A is crashed during R. The first *asynchronous cycle* of A in R is the minimal prefix of R such that each processor p_i in A communicates with every of its neighbors: At least one message m_j is sent by p_i to every neighbor p_j, such that p_j receives m_j during the asynchronous cycle.

The number of messages sent over a particular communication link during an asynchronous cycle is a function of the number of loop iterations the attached processors execute during this asynchronous cycle (note that a processor may execute any number of iterations before another processor completes a single loop iteration). Thus we consider a special execution to measure the communication complexity of an algorithm. A *very fair execution* is an execution in which every processor executes exactly a single iteration of its do-forever loop in every asynchronous cycle. The *communication complexity* is the total number of bits communicated over the communication links in a single asynchronous cycle of a *very fair execution*.

The set of *legal executions* includes all the executions that exhibit the desired behavior (input output relation) of the system for a task τ. For example, if τ is the mutual exclusion task, then at most one processor is executing the critical section in any configuration of a legal execution.

A *safe configuration* of the system is a configuration from which only legal executions, with respect to τ, start.

In this paper, the requirements are related to the *eventual* behavior of the system when the execution fulfills certain properties (unlike the requirements discussed in [10] see also [24]). We require that a self-stabilizing algorithm for group communication service will reach a safe configuration within a certain number of asynchronous cycles in any execution (that starts in an arbitrary con-

figuration) such that each processor p_i has a *fixed set of non crashed neighbors* during the execution[1].

We allow simultaneous existence of several groups. We do not consider however, interaction between the groups. Therefore, we choose a specific group g for describing the membership service. A boolean variable $member_i$ (logically) represents the intention of p_i to be included in g. A partition of the network may cause a "partition" of g as well. Therefore, we associate the set of *legal executions* for the group membership task, with the processors of a (fixed) connected component A, and include execution R such that the following properties hold:

1. If the value of $member_i = true$ ($member_i = false$) is fixed during R then there exists a suffix of R, in which p_i appears (does not appear, respectively) in all the views of group g in the connected component A.
2. If the value of $member_i$ of every processor p_i of group g in the connected component A is fixed during R then there exists a suffix, in which all the views of group g in connected component A are identical, the views have the same list of members and the *same view identifier*.

We note that the length of the prefix of R before the suffix mentioned in the above requirement is achieved by our algorithms is $\Theta(d)$ (which is the fastest possible).

The communication of a self-stabilizing algorithm is *adaptive* if the maximal communication complexity after reaching a safe configuration is smaller than the maximal communication complexity before reaching a safe configuration.

3 Self-Stabilizing Group Membership Service

In this section we present the first communication adaptive self-stabilizing algorithm for the membership service. Roughly speaking, a spanning tree of the system is constructed. This tree is used to execute the membership management tasks. The root of the tree is responsible for the management of the membership requests, and establishing new views. Several transient fault detectors monitor the consistency of the tree and the membership information. The transient fault detectors give fast indication on the occurrence of transient faults. Once a fault is detected the system changes state to a safe configuration executing a *propagation of information with feedback* (*PIF*) procedure [28,13], for several times (choosing random identifiers for these executions to ensure eventual stabilization).

The update algorithm informs each processor with the nodes in its connected component. The update algorithm stabilizes fast, as it takes $\Theta(d)$ asynchronous cycles before reaching a safe configuration. We use d to denote the actual diameter of the connected component. Unfortunately, the communication complexity of the update algorithm is $O(|E|n \log n)$ before *and after* a safe configuration is reached. In this section we present an algorithm that reduces the communication complexity to $O(|E| \log n + n^2 \log n) = O(n^2 \log n)$ once a safe configuration is reached.

[1] We note that we do not consider the time required to identify the status of the links and neighbors.

A transient fault detector is composed with the update algorithm to achieve the communication adaptability property (see [4,6,5] for definitions of transient fault detectors). The transient fault detector signals every processor whether or not it needs to activate the update algorithm. Our transient fault detector itself is obtained by using a new technique for composing transient fault detectors.

Roughly speaking, whenever a processor detects, by use of the transient fault detector, that the update algorithm is not in a safe configuration, the processor signals the processors in the system to start the activity of the update. The processor stops signaling the other processors to operate the update algorithm when it receives an indication that a safe configuration is reached.

3.1 Self-Stabilizing Update

We use the self-stabilizing update algorithm of [12,15]. We now sketch the main ideas used by the update algorithm. We start with the data structure used by a processor. Each processor has a list of no more than N tuples $\langle id, dis, parent\rangle$. When the update algorithm stabilizes it holds that the list of a processor p_i contains n tuples, exactly one tuple $\langle j, dis, k\rangle$ for each processor p_j that is in the same connected component with p_i. The value of dis is the number of edges in a shortest path from p_i to p_j and p_k is a neighbor of p_i that is in a shortest path to p_j. Thus, when the algorithm stabilizes every processor knows the identities of the other processors in its connected component.

The processors that execute the update algorithm repeatedly receive all the tuples from the tables of their neighbors and use the value received to calculate a new table (note that the current table is not used in calculating the new table). Every time a processor p_i finishes receiving the tuples of its neighbors it acts as follows: Let $\mathcal{TU}_i$ be the set of all tuples that a processor p_i reads from its neighbors. p_i adds 1 to the dis field of every tuple in $\mathcal{TU}_i$. p_i adds a tuple $\langle i, 0, nil\rangle$ to $\mathcal{TU}_i$. If there are several tuples with the same id in the resulting $\mathcal{TU}_i$ then p_i removes every such tuple except a single tuple among these tuple, a tuple with the minimal dis value. Finally, p_i removes every tuple $\langle id, dis, parent\rangle$ such that there exists a positive $z < dis$ and there is no tuple with $dis = z$ in $\mathcal{TU}_i$. The resulting set in $\mathcal{TU}_i$ is the new table of p_i.

3.2 Transient Fault Detectors for Reducing Communication Overhead

The communication complexity of the update algorithm is $O(|E|n \log n)$. Note that a naive approach for designing a transient fault detector is to repeatedly send $\mathcal{TU}_i$ to every neighbor. A fault will be detected whenever there should be a change in the value of $\mathcal{TU}_i$ (according to the update algorithm) when a message with $\mathcal{TU}_j$ arrives from a neighboring processor p_j. This approach results in communication complexity that is identical to the communication complexity of the update algorithm.

In this section we present a fault detector that reduces the communication complexity of our algorithm when the algorithm stabilizes (reaches a safe configuration). The communication complexity of the algorithm when a fault detector is used is $O(n^2 \log n)$.

The update algorithm informs each processor with the nodes in its connected component. The task of the transient fault detector is to detect a fault whenever there exists at least one processor that does not know the set of processors in its connected component.

We present a new scheme for combining fault detectors that results in low communication complexity. In order to reduce the communication complexity, we combine two transient fault detectors. The first one communicates short messages over all the links of the system and ensures that there is a marked rooted spanning tree. The short messages consist of the identifier of the common leader and the distance of the processor from this leader. The second transient fault detector assumes the existence of a spanning tree and communicates larger messages over the links of this tree. In fact, these messages consist of the description of the rooted spanning tree.

Transient Fault Detector for the Existence of a Tree Rooted at a Leader: The code for the first part of the transient fault detector appears in Figure 1. In the code we use the input $\langle leader_i, dis_i, parent_i \rangle$ which is defined by the output of the update algorithm. Let $\langle l, d, p \rangle$ be the tuple in $\mathcal{TU}_i$, such that l is the maximal value among the values of the *leader* variables in $\mathcal{TU}_i$. The value of $\langle leader_i, dis_i, parent_i \rangle$ is assigned by the values of $\langle l, d, p \rangle$. A change in the value of $\langle leader_i, dis_i, parent_i \rangle$ as well as in the $neighbors_i$ set triggers fault detection.

Lines 1 and 1a of the code ensure that the information for detection of a fault is sent from every processor to its neighbors once every timeout period. Line 1b ensures that the processor for which $leader_i = i$ has the value 0 in its *dis* variable and the value *nil* in the $parent_i$ variable. Line 2a ensures that all the processors have the same value in their leader variable and the distance of the parent of each (non-leader) processor p_i is one less than the distance of p_i from the leader.

Input: $\langle leader_i, dis_i, parent_i \rangle$ *(* updated by lower level *)*

1. **Upon timeout:**
   (a) for each $j \in neighbors_i$ send $\langle leader_i, dis_i, parent_i \rangle$.
   (b) if $leader_i = i$ and ($dis_i \neq 0$ or $parent_i \neq nil$) then fault is detected.
2. **Upon receiving** $\langle l, d, p \rangle$ **from** p_j
   (a) if ($l \neq leader_i$) or (($j = parent_i$) and ($dis_i \neq d + 1$)) then fault detected.

Fig. 1. Transient Fault Detector of p_i, for the Existence of a Tree Rooted at a Leader.

Define the directed graph $\mathcal{T} = (V, E)$ as follows: each node of the graph V represents a processor in the system (and vice versa). There exists a directed edge $(i, j) \in E$ if and only if the value of the parent field of the processor p_i, in the tuple of $\mathcal{TU}_i$ with the maximal id, is p_j.

Definition 1. *A directed graph is an in-tree if the undirected underlying graph is a tree and if every edge of the tree is directed towards a common root.*

To prove the correctness we show that if no processor detects faults during an asynchronous cycle, then $\mathcal{T}$ is an in-tree rooted at the common leader (the processor with the maximal identifier).

The Tree Update Algorithm: Before we continue with the transient fault detector, let us add a mechanism to distribute the description of $\mathcal{T}$ to every processor in the system. We augment each processor p_i with a variable $\mathcal{T}_i$ that should contain the description of $\mathcal{T}$.

Let $\mathcal{T}_i(p_j)$ be the component of $\mathcal{T}_i$ that is connected to p_i when the link from p_i to p_j is removed from $\mathcal{T}_i$. p_i repeatedly sends $\mathcal{T}_i(p_j)$ to every processor $p_j \in (\{parent_i\} \cup children_i)$. $parent_i$ is defined by the value p_j of the tuple $\langle l, d, p_j \rangle$ in $\mathcal{TU}_i$ such that $l = leader_i$. The $children_i$ set includes every neighbor p_j from which the last table $\mathcal{TU}_j$ received, includes a tuple $\langle l, d, p_i \rangle$ where $l = leader_i$. p_i repeatedly computes $\mathcal{T}_i$ using the last values of $\mathcal{T}_j(p_i)$ received from every processor $p_j \in (\{parent_i\} \cup children_i)$. p_i construct $\mathcal{T}_i$ from the above $\mathcal{T}_j(p_i)$ adding the links connecting itself to the processors in $(\{parent_i\} \cup children_i)$.

Input: $\mathcal{T}_i, parent_i, children_i$ *(* updated by lower level *)*

1. $consistent \leftarrow true$
2. if $\mathcal{T}_i$ does not encode a spanning in-tree then $consistent \leftarrow false$
3. if $children_i$ is different from the set of processors that are the children of p_i in $\mathcal{T}_i$ then $consistent \leftarrow false$
4. if $parent_i = nil$ and p_i is not the leader of $\mathcal{T}_i$ then $consistent \leftarrow false$
5. if $parent_i \neq nil$ and $parent_i$ is not the parent of p_i in $\mathcal{T}_i$ then $consistent \leftarrow false$
6. return $consistent$

Fig. 2. Consistency Test Function.

We now prove the correctness of the tree update algorithm. In the proof we consider an execution that starts in a *safe configuration* of the update algorithm and prove correctness of the tree update in such executions. A *safe configuration* of the update algorithm is a configuration in which the values of the tuples of all the processors are correct (and therefore are not changed in any execution that starts in such a safe configuration).

In the lemma we use the term *height* of a processor p_i in an in-tree for the maximal number of edges in a path from a leaf in the tree to p_i, such that the path does not include the root of the tree.

Lemma 1. *Consider any execution R of the tree update algorithm that starts in a safe configuration of the update algorithm and consists of at least $l + 1$ asynchronous cycles. Let p_j be a processor such that $\mathcal{T}(p_j)$ is a sub-in-tree of $\mathcal{T}$ (the in-tree defined by the update algorithm) that is rooted at p_j, and the height of $\mathcal{T}(p_j)$ is at most l. Let $\mathcal{T}_j(p_j)$ be the description of the tree rooted at p_j in the variable $\mathcal{T}_j$ of p_j. It holds that $\mathcal{T}(p_j) = \mathcal{T}_j(p_j)$ in the last configuration of R.*

A configuration, c, is safe with relation to the tree update algorithm iff c is safe for the update algorithm and for every processor p_i $\mathcal{T}_i = \mathcal{T}$. Moreover, in any execution that starts in c the value of $\mathcal{T}_i$ is not changed (this last requirement implies, in fact, that any message in transit from p_i to p_j contains $\mathcal{T}_i(p_j)$ that is the portion of $\mathcal{T}$ connected to p_i when the link from p_i to p_j is removed).

Corollary 1. *The tree update algorithm reaches a safe configuration following the first $O(d)$ asynchronous cycles and its communication complexity is $O(n^2 \log N)$.*

Transient Fault Detector for Correct Description of the Tree: The second transient fault detector assumes the existence of a rooted spanning tree $\mathcal{T}$ that is defined by the child parent relation and ensures that every processor p_i has the description of $\mathcal{T}$ in $\mathcal{T}_i$. Thus, ensures that every processor knows the set of processors in its connected component.

Let us first describe the consistency test function in Figure 2 that is used by our transient fault detector. In the code we use the input $\mathcal{T}_i, parent_i, children_i$ which is defined by the output of the tree update algorithm. The consistency test function uses a boolean variable *consistent*. First p_i assigns true to the *consistent* variable (line 1 of Figure 2). In line 2, p_i checks $\mathcal{T}_i$ to be a spanning *in-tree* — a directed tree for which every edge is directed towards a common root. Lines 3, 4 and 5 test the child parent relations of p_i (according to the update algorithm) to be correct in $\mathcal{T}_i$. The function returns the final value of *consistent*.

The transient fault detector is presented in Figure 3. The fault detector will ensure that all local values of $\mathcal{T}$ are identical and that every processor local tree neighborhood appears in $\mathcal{T}$. In the code we use the input $\mathcal{T}_i, parent_i, children_i$ which is defined by the output of the tree update algorithm (see the description of the code of Figure 2 for the values of the above inputs).

p_i repeatedly executes line 1a and 1b. In line 1a p_i sends $\mathcal{T}_i$ to its parent and children. p_i checks the consistency of $\mathcal{T}_i$ according to the consistency test described in Figure 2 and detects a fault accordingly. Whenever p_i receives $\mathcal{T}_j$ from p_j, p_i checks whether $\mathcal{T}_i = \mathcal{T}_j$ and detects a fault if this equation is not true (line 2a of the code).

Input: $\mathcal{T}_i, parent_i, children_i$ *(* updated by lower level *)*

1. **Upon timeout:**
   (a) for each $j \in \{parent_i\} \cup children_i$ send $\mathcal{T}_i$ to p_j.
   (b) if $\mathcal{T}_i$ is inconsistent then detected a fault.
2. **Upon receiving** $\mathcal{T}_j$ **from** p_j
   (a) if $\mathcal{T}_i \not= \mathcal{T}_j$ then detected a fault.

Fig. 3. Transient Fault Detector of p_i, for Correct Description of the Tree.

We prove the correctness of the second fault detector assuming that no fault is detected by the first transient fault detector.

Lastly, we combine both the fault detectors; the first fault detector messages are augmented with the second fault detector messages. (Note that the second fault detector sends messages only on tree links. The messages sent by the first fault detector on non-tree links are not augmented by a message of the second fault detector). We conclude the presentation and correctness proof of the transient fault detectors by the following corollary.

Corollary 2. *(1) The combined fault detector detects a fault during a single asynchronous cycle whenever there exists a processor p_i such that $\mathcal{T}_i$ does not consist of the processors in p_i's connected component, and (2) The communication complexity of the combined fault detector is $O(n^2 \log N)$.*

3.3 Lower Bound on the Communication Complexity

We now present a lower bound of $\Omega(n^2 \log(N/n-1))$ bits on the communication complexity. The lower bound is for any fault detector that detects a fault within a single asynchronous cycle (whenever a processor has an inconsistent knowledge on the set of processors in its connected component or view). Recall that group membership services notifies the application with the current view. To do so we consider an asynchronous cycle that starts with all processors sending to every one of their neighbors (where a processor can send *nil* messages in case no message should be sent to a neighbor), and the cycle terminates after all messages sent are received. We examine processors $p_1, p_2, \cdots p_n$ that are connected by a chain communication graph. Assume that n is even (a similar argument can be used for a chain with an odd number of processors).

Let $m_{k,k+1}$ ($m_{k+1,k}$) be the message sent from p_k to p_{k+1} (from p_{k+1} to p_k, respectively). We claim that the number of distinct combinations of $m_{k,k+1}, m_{k+1,k}$ must be at least $\Omega(n \log(N/n-1))$.

Let p_k be a processor in the chain and suppose that $k \leq n/2$. Fix a set of k distinct identifiers for the processors $p_1, p_2, \cdots, p_k$. We prove a lower bound by using the number of possible choices of different sets of $n-k$ distinct identifiers for the rest of the processors $p_{k+1}, p_{k+2}, \cdots, p_n$.

Let $\mathcal{X}_1$ and $\mathcal{X}_2$ be two such choices. Now we describe two different systems that differ in the way we assign identifiers to processors $p_{k+1}, p_{k+2}, \cdots, p_n$. The identifiers of the processors $p_{k+1}, p_{k+2}, \cdots, p_n$ in the first (second) system are the identifiers in $\mathcal{X}_1$ ($\mathcal{X}_2$, respectively). Clearly the communication over the edge connecting p_k to p_{k+1} must not be the same in the two systems above. Otherwise we may replace the two different portions of the two systems and no fault will be detected, while p_1 is not aware of the different set of processors in the system. The case of $kn/2$ is handled analogously fixing a set of k distinct identifiers for the processors $p_{n-k}, p_{n-k+1}, \cdots, p_n$. In both cases we conclude that the number of communication patterns needed are at least the number of choices of $n-k$ distinct identifiers for the processors $p_{k+1}, p_{k+2}, \cdots, p_n$, out of $N-k$ identifiers. $(N-k)!/((n-k)!((N-k)-(n-k))!) = (N-k)!/((n-k)!(N-n)!) = ((N-n+1)\cdots(N-k))/(n-k)! \geq ((N-n+1)/(n-k))^{n-k}$. We assume that $N \geq 2n$, thus we have that $(N-n+1)/(n-k) \geq 1$. By the assumption that $1 \leq k \leq n/2$, we have that $((N-n+1)/(n-k))^{n-k} \geq ((N-n)/n)^{n/2}$. Therefore,

for the communication between $m_{k,k+1}, m_{k+1,k}$, at least $\Omega(n \log(N/n-1))$ bits are needed. The communication complexity is a measure that considers all the links and therefore is $\Omega(n^2 \log(N/n-1))$ bits.

3.4 Group Membership and Voluntarily Join/Leave

In a legal execution, only the user is privileged to change his/hers membership status in a group. Such a change occurs in response to the application requests. Here we describe how, in a legal execution, processor p_i may join (leave) a group g by locally setting (resetting, respectively) $member_i$.

We use the self-stabilizing β-synchronizer algorithm [14] to coordinate view updates. The β-synchronizer is designed to be executed on a spanning tree of the system, in our case $\mathcal{T}$. There are two alternating phases for the β-synchronizer, propagation phase and convergecast phase. In a legal execution, processor p_l (the root of $\mathcal{T}$) is responsible for the membership updates. During the propagation phase, p_l propagates the view it maintains v_l. As v_l propagates through $\mathcal{T}$, every processor p_i assigns v_l to a local variable v_i that maintains its view. The value of $member_i$ of every processor p_i is accumulated during the convergecast phase. The value of $member_l$ of a leaf in $\mathcal{T}$ is delivered to p_k the parent of p_l. A parent of a leaf processor p_k, concatenates the values of the $member_c$, received from its children p_c, together with $member_k$ and delivers it to its parent, and so on. Once the convergecast phase terminates, the root sends the received concatenated information on the membership of all the processors, together with a view identifier (the view identifier is changed whenever the set of members is changed).

A transient fault detector monitors the consistency of the join/leave and membership information. Details are omitted from this extended abstract.

Before we turn to describe the actions taken upon a fault detection let us note that randomized transient failure detector can be used as well. In a legal execution, our deterministic transient failure detector repeatedly sends the same message through each link. Thus, the randomized technique proposed in [22], that uses a logarithmic size of the repeatedly sent message can be used here to further reduce the size of the messages sent. In such a case the failure detectors will detect a fault with high probability.

3.5 Fast Convergence

So far we have discussed transient fault detection, without describing the action taken when a fault is detected. The goal of the technique presented here is to ensure a fast convergence in the cost of a higher communication complexity. Once the transient fault detector detects a fault, we would like to activate the self-stabilizing tree update algorithm to regain consistency as soon as possible and then switch back to use transient fault detector.

Propagation of a Fault Detection: Once a fault is detected by a processor p_i, the processor propagates the fault indication to every other processor. Every tuple of the update tables is extended to include a *state* field, where the domain of the state is {*safe, dtc, act*}. We use the term the *source tuple* of p_i for the

single tuple of $\mathcal{TU}_i$ with i in the *id* field. p_i starts the propagation by assigning the values $\langle i, 0, nil, dtc \rangle$ in its source tuple. In the sequel we use the fourth field of p_i's source tuple as the state of p_i.

Every processor p_j that has at least one state field in a tuple of $\mathcal{TU}_j$ with a value not equal *safe* executes the update algorithm, sending messages through every attached link. When p_i sends the new value of $\mathcal{TU}_i$ to p_j and the state of p_j is *safe*, p_j changes it's state to *dtc*. The information on the fact that p_i detected a fault propagates to the entire system in the same way.

Our goal is to ensure that every processor p_k verifies that the tuples in the tables of the processors encode a fixed BFS tree rooted at p_k, and therefore the update algorithm is in a safe configuration. Then we allow the system to switch back to use the transient fault detector.

A central tool in achieving an indication on the completion of the reconstruction of the BFS trees is the *PIF* procedure. The propagation is done by flooding the system with the new information in the way we described above (for the case of *dtc*). The propagating processor, p_i, should receive a feedback on the completion of the propagation before finalizing the *PIF* procedure. The feedback is sent to a processor with a smaller distance from p_i, which p_i selects to be its parent in the tree. Every processor p_j uses the distance variable of the tuple with $id = i$ in $\mathcal{TU}_j$ as its (upper bound on the) distance to p_i.

A processor p_j sends a feedback only when the maximal distance difference of p_j to p_i, and the distance of any neighbor p_k of p_j to p_i is 1. The fact that the value in the distance fields is an upper bound on the distance from p_i guarantees that every neighbor p_j of p_i sends feedback when the value of its distance field is 1 and therefore has a fixed parent (namely, p_i). Moreover, p_j sends a feedback only when every of its neighbors has distance of at most 2. Thus, processors of distance 2 have the correct distance and therefore a fixed parent. Similar arguments hold for processors of greater distances, concluding that a fixed BFS rooted at p_i exists when p_i receives the feedback. More details can be found in [13]. We note that part of the new information that is propagated is a randomly chosen color that identifies (with high probability) the current *PIF* execution initiated by p_i, as a new *PIF* execution.

The fast convergence algorithm should ensure stabilization from an arbitrary state. We trace the activity of the system from the first fault detection. We would like the fault detection to ensure that every processor will start a *PIF* following the fault detection. Then, when every processor completes the *PIF* and verifies that its tree, is a fixed BFS tree we can stop executing the communication expensive tree update algorithm.

When p_i detected a fault it starts a *PIF* that causes every processor p_j either (1) to change a state from *safe* to *dtc* and start a *PIF* or (2) when p_j is in the state *act* to execute at least one more complete *PIF* before changing state to *safe*.

The update algorithm is executed by p_i whenever there exists a tuple in $\mathcal{TU}_i$ with a state field not equal *safe*. Otherwise, p_i responds to any $\mathcal{TU}_j$ message (sent by a neighbor p_j) by recomputing $\mathcal{TU}_i$ accordingly, and sending $\mathcal{TU}_i$ to p_j exclusively. (Note that the transient fault detector is disabled whenever there exists a tuple in $\mathcal{TU}_i$ with a state field not equal *safe*).

We may conclude that: once a transient fault is detected and propagated to the entire system it holds that (1) no processor is in a *safe* state, and (2) no transient fault is detected.

Upon Completing the Propagation of a Fault Detection: A processor p_i that has completed propagating the fault indication (completing a single *PIF*) changes state to *act*. Then p_i waits for all other processors to complete their propagation of fault detection, reaching a system state in which no processor (uses the failure detector to detect a fault and) starts propagating an indication of a failure. In other words, when p_i is in *act* state p_i repeatedly executes *PIF* until it receives an indication that no *dtc* tuple appears in any table.

The indication for the absence of *dtc* tuples, is collected using a *PIF* query. The *PIF* procedure is used to query the values of the state fields using the following procedure: Every tuple of the update tables is extended to include a *nodtc* bit field. When p_k chooses a new color, p_k set the *nodtc* bit $true$, and starts a *PIF*. A processor p_j sets the *nodtc* bit of every tuple in its table to $false$, whenever there exists a tuple with the state *dtc* in $\mathcal{TU}_j$. Whenever p_j sends feedback to its parent (as part of the *PIF*) p_j sends also the *and* result of the *nodtc* bit values of its children tables and its own table. Thus, a single $nodtc = false$ results in a $nodtc = false$ feedback that arrives to p_k.

We may conclude that once the *nodtc PIF* query procedure is completed with $nodtc = true$, then no processor is in a *dtc* state (and the transient fault detectors are disabled). Furthermore, let p_k be the first processor that changes its state from *act* to *safe*, after processor p_i had notified a fault detection. Let c be the configuration that immediately follows this state change of p_k. We will prove that, (1) the tree rooted at each processor in c is a fixed BFS tree, (2) the state field of every tuple in every table in c is *act* and, (3) no transient fault is detected.

Returning to Normal Operation: Once all the processors are in *act* state the system is ready to return to normal operation. A processor p_i changes state to a *safe* state when p_i is in *act* state and finds out that no *dtc* state exists in the system. Still p_i does not activate the transient failure detector until all processors change state to a *safe* state. p_i repeatedly executes *PIF* queries until it finds that the state of all the processors is *safe*. Thus, when a processor returns to use the transient failure detector all the processors are in a *safe* state and therefore a fault detection will result in a global state change to *dtc*, then to *act* and at last to *safe* after reaching a safe configuration.

The *PIF* query initiated by a processor in a *safe* state uses an additional *allsafe* bit field. When p_k chooses a new color, p_k set both the *nodtc* and the *allsafe* bits to *true*, and starts the *PIF* procedure. Recall that a processor p_j sets *nodtc* bit to $false$, whenever there exists a tuple with a *dtc* state in $\mathcal{TU}_j$. In addition, p_j sets the *allsafe* bit to $false$, whenever there exists a tuple with a state not equal to *safe* in $\mathcal{TU}_j$. p_k changes it state to *dtc* whenever there is a *dtc* tuple in $\mathcal{TU}_k$, or a feedback with $nodtc = false$ arrives. If the feedback carrying the *allsafe* bit is *true*, then p_k stops executing the update algorithm, and starts using the transient fault detector. If the *allsafe* bit is $false$ (and the *nodtc* bit is *true*) then p_k assigns *true* to both *nodtc* and *allsafe* bits, and repeats executing the *PIF* query.

We note that the tree description used by the transient fault detector should be identical in all the processors before switching back to normal operation. Thus, the *allsafe* bit is also used to indicate that the tree description of a processor and its neighbors are identical (otherwise the *allsafe* bit that arrives in the feedback is $false$).

We may conclude that when the feedback of the *allsafe PIF* query is true, it holds that all the processors are in a *safe* state. Furthermore, let p_k be the first processor that returns to use the transient fault detector, after p_i propagated a fault detection. Let c be the configuration in which p_k returns to use the transient fault detector. Then in c it holds that the system is in a safe configuration with relation to the update algorithm.

We now turn to a detailed presentation of the fast convergence algorithm. The code of the fast convergence algorithm appears in Figure 4. In the code, we use the *PIF* and the *PIF query* procedures. A formal description of the *PIF* procedure can be found in [13]. The *PIF* procedure is extended to *PIF* queries (*nodtc* and *allsafe queries*) as described above.

Lines 1, 2 and 3 of the code describe the actions p_i takes according to it's state. When p_i is in a *dtc* state (line 1), p_i executes a *PIF* (line 1a), once the *PIF* is completed p_i changes its state to *act* (line 1b). When p_i is in *act* state (line 2), p_i repeatedly executes a *PIF query* to ensure that no *dtc* tuple exists in the system (line 2a). Then, p_i changes its state to *safe* (line 2b). In a *safe* state (line 3), p_i repeatedly executes a *PIF query* to ensure that all the states (of the processors and the state fields of the tuples) are *safe* states (line 3a). If there exists a *dtc* tuple, then p_i changes its state to *dtc* (line 3b). If indeed there are only *safe* tuples in the system then, p_i returns to use the transient failure detector (line 3c). Once the failure detector is operating, p_i changes its state to *dtc* when a fault is detected (line 3c and 3d).

```
1. state=dtc — (* Notify *)
   (a) Execute PIF
   (b) state ← act
2. state=act — (* Finish Notification *)
   (a) Execute PIF query
       until no dtc in the system
   (b) state ← safe
3. state=safe — (* Back to TFD *)
   (a) Execute PIF query
       until all safe or exists dtc
   (b) if PIF query results dtc then
       state ← dtc
   (c) else execute transient failure
       detector until fault detection
   (d) state ← dtc
```

Fig. 4. Fast convergence algorithm of p_i.

4 Concluding Remarks

This paper present the first asynchronous self-stabilizing group membership service. We believe that the new ideas presented in this paper will enrich the set of techniques used in the design of robust group communication services. For example, we do not utilize the idea of token passing for detecting a crash. Instead we present a self-stabilizing scheme that detects a fault fast (in a single asynchronous cycle) and is still communication efficient. Our membership service can serve as the base for additional group communication services.

Acknowledgment

We thank Idit Keidar, Nancy Lynch, Jennifer Welch and Alex Shvartsman for helpful discussions and suggestions, and Ilana Petraru for improving the presentation.

References

1. Y. Afek and G. M. Brown, "Self-stabilization over unreliable communication media," *Distributed Computing*, 7:27–34, 1993.
2. T. Anker, D. Breitgand, D. Dolev, and Z. Levy, "Congress: CONnection-oriented Group-address RES-olution Service" TR CS96-23, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, December 1996.
3. G. Alari and A. Ciuffoletti, "Group membership in a synchronous distributed system," *Proc. of the 5th IEEE Symposium on Parallel and Distributed Processing*, pp. 490-493, 1993.
4. Y. Afek, and S. Dolev, "Local Stabilizer," *Proc. of the 5th Israeli Symposium on Theory of Computing and Systems*, pp. 74-84, 1997.
5. A. Arora and S. Kulkarni, "Detectors and correctors: A theory of fault-tolerance components," *International Conference on Distributed Computing Systems*, pp. 436-443, 1998.
6. J. Beauquier, S. Delaet, S. Dolev, and S. Tixeuil, "Transient Fault Detectors" *Proc. of the 12th International Symposium on Distributed Computing*, Springer-Verlag LNCS:1499, pp. 62-74, 1998.
7. O. Babaoglu, R. Davoli, L. Giachini and M. Baker, "Relacs: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems," *Proc. Hawaii International Conference on Computer and System Science,* 1995, vol. II, pp. 612–621.
8. K.P. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit,* IEEE Computer Society Press, Los Alamitos, CA, 1994.
9. F. Cristian, "Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems," *Distributed Computing*, vol. 4, no. 4, pp. 175-187, April 1991.
10. T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the Impossibility of Group Membership," *Proc. ACM Symposium on Principles of Distributed Computing,* pp. 322-330, 1996.
11. E. W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Communications of the ACM,* Vol. 17, No. 11, pp. 643-644, 1974.
12. S. Dolev, "Self-Stabilizing Routing and Related Protocols," *Journal of Parallel and Distributed Computing*, Vol. 42, pp. 122-127, May 1997.

13. S. Dolev, "Optimal Time Self-Stabilization in Uniform Dynamic Systems," *Parallel Processing Letters,* Vol. 8 No. 1, pp. 7-18, 1998
14. S. Dolev, *Self-Stabilization,* MIT Press, 2000.
15. S. Dolev and T. Herman, Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997.
16. R. De Prisco, A. Fekete, N. Lynch, and A.A. Shvartsman, "A Dynamic Primary Configuration Group Communication Service," *Proc. of the 13th International Conference on Distributed Computing (DISC)*, 1999.
17. D. Dolev and D. Malki, "The Transis Approach to High Availability Cluster Communications", *Comm. of the ACM*, vol. 39, no. 4, pp. 64-70, 1996.
18. Dolev, S., Schiller, E., "Communication Adaptive Self-Stabilizing Group Communication", Technical Report #00-02 Department of Computer Science Ben-Gurion University, Beer-Sheva, Israel, 2000.
19. P. Ezhilchelvan, R. Macedo and S. Shrivastava "Newtop: A fault-Tolerant Group Communication Protocol" in *Proc. of IEEE International Conference of Distributed Computing Systems*, pp. 296-306, 1995.
20. A. Fekete, N. Lynch and A. Shvartsman, "Specifying and Using a Partitionable Group Communication Service," *Proc. ACM Symposium on Principles of Distributed Computing,* pp. 53–62, 1997.
21. I. Keidar, J. Sussman, K. Marzullo, and D. Dolev "Moshe: A Group Membership Service for WANs". *MIT Technical Memorandum MIT-LCS-TM-593a*, revised September 2000.
22. E. Kushilevitz and N. Nisan *Communication Complexity*, Cambridge University Press 1998.
23. L.E. Moser, P.M. Melliar-Smith, D.A. Agarawal, R.K. Budhia and C.A. Lingley-Papadoplous, "Totem: A Fault-Tolerant Multicast Group Communication System", *Comm. of the ACM*, vol. 39, no. 4, pp. 54-63, 1996.
24. G. Neiger, "A new look at membership service", *Proc. of the 15th Annual ACM Symposium on Principles of Distributed Computing*, 1996.
25. R. van Renesse, K. P. Birman and S. Maffeis, "Horus:A Flexible Group Communication System," *Comm. of the ACM,* vol. 39, no. 4, pp. 76-83, 1996.
26. R. van Renesse, K. P. Birman, M. Hayden, A. Vasburd, and D. Karr, "Building adaptive systems using Ensemble," *Software-Practice and Experience,* 29(9):963-979, 1998.
27. R. van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service", *In Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, The Lake District, England, September 1998, pp. 55-70.
28. A. Segall, "Distributed Networks Protocols", *IEEE Trans. Comm.*, vol. IT-29, no. 1, pp. 23-35, Jan. 1983.

(Im)Possibilities of Predicate Detection in Crash-Affected Systems

Felix C. Gärtner[1] and Stefan Pleisch[2]

[1] Department of Computer Science
Darmstadt University of Technology, Darmstadt, Germany
felix@informatik.tu-darmstadt.de
[2] IBM Research, Zurich Research Laboratory
Rüschlikon, Switzerland
spl@zurich.ibm.com

Abstract. In an asynchronous system, where processes can crash, perfect predicate detection for general predicates is difficult to achieve. A general predicate thereby is of the form $\alpha \wedge \beta$, where α and β refer to a normal process variable and to the operational state of that process, respectively. Indeed, the accuracy of predicate detection largely depends on the quality of failure detection. In this paper, we investigate the predicate detection semantics that are achievable for general predicates using either failure detector classes $\Box\Diamond\mathcal{P}$, $\Diamond\mathcal{P}$, or $\mathcal{P}$. For this purpose, we introduce weaker variants of the predicate detection problem, which we call *stabilizing* and *infinitely often accurate*. We show that perfect predicate detection is impossible using the aforementioned failure detectors. Rather, $\Diamond\mathcal{P}$ and $\mathcal{P}$ only allow stabilizing predicate detection. Consequently, we explore alternative approaches to perfect predicate detection: introducing a stronger failure detector, called *ordered perfect*, or restricting the general nature of predicates.

1 Introduction

Testing and monitoring distributed programs involves the basic task of detecting whether a predicate holds during the execution of the system. For example, a software engineer might want to detect the predicate "variable x has changed to value 2" to find out at what point in the execution x takes on a bad value. Predicate detection in distributed settings is a well-understood problem and many techniques together with their detection semantics have been proposed [7]. However, most of the techniques have been proposed under the assumption that no faults occur in the system. Hence, most of the methods proposed in the literature are not robust in the sense that they offer no guarantees if faults such as message losses or process crashes occur in the system.

In an asynchronous system where processes can crash, a general predicate detection mechanism should also detect these crash events. Chandra and Toueg [3] proposed to encapsulate the functionality of failure detection into a separate module and specify it using axiomatic properties. Such a *failure detector* can be

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 98–113, 2001.

used to locally maintain information about the operational state of the processes. Based on the quality of failure detection, different classes of failure detectors can be defined. Most relevant to this paper are the classes of perfect, eventually perfect, and infinitely often accurate failure detectors (denoted $\mathcal{P}$, $\Diamond\mathcal{P}$, and $\Box\Diamond\mathcal{P}$, respectively) [3, 11].

Standard predicate detection techniques aim at monitoring conditions which are composed of predicates on the local state space of processes [7]. For instance, a predicate $x_i = 1 \wedge y_i = 2$ evaluates on the variables x and y in the local state of process p_i. In this paper, we consider the detection of predicates that are boolean combinations of predicates on the local state and predicates on the operational state of a process. This allows us to evaluate predicates of the form $x_i = 1 \wedge \mathit{crashed}_i$, where $\mathit{crashed}_i$ is a predicate that is true iff (if and only if) application process p_i has crashed. Ideally, a predicate detection algorithm never erroneously detects such a predicate and does not miss any occurrence of the predicate in the underlying computation. However, the quality of the underlying failure detector severely limits the quality of predicate detection. We show that *perfect* predicate detection is generally impossible with failure detectors of type $\Box\Diamond\mathcal{P}$ and $\Diamond\mathcal{P}$. Rather surprisingly, the impossibility still holds for $\mathcal{P}$. We investigate weaker variants of predicate detection which we call *stabilizing* and *infinitely often accurate*. Briefly spoken, a predicate detection algorithm is stabilizing if it eventually stops making false detections and it is infinitely often accurate if it has infinitely many phases where it does not issue false detections. We also investigate two conditions under which perfect predicate detection is solvable. The first is the existence of a novel type of failure detector which we call *ordered perfect* (denoted $\widehat{\mathcal{P}}$) and which is strictly stronger that $\mathcal{P}$. The second condition imposes restrictions on the generality of predicates.

Apart from clarifying the relation between predicate detection and failure detection, this work wishes to stress the connection between "stabilizing failure detectors" [3] and *self-stabilization* [8] (which has only partly been done by other authors [14, 2]) and, hence, argue that self-stabilization is a concept of eminent practical importance. Furthermore, our results manifest a drawback of the approach that uses abstract failure detectors to solve problems in distributed computing, namely, that for every problem it is necessary to adapt the failure detector properties, a highly non-trivial task.

Related Work. While predicate detection in fault-free environments has been intensely studied [7], solving the task in faulty environments is not yet very well understood. To our understanding, Shah and Toueg [16] were the first to investigate this by adapting the snapshot algorithm of Chandy and Lamport [4] with a simple timeout mechanism. Chandra and Toueg [3] later argued to define the functionality of failure detection in an abstract way and proposed a rich set of failure detector classes. However, these classes were meant to help solve the *consensus* problem and not the problem of predicate detection. Garg and Mitchell [10] investigate the predicate detection problem again and define an *infinitely often accurate* failure detector, i.e., a failure detector which is implementable in asynchronous systems [11], but they restrict the scope of the

predicates to *set-decreasing* predicates. A predicate is set-decreasing whenever it holds for a set of processes, it also holds for a subset of these processes. To our knowledge, our work is the first to investigate the relationship between predicate detection and failure detection in the general case.

While it is not clear whether Garg and Mitchell [10] or Shah and Toueg [16] consider predicates which contain references to the operational state of processes, Gärtner and Kloppenburg [12] explicitly allow these types of predicates but restrict themselves to environments where only infinitely often accurate failure detectors are available. Other authors have investigated the use of perfect failure detectors to detect special predicates, e.g., distributed deadlocks [19] or distributed termination [17].

The observation that perfect failure detectors do not allow to solve all problems which are solvable in synchronous systems has been previously made by Charron-Bost, Guerraoui, and Schiper [6] by exhibiting a problem that is solvable in synchronous systems but is not solvable in asynchronous systems augmented with perfect failure detectors (the *strongly dependent decision* problem). While Charron-Bost et al. [6] argue that this result has practical consequences with respect to the efficiency of atomic commitment, our result shows that there exists a practically relevant problem, namely predicate detection, which suffers from the deficiencies of perfect failure detectors.

Paper Organization. After introducing the system assumptions in Section 2, we define three different semantics for the predicate detection problem in Section 3. In Section 4, we consider systems where crash failures occur, augmented with infinitely often accurate (Section 4.2), with eventually perfect (Section 4.3), and perfect failure detectors (Section 4.4). Our focus is on a system with one application process and one observer. We then generalize to multiple application processes and observers in Section 5. Finally, Section 6 concludes the paper. For lack of space, we mostly only give proof sketches and relegate the full proofs to the extended version of the paper.

2 System Assumptions

2.1 System Model

We consider a system with n *application processes* $p_1, ..., p_n$ and m *monitor processes* $b_1, \ldots, b_m$ (i.e., observers) whose task is to monitor the execution on the application processes. Processes communicate by message passing via FIFO channels in a fully connected network. Communication is reliable, i.e., no messages are lost, duplicated, or altered. Our system is asynchronous, i.e., no boundaries on communication delays nor on relative processor speeds exist. Application processes can fail by crashing. Once a process has crashed, it does not recover any more during the execution. A process which crashes is called *faulty*. A non-faulty process is called *correct*. For simplicity, we assume that monitor processes do not fail. If it is clear from the context we will refer to application processes simply as "processes" and to monitor processes as "monitors".

Every application process p_i has a local state s_i consisting of an assignment of values to all of its variables. A *global state* $G = \{s_1, \ldots, s_n\}$ is a set containing

exactly one local state s_i from every process p_i. State changes are assumed to be atomic events local to some process such that an *execution* of the application processes can be modeled as a sequence of global stages $G_1, G_2, \ldots$, where G_{i+1} results from G_i by executing a local event on some process. Global state G_1 denotes the initial global state of the system.

2.2 Failure Detectors

Each monitor process has access to a local failure detector module that provides (possibly incorrect) information about failures that occur on application processes [3]. A process p_i is *suspected* by monitor b_j if the failure detector module associated with b_j detects the failure of p_i. Failure detectors are defined in terms of a *completeness* and an *accuracy* property. The completeness property requires that a failure detector eventually suspects processes that have crashed, while the accuracy property limits the number of mistakes a failure detector can make. We recall the definitions of accuracy and completeness which are relevant in this paper [3]:

- (strong completeness) Eventually every application process that crashes is permanently suspected by every monitor process.
- (strong accuracy) No application process is suspected before it crashes.
- (eventual strong accuracy) There is a time after which correct application processes are not suspected by any monitor.

The failure detectors we consider in this paper all satisfy strong completeness. A *perfect failure detector* additionally satisfies strong accuracy. An *eventually perfect failure detector* satisfies eventual strong accuracy instead of strong accuracy. Garg and Mitchell [11] have introduced an additional accuracy property:

- (infinitely often accuracy) Correct application processes are not permanently suspected by any monitor.

An *infinitely often accurate failure detector* satisfies strong completeness and infinitely often accuracy.

Chandra and Toueg [3] assume that failure detectors are passive modules. A monitor queries the module to learn about the operational state of other processes. In this setting, an infinitely often accurate failure detector offers no "real" accuracy at all. This is because a monitor may always query the failure detector when it is not accurate, i.e., when it erroneously suspects the process. Garg and Mitchell [11] therefore assume that the complete history of a failure detector is available so that an application does not miss the change of a failure detector output. This is equivalent to a model where the failure detector asynchronously (via *interrupts*) notifies the application about any change in its state. This model clearly makes more assumptions than the *query model* of Chandra and Toueg [3]. However, we now argue that these two models are equivalent for perfect and eventually perfect failure detectors in the context of predicate detection.

Since we use infinitely often accurate failure detectors, we necessarily use the *interrupt model* of Garg and Mitchell [11] in this paper. Note that query based

failure detectors can be turned into interrupt style failure detectors by adding a concurrent task to the system which periodically queries the failure detector and interrupts the application whenever it detects state changes. For perfect failure detectors, this transformation can only increase the detection latency but maintains the accuracy and completeness properties. Similarly, for eventually perfect failure detectors, only false detections may go unnoticed. Hence, the predicate detection results we obtain for perfect and eventually perfect failure detectors in the interrupt model also hold in the query model.

Failure detectors are grouped into classes that represent the set of failure detectors satisfying the given properties. We denote the class of all perfect failure detectors by $\mathcal{P}$, the class of all eventually perfect failure detectors by $\Diamond\mathcal{P}$, and the class of all infinitely often accurate failure detectors by $\Box\Diamond\mathcal{P}$. Intuitively, a failure detector fd_1 is *stronger* than a failure detector fd_2 (denoted $fd_1 \succeq fd_2$) if there exists a distributed algorithm that can be used to emulate fd_2 using fd_1 [3]. If $fd_1 \succeq fd_2$ and $fd_2 \succeq fd_1$ we write $fd_1 \cong fd_2$. If $fd_1 \succeq fd_2$ but not $fd_2 \succeq fd_1$ we say that fd_1 *is strictly stronger than* fd_2 and write $fd_1 \succ fd_2$. The relation $\succeq$ can be defined for failure detector classes in an analogous way. From the literature [3,11] we know that the following relations hold: $\mathcal{P} \succ \Diamond\mathcal{P} \succ \Box\Diamond\mathcal{P}$.

3 Predicate Detection

Given some global predicate φ over the global state G of $p_1, \ldots, p_n$, we would like to have an algorithm which answers the question of whether or not φ holds in a given computation. Whenever an event occurs at some application process p_k, a *control message* about this event is sent from p_k to all the monitors (the normal computation messages are called *application messages*). One generally assumes that the execution of the event and the sending of the control message execute as one atomic action.

We define a *property* of the system as a set of executions. A system *satisfies* a property iff every execution which is possible by the system is an element of the property. Every such property can be written as the intersection of a *safety property* and a *liveness property* [1, 13].

3.1 Three Different Detection Semantics

We use the symbols $\Box$ ("always") and $\Diamond$ ("eventually") here in the following way: Let S be a safety property. We denote by $\Diamond S$ the property in which S eventually holds, i.e., the set in which every execution in S can be prefixed by an arbitrary but finite sequence of states. We denote by $\Box\Diamond S$ the property where S holds infinitely often, i.e., the set consisting of all traces which can be constructed from (infinitely) interleaving finite sequences from S and $\Diamond S$. Note that $S \subseteq \Diamond S \subseteq \Box\Diamond S$.

A detection algorithm for a global predicate φ should notify us by triggering a detection event on the monitor processes once φ holds in the computation. Formally, we seek an algorithm which satisfies the following two properties:

S (safety) If a monitor triggers a detection event then φ has held in the computation, and

L (liveness) once φ holds in the computation, a monitor will eventually trigger a detection event.

We assume that the algorithm triggers a positive signal on detection but we also require the algorithm to revoke its detection by issuing a "previous detection was wrong" signal to the application. If this occurs, we say that the algorithm *undetects* the predicate. Note that we detect whether the predicate φ *held* which is a stable predicate even if φ is unstable. A predicate is *stable*, if once it holds it holds forever.

Definition 1 (Detection Semantics). *Let S and L denote the safety and liveness properties of predicate detection. We define the three detection semantics Sem_1, Sem_2, and Sem_3 as follows (where + denotes set intersection):*

1. $Sem_1 = L + S$ *(perfect)*
2. $Sem_2 = L + \Diamond S$ *(stabilizing)*
3. $Sem_3 = L + \Box\Diamond S$ *(infinitely often accurate)*

To illustrate the use of these detection semantics, assume that φ is a debugging condition, i.e., a "bad state" which should not occur. On detection of such a state, the software developer usually wants to stop the application system and analyze the execution which caused the bad state. In this context, we would ideally like a detection algorithm *Alg* for predicate φ to satisfy Sem_1, i.e., *Alg* will make no mistakes and not miss any occurrence of φ. We call this *perfect* predicate detection. However, this is sometimes impossible to achieve. In particular, if φ contains conditions about the operational state of processes, the detection algorithm might mistakenly detect φ, i.e., violate S. In these cases *Alg* should at least satisfy Sem_2, i.e., *Alg* may (erroneously) detect the predicate and later undetect it again. We call this *stabilizing* predicate detection because it is guaranteed that the algorithm will eventually stop making wrong detections.

Note that from the user's point of view there is no immediate way to distinguish between a correct and an incorrect detection (this may only be achieved through other means, e.g., through halting the system and inspecting it). Stabilizing predicate detection may, however, still be useful, e.g., in situations where false detections merely effect the *efficiency* of an application (not its correctness) or in situations where achieving Sem_1 is provably impossible. Revisiting our debugging example the developer may want to detect a predicate φ in his distributed application in order to identify an invalid state of the application. Detection semantics Sem_2 are sufficient in most cases, as the developer can manually verify whether the predicate detection algorithm has been accurate. If it has not been, the predicate detection is continued.

But even Sem_2 is sometimes impossible to satisfy, i.e., *Alg* may make *infinitely* many mistakes about φ holding. In this case we would prefer that *Alg* behaves according to Sem_3, i.e., *Alg* continuously switches between phases where possible detections are accurate and phases where mistakes regarding φ are made. We call this *infinitely often accurate* predicate detection. This means that if φ never holds then every detection event will be followed by an undetection event. Semantics Sem_3 offer close to no guarantees and, hence, can be considered

as a "best effort" specification. But at least it is better than ignoring the safety part of the specification overall (i.e., we rule out trivial predicate detection which always issues a detection event even in cases where φ never holds). Note that $Sem_1 \subseteq Sem_2 \subseteq Sem_3$.

3.2 Global Predicates in Faulty Systems

In systems with crash faults, it is a natural desire to detect a class of predicates which makes no sense in fault-free systems because we now additionally want to detect predicates involving the operational state of processes. For example, we may want to detect the fact that "process p_i crashed while holding a lock." To express this information within a global predicate we assume that the operational state of a process is explicitly modeled by a boolean flag *crashed* which is part of the global state. More specifically, $crashed_i$ is true iff p_i has crashed. Using this flag, we formalize the above predicate as $lock_i = true \wedge crashed_i$.

With respect to the truth value of a predicate on the global state, the *crashed* variables are treated just like other variables local to the processes. Let α denote a local predicate which only references local variables of a process, e.g., $\alpha \equiv x_i = 1$, and let β denote a predicate which only contains references to the operational state of a process, e.g., $\beta \equiv crashed_i$. To detect α we can use a standard mechanism for predicate detection in fault free systems. Conversely, to detect β we can use a (reliable) failure detection algorithm. However, global predicates can be constructed from boolean combinations of α and β. If we have disjunctions of α and β, e.g., $x_i = 1 \vee crashed_i$, it is sufficient to run existing detection algorithms for α and β independently and issue a detection event as soon as one of the algorithms issues such an event. However, global predicates that are a conjunction of α and β are more difficult to detect and are the focus of this paper.

The following examples illustrate the types of predicates we address. Let ec_i denote a local variable of p_i which stores the sequence number of events ("event count") on p_i.

- $ec_i \geq 5 \wedge crashed_i$, i.e., "$p_i$ crashed after event 5"
- $ec_i = 5 \wedge crashed_i$, i.e., "$p_i$ crashed immediately after event 5"
- $ec_i < 5 \wedge crashed_i$, i.e., "$p_i$ crashed before reaching event 5"
- $ec_i = 5 \wedge \neg crashed_i$, i.e., "$p_i$ executed event 5".

Although we have not restricted the class of predicates, it should be noted that only those predicates are detectable that do not explicitly depend on global time. For instance, the predicate $\varphi \equiv$"p_i executed event e_i more than 10 seconds ago" is impossible to detect in an asynchronous system model where no notion of global time exists. In analogy to Charron-Bost et al. [6], we call predicates that do not refer to real time *time free*.

4 Predicate Detection in Faulty Systems

We now consider an asynchronous system where crash faults can happen and study what types of detection semantics (i.e., Sem_1, Sem_2, or Sem_3) are achievable using different classes of failure detectors. For simplicity, we will restrict our

On monitor b:

variables:
$G : (s_1, \ldots, s_n)$ **init** $(I_1, \ldots, I_n)$
$crashed[1..n]$ **array of** $\{true, false\}$ **init** $(false, \ldots, false)$
$\varphi : G \times crashed \rightarrow \{true, false\}$ **init** ⟨by application⟩
$h : \{true, false\}$ **init** $false$

1 **algorithm:**
2 **upon** ⟨a message (i, e) arrives⟩ **do**
3 ⟨update $G[i]$ according to e⟩
4 **if** $\varphi(G, crashed) \wedge \neg h$ **then**
5 $h := true$
6 ⟨trigger detection event⟩
7 **elsif** $\neg\varphi(G, crashed) \wedge h$ **then**
8 $h := false$
9 ⟨trigger undetection event⟩
10 **end**
11 **upon** ⟨p_i is suspected or rehabilitated by failure detector⟩ **do**
12 ⟨update $crashed[i]$ accordingly⟩
13 **if** $\varphi(G, crashed) \wedge \neg h$ **then**
14 $h := true$
15 ⟨trigger detection event⟩
16 **elsif** $\neg\varphi(G, crashed) \wedge h$ **then**
17 $h := false$
18 ⟨trigger undetection event⟩
19 **end**

Fig. 1. Generic algorithm for predicate detection in faulty environments.

attention to the case where $n = m = 1$, i.e., a system with two processes only, namely an application process and a monitor process. We discuss the case where $n, m > 1$ in Section 5.

If faults can happen, the predicate detection algorithm must cater for the fact that a failure detector issues a suspicion or rehabilitation of a process. A process is *rehabilitated* if it has been erroneously suspected and the failure detector revokes its suspicion. Figure 1 shows a generic detection algorithm for this case. A boolean flag h ("history") is used to record the type of the most recent event which was triggered at the interface.

4.1 Plausible Failure Detector

Generally, the failure detector module accessed by the monitor issues suspicion and rehabilitation events for a process p_i. We define the following two properties concerning these events:

- (alternation) Suspicion and rehabilitation events for p_i alternate, i.e., the failure detector module never issues two suspicion events without issuing a rehabilitation event inbetween, and vice versa.
- (plausibility) Reception of a control message from p_i is only possible in phases where p_i is not suspected.

We argue later that these properties are fundamental to the predicate detection problem. Based on these properties, we define a plausible failure detector as follows:

Definition 2 (Plausible Failure Detector). *A* plausible *failure detector is a failure detector which satisfies alternation and plausibility.*

As shown in the following lemma, for most failure detector classes a plausible failure detector still belongs to the same class of failure detectors as its non-plausible equivalent.

Lemma 1. *Let F be either failure detector class $\Box\Diamond\mathcal{P}$ or $\Diamond\mathcal{P}$ and let $\bar{F}$ denote the set of failure detectors from F which are plausible. Then $F \cong \bar{F}$.*

Proof. A failure detector is rendered plausible by wrapping the failure detector and the delivery module for control messages into a separate module. If a control message arrives after a suspicion event has been generated, a rehabilitation event is passed to the predicate detection algorithm before delivering the control message. Clearly, the wrapper can be implemented in asynchronous systems.

Transforming a failure detector *fd* in $\mathcal{P}$ into a plausible failure detector using the algorithm in Figure 2 may result in a weaker failure detector. Indeed, assume a system with one process p and one monitor b, where p has sent a control message *msg* and then crashes. Before b receives the control message, it detects the crash of p. On reception of *msg*, the plausible version of *fd* rehabilitates p in order to receive *msg*. Later, *fd* eventually suspects p again. However, a failure detector in $\mathcal{P}$ is not allowed to make mistakes, i.e., can never rehabilitate processes. Hence, if *fd* is made plausible with algorithm in Figure 2, it is no longer in class $\mathcal{P}$. This is the reason why Lemma 1—using this particular wrapper—does not hold for failure detectors in $\mathcal{P}$.

4.2 Using an Infinitely Often Accurate Failure Detector ($\Box\Diamond\mathcal{P}$)

Consider a purely asynchronous system in which crash faults can happen. Garg and Mitchell [11] have shown that a failure detector in $\Box\Diamond\mathcal{P}$, e.g., an *infinitely often accurate failure detector*, can be implemented in these systems. Additionally we assume that such a failure detector is plausible. Then, predicate detection is only achievable with semantics Sem_3.

Theorem 1. *In asynchronous systems with crash failures and any failure detector in $\Box\Diamond\mathcal{P}$ it is (a) possible to satisfy detection semantics Sem_3 but it is (b) impossible to satisfy detection semantics Sem_2 and Sem_1 for general predicates without a failure detector strictly stronger than $\Box\Diamond\mathcal{P}$.*

Proof. We prove (a) using the standard algorithm in Figure 1. The reliable channel assumption ensures satisfaction of the liveness requirement of Sem_3 and the plausible infinitely often accuracy of failure detection ensures the safety requirement of Sem_3. Part (b) is proven indirectly: We assume that an algorithm satisfying Sem_2 exists and use it to construct a failure detector that allows to solve consensus, a contradiction to the impossibility result by Fischer, Lynch and Paterson [9].

On monitor b:

variables:
$suspects$ **set of** ⟨processes⟩ **init** $\emptyset$
$wasSuspected : \{true, false\}$ **init** $false$

1 **algorithm:**
2 **upon** ⟨fd rehabilitates p_i or a control message from p_i arrives⟩ **do**
3 $wasSuspected := false$
4 **if** $p_i \in suspects$ **then**
5 $wasSuspected := true$
6 $suspects := suspects \setminus \{p_i\}$
7 ⟨trigger event "rehabilitation of p_i"⟩
8 **endif**
9 **if** ⟨control message was received in line 2⟩ **then**
10 ⟨deliver control message ⟩
11 **if** $wasSuspected$ **then**
12 $suspects := suspects \cup \{p_i\}$
13 ⟨trigger event "suspicion of p_i"⟩
14 **endif**
15 **endif**
16 **upon** ⟨fd suspects p_i⟩ **do**
17 **if** $p_i \notin suspects$ **then**
18 $suspects := suspects \cup \{p_i\}$
19 ⟨trigger event "suspicion of p_i"⟩
20 **endif**

Fig. 2. Implementation of a wrapper that makes any failure detector fd in $\Box\Diamond\mathcal{P}$ or $\Diamond\mathcal{P}$ plausible in asynchronous systems. Note that line 2 and 16 refer to events generated by fd while in lines 7, 10, 13, and 19 events at the interface of the plausible failure detector are triggered which are then processed at lines 2 and 11 in the algorithm of Figure 1.

4.3 Using an Eventually Perfect Failure Detector ($\Diamond\mathcal{P}$)

Defining stronger limitations on incorrect failure suspicions results in stronger predicate detection semantics. As $\mathcal{P}$ and $\Diamond\mathcal{P}$ are stronger than $\Box\Diamond\mathcal{P}$ [11], Theorem 1 (a) holds also for these failure detectors, i.e., Sem_3 can be achieved. Note that the failure detector must be made plausible[1] before being used in the algorithm of Figure 1.

Corollary 1. *In asynchronous systems with crash failures and any failure detector in $\Diamond\mathcal{P}$ it is possible to satisfy detection semantics Sem_3.*

However, although an eventually perfect failure detector is stronger, it still is not sufficient to detect predicates perfectly. Actually, even a perfect failure detector cannot achieve perfect predicate detection. The intuitive reason for this is

[1] Although making $\mathcal{P}$ plausible with the wrapper in Figure 2 weakens the failure detector to $\Diamond\mathcal{P}$, Theorem 1 (a) still holds.

depicted in Figure 3. Consider the case where a predicate φ is true iff p_1 crashes after event e_1, i.e., $\varphi \equiv ec_1 = 1 \wedge crashed_1$. After suspecting p_1 (see Figure 3 (b)) the monitor b_1 must eventually raise an exception to the application that the predicate held. However, b_1 can never be sure that the predicate detection is accurate, because a message from p_1 may arrive later informing it about an event e_2 (see Figure 3 (a)). Since the message can be delayed for an arbitrary amount of time, b_1 cannot distinguish between both scenarios.

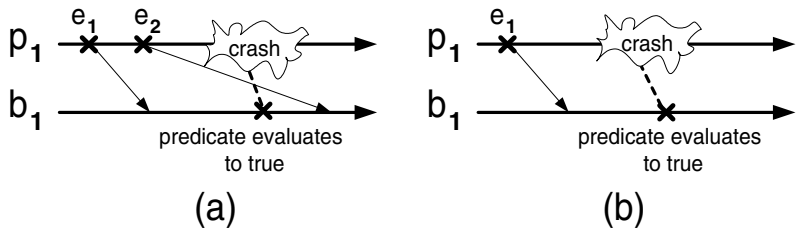


Fig. 3. Predicate $\varphi \equiv ec_1 = 1 \wedge crashed_1$ is not detectable according to detection semantics Sem_1 with any failure detector in $\mathcal{P}$.

Theorem 2. *In asynchronous systems with crash failures and any failure detector not strictly stronger than $\mathcal{P}$ it is impossible to satisfy detection semantics Sem_1.*

Corollary 2. *In asynchronous systems with crash failures and any failure detector not strictly stronger than $\Diamond\mathcal{P}$ it is impossible to satisfy detection semantics Sem_1.*

On the other hand, detecting Sem_2 with $\Diamond\mathcal{P}$ is achievable. However, the proof again relies on the fact that the given failure detector is plausible. Using a non-plausible failure detector may cause a miss of the occurrence of certain predicates and thus violate the liveness property. Assume, for instance, the predicate $\varphi \equiv x_i = 1 \wedge \neg crashed_i$, with x initially 0. Furthermore, assume that the failure detector fd in $\Diamond\mathcal{P}$ is not plausible and that it erroneously suspects process p_i. Although p_i sends the control messages about an event that sets x to 1 and back to 0 again, the monitor does not detect that φ has held.

Theorem 3. *In asynchronous systems with crash failures and any failure detector in $\Diamond\mathcal{P}$ it is possible to satisfy detection semantics Sem_2.*

4.4 Using a Perfect Failure Detector ($\mathcal{P}$)

Even a perfect failure detector is not sufficient to perfectly detect all possible predicates. Indeed, from the point of view of predicate detection for general predicates the strongest possible detection semantics are the same as for $\Diamond\mathcal{P}$. This

has already been shown in Theorem 2, i.e., it is impossible to detect with Sem_1 using any failure detector in $\mathcal{P}$. However, since Sem_2 is achievable using $\Diamond\mathcal{P}$, it is also achievable with $\mathcal{P}$.

Corollary 3. *In asynchronous systems with crash failures and a perfect failure detector it is possible to satisfy detection semantics* Sem_2.

Interestingly, we can detect predicates of the form $\alpha \wedge \beta$ according to Sem_1 using $\mathcal{P}$ if α is stable. The stability of α ensures that the predicate still holds, although control messages may still arrive from events that occurred immediately before the crash of the process (see Figure 3).

4.5 Introducing Failure Detector Class $\widehat{\mathcal{P}}$

A perfect failure detector is not sufficient to achieve optimal detection semantics in asynchronous systems. Intuitively, this is because $\mathcal{P}$ offers no information about the relative ordering of the crashes with respect to other application events. Consequently, we require a *plausible* failure detector that is still in $\mathcal{P}$. However, we show that this plausible failure detector is actually stronger than any failure detector in $\mathcal{P}$.

Definition 3 (Ordered Perfect Failure Detector). *An* ordered perfect failure detector *is a perfect failure detector which satisfies the following additional order property: Together with every "suspicion of* p_i*" event, the failure detector issues the event number of the last event that happened on* p_i.

We denote the class of all ordered perfect failure detectors by $\widehat{\mathcal{P}}$.

Theorem 4. $\widehat{\mathcal{P}} \succ \mathcal{P}$.

Proof. The fact that $\widehat{\mathcal{P}}$ is at least as strong as $\mathcal{P}$ is obvious. The proof that $\mathcal{P}$ is not at least as strong as $\widehat{\mathcal{P}}$ reuses the idea of Theorem 2 since $\widehat{\mathcal{P}}$ allows to distinguish the two situations which were indistinguishable if only $\mathcal{P}$ is available.

An ordered perfect failure detector allows to order crashes and normal process events causally, i.e., if a suspicion is issued by the failure detector and the associated sequence number is x, then delivery of the suspicion event can be held back until all control messages which have sequence numbers below or equal to x have been delivered. Hence, plausibility for an ordered perfect failure detector is achieved, which in turn means that the detection algorithm from Figure 1 allows to detect predicates with detection semantics Sem_1.

Theorem 5. *In asynchronous systems with crash failures and an ordered perfect failure detector it is possible to detect general predicates with detection semantics* Sem_1.

Overall, perfect detection of general predicates in asynchronous systems is achievable only if we postulate a failure detector that is strictly stronger than a perfect failure detector. This is somewhat disappointing since even perfect failure detectors are very hard to implement in practice. (This also shows that in these situations stabilizing service semantics (like Sem_2) are reasonable path to follow.) However, ordered perfect failure detectors can still be implemented using a *timely computing base* [18]. In such an approach, an asynchronous network is enhanced by a synchronous real-time control network. The asynchronous network is assumed to be high-bandwidth and is used for regular "payload" traffic while the synchronous network is only used for small control messages and therefore can be low-bandwidth. Under these assumptions it is possible to build a failure detection service that satisfies the order requirement of $\widehat{\mathcal{P}}$ by synchronously passing information about sent messages over the control network. Unfortunately, the execution of the event, and the sending on the synchronous and asynchronous network together have to be executed as an atomic action, which is a rather strong assumption. However, with this approach, a remote process accurately detects process crashes and is aware of the number of control messages sent prior to the crash.

5 Generalization to n Processes and m Monitors

In the previous sections we consider predicates local to one process in conjunction with a predicate on the operational state of this process, i.e., $\alpha \wedge \beta$. This section generalizes our results to scenarios with multiple processes (i.e., $n > 1$) and to multiple monitors (i.e., $m > 1$). The algorithms presented in Figures 1 and 2 are thus executed on every monitor. In the context of n processes and m monitors, the predicates are of the form $(\alpha_1 \wedge \beta_1)$ *op* $(\alpha_2 \wedge \beta_2)$ *op* ..., where *op* denotes either $\wedge$ or $\vee$.

In a system with n processes and m monitors, a causal broadcast mechanism is used so that the control messages are received by the monitors in *causal order* [15].

5.1 Observer Independence

Generalizing predicate detection to systems with multiple processes and multiple monitors gives rise to the issue of *observer independence*. Depending on the predicate φ and the setting in which it is evaluated, the validity of certain predicates depends on the observer [15]. Observer independence is achieved if *all possible* observations of the system result in the same truth value for φ [5]. Assume, for instance, that process p_1 executes an assignment $x := x+1$ (i.e., event e_1^1 in Figure 4) and p_2 an assignment $y := y+1$ (i.e., event e_1^2) on variables x and y which are initially 1. While monitor b_1 detects the predicate $\varphi \equiv x = 1 \wedge y = 2$, b_2 does not; the predicate φ is thus not observer independent, although the corresponding local predicates (i.e., $x = 1$ and $y = 2$) are. Charron-Bost et al. [5] have shown that observer independence is maintained for the disjunction of observer independent predicates, whereas it generally is not for the conjunction.

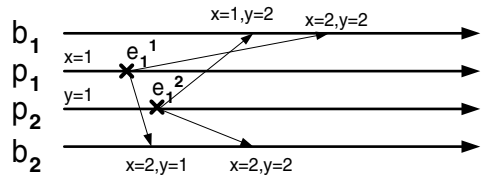


Fig. 4. Example of an observer dependent predicate, where e_1^1 specifies the event $x := x + 1$ and e_1^2 the event $y := y + 1$.

In general, two approaches are possible to address the problem of observer independence: (a) limiting the set of observed predicates or (b) defining a different notion of what it means for φ to hold. We will focus on the former approach here. The latter approach has been studied by Gärtner and Kloppenburg [12].

5.2 Limiting the Set of Observed Predicates

The global predicates we are considering consist of the conjunction and disjunction of local predicates α_i and predicates about the operational state of processes β_i. Unreliable failure detection introduces a new source of observer dependence; observer independence for a global predicate generally depends on the failure detector. Obviously, β_i is detectable in an observer independent way if a perfect failure detector is available. However, predicates of type $\alpha_i \wedge \beta_i$ need an ordered perfect failure detector to be detectable in an observer independent way. A failure detector of class $\mathcal{P}$ is sufficient, if α_i is stable. On the other hand, a failure detector in $\Diamond\mathcal{P}$ only achieves "eventual" observer independence, whereas with $\Box\Diamond\mathcal{P}$, observer independence may never be achieved.

Limiting the set of observed predicates to observer independent predicates considerably reduces the number of global predicates that can be detected. However, following Charron-Bost et al. [5] and the above findings, we can construct new observer independent global predicates from smaller building blocks. For example, disjunctions of stable local predicates in conjunction with predicates on the operational state of processes, i.e., $(\alpha_i \wedge \beta_i) \vee (\alpha_j \wedge \beta_j)$, remain observer independent if a failure detector in $\mathcal{P}$ is available. On the other hand, conjunctions of observer independent local predicates and predicates about the operational state of processes, i.e., $(\alpha_i \wedge \beta_i) \wedge (\alpha_j \wedge \beta_j)$, generally are not observer independent.

6 Conclusions

This paper investigates the predicate detection semantics that are achievable for general predicates using either failure detector classes $\Box\Diamond\mathcal{P}$, $\Diamond\mathcal{P}$, or $\mathcal{P}$. A general predicate thereby is of the form $\alpha \wedge \beta$, where α is a local predicate and β denotes a predicate on the operational state of a process, i.e., specifies whether a process has crashed or not. We define three different predicate detection semantics: perfect (i.e., Sem_1), stabilizing (Sem_2), and infinitely often accurate (Sem_3). Our

Table 1. Strongest achievable predicate detection semantics with respect to types of predicates and the failure detector class available. Again, α denotes a predicate which refers only to normal process variables and β is a predicate referring only to the operational state of the process.

Failures	Predicates	Failure Detector Class	Achievable Semantics	Reference
none	α	none	Sem_1	[7]
crash	α	none	Sem_1	[7]
crash	β	$\mathcal{P}$	Sem_1	[3]
crash	β	$\Diamond\mathcal{P}$	Sem_2	[3]
crash	β	$\Box\Diamond\mathcal{P}$	Sem_3	[10]
crash	$\alpha \vee \beta$	$\mathcal{P}$	Sem_1	Sect. 3.2
crash	$\alpha \wedge \beta$	$\Box\Diamond\mathcal{P}$	Sem_3	Thm 1
crash	$\alpha \wedge \beta$	$\Diamond\mathcal{P}$	Sem_2	Thm 3
crash	$\alpha \wedge \beta$	$\mathcal{P}$	Sem_2	Cor. 3
crash	$\alpha \wedge \beta$, α stable	$\mathcal{P}$	Sem_1	Sect. 4.4
crash	$\alpha \wedge \beta$	$\widehat{\mathcal{P}}$	Sem_1	Thm 5

results show that failure detector class $\Box\Diamond\mathcal{P}$ allows to detect general predicates according to Sem_3, whereas $\Diamond\mathcal{P}$ enables Sem_2. Somewhat surprisingly, a perfect failure detector is not sufficient to detect general predicates according to Sem_1, indicating the importance of stabilizing detection semantics. This leads to the definition of a stronger failure detector, called ordered perfect and denoted $\widehat{\mathcal{P}}$. With $\widehat{\mathcal{P}}$, perfect predicate detection (i.e., Sem_1) is achievable. An overview of our results is shown in Table 1.

In the future, we plan to further investigate issues of observer independence in systems with n processes and m monitors and consider predicate detection under more severe fault assumptions, e.g., crash-recovery.

Acknowledgments

We wish to thank Sven Kloppenburg, Klaus Kursawe, and the anonymous reviewers for their very insightful comments on this paper.

References

1. B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
2. J. Beauquier and S. Kekkonen-Moneta. Fault-tolerance and self-stabilization: Impossibility results and solutions using self-stabilizing failure detectors. *International Journal of System Science*, 28(11):1177–1187, 1997.
3. T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.
4. K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computing Systems*, 3(1):63–75, 1985.

5. B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and temporal predicates in distributed systems. *ACM Transactions on Programming Languages and Systems*, 17(1):157–179, Jan. 1995.
6. B. Charron-Bost, R. Guerraoui, and A. Schiper. Synchronous system and perfect failure detector: Solvability and efficiency issues. In *International Conference on Dependable Systems and Networks (IEEE Computer Society)*, 2000.
7. C.M. Chase and V.K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.
8. S. Dolev. *Self-Stabilization*. MIT Press, 2000.
9. M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.
10. V.K. Garg and J.R. Mitchell. Distributed predicate detection in a faulty environment. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, 1998.
11. V.K. Garg and J.R. Mitchell. Implementable failure detectors in asynchronous systems. In *Proc. 18th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 1530 in Lecture Notes in Computer Science, Chennai, India, Dec. 1998. Springer-Verlag.
12. F.C. Gärtner and S. Kloppenburg. Consistent detection of global predicates under a weak fault assumption. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS 2000)*, pages 94–103, Nürnberg, Germany, Oct. 2000. IEEE Computer Society Press.
13. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, Mar. 1977.
14. H. Matsui, M. Inoue, T. Masuzawa, and H. Fujiwara. Fault-tolerant and self-stabilizing protocols using an unreliable failure detector. *IEICE Transactions*, E83-D(10):1831–1840, Oct. 2000.
15. R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7:149–174, 1994.
16. A. Shah and S. Toueg. Distributed snapshots in spite of failures. Technical Report TR84-624, Cornell University, Computer Science Department, July 1984.
17. S. Venkatesan. Reliable protocols for distributed termination detection. *IEEE Transactions on Reliability*, 38(1):103–110, Apr. 1989.
18. P. Veríssimo, A. Casimiro, and C. Fetzer. The timely computing base: Timely actions in the presence of uncertain timeliness. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 533–542, New York City, USA, June 2000. IEEE Computer Society Press.
19. P. yu Li and B. McMillin. Fault-tolerant distributed deadlock detection/resolution. In *Proceedings of the 17th Annual International Computer Software and Applications Conference (COMPSAC'93)*, pages 224–230, Nov. 1993.

The Theory of Weak Stabilization*

Mohamed G. Gouda[1]

Department of Computer Sciences, The University of Texas at Austin
Austin, TX 78712-1188, U.S.A.
gouda@cs.utexas.edu

Abstract. We investigate a new property of computing systems called weak stabilization. Although this property is strictly weaker than the well-known property of stabilization, weak stabilization is superior to stabilization in several respects. In particular, adding delays to a system preserves the system property of weak stabilization, but does not necessarily preserve its stabilization property. Because most implementations are bound to add arbitrary delays to the systems being implemented, weakly stabilizing systems are much easier to implement than stabilizing systems. We also prove the following important result. A weakly stabilizing system that has a finite number of states is in fact stabilizing assuming that the system execution is strongly fair. Finally, we discuss an interesting method for composing several weakly stabilizing systems into a single weakly stabilizing system.

1 Introduction

There has been a growing interest in recent years to design and implement stabilizing computing systems. See for instance, [3], [4], [5], [6], and [7]. Unfortunately, stabilizing systems are difficult to implement in such a way as to preserve their stabilization properties. (This fact is often ignored in light of the immense intellectual pleasure that one can derive from designing such systems.) The main reason for this difficulty is that stabilization properties are extremely delay sensitive. The simple transformation of adding a small delay unit to a stabilizing system can yield this system non-stabilizing [1]. Because every system implementation is bound to add one or more delay units to the system being implemented, the implemented system often ends up being non-stabilizing.

This situation leaves the designers of stabilizing systems, who wish to implement their designs, with three options.

The first option is to identify all the delay units that may be added to the system during its implementation, then ensure that the system with the added delay units is still stabilizing. This option is not attractive because a system with many delay units has a relatively large state space, and the task of ensuring that such a system is stabilizing is usually hard.

The second option is to implement the system in a way that does not necessarily preserve its stabilization properties, then hope for the best, namely that

* This work is supported in part by DARPA contract F33615-01-C-1901.

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 114–123, 2001.

the implemented system will turn out to be "almost stabilizing". In this case, the system designer does not plan on validating this hope, and so the concept of "almost stabilization" is left vague and undefined.

The third option is to introduce a weaker version of the stabilization property, then show that this weak stabilization is delay insensitive. In this case, most sensible implementations of stabilizing (or even weak stabilizing) systems are guaranteed to be weakly stabilizing.

In this paper, we adopt this third option, and give a formal characterization of the weak stabilization property. In particular, we show that weak stabilization is delay insensitive and that it is a good approximation of the original stabilization property.

2 Stabilization and Weak Stabilization

A (computing) system is a nonempty set of variables, whose values are from predefined domains, and a nonempty set of actions that can be executed to update the values of the variables. Each action is of the form:

$\langle guard \rangle \rightarrow \langle assignment \rangle$

The $\langle guard \rangle$ is a Boolean expression over the system variables, and $\langle assignment \rangle$ is an assignment statement of the form:

$\langle variable \rangle := \langle expression \rangle$

The $\langle expression \rangle$ is an expression over the system variables and its value is from the domain of $\langle variable \rangle$.

For simplicity, we require that each variable of a system S be in the left-hand side of the assignment of at most one action of system S. In this case, the v action refers to the action where variable v is in the left-hand side of its assignment.

A state of a system S is a function that assigns a value to each variable of S. The value assigned to each variable is from the domain of that variable.

An action of a system S is enabled at a state p of S iff the guard of the action is true at state p. For simplicity, we assume that at least one action of a system S is enabled at each state of S.

A transition of a system S is a pair (p, q), where p and q are states of S and there is an action of S that is enabled at state p and executing this action starting at state p yields system S in state q.

A computation of a system S is an infinite sequence p.0, p.1, ... of S states such that every pair (p.i, p.(i+1)) of successive states in the sequence is a transition of S.

A state predicate of a system S is a function that has a Boolean value, true or false, at each state of S. Let true denote the state predicate whose value is true at each state of S, and false denote the state predicate whose value is false at each state of S.

A state p of a system S is a P state iff P is a state predicate of S whose value is true at state p.

Let P and Q be state predicates of a system S. Predicate P equals predicate Q, denoted P = Q, in system S iff P and Q have equal values at every state of S.

Predicate P implies predicate Q, denoted P ⇒ Q, in system S iff for every state p of system S, if P is true at p then Q is true at p.

A state predicate P of a system S is closed in S iff for each transition (p, q) of S, if p is a P state, then q is a P state.

A system S is stabilizing to a state predicate P iff the following two conditions hold. First, P is closed in S. Second, for every state p, every computation of S that starts at p has a P state.

A system S is weakly stabilizing to a state predicate P iff the following two conditions hold. First, P is closed in S. Second, for every state p, there is a computation of S that starts at p and has a P state.

Theorem 1. *If a system S is stabilizing to a state predicate P, then S is weakly stabilizing to P. The converse does not necessarily hold.*

Proof. Stabilization clearly implies weak stabilization. It remains to show that the converse does not necessarily hold. Consider a unidirectional token ring similar to that discussed in [2]. Assume that there are four or more actions in the ring and that the domain of values for each variable is 0..1. It is straightforward to show that this ring is weakly stabilizing but not stabilizing to an appropriate state predicate P.

3 Proof Obligations

In this section, we state proof obligations that can be used to prove that a system S is stabilizing, or weakly stabilizing, to a state predicate P. But first, we need to introduce the concepts of well-founded domain and ranking function.

A well-founded domain is a pair (D, >) where D is a set of elements and > is a total order relation over the elements of D such that each sequence of elements of D, that is decreasing with respect to the relation >, is finite.

A ranking function F of a system S is a function that assigns to each state p of S, a value F.p from a well-founded domain.

The following three theorems state proof obligations for stabilization and weak stabilization. Correctness of these theorems is straightforward.

Theorem 2. *(Closure of P)*

If *for every P state p of a system S and every transition (p, q) of S,*
q is a P state
then *P is closed in S.*

Theorem 3. *(Convergence to P)*

If *there is a ranking function F of a system S such that*
for every state p of S and every transition (p, q) of S
F.p > F.q or q is a P state
then *for every state p of S, every computation of S,*
that starts at p, has a P state.

Theorem 4. *(Weak Convergence to P)*

If *there is a ranking function F of a system S such that*
for every state p of S, there is a transition (p, q) of S where
F.p > F.q or q is a P state
then *for every state p of S, there is a computation of S*
that starts at p and has a P state.

Note that a ranking function for proving stabilization (or convergence) of a system S needs to be decreased by every action of S, whereas a ranking function for proving weak stabilization (or weak convergence) of S needs to be decreased by at least one action of S. Thus, a ranking function for proving stabilization of a system is stricter than one for proving weak stabilization of the same system.

4 Delay Insensitivity

In this section, we discuss how to transform a system by adding a delay to it, and show that in general such a transformation preserves weak stabilization of the system but not its stabilization.

Let v be a variable of a system S. System S can be transformed, by adding a delay to variable v, as follows.

i. Add to system S a new variable dv, whose domain of values is the same as that of v.
ii. Modify every action "g → s" of S by replacing every occurrence of v in the guard g and in the right-hand side of the assignment s by an occurrence of dv.
iii. Add the action "dv /= v↦ dv := v" to system S.

The resulting system after this transformation is denoted S<v>. The added variable dv in the transformed system can be thought of as a delayed version of variable v. The added dv-action in the transformed system can be thought of as the added delay to variable v.

Next, we discuss the effect of adding a delay to some variable of a system on the closed predicates of that system. Let P be a closed predicate of a system S. Is predicate P also closed in the transformed system S<v>? The answer to this question is "no" in general. This is because the added variable dv does not occur in P. Thus, the value of dv can be arbitrary at a P state of system S<v>. Starting from this state and executing an action where some variable that occurs in P is updated using variable dv can yield system S in a state where predicate P no longer holds. This shows that P is not closed in S<v>.

Though a predicate P that is closed in a system S is not necessarily closed in the transformed system S<v>, we show (in Theorem 5 below) that another predicate, related to P, is closed in S<v>. But first we need to introduce the concept of a state predicate being exclusive with respect to some variable in its system.

Let v be a variable of a system S, and P be a state predicate of S. Predicate P is v-exclusive in S iff for every P state p, if the v action in S, if any, is enabled at p, then no other action of S is enabled at p.

Theorem 5.

If *P is a closed state predicate of a system S, and*
P is v-exclusive in S,
then *the state predicate (P ∧ (dv = v ∨ G.v)) is closed in S<v>,*
where G.v is the negation of the disjunction of the guards of all actions, other than the dv action and the v action, in system S<v>.

Proof. Let p be a (P ∧ (dv = v ∨ G.v)) state of system S<v>, and assume that an execution of an action c of S<v> starting at state p yields the system in a state q. We need to show that q is a (P ∧ (dv = v ∨ G.v)) state. There are six cases to consider.

Case 1: *(p is a (P ∧ dv = v) state and c is the dv action)*

This case is not valid because the dv action is not enabled at p.

Case 2: *(p is a (P ∧ dv = v) state and c is the v action)*

In this case, P is true at state q because P is true at state p and P is closed in system S. Also, G.v is true at state p because p is a P state and P is v-exclusive in S. Predicate G.v remains true at state q because v, the only variable updated by action c, does not occur in G.v. Thus, q is a (P ∧ G.v) state.

Case 3: *(p is a (P ∧ dv = v) state and c is any other action)*

In this case, P is true at state q because P is true at state p and P is closed in system S. Also, dv = v at state q because dv = v at state p and neither dv nor v are updated by action c. Thus, q is a (P ∧ dv = v) state.

Case 4: *(p is a (P ∧ G.v) state and c is the dv action)*

In this case, P is true at state q because P is true at state p and dv, the only variable updated by action c, does not occur in P. Also, dv = v at state q because c is the dv-action. Thus, q is a (P ∧ dv = v) state.

Case 5: *(p is a (P ∧ G.v) state and c is the v action)*

In this case, P is true at state q because P is true at state p and P is closed in system S. Also, G.v is true at state p, and G.v remains true at state q because v, the only variable updated by action c, does not occur in G.v. Thus, q is a (P ∧ G.v) state.

Case 6: *(p is a (P ∧ G.v) state and c is any other action)*

This case is not valid because no action, other than the dv action and the v action, can be enabled at p where G.v is true.

Having established that the state predicate (P ∧ (dv = v ∨ G.v)) is closed in the transformed system S<v>, it seems reasonable to ask the following two questions.

Given that system S is stabilizing to P, is the transformed system S<v> stabilizing to (P ∧ (dv = v ∨ G.v))? Given that S is weakly stabilizing to P, is S<v> weakly stabilizing to (P ∧ (dv = v ∨ G.v))? The next two theorems answer these two questions with "not necessarily" and "yes" respectively.

Theorem 6.

If *P is a v-exclusive state predicate in a system S, and*
S is stabilizing to P,
then *S<v> is not necessarily stabilizing to (P ∧ (dv = v ∨ G.v)).*

Proof. We exhibit a system S that is stabilizing to a v-exclusive state predicate P, and show that the transformed system S<v> is not stabilizing to (P ∧ (dv = v ∨ G.v)). Consider a system S that has three binary variables, named "u", "v", and "out", and the following three actions:

1 : u /= v ↦ u := v
2 : v /= u ↦ v := u
3 : u = v → out := u

It is straightforward to show that the state predicate P, defined as (u = v), is v-exclusive in S and that S is stabilizing to P.

Now consider the transformed system S<v>. This system has four binary variables, namely "u", "v", "out", and "dv", and the following four actions:

1 : u /= dv ↦ u := dv
2 : dv /= u ↦ v := u
3 : u = dv → out := u
4 : dv /= v ↦ dv := v

The predicate G.v, which is the negation of the disjunction of the guards of all actions other than the dv action and the v action in S<v>, is defined as follows.

G.v = ¬ (u /= dv ∨ u = dv)
= false

Thus, the state predicate (P ∧ (dv = v ∨ G.v)) is defined as (u = v ∧ dv = v).

To show that the transformed system S<v> is not stabilizing to (u = v ∧ dv = v), it is sufficient to exhibit an infinite computation of S<v> that does not have a state where (u = v ∧ dv = v) holds. Consider a state p of S<v> where (u = v ∧ u /= dv) holds. If system S<v> starts at state p and the four actions 1, then 3, then 4, then 2 are executed repeatedly in this order, then S<v> will never reach a state where (u = v ∧ dv = v) holds. Thus, S<v> is not stabilizing to (u = v ∧ dv = v).

Theorem 7.

If *P is a v-exclusive state predicate in a system S, and*
S is weakly stabilizing to P
then *S<v> is weakly stabilizing to (P ∧ (dv = v ∨ G.v)).*

Proof. By Theorem 5, the state predicate (P ∧ (dv = v ∨ G.v)) is closed in the transformed system S<v>. It remains to be shown that for every state p of S<v>, there is a computation of S<v> that starts at p and has a (P ∧ (dv = v ∨ G.v)) state. Let p be any state of system S<v>, and q be the corresponding state of system S. Thus, the value of each variable of system S at state q equals the value of the corresponding variable of system S<v> at state p. Because S is

weakly stabilizing to P, there is a computation x of S that starts at q and has a P state.

Now consider computation y of S<v> that is generated to track the generation of computation x as follows. First, computation y starts at state p. Second, the sequence of actions of system S<v> that is executed to generate the states in y is the same as the sequence of actions of system S that is executed to generate the states in x, with the following exception. If the values of variables dv and v are different in the last state generated in y, then the next executed action in y is the dv action. Then, the generation of computation y continues to track the generation of x. Because computation x eventually reaches a P state, computation y eventually reaches a (P $\wedge$ dv = v) state. This completes the proof that the transformed system S<v> is weakly stabilizing to (P $\wedge$ (dv = v $\vee$ G.v)).

5 Achieving Stabilization

In the last section, we established that adding delays to a system S preserves the weak stabilization of S, but does not necessarily preserve the stabilization of S. In this section, we establish that weak stabilization is a "good approximation" of stabilization. In particular, we show that under reasonable conditions, namely that the system has a finite number of states and its execution is strongly fair, weak stabilization of the system implies stabilization of the same system.

A computation of a system S is strongly fair iff for every transition (p, q) of system S, if state p occurs infinitely many times in the computation, then transition (p, q) occurs infinitely many times in the computation.

A system S is stabilizing to a state predicate P under strong fairness iff P is closed in S, and for every state p of S, every strongly fair computation of S, that starts at p, has a P state.

Theorem 8.

If *S is a system that has a finite number of states, and*
S is weakly stabilizing to a state predicate P,
then *S is stabilizing to P under strong fairness.*

Proof. Because S is weakly stabilizing to P, P is closed in S. It remains to be shown that for every state p of S, every strongly fair computation of S that starts at p has a P state. Let x be a strongly fair computation, of the form (p.0, p.1, ...), that starts at p (i. e. p = p.0). We need to show that computation x has a P state.

Because S has a finite number of states, at least one state q occurs infinitely many times in computation x. Because S is weakly stabilizing to P, there is a computation (q, q.1, q.2, ...) that starts at q and has a P state. Thus, some q.i in the computation (q, q.1, q.2, ...) is a P state. Because q occurs infinitely many times in the strongly fair computation x, both transition (q, q.1) and state q.1 occur infinitely many times in x. Similarly, because q.1 occurs infinitely many times in x, both transition (q.1, q.2) and state q.2 occur infinitely many times in x. This argument can be extended to show that a P state, namely state q.i,

occurs (infinitely many times) in computation x. This completes the proof that S is stabilizing to P under strong fairness.

6 Theorem of Weak Stabilization

The next three theorems state interesting properties of weak stabilization. Because similar properties hold for stabilization, these theorems serve as further evidence that weak stabilization is a "good approximation" of stabilization.

Theorem 9. *(Base Theorem)*
Each system is weakly stabilizing to the state predicate true.

Theorem 10. *(Theorem of Junction)*

If *S is weakly stabilizing to a state predicate P, and*
S is weakly stabilizing to a state predicate Q,
then *S is weakly stabilizing to (P ∨ Q), and*
S is weakly stabilizing to (P ∧ Q), provided that P ∧ Q /≡false.

Theorem 11. *(Theorem of Weakening)*

If *S is weakly stabilizing to a state predicate P,*
Q is a closed state predicate in S, and
P ⇒ Q in S,
then *S is weakly stabilizing to Q.*

Proofs of these three theorems are rather straightforward. We present here the most interesting proof, namely the proof of the second part of Theorem 10.

Because both P and Q are closed in system S, then (P ∧ Q) is closed in S. It remains to show that for each state p of S, there is a computation that starts at p and has a (P ∧ Q) state. Because S is weakly stabilizing to P, there is a computation (p.0, p.1, ...) that starts at p (i.e. p = p.0) and has a P state p' (i.e. p' = p.i for some i). If p' is a Q state, then the required computation is (p.0, p.1, ...) itself. Otherwise, p' is not a Q state. But because S is also weakly stabilizing to Q, there is a computation (q.0, q.1, ...) that starts at p' (i.e. p' = q.0) and has a Q state q (i.e. q = q.j for some j). Because p' is a P state and P is closed in S, then every state in the computation (q.0, q.1, ...) is a P state. Thus, state q in the computation (q.0, q.1, ...) is a (P ∧ Q) state. In this case, the required computation is (p.0, ..., p.(i-1), p', q.1, ..., q.j, ...). Thus, S is weakly stabilizing to (P ∧ Q).

7 Composition of Weak Stabilization

In this section, we describe a method for composing several weakly stabilizing systems into a single weakly stabilizing system. Note that although there

are methods for composing several stabilizing systems into a single stabilizing system, the methods for composing weakly stabilizing systems seem richer than those for composing stabilizing systems. In particular, the method for composing weakly stabilizing systems described in this section cannot be used to compose stabilizing systems.

A state predicate P of a system S is called a fixed point in S iff every action "g → v := E" of S is such that g = false or v = E at every P state.

Let v and w be two variables of a system S. Variable v is an input of S iff v does not occur in the left-hand side of any assignment (in an action) in S. Variable w is an output of S iff w does not occur in the guard or in the right-hand side of any assignment (in an action) in S. We adopt the notation S[v, w] to denote a system S that has an input variable v and an output variable w.

Two systems S[v, w] and T[w, v] are compatible iff they have no common variables other than v and w.

Two compatible systems S[v, w] and T[w, v] can be composed into a single system, denoted S[v, w] ‖ T[w, v], as follows. First, the set of variables of the composed system is the union of the two sets of variables of systems S[v, w] and T[w, v]. Second, the set of actions of the composed system is the union of the two sets of actions of systems S[v, w] and T[w, v].

Theorem 12.

If *S[v, w] is weakly stabilizing to a fixed point P,*
T[w, v] is weakly stabilizing to a fixed point Q,
S[v, w] and T[w, v] are compatible,
there is one-to-one correspondence between the values of v and the values of w such that
P holds only at the corresponding value pairs, and
Q holds only at the corresponding value pairs,
then *S[v, w] ‖ T[w, v] is weakly stabilizing to (P ∧ Q).*

Proof. Because P is a fixed point in system S, each action "g → v := E" of S is such that g = false or v = E at every P state. Also, because Q is a fixed point in system T, each action "g → v := E" of T is such that g = false or v = E at every Q state. Thus, each action "g → v := E" of system S ‖ T is such that g = false or v = E at each (P ∧ Q) state, and the state predicate (P ∧ Q) is a fixed point in the system S ‖ T. It remains to show that for every state b of system S ‖ T, there is a computation of system S ‖ T that starts at state b and has a (P ∧ Q) state.

Let b be any state of system S ‖ T, and let b|S be the "projection" of state b on system S. Because system S is weakly stabilizing to P, there is a computation x of S that starts at state b|S and has a P state. Thus, there is a computation (b.0, b.1, ...) of system S ‖ T that starts at state b and has the same sequence of actions executed in computation x. Computation (b.0, b.1, ...) has a state c, i.e. b.i = c for some i, such that c|S is a P state that occurs in computation x. Because c|S is a P state, the values of the two variables v and w at state c constitute a corresponding value pair.

Because system T is weakly stabilizing to Q, there is a computation y of T that starts at state c|T and has a Q state. Thus, there is a computation (c.0, c.1, ...) of system S ‖ T that starts at state c and has the same sequence of actions executed in computation y. Computation (c.0, c.1, ...) has a state c.j such that state c.j|T is a Q state that occurs in computation y. Because c.j|T is a Q state, the values of the two variables v and w at state c.j constitute a corresponding value pair. Thus, the values of variables v and w at state c.j is the same as the values of their values at state c. Therefore, c.j is a (P ∧ Q) state. In other words, the computation (b.0, b.1, ..., b.i, c.1, ..., c.j, ...) starts at state b and has a (P ∧ Q) state. This completes our proof that S ‖ T is weakly stabilizing to (P ∧ Q).

8 Concluding Remarks

In this paper, we have introduced the property of weak stabilization and showed that this property is superior to the (strictly stronger) property of stabilization in many respects. First, we showed that the proof obligations for weak stabilization are less severe than those for stabilization. This suggests that verifying weak stabilization is easier than verifying stabilization. Second, we showed that adding delays to a system preserves the weak stabilization properties of that system but does not necessarily preserve the stabilization properties of the system. This suggests that weak stabilizing systems are easier to implement than stabilizing systems. Third, we showed that weakly stabilizing systems that have a finite numbers of states are in fact stabilizing under strong fairness, and showed that weak stabilization satisfies several interesting theorems that are also satisfied by stabilization. This suggests that weak stabilization is a "good approximation" of stabilization. Fourth, we described a method for combining weakly stabilizing systems and argued that this method cannot be used to compose stabilizing systems. This suggests that weak stabilizing systems are easier to compose than stabilizing systems.

References

1. Arora, A., Gouda, M.G.: Delay-Insensitive Stabilization. Proceedings of the Third Workshop on Self-Stabilizing Systems (1997) 95–109
2. Dijkstra, E.W.: Self-Stabilizing Systems in Spite of Distributed Control. Communications of the ACM, Vol. 17 (1974) 643–644
3. Dolev, S.: Self-Stabilization. 1st edn. MIT Press, Cambridge Massachusetts (2000)
4. Flatebo, M., Datta, A.K.: Self-Stabilization in Distributed Systems. In: Casavant, T.L., Singhal, M. (eds.): Readings in Distributed Computing Systems. Lecture Notes in Computer Science, Vol. 1281. Springer-Verlag, Berlin Heidelberg New York (1994) 100–114
5. Gouda, M.G.: The Triumph and Tribulation of System Stabilization. In: Helary, J.M., Raynal, M. (eds.): Proceedings of the International Workshop on Distributed Algorithms. Lecture Notes in Computer Science, Vol. 972. Springer-Verlag, Berlin Heidelberg (1995) 1–18
6. Herman, T.: A Comprehensive Bibliography on Self-Stabilization. `http://www.cs.uiowa.edu/ftp/selfstab/bibliography/2000` (2000)
7. Schneider, M.: Self-Stabilization. ACM Computing Surveys, Vol. 25 (1993) 45–67

On the Security and Vulnerability of PING*

Mohamed G. Gouda[1], Chin-Tser Huang[1], and Anish Arora[2]

[1] Department of Computer Sciences, The University of Texas at Austin
Austin, TX 78712-1188, U.S.A.
{gouda,chuang}@cs.utexas.edu
[2] Department of Computer and Information Science, The Ohio State University
Columbus, OH 43210-1277, U.S.A.
anish@cis.ohio-state.edu

Abstract. We present a formal specification of the PING protocol, and use three concepts of convergence theory, namely closure, convergence, and protection, to show that this protocol is secure against weak adversaries (and insecure against strong ones). We then argue that despite the security of PING against weak adversaries, the natural vulnerability of this protocol (or of any other protocol for that matter) can be exploited by a weak adversary to launch a denial of service attack against any computer that hosts the protocol. Finally, we discuss three mechanisms, namely ingress filtering, hop integrity, and soft firewalls that can be used to prevent denial of service attacks in the Internet.

1 Introduction

Recent intrusion attacks on the Internet, the so called denial of service attacks, have been through the well-known PING protocol [3]. These repeated attacks raise the following two important questions: Is the PING protocol secure? Can the PING protocol be made secure enough to prevent denial of service attacks? In this paper, we use several concepts of convergence theory to answer these two important questions. In particular, we use the three concepts of closure, convergence, and protection to show that the PING protocol is in fact secure. We also argue that the PING protocol cannot be secure enough to prevent denial of service attacks. An adversary can always exploit the natural vulnerability of any (possibly secure) protocol to launch a denial of service attack against any computer that hosts this protocol. We briefly discuss several techniques that can be used to safeguard the computers in any network against denial of service attacks on that network.

The PING protocol in this paper is specified using a version of the Abstract Protocol Notation presented in [6]. Using this notation, each process in a protocol is defined by a set of constants, a set of variables, a set of parameters, and a set of actions. For example, in a protocol consisting of two processes p and q and two channels (one from p to q and one from q to p), process p can be defined as follows.

* This work is supported in part by DARPA contract F33615-01-C-1901.

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 124–135, 2001.

```
process p
const   <name of constant>  :   <type of constant>
        ...
        <name of constant>  :   <type of constant>
var     <name of variable>  :   <type of variable>
        ...
        <name of variable>  :   <type of variable>
par     <name of parameter> :   <type of parameter>
        ...
        <name of parameter> :   <type of parameter>
begin
        <action>
[]      <action>
        ...
[]      <action>
end
```

The constants of process p have fixed values. The variables of process p can be read and updated by the actions of process p. Comments can be added anywhere in a process definition; each comment is placed between the two brackets { and }.

Each `<action>` of process p is of the form:

`<guard> -> <statement>`

The guard of an action of p is one of the following three forms: a boolean expression over the constants and variables of p, a receive guard of the form `rcv <message> from q`, or a timeout guard that contains a boolean expression over the constants and variables of every process and the contents of the two channels in the protocol.

Each parameter declared in a process is used to write a finite set of actions as one action, with one action for each possible value of the parameter. For example, if process p has the following variable x and parameter i,

```
var     x :   0 .. n-1
par     i :   0 .. n-1
```

then the following action in process p

```
        x = i   ->   x := x + i
```

is a shorthand notation for the following n actions.

```
        x = 0   ->   x := x + 0
[]      ...
[]      x = n-1 ->   x := x + n-1
```

Executing an action consists of executing the statement of this action. Executing the actions of different processes in a protocol proceeds according to the following three rules. First, an action is executed only when its guard is true. Second, the actions in a protocol are executed one at a time. Third, an action whose guard is continuously true is eventually executed.

The `<statement>` of an action of process p is a sequence of `<skip>`, `<send>`, `<assignment>`, `<selection>`, or `<iteration>` statements of the following forms:

```
<skip>          :     skip
<send>          :     send <message> to q
<assignment>    :     <variable in p>          :=   <expression>
<selection>     :     if <boolean expression>  ->   <statement>
                      ...
                      [] <boolean expression>  ->   <statement>
                      fi
<iteration>     :     do <boolean expression>  ->   <statement>
                      od
```

Executing an action of process p can cause a message to be sent to process q. There are two channels between the two processes: one is from p to q, and the other is from q to p. Each sent message from p to q remains in the channel from p to q until it is eventually received by process q or is lost. Messages that reside simultaneously in a channel form a set and so they are received or lost, one at a time, in any order and not necessarily in the same order in which they were sent.

2 The PING Protocol

The PING protocol (which stands for the Packet Internet Groper protocol) allows a computer in the Internet to test whether a specified computer in the Internet is up [9]. The test is carried out as follows. First, the testing computer p sends several echo request messages to the computer q[i] being tested. Second, the testing computer p waits to receive one or more echo reply messages from computer q[i]. Third, if the testing computer p receives one or more echo reply messages from computer q[i], p concludes that computer q[i] is up. On the other hand, if the testing computer p receives no echo reply message from computer q[i], p concludes that q[i] may not be up. The conclusion in this case is not certain because it is possible (though unlikely) that all the echo request messages sent from p to q[i] or the corresponding echo reply messages from q[i] to p are lost during transmission.

The testing computer p stores the test results in a local variable array named up that is declared as follows.

```
var     up :    array [0 .. n-1] of boolean
```

Note that n is the number of computers being tested. If at the end of a session of the PING protocol, up[i] = true in computer p, then computer q[i] was up sometime during that session. On the other hand, if at the end of a session, up[i] = false in computer p, then no firm conclusions can be reached (but it is likely that q[i] was down during that session).

For computer p to ensure that a received echo reply message from a computer q[i] corresponds to the echo request message that p has sent earlier to q[i], p adds a random identifier id[i] to all the echo request messages that p sends to q[i] in a session of the PING protocol. When a computer q[i] receives an erqst(id[i]) message from computer p, q[i] replies by sending an erply(id[i]) message to p.

When computer p receives an erply(id[i]) message, p checks whether id[i] is the random identifier of the current protocol session with q[i]. If so, p accepts the message and assigns the corresponding up[i] element the value true. Otherwise, p discards the message.

The process of the testing computer p in the PING protocol is defined as follows.

```
process p
const   n, idmax, cmax
var     up :    array [0 .. n-1] of boolean
        wait :  array [0 .. n-1] of boolean
        id :    array [0 .. n-1] of 0 .. idmax
        x :     0 .. idmax
        c :     0 .. cmax
par     i :     0 .. n-1
begin
        ~wait[i] ->     up[i] := false;
                        id[i] := random;
                        c := 0;
                        do (c < cmax) ->
                                send erqst(id[i]) to q[i];
                                c := c + 1;
                        od;
                        wait[i] := true

[]      rcv erply(x) from q[i] ->
                        if   wait[i]  ^  x = id[i]  ->
                                up[i] := true
                        [] ~ wait[i]  v  x <> id[i] ->
                                {discard erply} skip
                        fi

[]      timeout  ( wait[i] ^
         erqst(id[i])#ch.p.q[i] + erply(id[i])#ch.q[i].p = 0 ) ->
                        wait[i] := false
end
```

Process p has three actions. In the first action, p recognizes that it is no longer waiting for any erply messages from its last session of the protocol with q[i], and starts its next session with q[i]. Process p starts the next session with q[i] by selecting a new random identifier id[i] for the new session and sending cmax many erqst(id[i]) messages to process q[i]. In the second action, process p receives an erply(id[i]) message from any process q[i] and decides whether to accept the message and assign up[i] the value true, or discard the message. In the third action, process p recognizes that a long time has passed since p has sent the erqst(id[i]) messages in the current session, and so the number of erqst(id[i]) messages in the channel from p to q[i], denoted erqst(id[i])#ch.p.q[i], is zero and the number of erply(id[i]) messages in the channel from q[i] to p, denoted

erply(id[i])#ch.q[i].p, is also zero. In this case, p terminates the current session with q[i] by assigning variable wait[i] the value false.

The process for any computer q[i] being tested is defined as follows.

```
process q[i: 0 .. n-1]
const   n, idmax
input   up :    boolean
var     x :     0 .. idmax
begin
        rcv erqst(x) from p ->
                        if   up ->    send erply(x) to p
                        [] ~ up ->    skip
                        fi

end
```

Process q[i] has a boolean input named up that describes the current state of q[i]. Clearly, the value of input up can change over time to reflect the change in the state of q[i]. Nevertheless, to keep our analysis of the PING protocol simple, we assume that the value of input up remains constant.

Process q[i] has only one action. In this action, q[i] receives an erqst(x) message from p and either sends an erply(x) message to p (if input up in q[i] is true), or discards the received erqst(x) message (if input up in q[i] is false).

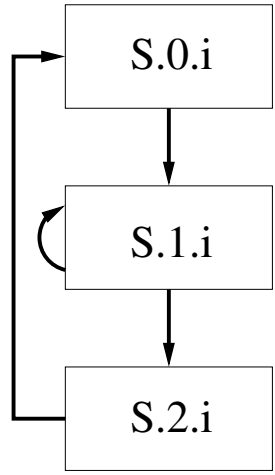


Fig. 1. State transition diagram of PING.

The state transition diagram for the PING protocol is shown in Figure 1. There are three nodes in this diagram. Each of the three nodes represents a set of states of the PING protocol. Each node v is labeled with a state predicate S.v.i, whose value is true at every state represented by node v. The three state

predicates in the state transition diagram, namely S.0.i, S.1.i, and S.2.i, are defined as follows

$$\begin{aligned} \text{S.0.i} &= \sim \text{wait[i]} \wedge \text{B.i} = 0 \wedge \text{C.i} = 0 \\ \text{S.1.i} &= \text{wait[i]} \wedge \text{B.i} > 0 \wedge \text{C.i} = 0 \wedge \text{X.i} \wedge \text{Y.i} \\ \text{S.2.i} &= \text{wait[i]} \wedge \text{B.i} = 0 \wedge \text{C.i} = 0 \wedge \text{Y.i} \end{aligned}$$

where

$\text{B.i} = \text{erqst(id[i])\#ch.p.q[i]} + \text{erply(id[i])\#ch.q[i].p}$
B.i is the number of messages (in the two channels between p and q[i]) whose identifiers are equal to the value of variable id[i] in p.

$\text{C.i} = \sum_{\text{r} \neq \text{id}} (\text{erqst(r)\#ch.p.q[i]} + \text{erply(r)\#ch.q[i].p})$
C.i is the number of messages (in the two channels between p and q[i]) whose identifiers are different from the value of variable id[i] in p.

$\text{X.i} = (\ (\text{erply(id[i])\#ch.q[i].p} > 0) \Rightarrow (\text{up} = \text{true in q[i]})\ )$
X.i states that if there is one or more erply(id[i]) message in the channel from a process q[i] to process p, then input up in q[i] has the value true.

$\text{Y.i} = (\ (\text{up[i]} = \text{true in p}) \Rightarrow (\text{up} = \text{true in q[i]})\ )$
Y.i states that if an element up[i] in process p has the value true, then input up in process q[i] has the value true.

The directed edges in the state transition diagram in Figure 1 represent executions of actions in processes p and q[i]. The directed edge from node S.0.i to node S.1.i represents an execution of the first action in process p. The directed edge from node S.1.i to node S.2.i represents an execution of the second action in process p. The directed edge from node S.2.i to node S.0.i represents execution of the third action in process p. The self-loop at node S.1.i represents any of the following: an execution of the action in process q[i], an execution of the second action in process p, and a loss of one message from one of the two channels between p and q[i].

3 The PING Adversary

For two reasons, the PING protocol is designed to overcome the activities of a weak adversary, rather than a strong adversary. First, this assumption keeps the PING protocol, which performs a basic task (of allowing any computer to test whether another computer in the Internet is up) both simple and efficient. Second, by disrupting the PING protocol, a strong adversary achieves very little: merely convincing one computer that another computer in the Internet is up when in fact that other computer is not up. It is not clear what does a strong adversary gain by such disruption, and so it is doubtful that a strong adversary will attempt to use its strength to disrupt the PING protocol.

The weak adversary considered in designing the PING protocol is one that can insert a finite number of erqst(x) messages into the channel from process p to a process q[i], and can insert a finite number of erply(x) messages into the channel from a process q[i] to process p. Identifiers of the messages inserted by

this adversary at any instant are different from the identifiers of the current session and any future session of the PING protocol. Thus, if the adversary inserts an erply(x) message in the channel from process q[i] to process p at some time instant, and if p receives this message at some future instant, then p can still detect that x is different from the current value of variable id and discard the message. For convenience, we refer to every message whose identifier is different from the identifiers of the current session and every future session as an adversary message.

Figure 2 shows the state transition diagram that describes the activities of both the PING protocol and its (weak) adversary. Note that this diagram has three additional nodes over the diagram in Figure 1. These three nodes are labeled with the state predicates U.0.i, U.1.i, and U.2.i defined as follows.

$$\begin{aligned} \text{U.0.i} &= \sim \text{wait[i]} \wedge \text{B.i} = 0 \wedge \text{C.i} > 0 \\ \text{U.1.i} &= \text{wait[i]} \wedge \text{B.i} > 0 \wedge \text{C.i} > 0 \wedge \text{X.i} \wedge \text{Y.i} \\ \text{U.2.i} &= \text{wait[i]} \wedge \text{B.i} = 0 \wedge \text{C.i} > 0 \wedge \text{Y.i} \end{aligned}$$

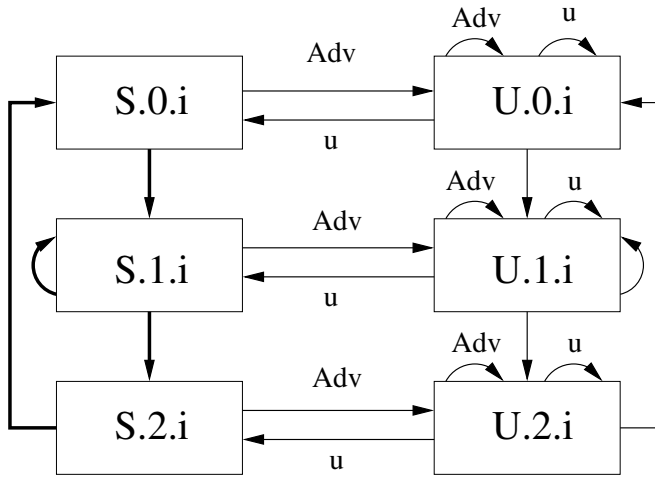


Fig. 2. State transition diagram of PING and adversary.

Note that B.i, C.i, X.i, and Y.i are defined above in Section 2. Note also that each predicate U.v.i is the same as the corresponding predicate S.v.i except that the conjunct C.i = 0 in S.v.i is replaced by the conjunct C.i > 0 in U.v.i. Thus, each U.v.i state is the same as a corresponding S.v.i state except that some adversary messages are inserted into some channels in the protocol.

In the state transition diagram in Figure 2, each edge labeled "Adv" represents an adversary action where one or more adversary messages are inserted into some channels in the protocol. Each edge or self-loop labeled "u" represents an execution of some protocol action where an adversary message is either received by a process q[i] (and another adversary message is sent from q[i] to p) or received by process p (and discarded).

Note that, despite the adversary involvement, the state predicate Y.i holds at every S.2.i state and every U.2.i state. Thus, Y.i holds at the end of every session between process p and process q[i] of the PING protocol.

4 Security of PING

In this section, we use three concepts of the theory of convergence [5], namely closure, convergence, and protection, to show that the PING protocol (presented in Section 2) is secure against the weak adversary (presented in Section 3). In general, to show that a protocol P is secure against an adversary D, one needs to partition the reachable states of P into safe states and unsafe states, then identify the critical variables of P (those that need to be protected from the actions of D), and show that the following three conditions hold ([1] and [7]).

i. **Closure**:
The set of safe states is closed under any execution of a P action, and the set of reachable states (i.e. the union of the safe state set and the unsafe state set) is closed under any execution of a P action or a D action.

ii. **Convergence**:
Starting from any unsafe state, any infinite execution of the P actions leads P to safe states.

iii. **Protection**:
If an execution of a P action starting at an unsafe state s changes the values of the critical variables of P from V to V', then there is a safe state s' such that the values of the critical variables in s equals to V, and execution of the same action starting at s changes the values of the critical variables of P from V to V'. (Note that this condition is a generalization of the corresponding condition in [7] which states that each execution of a P action starting at an unsafe state cannot change the values of the critical variables of P.)

Following this definition, the security of the PING protocol can be established by identifying the safe, unsafe, and reachable states of the protocol, then identifying its critical variables, and finally showing that the PING protocol satisfies the three conditions of closure, convergence, and protection.

The safe states of the PING protocol are specified by the state predicate S.i, where

$$S.i = S.0.i \vee S.1.i \vee S.2.i$$

The unsafe states of PING are specified by the state predicate U.i, where

$$U.i = U.0.i \vee U.1.i \vee U.2.i$$

Thus, the reachable states of the protocol are specified by the state predicate $S.i \vee U.i$.

The PING protocol has only one critical variable, namely array up in process p. It remains now to show that the protocol satisfies the above three conditions of closure, convergence, and protection.

Satisfying the Closure Condition: From the state transition diagram in Figure 1, the set of safe states is closed under any execution of an action of the PING protocol. From the state transition diagram in Figure 2, the set of reachable states is closed under any execution of an action of the PING protocol or an action of the weak adversary.

Satisfying the Convergence Condition: Along any infinite execution of the actions of the PING protocol, the following three conditions hold.

i. No adversary message is added to the channel from process p to a process q[i].
ii. Each adversary message in a channel from process p to a process q[i] is eventually discarded (if up = false in q[i]), or replaced by another adversary message in the channel from q[i] to p (if up = true in q[i]).
iii. Each adversary message in a channel from a process q[i] to process p is eventually discarded.

Thus, starting from an unsafe U.i state, any infinite execution of the actions of the PING protocol leads the protocol to a safe S.i state where no channel has adversary messages.

Satisfying the Protection Condition: Assume that an execution of an action of the PING protocol starting at an unsafe state s changes the value of array up in process p. Then, the executed action is one where process p receives an erply(x) message from a process q[i], where x = id. Thus, the received erply(x) message is not an adversary message, and receiving this message causes the value of element up[i] in p to change from false to true. Let s' be the state that results from removing all the adversary messages that exist in state s. From the state transition diagram in Figure 2, state s' is a safe state. At state s', message erply(x) is still in the channel from process q[i] to process p. Thus, executing the action where process p receives the erply(x) message, starting at state s', causes the value of element up[i] in p to change from false to true. This completes our proof of the security of the PING protocol against the weak adversary.

It is also straightforward to show that the PING protocol is not secure against a strong adversary that can insert messages whose identifiers are equal to the identifier of the current session. Consider an unsafe protocol state s where the adversary has inserted an erply(x) message, where x = id, at the channel from a process q[i], whose input up is false, to process p. Executing the action where process p receives the inserted erply(x) message, starting at the unsafe state s, changes the value of element up[i] in p from false to true (even though the value of input up in q[i] is false). Because no action execution, starting at any safe state, changes the value of element up[i] in p from false to true (given that the value of input up in q[i] is false), then the protection condition does not hold. This argument shows that PING is not secure against a strong adversary.

5 Vulnerability of PING

Security of the PING protocol against the weak adversary is established in the last section by showing that the protocol satisfies the three conditions of closure, convergence, and protection. The closure condition states that the unsafe states (specified by U.i) are the furthest that the adversary can lead the protocol away from its safe states (specified by S.i). The convergence condition states that, when the adversary stops inserting adversary messages into the protocol channels, the PING protocol eventually converges from its current unsafe state to the safe states. The protection condition states that while the protocol is in its unsafe states (due to the influence of the adversary), the critical array "up" in process p is updated as if the protocol is in a safe state.

Despite the security of PING against its weak adversary, the weak adversary can exploit the natural vulnerability of the PING protocol to attack any computer that hosts PING as follows. The adversary inserts a very large number of adversary messages into the protocol channels. The protocol processes p and q[i: 0..n-1] become very busy processing and eventually discarding these messages. Thus, the computers that host these processes become very busy and unable to perform any other service. Such an attack is usually referred to as a denial of service attack.

It follows that denial of service attacks by weak adversaries can succeed by exploiting the natural vulnerability of any (even the most secure) protocols. The only way to prevent denial of service attacks is to prevent the adversary messages from reaching the protocol processes. In other words, the adversary messages need to be detected as such and discarded before they reach their destination processes. The question now is how to detect the adversary messages?

The answer to this question, in the case of the PING protocol, is straightforward. In the known denial of service attacks that exploit the PING protocol, the adversary inserts messages whose source addresses are wrong. Because the source of the inserted messages is the adversary itself, the source address in each of these messages should have been the address of the adversary. However, the source address in each inserted erqst(x) message is recorded to be the address of p so that the reply to the message is sent to p. Also, the source address in each inserted erply(x) message is recorded to be the address of some q[i] in order to hide the identity of the adversary.

6 Preventing Denial of Service Attacks

To prevent denial of service attacks, the routers in the Internet need to be modified to perform the following task: detect and discard any message whose source address is wrong. Note that this task can prevent any adversary from exploiting the natural vulnerability of any protocol, not only PING, to launch a denial of service attack against any host of that protocol. This task can be achieved using two complementary mechanisms named "Ingress Filtering" [4] and "Hop Integrity" [8]. These two complementary mechanisms can be described as follows:

i. **Ingress Filtering**:
A router that receives a message, supposedly from an adjacent host H, forwards the message only if the source address recorded in the message is that of H.

ii. **Hop Integrity**:
A router that receives a message, supposedly from an adjacent router R, forwards the message only after it checks that the message was indeed sent by R.

These two mechanisms can be used together to detect and discard any message, whose source address is wrong, that is inserted by a weak or strong adversary into the Internet. Thus, a large percentage of denial of service attacks can be prevented.

Another mechanism for detecting and discarding adversary messages is called soft firewalls. A soft firewall for a process p is another process fp that satisfies the following three conditions:

i. **Output Observation**:
Each message that process p intends for another process q is first sent to the firewall process fp before it is forwarded to process q.

ii. **Input Observation**:
Each message from another process q intended for process p is first sent to the firewall process fp before it is forwarded to process p.

iii. **Input Filtering**:
The firewall process fp maintains a coarse image of the local state of process p, and uses this image to detect and discard any inappropriate message intended for p from any other process or from any adversary.

(Note that the soft firewall processes described here are similar to stateless firewall processes described in [2], with one exception. A stateless firewall does not maintain any image of the local state of the process behind the firewall, whereas a soft firewall process maintains a soft state image of the local state of the process behind the firewall.)

A possible soft firewall for process p in the PING protocol is a process fp that maintains one bit "w" as a coarse state for array "wait" in process p. Whenever fp receives an erqst(x) message from process p intended for process q[i], fp assigns its bit w the value 1. Process fp keeps the value of bit w "1" for one minute, since fp received the last erqst(x) message from p, then fp assigns bit w the value "0". (The one minute is an ample time for the sent erqst(x) message to reach the intended q[i] and for the resulting erply(x) to return from q[i] to p.) Whenever the firewall process fp receives an erply(x) message intended for p, fp checks the current value of bit w and forwards the received erply(x) message to process p only if the value of bit w is "1". Thus, all erply(x) messages, that are generated by the weak adversary, are discarded by fp before they reach process p (as long as p itself does not send erqst(x) messages to any q[i]).

7 Concluding Remarks

Our objective in this paper is three-fold. First, we want to demonstrate the utility of a new definition of system security [7] that is based on the three concepts of closure, convergence, and protection of convergence theory. Our demonstration show that the concept of protection as presented in [7] is too strong, and suggest a sensible weakening of this concept, discussed in Section 4. Second, we want to formally show that the PING protocol is secure (against a weak adversary), despite the fact that this protocol has been used repeatedly to launch denial of service attacks against several computers in the Internet. Our proof, based on the two state transition diagrams in Figures 1 and 2, is both simple and straightforward. Third, we want to make the point that every protocol (whether secure or unsecure) has a natural vulnerability that can be exploited by (possibly weak) adversaries to attack any computer that hosts this protocol. Such attacks can be foiled, not by making protocols more secure which is impossible, but by detecting the attacking messages early on and discarding them promptly. In this regard, the ideas of ingress filtering, hop integrity, and soft firewalls offer much hope.

References

1. Arora, A., Gouda, M.G.: Closure and convergence: A foundation for fault-tolerant computing. IEEE Transactions on Software Engineering, Vol. 19, No. 3 (1993) 1015–1027
2. Cheswick, W.R., Bellovin, S.M.: Firewalls and Internet Security. 1st edn. Addison-Wesley Publishing Co., Reading, Massachusetts (1994)
3. CERT Advisory: Smurf IP Denial-of-Service Attacks. CERT Advisory CA-1998-01, `http://www.cert.org/` (1998)
4. Ferguson, P., Senie, D.: Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2267 (1998)
5. Dolev, S.: Self-Stabilization. 1st edn. MIT Press, Cambridge Massachusetts (2000)
6. Gouda, M.G.: Elements of Network Protocol Design. 1st edn. John Wiley & Sons, New York, New York (1998)
7. Gouda, M.G.: Elements of security: Closure, convergence, and protection. Information Processing Letters, Vol. 77, Nos. 2–4 (2001) 109–114
8. Gouda, M.G., Elnozahy, E.N., Huang, C.-T., McGuire, T.M.: Hop Integrity in Computer Networks. Proceedings of the 8th IEEE International Conference on Network Protocols (2000) 3–11
9. Postel, J.: Internet Control Message Protocol. RFC 792 (1981)

A New Efficient Tool for the Design of Self-Stabilizing ℓ-Exclusion Algorithms: The Controller

Rachid Hadid* and Vincent Villain

LaRIA, Université de Picardie Jules Verne
5, rue de Moulin Neuf, 80000 Amiens, France
Tel.: (33) 3 22 82 88 76, Fax: (33) 3 22 82 75 02
villain@laria.u-picardie.fr

Abstract. In this paper, we present the first self-stabilizing protocols for ℓ-exclusion problem in the message passing model. The ℓ-exclusion problem is a generalization of the mutual exclusion problem—we allow ℓ ($\ell \geq 1$) processors, instead of 1, to use a shared resource. We propose a new technique for the design of self-stabilizing ℓ-exclusion: the controller. This tool allows to count tokens of the system without any counter variable for all processors except one called Root. We also introduce a new protocol composition called *parametric composition*. Then we present protocols on rings and on trees. The space requirement of both algorithms is independent of ℓ for all processors except Root. The stabilization time of the first protocol is $3n$ time, where n is the ring size and the stabilization time of the second one is $6h + 2$ time, where h is the tree height.

1 Introduction

Fault-tolerance is one of the most important requirements of modern distributed systems. Various types of faults are likely to occur at various parts of the system. The distributed systems go through the transient faults because they are exposed to constant change of their environment. The concept of self-stabilization [7] is the most general technique to design a system to tolerate arbitrary transient faults. A self-stabilizing system, regardless of the initial states of the processors and initial messages in the links, is guaranteed to converge to the intended behavior in finite time. In 1974, Dijkstra introduced the property of self-stabilization in distributed systems and applied it to algorithms for mutual exclusion [7]. In the mutual exclusion problem, there is an activity, that of executing a critical section of code, that only one process can do at a time. The *ℓ-exclusion problem* is a generalization of the mutual exclusion problem—ℓ processors are now allowed to execute the critical section concurrently. Algorithms for ℓ-exclusion are given in [2, 6, 10, 12]. The problem was first defined and solved by Fischer, Lynch,

* This work was supported in part by the Pôle de Modélisation de Picardie, France, and the Fonds Social Européen.

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 136–151, 2001.

Burns, and Borodin [9]. The first self-stabilizing algorithm for the ℓ-exclusion problem was presented in [8]. This solution is a generalization of Dijkstra's algorithm [7]. The algorithm in [3] is the second self-stabilizing solution to the ℓ-exclusion problem, but in the shared memory model. In both cases the space requirement depends on the size of the network and ℓ. Algorithms in [3] and [8] require at least $O(2^n)$ and $\Omega(n\ell)$ states per process, respectively, where n is the size of the network. The first attempt to solve this problem with a space complexity independent of n (and almost independent of ℓ) are presented in [15] (on chains), and [11] (on trees). All those algorithms run in the state model ([8,11,15]), or in the shared memory model ([3]).

Contributions. In this paper, we present the first self-stabilizing protocols for ℓ-exclusion problem in the message passing model. We propose a new technique for the design of self-stabilizing ℓ-exclusion: the controller. This tool allows to count tokens of the system without any counter variable for all processors except one called Root. We also introduce a new protocol composition called *parametric composition*. The parametric composition of protocols P_1 and P_2 allows interactions both from P_1 to P_2 and from P_2 to P_1. This is a generalization of the *collateral composition* in [13] and of the *conditional composition* in [5]. Using the *controller* on uni-directional rings, a simple circulation of ℓ tokens, and the parametric composition we design a self-stabilizing ℓ-exclusion protocol on rings. Using the *controller* on trees, a simple distribution of ℓ tokens, and the parametric composition we then design a self-stabilizing ℓ-exclusion protocol on trees. The space requirement of both algorithms is independent of ℓ for all processors except Root. The stabilization time of the first protocol is $3n$ time, where n is the ring size and the stabilization time of the second one is $6h + 2$ time, where h is the tree height. Using the power of both the controller and the parametric composition, we also discuss some adaptations of these protocols.

Outline of the Paper. In Section 2, we describe the distributed system, the model we use in this paper, and also, state the specification of the problem solved in this paper. In Section 3 we present the parametric composition used to write our algorithms. In Section 4, we present a self-stabilizing ℓ-exclusion protocol on rings. Then we present an implementation of this solution on tree networks in Section 5. Finally, we make some concluding remarks in Section 6.

2 Preliminaries

2.1 Model

Distributed systems we consider in this paper are asynchronous networks. We number the processors from 0 to n-1 for ease of notation only. We assume there is a distinguished processor (processor 0) that we often refer to as Root. Processors communicate with their neighbors by sending them *messages*. We assume that the time of message transit is finite but not bounded and as long as a message is not processed, we consider that it is in transit. Moreover, each link is assumed

to be bounded, messages transmitted over the links are not lost and arrive error free and in the order sent (FIFO) during and after the stabilization phase. We consider *semi-uniform* protocols [6]. So, every processor with the same degree executes the same protocol, except one processor, Root. The protocol consists of a collection of actions. An action is of the form: $< guard > \longrightarrow < statement >$. A *guard* is a boolean expression over the variables of the processor and/or an *input* message. A *statement* is a sequence of assignments and/or message sendings. An action can be executed only if its guard evaluates to true. When several actions of a processor are simultaneously enabled then the first in the text of the protocol is executed only. The *state* of a processor is defined by the values of its variables. The *state* of a system is a vector of $n+1$ components where the first n components represent the state of n processors, and the last one refers to the multiset of messages in transit in the links. In the sequel, we refer to the state of a processor and the system as a (*local*) *state* and *configuration*, respectively. Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\rightarrow$, on $\mathcal{C}$, the set of all possible configurations of the system. A *computation* of a protocol $\mathcal{P}$ is a *maximal* sequence of configurations $e = \gamma_0, \gamma_1, ..., \gamma_i, \gamma_{i+1}, ...$, such that for $i \geq 0, \gamma_i \rightarrow \gamma_{i+1}$ (a single *computation step*) if γ_{i+1} exists, or γ_i is a terminal configuration.

2.2 Self-Stabilization

Definition 1 (Self-Stabilization). *A protocol $\mathcal{P}$ is self-stabilizing for a specification $\mathcal{SP}$ (predicate over the computations) if and only if every execution starting from an arbitrary configuration will eventually reach (convergence property) a configuration from which it satisfies $\mathcal{SP}$ forever (closure property).*

In practice, we associate to $\mathcal{P}$ a predicate on the system configurations, denoted $\mathcal{L}_{\mathcal{P}}$ and called the *legitimacy predicate*. We define $\mathcal{L}_{\mathcal{P}}$ as follows: starting from a configuration α satisfying $\mathcal{L}_{\mathcal{P}}$, $\mathcal{P}$ always behaves according to $\mathcal{SP}$, and any configuration reachable from α satisfies $\mathcal{L}_{\mathcal{P}}$ (closure property). Moreover if any execution of $\mathcal{P}$ starting from an arbitrary configuration eventually reaches a configuration satisfying $\mathcal{L}_{\mathcal{P}}$ (convergence property), we say that $\mathcal{P}$ stabilizes for $\mathcal{L}_{\mathcal{P}}$ (hence for $\mathcal{SP}$).

2.3 ℓ-Exclusion

Specification of the ℓ-Exclusion Protocol.
Safety. In any computation e, at most ℓ processors can execute the critical section concurrently.
Liveness.
1. **Fairness.** In any computation e, each requesting processor can enter the critical section in a finite time.
2. **ℓ-Liveness.** In any computation e, if $x < \ell$ processors execute the critical section forever and some other processors are requesting the critical section, then eventually at least another processor will eventually enter the critical section.

Self-Stabilizing ℓ-Exclusion Protocol. An ℓ-exclusion algorithm is self-stabilizing if every computation starting from an arbitrary initial configuration, eventually satisfies the above safety and liveness requirements.

3 Parametric Composition

The parametric composition of protocols P_1 and P_2 is a generalization of the *collateral composition* in [13] because it allows not only P_2 to read the variables written by P_1 but also P_1 to read the variables written by P_2. This is also a generalization of the *conditional composition* in [5] because it allows not only P_2 to use the predicates of P_1 but also P_1 to use the predicates of P_2. Informally, P_1 can be seen as a tool used by P_2, where P_1 call some "public" functions of P_2 (we use the term *function* with a generic meaning: it can be the variables used in the collateral composition or the predicates as in the conditional composition...), and P_2 can also use some functions of P_1 through the medium of parameters.

Definition 2 (Parametric Composition). *Let P_1 be a protocol with* parameters *and a* public *part. Let P_2 be a protocol such that P_2 uses P_1 as an external protocol. By the* parameters *P_2 allows P_1 to use some of its functions (function may return no result). By the* public part *protocol P_1 allows protocol P_2 to call some of its functions. The* parametric composition *of P_1 and P_2, denoted $P_1 \triangleright_{\mathsf{P}} P_2$, is a protocol that has all the variables and all the actions of P_1 and P_2.*

The implementation scheme of P_1 and P_2 is given in Algorithm 1. Protocol P_2 allows P_1 to use functions F_1,...,F_α (called $\mathsf{F'}_1$,...,$\mathsf{F'}_\alpha$ in P_1) and P_1 allows P_2 to use its public functions Pub_1,...,Pub_β (called $\mathsf{P}_1.\mathsf{Pub}_1$,..., $\mathsf{P}_1.\mathsf{Pub}_\beta$ in P_2)

Algorithm 1 $P_1 \triangleright_{\mathsf{P}} P_2$

```
Protocol P1( F'1 : TF1, F'2 : TF2, ..., F'α : TFα);
Public
    Pub1 : TP1
        /* definition of Function Pub1 */
    ...
    Pubβ : TPβ
        /* definition of Function Pubβ */

begin
...
[]  < Guard > ⟶ < statement >
      /* Functions F'i can be used
      in Guard and/or statement */
...
end
```

```
Protocol P2
External Protocol P1(F1, F2,...,Fα);
Parameters
        F1 : TF1
            /* definition of Function F1 */
        ...
        Fα : TFα
            /* definition of Function Fα */

begin
...
[]  < Guard > ⟶ < statement >
      /* Functions P1.Pubi can be used
      in Guard and/or statement */
...
end
```

Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be predicates over the variables of P_1 and P_2, respectively. We now define a *fair* composition w.r.t. both protocols and define what it means for a parametric composite algorithm to be self-stabilizing.

Definition 3 (Fair Execution). *An execution e of the composition of P_1 and P_2 is fair w.r.t. P_i ($i \in \{1, 2\}$) if one of these conditions holds:*

1. *e is finite.*
2. *e contains infinitely many steps of P_i, or contains an infinite suffix in which no step of P_i is enabled.*

Definition 4 (Fair Composition). *The composition of P_1 and P_2 is fair w.r.t. P_i ($i \in \{1, 2\}$) if any execution of the composition of P_1 and P_2 is fair w.r.t. P_i.*

The following composition theorem and its corollary are obvious:

Theorem 1. *If the following four conditions hold:*

1. *composition is fair w.r.t. P_1,*
2. *composition is fair w.r.t. P_2 if P_1 is stabilized for $\mathcal{L}_1$,*
3. *protocol P_1 stabilizes for $\mathcal{L}_1$ even if P_2 is not stabilized for $\mathcal{L}_2$, and*
4. *protocol P_2 stabilizes for $\mathcal{L}_2$ if $\mathcal{L}_1$ is satisfied,*

then $P_1 \triangleright_{\mathsf{P}} P_2$ stabilizes for $\mathcal{L}_1 \wedge \mathcal{L}_2$.

Corollary 1. *Let $P_1 \triangleright_{\mathsf{P}} P_2$ be a self-stabilizing protocol. If Protocol P_1 stabilizes in t_1 time for $\mathcal{L}_1$ even if P_2 is not stabilized for $\mathcal{L}_2$ and Protocol P_2 stabilizes in t_2 time for $\mathcal{L}_2$ when P_1 is stabilized for $\mathcal{L}_1$, then $P_1 \triangleright_{\mathsf{P}} P_2$ stabilizes for $\mathcal{L}_1 \wedge \mathcal{L}_2$ in $t_1 + t_2$ time.*

4 Self-Stabilizing ℓ-Exclusion Protocol on Rings

In this section, we present a self-stabilizing ℓ-exclusion protocol on *unidirectional* rings. Each processor can distinguish its two neighbors: the left neighbor from which it can receive messages, and the right neighbor to which it can send messages. For a processor i the left neighbor (predecessor) is the processor i-1 and the right neighbor (successor) is the processor i+1, where indices are modulo n. The protocol we present is based on the concept of tokens: each time a requesting processor owns a token, it can enter the critical section. A simple solution is a circulation of ℓ tokens such that no processor can keep more than one token while it is in critical section. We use a *controller mechanism* to ensure that the number of tokens will eventually be equal to ℓ. The protocol we propose is a parametric composition of two protocols: *ℓ-Token-Circulation* (see Algorithm 3) and *Ring-Controller* (see Algorithm 2). In the rest of the paper we call *E-tokens* (with E for Exclusion) the tokens used by ℓ-Token-Circulation and *C-token* (with C for Control) the token used by Ring-Controller.

4.1 Ring-Controller

Ring-Controller allows to count the E-tokens of the system without any counter variable for all processors except Root. Ring-Controller uses a single token (the C-token) circulation. The C-token continuously makes ring traversals. At the beginning of each traversal, Ring-Controller informs Root that it can both end the current E-token counting and start a new counting (see Function START in ℓ-Token-Circulation). Roughly speaking, the C-token cannot pass any E-token and no E-token can pass the C-token, so during the C-token traversal, all the E-tokens will visit Root exactly once. Root knows the number of E-tokens after a complete traversal of the C-token. Since the C-token cannot pass an E-token (for fear of counting error, but we will see an exception in Section 4.2), it must be stopped by a processor in critical section. As in [1,14] the C-token circulation is implemented by the sending of $SeqVal$ message whose value is different from that of the previous $SeqVal$ message. We use two local variables ($MySeq$ and $NextSeq$) to store the values of $SeqVal$ messages. If a processor i ($i \not=$Root) does not hold any E-token when it receives the C-token ($SeqVal \not= MySeq$), then it executes a_{c3}: it copies $SeqVal$ in $NextVal$ and in $MySeq$, then it sends the $SeqVal$ message to $i+1$. If i is in critical section (i holds an E-token), it stops the C-token: Function STOP returns *true* and i just copies $SeqVal$ in $NextVal$ (Action a_{c4}). If i is still in critical section when it receives a copy of the $SeqVal$ message, it does nothing ($SeqVal = NextSeq$). Processor i will send again the C-token after i exits the critical section and receives a new copy of the $SeqVal$ message: in this case Function STOP returns $false$ and i can execute Action a_{c3}.

Algorithm 2 Ring-Controller.

For Root

RING-CONTROLLER(START)
/* START is used to inform Root that the C-token has just passed */

Variables $MySeq : 0..Max$

begin
(a_{c1}) [] (**receive** $SeqVal$ **from** $n-1$)
$\wedge(MySeq = SeqVal) \longrightarrow$
$MySeq := MySeq + 1$
send $MySeq$ to 1
START

(a_{c2}) [] **timeout** $\longrightarrow$
send $MySeq$ to 1

end

For another processor

RING-CONTROLLER (STOP: Boolean)
/* STOP is used to stop Controller from sending messages, so it must be fair w.r.t. Controller */
Public Function Csend()
$MySeq := NextSeq$
send $MySeq$ to $i+1$
end Function

Variables $MySeq, NextSeq : 0..Max$

begin
(a_{c3}) [] (**receive** $SeqVal$ **from** $i-1$)
$\wedge((MySeq \not= SeqVal) \Rightarrow \neg$ STOP) $\longrightarrow$
$NextSeq := SeqVal$
$Csend$

(a_{c4}) [] (**receive** $SeqVal$ **from** $i-1$)
$\wedge(NextSeq \not= SeqVal) \wedge$STOP $\longrightarrow$
$NextSeq := SeqVal$

end

4.2 Self-Stabilizing ℓ-Token-Circulation

To explain the behavior of ℓ-Token-Circulation (see Algorithm 3) we assume that Ring-Controller is stabilized (i.e., the C-token is unique).

The protocol uses two functions from the application which needs the ℓ-exclusion:

1. Function STATE in $\{Request, In, Out\}$ meaning that the application is requesting for the critical section, is in the critical section, or out of the critical section while not requesting it, respectively,
2. Function ECS which does not return any value. This function allows the application to enter the critical section.

Algorithm 3 ℓ-Token-Circulation.

```
For Root

ℓ-TOKEN-CIRCULATION(STATE:
                {Request, In, Out }, ECS)
External
  RING-CONTROLLER(START)
Parameters
  Function START()
    for k = 1 to ℓ-Cpt do
      send Token to 1
    Cpt := ℓ − Cpt + T
  end Function
Variables     T : 0..1
              Cpt : 0..ℓ
begin
(a_l1)  [] (STATE ∈ {Request, Out})
        ∧(T = 1) ⟶
           if STATE = Out then
              T := 0
              send Token to 1
           else ECS

(a_l2)  [] (receive Token from n − 1)
        ∧ (T = 0) ∧(Cpt < ℓ) ⟶
           Cpt := Cpt + 1
           if STATE = Request then
              T := 1
              ECS
           else send Token to 1

(a_l3)  [] (receive Token from n − 1)
        ∧ (T = 1) ∧(Cpt < ℓ) ⟶
           Cpt := Cpt + 1
           send Token to 1
end
```

```
For another processor

ℓ-TOKEN-CIRCULATION(STATE:
                {Request, In, Out }, ECS)
External
  RING-CONTROLLER(INCS: Boolean)
Parameters
  Function INCS(): Boolean
     Return(STATE = In)
  end Function


Variables     T : 0..1

begin
(a_l4)  [] (STATE ∈ {Request, Out})
        ∧(T = 1) ⟶
           if STATE = Out then
              T := 0
              send Token to i+1
              RING-CONTROLLER.Csend
           else ECS

(a_l5)  [] (receive Token from i-1)
        ∧ (T = 0) ⟶
           if STATE = Request then
              T := 1
              ECS
           else send Token to i+1


(a_l6)  [] (receive Token from i-1)
        ∧ (T = 1) ⟶
           send Token to i+1
           RING-CONTROLLER.Csend
end
```

T is a binary variable and means that the processor owns an E-token ($T = 1$) or does not ($T = 0$). When a processor receives an E-token, either it immediately sends it to its neighbor if it does not need it (STATE $\neq$ *Request* or $T = 1$) or it keeps it if it is requesting the critical section (STATE = *Request*).

When a processor leaves the critical section ($\mathsf{STATE} = \mathit{Out}$ and $T = 1$) it sends the E-token it used to its neighbor. The last case ($\mathsf{STATE} = \mathit{Request}$ and $T = 1$) only appears in an initial configuration because during the execution of the protocol an E-token is kept only if the processor is requesting the critical section (see Actions a_{l2} and a_{l5}).

To ensure a right E-token account we must forbid an E-token to pass the C-token. Unfortunately, this obligation leads the system to not verify the ℓ-liveness: if the C-token is stopped by a processor in critical section forever then the other E-tokens will be stopped by the C-token. To prevent this drawback, our protocol allows another E-token to pass both the E-token used by the processor and the stopped C-token. Then the C-token can leave the processor which stopped it before a second E-token passes it (Action a_{l6}). We explain this *double-passing* mechanism below:

Consider the following situation where a processor i has an E-token (t_1) and is executing its critical section. Moreover, i has stopped the C-token (C). An other E-token (t_2) is coming from $i - 1$:

$$\xrightarrow{t_2} \boxed{Ct_1}$$

When i receives t_2 it executes a_{l6}. So it sends t_2 followed by C:

$$\boxed{t_1} \xrightarrow{C\ t_2}$$

It is obvious that the above configuration is equivalent to:

$$\boxed{t_2} \xrightarrow{C\ t_1}$$

From the above observation we can deduce that the *double-passing* mechanism has no effect on Root counting. More precisely, assuming the above equivalence, we can claim the following property:

Property 1. Assuming *Ring-Controller* is stabilized for the predicate "the C-token is unique", Root is visited exactly once by each E-token during a complete turn of the C-token.

Moreover this mechanism allows us to claim a second property:

Property 2. The circulation of the C-token and the E-tokens cannot be stopped forever.

4.3 Correctness Proof

From [1, 14] and Property 2 we can deduce the following lemma:

Lemma 1. *The composition* Ring-Controller $\triangleright_{\mathsf{P}}$ ℓ-Token-Circulation *is fair w.r.t.* Ring-Controller.

Lemma 2. *The composition* Ring-Controller $\triangleright_{\mathsf{P}}$ ℓ-Token-Circulation *is fair w.r.t.* ℓ-Token-Circulation *if* Ring-Controller *is stabilized.*

Proof. Suppose in the way of contradiction that there exists an execution of the composition which eventually contains no steps of the *ℓ-Token-Circulation*. By Property 2 this implies that no E-token is present in the ring. Since Root initiates infinitely often the C-token traversal, after one round trip delay, Root detects that there is no E-token in the system. Then it can add ℓ E-tokens when it initiates the next C-token traversal (Function START), a contradiction. □

From [1, 14] we can claim that Ring-Controller stabilizes for the predicate "the C-token is unique" if the two following conditions hold:

1. Constant $Max \geq nL_{Max} + 1$, where n is the ring size and L_{Max} is the maximum capacity of communication links.
2. A C-token cannot be stopped forever.

So, by Lemma 1 we can deduce the following result:

Lemma 3. Ring-Controller *stabilizes for the predicate "the C-token is unique" even if* ℓ-Token-Circulation *is not stabilized.*

Lemma 4. *Assuming* Ring-Controller *is stabilized for the predicate "the C-token is unique" then* ℓ-Token-Circulation *stabilizes for the predicate "there exist exactly ℓ E-tokens in the ring".*

Proof. Once Ring-Controller is stabilized, by Property 1 Root can count the E-tokens without any error. More precisely, if the system contains less than ℓ E-tokens, Root will generate the missing E-tokens when the C-token starts a new traversal (Function START). If the system contains more than ℓ E-tokens, Root will erase all the extra E-tokens during the C-token traversal (see Predicate $(Cpt < \ell)$ of Actions a_{l2} and a_{l3}). □

Then it is easy to verify that if both above predicates are satisfied then the safety (the number of E-tokens is equal to ℓ forever), fairness, and ℓ-liveness (the circulation of the E-tokens which are not required by some processors cannot be stopped forever) requirement are satisfied forever. So from Theorem 1 and Lemmas 1, 2, 3, and 4 we can claim the following theorem:

Theorem 2. Ring-Controller $\triangleright_{\mathsf{P}}$ ℓ-Token-Circulation *stabilizes for the ℓ-exclusion specification.*

Stabilization Time Complexity. The stabilization time can be easily evaluated from [14] and Corollary 1. *Ring-Controller* stabilizes in two *round trip delay* or in $2n$ time. Once *Ring-Controller* stabilized, the next round trip ensures the right counting at Root. So, the stabilization time is at most $3n$ time.

4.4 Extension

The ℓ-exclusion protocol works on oriented bi-directional rings but in this kind of systems we can use this property to improve the protocol in order to obtain a version which tolerates the loss of messages. Moreover, we can reduce the value of the constant Max to $3(L_{Max}+1)$ by using the *self-stabilizing alternating bit protocol* in [1] and the *self-stabilizing three state ring protocol* in [15]. We can also design a version for rings with unbounded communication links by using a randomized version of the *self-stabilizing alternating bit protocol* [1]. At last, using a self-stabilizing tree construction and the Euler tour of the tree to build a virtual ring, any of these versions can work on general systems. Unfortunately, with such a solution, the stabilization time of the ℓ-exclusion protocol depends on n instead of h (the tree height) as we could expect.

5 Self-Stabilizing ℓ-Exclusion Protocol on Trees

In this section, we present a self-stabilizing ℓ-exclusion protocol on rooted tree networks. The links are bi-directional. We assume that each processor i (i /= Root) knows its *parent*, denoted by MyP. We denote the set of *children* of any processor by $Children$. Each processor (except leaf) locally distinguishes its children by some ordering denoted by $1..\Delta_i - 1$ if i is an internal processor and $1..\Delta_i$ if $i =$ Root, where Δ_i is the degree of i.

The basic idea of this algorithm is as follows: Root infinitely often sends a wave of E-tokens (0...ℓ E-tokens) down the tree. Each E-token follows a branch consistently from Root to a leaf. Once arrived to the leaf, an E-token disappears. Before to send a new wave of E-tokens, Root uses a *controller* in order to detect the number of E-tokens of the previous wave which are not disappeared yet. The self-stabilizing Tree ℓ-exclusion protocol is a parametric composition of Tree-Controller (Algorithm 4) and ℓ-Token-Distribution (Algorithm 5).

5.1 Tree-Controller

The basis of *Tree-Controller* is the Propagation of Information with Feedback (PIF) in [14]. In the PIF scheme, Root broadcasts a sequence of values ($SeqVal$, see Algorithm 4) to every processor in the network. In order to detect when the current broadcast has terminated Root needs to obtain feedback (of the same value it sent) from the others processors. When an internal processor receives a new value from its parent, then it broadcasts this value to its children. When a leaf of the tree gets a new value, it simply sends an acknowledgment up to its parent (the same value). An internal processor sends an acknowledgment up to its parent, when it has received acknowledgments from all its children. When Root receives an acknowledgment from all its children, Root starts to broadcast a new value.

More formally we can define the specification of the PIF Scheme from that of PIF Cycle as follows:

Algorithm 4 Tree-Controller.

```
Function Forward(Mes)
    for all j ∈ Children do send Mes to j
end Function

Function Feedback(Mes)
    send Mes to MyP
end Function

For Root
TREE-CONTROLLER(START, SIGNAL)
Variables      MySeq: 0..Max
Answer[j] : boolean flag for each
            j ∈ Children
begin
(ac1)  [] receive SeqVal from j ∈ Children ⟶
          if (MySeq = SeqVal) then
             Answer[j] := true
          if (∀j ∈ Children ::
             Answer[j] = true) then
             MySeqVal := MySeqVal + 1
             for all j ∈ Children do
                 Answer[j] := false
             START
             Forward(MySeq)

(ac2)  [] receive Mark from j ∈ Children ⟶
          SIGNAL

(ac3)  [] timeout ⟶
          Forward(MySeq)

end

For an internal processor
TREE-CONTROLLER(STOP: Boolean)
Variables      MySeq: 0..Max
Answer[j] : boolean flag for each
            j ∈ Children
begin
(ac4)  [] receive SeqVal from MyP ⟶
          if (MySeq /= SeqVal) then
             MySeq := SeqVal
             for all j ∈ Children do
                 Answer[j] := false
             if STOP then
                 Feedback(Mark)
          Forward(MySeq)

(ac5)  [] receive SeqVal from j ∈ Children
          ⟶
          if (MySeq = SeqVal) then
             Answer[j] := True
          if (∀j ∈ Children ::
            Answer[j] = true) then
             Feedback(MySeq)

(al5)  [] receive Mark from MyP ⟶
          Feedback(Mark)
end

For a leaf processor
TREE-CONTROLLER ( STOP: Boolean )
Variables      MySeq: 0..Max
begin
(ac7)  [] receive SeqVal from MyP ⟶   if (MySeq /= SeqVal) then
                                         MySeq := SeqVal
                                         if STOP then Feedback(Mark)
                                      Feedback(MySeq)
end
```

Specification 1 (PIF Cycle).

[PIF1] Root *initiates a cycle by broadcasting a message* m.
[PIF2] *The cycle terminates in* Root.
[PIF3] Root *terminates the cycle if and only if all processors acknowledged the receipt of* m.

Specification 2 (PIF Scheme). *The* PIF scheme *is an infinite sequence of PIF Cycles.*

In Tree-Controller we add a special message: $Mark$. This message is sent to Root by a processor in critical section (see function STOP) when it receives a $SeqVal$ message. This $Mark$ message informs Root that an E-token is used. Tree-Controller informs Root that the PIF has just started by using Function START.

5.2 Self-Stabilizing ℓ-Token-Distribution

To explain the behavior of ℓ-Token-Distribution (see Algorithm 5) we assume that Tree-Controller is stabilized.

Algorithm 5 ℓ-Token-Distribution.

ℓ-**TOKEN-DISTRIBUTION**(STATE: { Request, In, Out }, ECS)

For Root
Macro $Next = \begin{cases} Ch+1 \text{ if } Ch < \Delta \\ 1 \quad \text{Otherwise} \end{cases}$
External
TREE-CONTROLLER(**START**, **SIGNAL**)

Parameters
Function START()
if (STATE = $Request$)$\wedge$
$((\ell - Cpt) > 0)$ **then**
$T := 1$
for $k = 1$ to ℓ-$(Cpt + T)$ **do**
send $Token$ to Ch
$Ch := Next$
$Cpt := 0$
end Function

Function SIGNAL()
If $Cpt < \ell$ **then**
$Cpt := Cpt + 1$
end Function

Variables $T : 0..1$
$Cpt : 0..\ell$

begin
(a_{l1}) [] (STATE $\in \{Request, Out\}$)
$\wedge(T = 1) \longrightarrow$
if STATE = Out **then**
$T := 0$
send $Token$ to Ch
$Ch := Next$
else ECS
end

For an internal processor
Macro $Next = \begin{cases} Ch+1 \text{ if } Ch < \Delta - 1 \\ 1 \quad \text{Otherwise} \end{cases}$
External
TREE-CONTROLLER(**INCS**)

Parameters
Function INCS(): Boolean
Return(STATE = In)
end Function

Variables $T : 0..1$

begin
(a_{l2}) [] (STATE $\in \{Request, Out\}$)
$\wedge(T = 1) \longrightarrow$
if STATE = Out **then**
$T := 0$
send $Token$ to Ch
$Ch := Next$
else ECS

(a_{l3}) [] **receive** $Token$ **from** MyP
if STATE = $Request$ **then**
$T := 1$
ECS
else send $Token$ to Ch
$Ch := Next$

(a_{l4}) [] (**receive** $Token$ **from** MyP)
$\wedge(T = 1) \longrightarrow$
send $Token$ to Ch
$Ch := Next$
end

For a leaf processor
External TREE-CONTROLLER(**INCS**: Boolean)
Parameters Function INCS : Boolean
Return(STATE = In)
end Function
Variables $T : 0..1$

begin
(a_{l5}) [] (STATE $\in \{Request, Out\}$) $\wedge$ $(T = 1) \longrightarrow$ **if** STATE = Out **then** $T := 0$ **else** ECS

(a_{l6}) [] (**receive** $Token$ **from** MyP) $\wedge$ $(T = 0) \longrightarrow$ **if** STATE = $Request$ **then** $T := 1$; ECS
end

We use a *switch mechanism* [11] to ensure the fairness. The switch mechanism is implemented at Root and every internal processor by using a pointer variable Ch, which holds a value in $Children$. By using this mechanism, E-tokens are sent in a *breadth-first* manner. We use a *macro* (not a variable but dynamically evaluated) called $Next$ to choose the next child to visit. Recall that every processor maintains a local ordering among its children.

Each time Tree-Controller calls Function START, Root sends a wave of E-tokens. Remark that if Root requests to enter its critical section, it keeps one E-token and enters its critical section (See Function START and Action a_{l1}).

When a processor i receives an E-token from its parent, i does the following: If i requests to enter its critical section (State $= Request$), then i sets $T = 1$ and enters its critical section (See Action a_{l2} for an internal processor and Action a_{l6} for a leaf processor). Otherwise, either i is an internal processor and i sends the E-token to Ch (a_{l4}) or i is a leaf processor and the E-token disappears. When a processor i exits its critical section, it sends its E-token to Ch_i if i is Root or an internal processor (See Action a_{l1} for Root and Action a_{l2} for an internal processor) otherwise the E-token disappears (See Action a_{l5}).

The principle of Tree-Controller is not the same as Ring-Controller. It allows Root to count the E-tokens which are may be still used (Cpt). Once Tree-Controller is stabilized, no processor in critical section can send more than one $Mark$ message. Moreover, the links are FIFO, so we are sure that the unused E-tokens have disappeared at the end of the control broadcast phase. So when Root sends $\ell - Cpt$ E-tokens, the total number of E-tokens is always less than or equal to ℓ.

5.3 Correctness Proof

The progression of the PIF cycle and the E-tokens are almost independent since an E-token does not stop the broadcast message. They just synchronize at the beginning of the PIF cycle (Action a_{c1} and Function START). So we can claim:

Lemma 5. *The composition* Tree-Controller $\triangleright_{\mathsf{P}}$ ℓ-Token-Distribution *is fair w.r.t.* Tree-Controller.

Lemma 6. *The composition* Tree-Controller $\triangleright_{\mathsf{P}}$ ℓ-Token-Distribution *is fair w.r.t.* ℓ-Token-Distribution *if the* Tree-Controller *is stabilized.*

Proof. Suppose in the way of contradiction that there exists an execution of the composition which contain no steps of the ℓ-exclusion protocol. This implies that no E-token is present in the tree. Since, Root initiates infinitely often the controller and after $2h$ time, Root detects that there is no E-token in the system. Then it can add ℓ E-tokens when it initiates the next controller, a contradiction. □

From [1,14] we can claim that the *Tree-Controller* stabilizes for the specification 2 if $Max > nL_{max}$, where L_{max} is the maximum capacity of communication links, and n is the number of tree nodes. So, we can claim the following result:

Lemma 7. Tree-Controller *stabilizes for Specification 2 even if* ℓ-Token-Distribution *is not stabilized.*

Lemma 8. *(Safety) Assuming* Tree-Controller *is stabilized for Specification 2 then the system will eventually contain at most* ℓ *E-tokens forever.*

Proof. Let a configuration γ_{start} be the end of the PIF cycle of the *Tree-Controller* and Cpt_{start} the value of Cpt at γ_{start}. We denote by $\sharp Mark_{start}$ the number of $Mark$ messages received by the root during the PIF cycle and by $\sharp E\text{-}tokens_{start}$ the number of E-tokens in the tree at γ_{start}. By Action a_{c2} and Function SIGNAL, we have $Cpt_{start} = min(\sharp Mark_{start}, \ell)$. Since during the PIF cycle no extra E-token is created but an E-token which generated a Mark message may disappear at a leaf, we can claim that $\sharp Mark_{start} \geq \sharp E\text{-}tokens_{start}$. Then when Root initiates the next PIF cycle, we consider two cases.

1. $\sharp Mark_{start} \geq \ell$. In this case $Cpt_{start} = \ell$ and Root does not send any new E-token but eventually starts a new PIF cycle (see Function START). So if $\sharp E\text{-}tokens_{start} \geq \ell$, then $Mark_{start} \geq \ell$ and Root does not send any new E-token. Since E-tokens eventually disappear at a leaf the number of E-tokens decreases until it becomes less than ℓ.
2. $\sharp Mark_{start} < \ell$. In this case $Cpt_{start} = \sharp Mark_{start}$ and Root sends $\ell - Cpt_{start} \leq \ell - \sharp E\text{-}tokens_{start}$. So the number of E-tokens $(\ell - Cpt_{start} + \sharp E\text{-}tokens_{start})$ is always less than or equal to ℓ.

□

We need the following definition to prove the next lemma. A *path* μ_{ij} is a sequence $(i_0, i_1, ..., i_{x-1}, i_x)$ such that the following conditions are true: $(1)i = i_0$, $(2)j = i_x$, $(3)x \geq 1$, $(iv)\forall m \in [1, x-1]$, $i_m = P_{i_{m+1}}$, where P_i denotes the parent of i. The *length* x of a path μ_{ij} is denoted by $\overline{\mu_{ij}}$.

Lemma 9. *(Fairness) Assuming* Tree-Controller *is stabilized for Specification 2, in any execution e, each processor i receives (at least) an E-token every W E-tokens sent by* Root *such that* $W = (\prod_{j \in \mu_{rP_i}} (\sharp Children_j))$ *if* $i \neq r$ *and* $W = 1$ *if* $i = r$.

Proof. We will prove this theorem by induction on the length of the path μ_{ri} from Root to i $(\overline{\mu_{ri}})$.
1. Basic step. The case where $i = r$ is obvious. Consider the case where $\overline{\mu_{ri}} = 1$, so $i \in Children_r$. It is easy to see that if $W = \sharp Children_r$, then by the switch mechanism each processor i receives an E-token.
2. Induction step. Assume that the theorem is true for $\overline{\mu_{ri}} \leq k-1$, $k > 1$. Consider the case where $\overline{\mu_{ri}} = k$. Let W be an integer such that

$$W = (\prod_{j \in \mu_{rP_i}} (\sharp Children_j)) = (\sharp Children_{P_i})(\prod_{j \in \mu_{rP_{P_i}}} (\sharp Children_j))$$

By hypothesis, P_i receives at least $\sharp Children_{P_i}$ E-tokens every W E-tokens sent by Root, by the switch mechanism each of its children receives an E-token, in particular Processor i. □

Lemma 10. *(ℓ-Liveness) Assuming* Tree-Controller *is stabilized for Specification 2, any execution e satisfies the ℓ-liveness property.*

Proof. Suppose in the way of contradiction that there exist an execution e such that x, $x < \ell$, processors execute the critical section forever and any other requesting processor can never get any E-token.

Since no other processor can enter the critical section forever then the $\ell - x$ E-tokens are unused forever. So, eventually each PIF wave of Tree-Controller will send exactly x $Mark$ messages to Root forever. Then between each PIF wave Root will send exactly $\ell - x$ E-tokens. Since $\ell - x > 0$, by Lemma 9, any processor will eventually receive an E-token. This contradicts our assumption. □

From Lemmas 8, 9, and 10 we can claim the following result:

Lemma 11. *Assuming* Tree-Controller *is stabilized for Specification 2 then* ℓ-Token-Distribution *stabilizes for the* ℓ*-exclusion specification.*

So from Theorem 1 and Lemmas 5, 6, 7, and 11 we can claim the following theorem:

Theorem 3. Tree-Controller $\triangleright_{\mathsf{P}}$ ℓ- Token-Distribution *stabilizes for the* ℓ*-exclusion specification.*

Stabilization Time Complexity. The stabilization time of *Tree-Controller* is $4h+2$ time [14]. Once *Tree-Controller* is stabilized, after a new PIF wave Root will never send too much E-tokens in the system. So the stabilization time of the composition is bounded by $6h + 2$ time.

5.4 Extension

The tree ℓ-exclusion protocol has a drawback: the number of times that another processor can enter the critical section before a processor p can do it $O(CH^h \times h)$, where $CH = \max_i(\sharp Children_i)$. We can remove this drawback by changing the switch mechanism as follows: every requesting processor sends a request to Root, every intermediate processor enqueues the link number the request is coming from. Now the E-tokens follow the paths described by the local queues. After they have been consumed the E-tokens are erased by the processors they used them. In this case the delay is only $O(n)$. But the drawback of this solution is that the size of the queue is $O(n)$. Nevertheless, we can also improve the algorithm as in rings in order to obtain a version which tolerates the lost of messages. We can also reduce the value of the constant Max to $3(L_{Max} + 1)$ by using the *self-stabilizing alternating bit protocol* in [1] and the *self-stabilizing three state PIF protocol* in [4].

6 Conclusion

In this paper, we present the first self-stabilizing protocols for ℓ-exclusion problem in the message passing model. We propose a new technique for the design of self-stabilizing ℓ-exclusion: the controller. This tool allows to count the tokens

of the system without any counter variable for all processors except one called Root. We also introduce a new protocol composition called *parametric composition*. This is a generalization of the *collateral composition* in [13] and of the *conditional composition* in [5]. Then we present protocols on rings and on trees. The space requirement of both algorithms is independent of ℓ for all processors except Root. The stabilization time of the first protocol is $3n$ time, where n is the ring size and the stabilization time of the second one is $6h + 2$ time, where h is the tree height. We also discuss numerous adaptations following various models.

References

1. Afek, Y., Brown, G.M : Self-stabilization over unreliable communication media. Distributed Computing, Vol. 7 (1993) 27–34
2. Afek, Y., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: A bounded first-in, first-enabled solution to the ℓ-exclusion problem. In Proceedings of the 4th International Workshop on Distributed Algorithms, Springer-Verlag, LNCS 486 (1990) 422–431.
3. Abraham, U., Dolev, S., Herman, T., Koll, I.: Self-Stabilizing ℓ-exclusion. In Proceedings of the third Workshop on Self-Stabilizing Systems, International Informatics Series 7, Carleton University Press (1997) 48–63
4. Bui, A., Datta, A.K., Petit, F., Villain V.: State-optimal snap-stabilizing PIF in tree networks. In Proceedings of the Fourth Workshop on Self-Stabilizing Systems (1999) 78–85
5. Datta, A.K., Gurumurthy, S., Petit, F., Villain V.: Self-stabilizing network orientation algorithms in arbitrary rooted networks. In Proceedings of the 20th IEEE International Conference on Distributed Computing Systems (2000) 576–583
6. Dolev, D., Gafni, E., Shavit, N.: Toward a non-atomic era: ℓ-exclusion as test case. In Proceeding of the 20th Annual ACM Symposium on Theory of Computing, Chicago (1988) 78–92
7. Dijkstra, EW.: Self stabilizing systems in spite of distributed control. Communications of the Association of the Computing Machinery, Vol. 17, No. 11 (1974) 643–644
8. Flatebo, M., Datta, A.K., Schoone, A.A.: Self-stabilizing multi-token rings. Distributed Computing, Vol. 8 (1994) 133-142
9. Fisher M., Lynch N., Burns, J., Borondin A.: Resource allocation with immunity to limited process failure. In Proceedings of the 20th IEEE Annual Symposium on Foundations of Computer Science (1979) 234–254
10. Fisher M., Lynch, N., Burns J., Borondin A.: A Distributed Fifo allocation of identical resources using small shared space. ACM Transactions on Programming Languages and Systems, Vol 11 (1989) 90–114
11. Rachid, H.: Space and time efficient self-stabilizing ℓ-exclusion in tree networks. In Proceedings of the 14th IEEE International Parallel and Distributed Processing Symposium (2000) 529–534
12. Peterson, G.: Observation on ℓ-exclusion. In Proceedings of the 28th Annual Allerton Conference on Communication, Control and computing, Monticello (1990) 568–577
13. Tel, G.: Introduction to distributed algorithms. Cambrige University Press (1994)
14. Varghese G.: Self-stabilizing by counter flushing. Technical Report, Washington University (1993)
15. Villain V.: A key tool for optimality in the state model. In DIMACS'99, The 2nd Workshop on Distributed Data and Structures, Carleton University Press (1999) 133–148

Self-Stabilizing Agent Traversal

Ted Herman[1]* and Toshimitsu Masuzawa[2]**

[1] University of Iowa
herman@cs.uiowa.edu
[2] Osaka University, Japan
masuzawa@ics.es.osaka-u.ac.jp

Abstract. This paper introduces the problem of n mobile agents that repeatedly visit all n nodes of a given network, subject to the constraint that no two agents can simultaneously occupy a node. It is shown for a tree network and a synchronous model that this problem has $O(\Delta n)$ upper and lower time bounds where Δ is the maximum degree of any vertex in the communication network. The synchronous algorithm is self-stabilizing and can also be used for an asynchronous system. A second algorithm is presented and analyzed to show $O(n)$ round complexity for the case of a line of n asynchronous processes.

1 Introduction

A fundamental task for (mobile) agent-based computing is search, or visitation, of a group of network nodes. Agents are convenient programming entities for distributed systems because they encapsulate procedures for a sequence of locations without having to specify programs at each location. A collection of agents can have different functions or cooperatively work for a single function (some recent research [5] metaphorically calls collections of agents "swarms of insects"). Ideally, agents are autonomous, and agent-based computation is self-stabilizing. Autonomous agents are capable of independent action in open, unpredictable environments. If we interpret a mismatch between the internal variables of agents and the environmental variables as a faulty condition, then autonomous agents can be fault tolerant by taking local actions to correct their internal variables to be consistent with the environment. The paradigm of self-stabilization generalizes this problem, tolerating even initial situations where internal variables are mutually inconsistent. This generalization can simplify agent design by avoiding detailed case analysis based on all the ways that the environment can change. These considerations motivate the study of self-stabilizing protocols based on the mobile agent paradigm.

We suppose in this paper that agents require considerable resource support with respect to the host platform. For instance, the host could be an embedded

* Research supported by NSF award CAREER 97-9953 and DARPA contract F33615-01-C-1901.

** This work is supported in part by Japan Society for the Promotion of Science (JSPS), Grant-in-Aid for Scientific Research((c)(2)12680349).

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 152–166, 2001.

controller with limited memory and processing power. To model this kind of limitation, we propose the following constraint: no node can contain more than one agent at a time. Therefore, for an agent to move about the system, other agents may need to be displaced in its path. In this paper, the fundamental operation that satisfies this constraint is the *agent swap* operation. Agents at neighboring nodes are permitted to exchange locations in a single atomic swap operation.

Related Work. The topic of mobile agents in a self-stabilizing system appears in [10] and is also motivated by [4]. Other papers that study mobility issues in the context of self-stabilization include [14, 8]. Except for some motivation, general programming treatments of mobile agents [17] have little relation to this paper. Remarks in the conclusion explain that previous work on supporting serial execution models in distributed systems for self-stabilizing algorithms [16,20,12,19,11,18] are related by similar techniques to the algorithms presented here. The protocols presented in Sections 4.1 and 5 have the simple character of wave algorithms [21] (though there are no initiators or decide events), and phase clock protocols [13, 1].

Summary of Results and Contributions. The main results of this paper are the definition of the problem of n-agent traversal, lemmas that show lower bound time complexity in a tree, and a construction with an upper bound complexity matching this lower bound. An additional algorithm solving the problem for a tree is very simple (essentially requiring no convergence, since every state is legitimate) and we analyze its time complexity for the case of a linear array of processes.

2 Network, Agents, and Self-Stabilization

A complete description of a mobile agent architecture would discern between variables that accompany agents, the so-called *briefcase* variables of [10], and variables that are attached to hosts, that can be read and written by agents. Since the focus of this paper is a certain behavior of agents (traversing the network) rather than the application invoked by agents as they travel, we can use a simplified model. We specify the traversal protocol using traditional guarded assignments, but extend the execution model to support the swap operation.

The system consists of $n > 1$ nodes in a network described by a connected, undirected graph; let $j \in N_i$ denote that node j is a neighbor of node i in the graph. Let δ_i be the degree of node i in the graph, that is, $\delta_i = |N_i|$. We suppose N_i has a fixed ordering, and $N_i[k]$ denotes the k-th node in this ordering.

Programs for the nodes are specified by guarded assignments of the form $g_i \rightarrow s_i$, where g_i is a boolean function and s_i is an assignment to the variables of node i; both g_i and s_i may refer to variables of nodes in the set $\{i\} \cup N_i$. The guarded assignments are also called *actions* of the system.

A state of the system is a specification of values for the variables of all nodes. With respect to any given state σ, a guarded assignment $g_i \rightarrow s_i$ is termed *enabled* if g_i holds at σ and is otherwise *disabled*. We assume that programs satisfy

the following property: if $g_i \rightarrow s_i$ is enabled at a state σ, and execution of the assignment s_i at σ results in state σ', then $g_i \rightarrow s_i$ is disabled at σ'. Thus any enabled guarded assignment will disable itself if executed.

One variable of each node is reserved for an entity called an *agent*. The domain of such a variable is such that it always "contains" one agent. We suppose that each agent has a unique identity, and a basic axiom for any state (including erroneous initial states) is that no agent can be located at more than one node at any state.

Let us review the standard execution semantics for guarded assignments typically used for self-stabilizing algorithms. A *synchronous computation* of the system is a maximal sequence of states so that each consecutive pair of states (σ, σ') includes the (parallel) execution of one guarded assignment at each node that has an enabled guarded assignment. An *asynchronous computation* of the system is a maximal sequence of states so that each consecutive pair of states (σ, σ') corresponds to the execution of one enabled guarded assignment at state σ. Within a computation (synchronous or asynchronous) any pair of consecutive states is called a *step* of the computation. A *weakly fair* asynchronous computation is an asynchronous computation containing no suffix in which some node has an enabled guarded assignment at each state in the suffix (because we suppose each node has only one guarded assignment, the condition of being continuously enabled implies that the node does not execute the assignment throughout the sequence). In an asynchronous computation, time is measured by *rounds*. The first round is the minimal prefix of the computation so that, for each node i, either i's guarded assignment is disabled at some state of the prefix, or some consecutive pair of states corresponds to the execution of a guarded assignment by node i. Round i, for $i > 1$ is defined recursively, by applying the definition of the first round to the remaining suffix of states in the computation.

We now propose an extension of the execution semantics of guarded assignments to include an *agent swap* operation. An agent swap operation in an asynchronous computation is a consecutive pair of states (σ, σ') that changes variables of two neighboring nodes, say i and j. If at σ two agents X and Y are located at nodes i and j respectively, then an agent swap reverses the locations of X and Y. The swap operation (σ, σ') may also change values of (non-agent) variables at nodes i and j. This notion of agent swap can be generalized to an agent move, for the case of an agent atomically moving from one node to another, and agent swaps for synchronous computations can be defined in the obvious way (we omit details). While it is unusual to consider the type of parallel assignment for an agent swap in guarded assignments of self-stabilizing algorithms, recall that CSP's communication mechanism [15] does something similar, changing the local state of two processes atomically, based on the "willingness" of both parties. Rather than developing a formal programming notation for agent swap, we use informal descriptions for the few protocols presented in subsequent sections.

A subset B of all possible computations (synchronous or asynchronous) specifies the *legitimate behaviors* of the system. A system is *self-stabilizing* if every computation has a suffix contained in the set B. The *stabilization time* of a

system is the worst case, taken over all possible computations, of the length of the minimum prefix beyond which the remaining suffix is an element of B. For asynchronous computations, the stabilization time is determined by the number of the rounds in the prefix rather than its length.

3 Problem Statement and Lower Bound

Given are n distinct agents, initially present at arbitrary nodes of the network so that no two agents are located at a common node. Initial values of non-agent variables at the nodes can be arbitrary. Legitimate behavior for the system is any computation so that, for each agent A and node p, the agent A is located at node p at infinitely many states. The *n-agent traversal problem* is to specify a protocol so that all its computations are legitimate behaviors.

Efficiency of a solution to the traversal problem is measured by the time required for all agents to visit all nodes. An *agent walk* is a sequence of nodes so that there is a graph edge for every consecutive pair of nodes. A *traversal* is an agent walk that includes each node at least once and has the same node for the initial and final item in the sequence. Efficiency is determined by measuring the worst-case minimum time for every agent to complete a traversal. A trivial lower bound of $\Omega(n)$ for all agents to complete a traversal applies to all networks, however tighter lower bounds on efficiency are topology-specific. For the case of a tree network, we have a more precise lower bound.

Lemma 1. *Given a tree network with maximum degree Δ, the efficiency of any n-agent traversal protocol is $\Omega(\Delta n)$.*

Proof. By counting agent visits to a particular node, the bound is derived. Consider first the case of a synchronous computation. Let v be some node having degree Δ in the communication graph. Let P be the agent initially located at v. To complete a traversal, P leaves v and returns to v at least once for each neighbor (this follows because the graph is a tree). Because each agent swap can contribute to two traversals, we count time only for agents that move to v by a swap. In this accounting, agent P's contribution is Δ — once for each neighbor of v. Consider some agent $Q \neq P$. Agent Q enters v and departs from v at least once for each edge incident on v, hence the accounting for Q is the same as for P. Thus, all n agents contribute Δ swap operations just for arrivals at v, which provides an $\Omega(\Delta n)$ bound. For an asynchronous computation, we have only to consider computations in which the operation by v (if any is enabled) occurs last in the round. This ensures that v participates in at most one swap per round, and the $\Omega(\Delta n)$ bound applies. □

4 Phase-Based Algorithm

The main result of this section is an algorithm that completes n-agent traversal for a tree in $O(\Delta n)$ time; the algorithm also applies to rings, but does not work for all topologies. We present the algorithm first for a synchronous computation and then discuss adaptation to the asynchronous model.

4.1 Synchronous Algorithm

Two building blocks for the algorithm are edge coloring [7] and phase synchronization [13,1], which are tasks having published self-stabilizing solutions. Colors are represented by natural numbers in the domain $[0, (\Delta - 1)]$ and every node v has a non-agent variable $color_v[w]$ for each neighbor $w \in N_v$ (in the presentation below, v and w are implicitly typed as neighboring nodes). Phases are natural numbers, and every node v has a non-agent variable $clock_v$ containing a phase number. The domain R of phase numbers is either infinite or a given domain $[0, m-1]$ for some $m \geq \Delta$. A self-stabilizing edge coloring algorithm reaches a fixed point satisfying the *Colored* predicate:

$$\begin{aligned} \mathit{Colored} \equiv\ & (\forall v, w ::\ color_v[w] \in [0, (\Delta - 1)]) \\ & \wedge\ (\forall v, w ::\ color_v[w] = color_w[v]) \\ & \wedge\ (\forall v ::\ |\{\ color_v[w] \mid w \in N_v\ \}| = \delta_v) \end{aligned}$$

A self-stabilizing phase synchronization protocol converges to a stable *Synchronized* predicate (recall that we consider a synchronous model of computation for this first algorithm):

$$\mathit{Synchronized} \quad \equiv \quad (\forall v ::\ clock_v \in R)\ \wedge\ (\forall v, w ::\ clock_v = clock_w)$$

Additionally, a phase synchronization protocol satisfies this liveness property: the value of $clock_v$ increments in each step of the computation if R is infinite, and increments modulo m if R is finite.

Our traversal algorithm is the parallel composition [9] of stabilizing coloring, stabilizing phase synchronization, and a swapping protocol. In the case where the phase clock is unbounded, the swapping protocol is given by the single guarded action (per node)

$$(clock_v \bmod \Delta) = color_v[w] \rightarrow \textit{swap agent at } v \textit{ with agent at } w$$

If the phase clock has finite domain $[0, m-1]$ for $m \geq \Delta$, then there exists g, the largest integer that is a multiple of Δ and at most m, for which the swapping protocol is given by

$$\begin{aligned} & (clock_v < g)\ \wedge\ (clock_v \bmod \Delta) = color_v[w] \\ & \quad \rightarrow \textit{swap agent at } v \textit{ with agent at } w \end{aligned}$$

At each time unit, a node either initiates an agent swap or the node does nothing more than increment its phase clock.

Lemma 2. Let the communication graph be a tree network and let σ be a system state satisfying *Colored* and *Synchronized* predicates. Let A be an agent located at some node v such that $color_v[w] = clock_v$ at state σ. Let T be the subtree of the communication graph formed from the union of all paths that contain w and have v as one endpoint of the path. Then, in any subsequent computation, agent A crosses over each edge of T exactly twice before returning to v.

Proof. Note first that, for any state satisfying *Colored* and *Synchronized*, that every agent eventually moves via an agent swap operation. This follows from the liveness property of the phase clock and the edge coloring, which ensures that every color enables a swap infinitely often in a computation. The remainder of the proof is by induction on the size of T. The first swap at v following state σ sends agent A to node w. If w is a leaf of T, then the next swap by w returns A to v and proves the claim. Otherwise, by hypothesis, agent A completes a tour of one subtree rooted at w (that doesn't include v) and returns to w; the return to w also advances the phase clock to the next color, and thus selects another subtree. Because the ordering of colors and edge coloring is fixed, only after visiting all subtrees of w (an visiting none of these more than once), agent A's next swap returns it to v, which completes the proof. □

Theorem 1. The composition of a stabilizing coloring algorithm, a stabilizing phase clock, and the agent swapping protocol is a stabilizing solution to the n-agent traversal problem for tree networks.

Proof. The stabilization of the coloring and phase clock are given. The previous lemma shows that a depth-first search order of agent circulation holds for a subtree; the same argument applies in succession to all subtrees, which demonstrates agent traversal. □

Lemma 3. Starting from a state satisfying *Colored* and *Synchronized*, all agents traverse the tree network within $O(\Delta n)$ time.

Proof. We show the proof for a particular case, that is, where the clock modulus m satisfies $m \bmod \Delta = 0$ and where the initial state has $clock_v = 0$ for every $v \in V$ (other cases have similar arguments). In the following Δ steps in a computation from such an initial state, there is exactly one swap for each edge of the network, thus there are $n - 1$ swaps in Δ steps. Because each agent moves only in a DFS tour, each agent uses exactly $2(n - 1)$ swaps to complete its tour. All agents move in parallel and each agent moves at least once in each sequence of Δ steps. So after $2(n - 1)\Delta$ steps, every agent has completed traversal. □

Lemma 3 shows that the agent traversal algorithm is asymptotically optimal, since the traversal time matches the lower bound. Moreover, by examining the algorithm with much care, we can show that all agents traverse the tree network exactly in Δn time that exactly matches the lower bound. The asymptotic optimality also holds for a suboptimal coloring, provided the number of colors is $O(\Delta)$.

Lemma 4. The composition of a stabilizing coloring algorithm, a stabilizing phase clock, and the agent swapping protocol is a stabilizing solution to the n-agent traversal problem for ring networks.

Proof. First we establish that each agent in the ring has a fixed orientation (clockwise or counterclockwise) throughout any computation. Since, by the swapping protocol, all colors selected for swaps infinitely often in any computation, the result follows. By properties of the phase clock and the *Colored* predicate, there occurs a swap along each edge of the communication graph infinitely often in any computation. Therefore, we may consider an arbitrary agent A and the first time it swaps with some neighboring agent. This event occurs on an edge (v, w) with some color r, moving A from v to w. In the ring, there are two edges incident on w, namely (v, w) and (w, x), where (v, x) has color $s \neq r$. The next swap event that involves A is along (w, x), sending A to x, since the phase clock's activation of color s occurs before color r, because colors are selected cyclically and r was the last color selection. This establishes that agent A has the orientation (thus far in its movement) of "v to w to x." By an inductive argument it follows that agent A retains the same orientation continuously around the ring. □

To establish asymptotically optimal efficiency of the swapping protocol for the ring, we require that the coloring algorithm use only a constant number of colors, since this implies that each edge is engaged in a swap within a constant number of steps, so that each agent completes traversal of the ring within $O(n)$ steps.

That the agent traversal algorithm fails to solve n-agent traversal for general communication graphs is shown by the counter-example of Figure 1. The colors 0, 1, and 2 are shown on the edges of this 4-vertex graph. Agent a is located centrally in the figure, and clocks start at zero. From this initial state, agent a perpetually swaps along the graph's cycle, but fails to visit the leaf node.

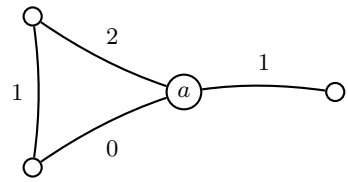


Fig. 1. Counter example for agent traversal.

The algorithm does solve n-agent traversal on graphs that are hybrids of trees and a ring; in fact, it is not difficult to show there exists a coloring (generally far from optimum) for graphs with an Euler walk, so that agent swaps are sequenced to move agents cyclically over all edges of the communication graph.

4.2 Adaptation to Asynchronous Computation

The synchronous algorithm can be adapted to the asynchronous case by using a stabilizing asynchronous phase clock and a stabilizing asynchronous edge coloring algorithm. A technical challenge for such an adaptation is that neighboring clocks can differ in the asynchronous case, so extra measures are needed to prevent conflict in the scheduling of agent swaps. Recall that an invariant for an asynchronous phase clock is $(\forall v :: \mathit{legitClock}_v)$ where

$$\begin{aligned} \mathit{legitClock}_v \quad \equiv \quad & (\forall w : w \in N_v : (\mathit{clock}_v - \mathit{clock}_w) \bmod m \leq 1 \quad \vee \\ & \qquad (\mathit{clock}_w - \mathit{clock}_v) \bmod m \leq 1 \quad) \end{aligned}$$

for some constant m. Informally, the invariant is that neighboring *clock* variables differ by at most one (the protocol of [6] has a more complicated definition of a legitimate state because variables other than *clock* are introduced; however for our purposes, *legitClock* suffices). The condition for a *clock* to increment (modulo some constant) is that its value be "minimal" with respect to all neighboring node *clock* variables, where minimal is with respect to the a cyclic ordering $\preceq$ of the range of the clock modulus (called the "`beh`" relation in [6]).

We suggest the following modification to the phase clock protocol. The guard for advancing the phase clock at v should be strengthened in the locally legitimate case, that is, when clock_v is minimal for its neighborhood (to simplify presentation, we are considering the case $m \bmod \Delta = 0$). This advance of the phase should be prevented in the case where two neighboring clock values are equal and the edge color for these neighbors indicates a swap — then the value of both clocks should increment in parallel, accompanied by an agent swap, in one step of the computation. Figure 2 illustrates the situation for a network of five nodes. Edges in the figure are labeled with colors in the domain $[0, 3]$ and the ordering of *clock* values satisfies $0 \prec 1 \prec 2 \cdots \prec (m-1) \prec 0$. Nodes are labeled by their *clock* values. In this state, the leaf node with $\mathit{clock} = 1$ would, in the usual asynchronous phase clock protocol, be enabled to increment (modulo m) because its *clock* is locally minimal. We have proposed that a swap should accompany such an increment since the edge color is equal to the phase number, however, this cannot occur unless *both* nodes incident on the edge simultaneously advance their *clock* variables, and local minimality does not hold for the figure's central node.

To implement our proposal, let each node v have a new boolean variable b_v. The protocol for v has a new action

$$\neg b_v \ \wedge \ (\forall w : \ w \in N_v : \ \mathit{clock}_v \preceq \mathit{clock}_w) \quad \rightarrow \quad b_v := \mathit{true}$$

and any change to clock_v is modified to also falsify b_v. With these modifications, when b_v is *true* in a legitimate state, then v has a locally minimal *clock*. To define legitimacy, we consider both *clock* and *color* values. A legitimate state is

one satisfying $(\forall v :: \mathit{legit}_v)$ where

$$\begin{aligned}\mathit{legit}_v \;\equiv\; & \mathit{legitClock}_v \;\wedge \\ & (b_v \;\Rightarrow\; (\forall w :\; w \in N_v :\; \mathit{clock}_v \preceq \mathit{clock}_w)) \;\wedge \\ & (\forall w :\; w \in N_v :\; (\mathit{clock}_v \bmod \Delta) = \mathit{color}_{vw} \;\Rightarrow\; \mathit{clock}_w \preceq \mathit{clock}_v)\end{aligned}$$

The last conjunct of legit_v is needed to specify that when v's *clock* advances to equal (mod Δ) the color of an incident edge (v, w), it should not be the case that $\mathit{clock}_v \prec \mathit{clock}_w$ — since deadlock would result by v waiting for w to advance its phase "backward" to equal the (v, w) edge color. Notice that the definition of legit_v has been engineered to be locally testable.

Node v's swap protocol combined with the phase clock protocol has the following form (omitted are corrective actions for illegitimate states [6] and the assignments to b_v mentioned above).

$$\begin{aligned}&(\forall w :\; w \in N_v :\; \neg \mathit{legit}_v \;\vee\; (\mathit{clock}_v \bmod \Delta) \;/\!=\! \mathit{color}_{vw}) \quad \rightarrow \\ &\quad \textit{use normal phase clock protocol} \\ &[]\quad ([]w :\; w \in N_v :\; \mathit{legit}_v \;\wedge\; (\mathit{clock}_v \bmod \Delta) = \mathit{color}_{vw} \;\wedge\; b_v \;\wedge\; b_w \quad \rightarrow \\ &\quad \textit{advance } \mathit{clock}_v \;; \\ &\quad \textit{swap agents with node } w \;; \\ &\quad b_v := \mathit{false}\;)\end{aligned}$$

Recall that synchronously parallel assignment to variables of two neighboring nodes is implicit in our model, which enables agent swap as a fundamental primitive (even in an asynchronous computation).

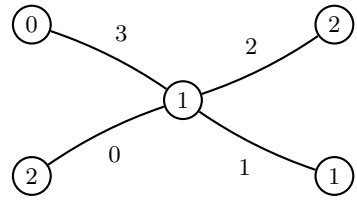


Fig. 2. Example of asynchronous phase clock.

5 Non-phased Algorithm

The agent traversal algorithm of Section 4.1 relies on coloring and phase control to correctly swap agents. This subsection introduces a very simple protocol that requires no such machinery. In fact, the protocol presented below has no illegitimate states (it is snap-stabilizing [2,3]) and is suited to both synchronous and asynchronous models of computation. This simple protocol does not apply to as many topologies as the protocol of Section 4.1 — it is restricted to tree networks.

5.1 Presentation and Verification

Figure 3 presents the new protocol. Only one non-agent variable is used by each node v, $swapnext_v$, which is the index with respect to N_v of one neighbor. Intuitively, $swapnext_v$ specifies which neighbor v should next engage in an agent swap operation, as soon as that neighbor agrees to the swap. Recall that a swap is an atomic step changing the state of two nodes. When node v swaps agents with neighbor w, then both $swapnext_v$ and $swapnext_w$ increment, modulo the size of their respective neighborhoods, in the same step. Node v can be viewed as "waiting" at states where $N_v[swapnext_v] = w$, but $N_w[swapnext_w] \neq v$. This possibility of waiting motivates the following lemma to prove absence of deadlock (however deadlock can occur for a cyclic communication graph).

$$N_v[swapnext_v] = w \ \wedge \ N_w[swapnext_w] = v$$
$$\rightarrow$$
swap agent at v with agent at w ;
$$swapnext_v \ := \ (swapnext_v + 1) \bmod \delta_v$$

Fig. 3. Agent traversal protocol for node v.

Lemma 5. In any state, at least one agent swap is enabled for the protocol of Figure 3.

Proof. We define, for any state σ, a directed graph $G = (V, E)$ as follows. The set of vertices V is the set of nodes in the tree; edge $(v, w) \in E$ iff $w = N_v[swapnext_v]$. Observe that G contains a cycle $\{(v, w), (w, v)\}$ iff an agent swap between v and w is enabled at σ. The proof consists of showing the existence of such a cycle. To show that G has at least one cycle, we use a path argument. For any leaf node v of the tree, there is an edge (v, w_1) in E because the domain of $swapnext_v$ is the singleton set $\{0\}$. If $(w_1, v) \in E$ then there is a cycle and the lemma holds; otherwise there is some edge (w_1, w_2). Now we repeat this case analysis inductively, either finding a cycle or constructing a directed path $w_1, w_2, \ldots, w_k$ such that w_k is a leaf vertex in the tree — which then implies a cycle because $(w_k, w_{k-1}) \in E$. □

Lemma 6. Each node participates in an agent swap infinitely often in any computation.

Proof. Suppose, heading for a contradiction, that some node participates only finitely often in some computation. Then there exist $x, y \in V$ such that y participates in infinitely many swaps (by weak fairness and Lemma 5) and x does not, but $x \in N_y$. However, because y increments $nextswap_y$ to eventually satisfy $N_y[nextswap_y] = x$ following any state of the computation, it follows that x thereafter swaps with y because no further step involving y can occur without a swap with x. □

A corollary of Lemma 6 is that each agent is swapped infinitely often. To show other properties of the protocol, we borrow concepts from previous papers that implement self-stabilizing construction of a depth-first search tree. Given a rooted tree network with a fixed neighborhood enumeration (such as N_v) and distinct node identifiers, there is a lexicographic ordering of paths from the root to every other node in the tree.

Lemma 7. The sequence of nodes visited by any agent is a DFS traversal of the network.

Proof. Let r be an arbitrary node hosting an agent A in the initial state, which we consider to be the root for purposes of the proof. To fix the order of paths with respect to r, we show first by induction that agent A moves to a initial path from r to some leaf. For the base case, if r has a neighbor v and the first swap of A involves (r, v), then the initial path is established. Otherwise, A moves to some $w \in N_r$ so that $N_w[swapnext_w] \neq r$. Applying the argument inductively, A continues to move along some path, but the tree topology prevents the formation of a cycle, and because the number of nodes is finite, A arrives at some leaf node x after at most $n-1$ swaps involving agent A. The second part of the proof exploits the fixed neighborhood enumeration. Suppose, inductively, that agent A has visited some proper subset $W \subset V$ in DFS order with respect to root r with x as the first leaf visited. There are two cases for the last swap in such a visitation sequence. If the last swap places A at some node y it visited previously, then the backtrack swap moving A to y either increments $nextswap_y$ to the next node in the DFS order (if such a previously unvisited node exists) or the increment of $nextswap_y$ refers to the first neighbor of y upon which A arrived, that is, the backtrack neighbor. This verifies the inductive hypothesis. □

Theorem 2. The protocol of Figure 3 is a stabilizing solution to the n-agent traversal problem.

Proof. Lemma 7 makes no particular assumption about the initial state, hence every computation moves all agents in DFS order, which is a solution to the n-agent traversal. □

5.2 Complexity for a Line

The results of this subsection are for a line of n processes. In this linear topology, $\delta_i = 2$ for all but the endpoints of the line, which simplifies the analysis. For any given state of the linear network, an agent A has an orientation, right or left, denoted $A \rightarrow$ or $\leftarrow A$. If A and B are neighboring nodes with A to the left of B, then one of the four possibilities of their agent's orientation is $A \rightarrow\!\leftarrow B$, which we denote by $A \leftrightarrows B$.

Lemma 8. Agents swap in the direction of orientation; agents retains one orientation until swapped to an endpoint of the line; the orientation reverses at the endpoint of the line.

Proof. These simple properties result from the fact that each swap reverses the orientations at the pair of nodes involved, but a precondition for a swap is the $\leftrightarrows$ condition. □

Lemma 9. If A and B have the same orientation and A is nearer to its target endpoint than B, then A continues to be nearer to its target endpoint so long as A and B have the same orientation.

Proof. Up to when one of A or B reaches the endpoint node, it suffices to consider the case when the two are neighbors, since they would have to be neighbors as a precondition for one overtaking the other. However, if A and B are neighbors, preconditions $A \leftrightarrows B$ or $B \leftrightarrows A$ are not possible, by Lemma 8. □

It is possible to extend Lemma 9 to show that agents arrive at nodes in deterministic order, that is, from an initial state σ, if the order of agent arrival at some node v is $\mathcal{A} = A_1, A_2, A_3, \ldots$ for one computation, then the same order $\mathcal{A}$ of agent arrival holds for *every* computation beginning from σ.

Let a sequence of consecutive nodes in the line be called an *orientation chain* if all their agents have the same orientation. For any given state the orientation chains are described by a tuple of the form $\langle r_1, \ell_1, r_2, \ell_2, \ldots, r_k, \ell_k \rangle$, where each term r_i specifies the number of nodes in a $\rightarrow$ chain and ℓ_i specifies the number of nodes in the following $\leftarrow$ chain (the endpoint orientations are constant). The *short chain* predicate Sc is defined to hold iff each term of the orientation tuple is 1, with possible exception of the endpoint terms, which are either 1 or 2.

Lemma 10. In any synchronous computation, the protocol stabilizes to Sc within $O(n)$ steps.

Proof. (Sketch.) It can be shown, for a synchronous computation, that with the exception of merging endpoint chains of length 1, no chain increases length by any step (the proof is by induction, using the fact that only swaps can modify chain lengths, and every chain terminates, following its orientation, in a $\leftrightarrows$ condition). An example of this exception is the following step: $\langle 1, 3, 1, 1 \rangle$ becomes $\langle 2, 4 \rangle$, which is depicted as follows:

$$\circ \leftrightarrows \circ \leftarrow \circ \leftarrow \circ \quad \circ \leftrightarrows \circ \quad \Longrightarrow \quad \circ \rightarrow \circ \leftrightarrows \circ \leftarrow \circ \leftarrow \circ \leftarrow \circ$$

Endpoint chains make the exception because orientations at the end of a chain do not reverse by a swap operations. For chains not subject to this exception, it remains only to show that chains of length $t > 2$ shorten in a computation. Consider a chain of length t and two subsequent steps of the computation. Since the chain terminates at a $\leftrightarrows$ pair, it will shorten by one node as the result of this swap; however, the chain could also lengthen because some swap adjacent to the opposite end adds a new node. Thus in one step, the chain either retains its length or shortens. In the case where the length is unchanged, let us suppose the chain lengthens by a new node to the left and diminishes from the right. This implies that the sum of terms occurring before the chain's terms in the

orientation tuple decreases. Therefore, after at most $O(n)$ steps, the chain must decrease in length. Since these observations hold in parallel for all chains, the protocol stabilizes to Sc within $O(n)$ steps. □

Lemma 11. In any synchronous computation, every agent traverses the network within $O(n)$ steps.

Proof. Once Sc holds, there occur at least $n/4$ swaps per step. Each agent therefore advances in the DFS visitation at least once every four steps, so $4n$ steps suffice to guarantee n-agent traversal. □

The more difficult case is asynchronous computation. To prepare for the analysis, we propose an ordering on the swap operations in a computation. Let σ be an initial state and construct an infinite directed graph, which has vertices taken from the states of a synchronous computation. More precisely, the graph is constructed inductively from an initially empty set of vertices, and then adding the n vertices representing the nodes of the network for each step of the synchronous computation. In this construction, we label each vertex as $v : i$ by the name of its network node v and the step number i from which it derives. For each step from time i to time $i + 1$, add an edge $(v : i, w : i + 1)$ to the graph iff an agent moves from node v at time i to node w at time $i + 1$. By the arguments of Lemma 11, the number of edges added for each step eventually becomes at least $n/2$. By extension of Lemma 9, the order defined by this graph is the same dependency order for agent swaps in any computation, including asynchronous ones.

Now consider an asynchronous computation, and the same graph construction to model the computation as in the previous paragraph. Here, in addition to labeling vertices by node and step, we may add a third label to specify the round number, so each vertex has a label of the form $v : i : j$. The step number i refers to the step number of the *synchronous computation* where this swap would occur rather than the step number in the asynchronous computation. Although our computational model is not a message-based one, we can use the notion of a consistent cut to relate states from the asynchronous graph to the synchronous one. A cut is a subgraph that causally includes, for each of its vertices other than ones from the initial state, all predecessor edges and vertices.

Lemma 12. Let $v : i$ be a vertex with maximum value i in a cut and $w : k$ be a vertex with minimum value k in the same cut; then $i - k < n$.

Proof. (Sketch.) An asynchronous computation allows some swaps to occur in different order from the synchronous computation, but since agents move in DFS order and the graph dependencies for swaps are the same for any computation, dependencies "fan out" to include all n nodes if one agent swaps $n - 2$ times in some period while other agents do not swap. □

Lemma 13. In any asynchronous computation, every agent traverses the network within $O(n)$ rounds.

Proof. (Sketch.) In each round, every agent located at a vertex of the form $w : k$ where k is minimal completes a swap because all causal dependencies are satisfied. The total "stretch" for such dependencies is n by Lemma 12, hence any asynchronous computation's maximum cut is at most $O(n)$ swaps behind some synchronous computation with the comparable time. □

6 Conclusion

This paper defines and investigates the problem of n-agent traversal in a model where only one agent per node is permitted in any state. The basic operation for agent movement is the agent swap, which differs from previous research in self-stabilization because the swap operation atomically changes the state of two processes. Although this model of mutual engagement differs from classical work in the area, the algorithmic techniques we used are widely used. Readers familiar with central daemon emulations will recognize one of the basic requirements of the model: two simultaneous swaps are not allowed if they are incident on a common node (in the central daemon emulation, neighbors are not allowed to execute actions in parallel). It is therefore not surprising to find self-stabilizing daemon emulation algorithms similar to the ones in this paper [16, 20, 12, 19, 11, 18].

The line topology protocol presented in Section 5 resembles algorithms given in [16, 11]. We conjecture that our analysis of traversal time can be applied to these daemon emulators. The combination of "maximal concurrency" complexity [11] and the causal dependency of the algorithm could give insight for measuring round-based complexity of these emulators.

References

1. A Arora, S Dolev, and MG Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1:11–18, 1991.
2. A Bui, AK Datta, F Petit and V Villain. Snap-stabilizing PIF algorithm in the tree networks without sense of direction. In *Proceedings of the 6th International Colloquium on Structural Information and Communication Complexity (SIROCCO'99)*, Carleton University Press, pp. 32–46, 1999.
3. A Bui, AK Datta, F Petit and V Villain. State-Optimal Snap-Stabilizing PIF in Tree Networks. In *Proceedings of the 4th Workshop on Self-Stabilizing Systems (WSS'99)*, IEEE Computer Society Press, pp. 78–85, 1999.
4. M Bui, AK Datta, O Flauzac, DT Nguyen. Randomized adaptive routing based on mobile agents. *Proceedings of International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN)*, pp. 380-385, 1999.
5. E Bonabeau and G Théraulaz. Swarm smarts. *Scientific American*, pp. 72-79, March 2000.
6. JM Couvreur, N Francez, and MG Gouda. Asynchronous unison. In *ICDCS'92 Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 486–493, 1992.
7. S Dolev and T Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 3(4), 1997.

8. S Dolev, DK Pradhan, and JL Welch. Modified tree structure for location management in mobile environments. *Computer Communications*, 19:335–345, 1996.
9. S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.
10. S Ghosh. Agents, distributed algorithms, and stabilization. In *Computing and Combinatorics (COCOON 2000), Springer-Verlag LNCS:1858*, pp. 242–251, 2000.
11. MG Gouda and F Haddix. The linear alternator. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pp. 31–47. Carleton University Press, 1997.
12. MG Gouda and F Haddix. The alternator. In *Proceedings of the Third Workshop on Self-Stabilizing Systems (published in association with ICDCS'99 The 19th IEEE International Conference on Distributed Computing Systems)*, pp. 48–53. IEEE Computer Society, 1999.
13. MG Gouda and T Herman. Stabilizing unison. *Information Processing Letters*, 35:171–175, 1990.
14. SKS Gupta, A Bouabdallah, and PK Srimani. Self-stabilizing protocol for shortest path tree for multi-cast routing in mobile networks (research note). In *Euro-par 2000 Parallel Processing, Proceedings LNCS:1900*, pp. 600–604, 2000.
15. CAR Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.
16. ST Huang. The fuzzy philosophers. In *Parallel and Distributed Processing (IPDPS Workshops 2000), Springer-Verlag LNCS:1800*, pp. 130–136, 2000.
17. DS Milojicic, F Douglis (Editor), RG Wheeler. *Mobility: Processes, Computers, and Agents*, Addison-Wesley, 1999.
18. M Mizuno and H Kakugawa. A timestamp based transformation of self-stabilizing programs for distributed computing environments. In *WDAG'96 Distributed Algorithms 10th International Workshop Proceedings, Springer-Verlag LNCS:1151*, pp. 304–321, 1996.
19. M Mizuno and M Nesterenko. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letters*, 66(6):285–290, 1998.
20. M Nesterenko and A Arora. Stabilization-preserving atomicity refinement. In *DISC99 Distributed Computing 13th International Symposium, Springer-Verlag LNCS:1693*, pp. 254–268, 1999.
21. G Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

A Composite Stabilizing Data Structure*

Ted Herman and Imran Pirwani

Department of Computer Science, University of Iowa
{herman,pirwani}@cs.uiowa.edu

Abstract. A data structure is stabilizing if, for any arbitrary (and possibly illegitimate) initial state, any sequence of sufficiently many operations brings the data structure to a legitimate state. A data structure is available if, for any arbitrary state, the effect of any operation on the structure is consistent with the operation's response. This paper presents an available stabilizing data structure made from two constituents, a heap and a search tree. These constituents are themselves available and stabilizing data structures described in previous papers. Each item of the composite data structure is a pair (key,value), which allows items to be removed by either minimum value (via the heap) or by key (via the search tree) in logarithmic time. This is the first research to address the problem of constructing larger data structures from smaller ones that have desired availability and stabilization properties.

1 Introduction

Availability is an important topic in online system design. Ideally, a system should respond to requests in a timely manner in spite of hardware failures, bursts in load, internal reconfigurations, and other disruptive factors. The usual technique for ensuring availability is to engineer a system with sufficient redundancy to overcome failures and resource shortages [13, 11, 5].

One attraction of self-stabilization is that it does not require the traditional type of resource redundancy to deal with faults. The question then comes to mind, can self-stabilization be enhanced to support system availability? We begin to address this question with a low-level task, which is to consider data structures. Since data structures are frequently used in software for systems, the key question is: can data structures that support availability in spite of transient failures be constructed? The answer is not obvious, since most operations on data structures either abort, get caught in a loop, throw exceptions, or have unpredictable behavior when internal variables have invalid values. Specifically, we study one data structure in this paper, a composite data structure made from a heap and a search tree. This data structure is of interest because it shows how one available data structure can be constructed from smaller, available data structures (of course, general compositional methods are the ultimate goal, but examples are helpful to understand the technical difficulties).

* This research is sponsored by NSF award CAREER 97-9953 and and DARPA contract F33615-01-C-1901.

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 167–182, 2001.

The literature of self-stabilization differs from our treatment in several ways. First, we do not consider a distributed system, which is the normal model for self-stabilization [3]. Second, most self-stabilizing algorithms make no guarantees about behavior before the system reaches a legitimate state — whereas we require some guarantees for availability. Third, the data structure model of operations in this paper restricts transient faults to effect only the variables of the data structure and not the internal variables of an operation underway (in principle, our results could be extended to allow such corruption as well). Our data structure results are stabilizing in the following sense. Initially, the content of a data structure may be arbitrary and corrupt. During some initial sequence of operations on the data structure, the response time of each of these operations could be abnormally large, but not larger than that for a legitimately full data structure; and some operations during this sequence could respond with errors, such as reporting that an item could not be inserted, even though the data structure is not full. In all cases, however, the response is consistent with how the operation changed the data structure. Finally, after a sufficiently long sequence of operations, the data structure's state is legitimate, and thereafter all operations have normal running times and responses.

While there are numerous studies of fault-tolerant data structures, the fault model for these studies does not consider recovery from unlimited transient faults in the data structure. Self-stabilization is required to deal with unlimited transient faults, yet very few papers in the area of self-stabilization treat data structures in the model of operations applied to the structures. Two works [12, 10] mention self-stabilization the context of objects that undergo operations before a legitimate state is reached. The challenge of [10] is that operations are concurrent and wait-free, which is an issue beyond the scope of our present investigation. The notion of an available and stabilizing data structure is new in [8], which presents an available, stabilizing, binary heap. An available and stabilizing 2-3 tree is described in [9].

The data structures investigated in this paper and related papers [8, 9] are fixed-capacity structures. The reason for fixing the capacity is to include the design of dynamic storage allocation in the implementation. Standard texts presenting data structures may gloss over the role of allocation in dynamic structures, but for the case of stabilization, an incorrect initial state can make the storage allocation pool appear empty even though the data structure contains very few items. The situation is even more complex when numerous data structures share a common allocation pool. At this stage of research, we investigate single data structures, hence the constraint of fixed capacity to make the presentation self-contained. Also, the data structure operations considered here are single-item operations (insert, delete, find); operations such as set intersection or union are not investigated. Many research directions thus remain unexplored, including inter-structure operations, data structures of unlimited capacity, and shared allocation pools (and of course, the question of concurrent operations).

Organization of Paper. Section 2 describes the model of data structures and operations, and then defines the availability and stabilization precisely. Section 3

presents a simple example of an data structure and briefly reviews the constructions for heap and search tree. Section 4 defines the main problem we consider, and sketches an impossibility result related to this problem. Section 5 gives an overview of the main construction. Detailed correctness proofs of the construction are omitted from this extended abstract. The paper ends with discussion in Section 6.

2 Availability, Stabilization, and Operations

A data structure is an object containing items. The object is manipulated by a fixed set of operations. Each type of operation is defined by a signature (operation name and invocation parameters) and a set of responses for that signature. The relation between a signature and its response is specified by sets of sequential histories of operations applied to the data structure. That is, we specify operation semantics by a collection of legal sequences rather than by presenting pseudocode. We do this in order to maximize the freedom of the object's implementor to choose data representation and algorithms.

A history is an infinite sequence of pairs $\langle (op_1\ res_1)\ (op_2\ res_2)\ \cdots \rangle$ where op_i is the i-th operation invocation (including its parameters) and res_i is the response to op_i. A *point* in a history refers to the state of the data structure either before any operation or between two operations op_i and op_{i+1}. The *content* of the data structure at a point t is defined directly if t is the initial point of the history; if t is not the initial point, then the content is defined in terms of the sequence of operations and responses leading up to point t. Let C_t denote the content at point t. Let $|C_t|$ denote the number of items in the data structure at point t. All data structures have a fixed *capacity*, which is an upper bound on the number of items the data structure is allowed to contain.

We illustrate content specification by history with a small example, a data structure for a set of at most K items with `insert` and `remove` operations. Let the content of the set be empty at the initial point of any history. The content C_t at point t following operation op_t is defined recursively: if op_t is an `insert`(x), and $x \not\in C_{t-1} \wedge |C_{t-1}| < K$, then $C_t = C_{t-1} \cup \{x\}$ and the response to the operation is *ack*; if op_t is an `insert`(x) and $|C_{t-1}| = K \vee x \in C_{t-1}$, then $C_t = C_{t-1}$ and the response to the operation is *full* if $|C_{t-1}| = K$ or *ack* otherwise. If op_t is a `remove`(x), then $C_t = C_{t-1} \setminus \{x\}$ and the response is *ack*.

Throughout the paper we assume the following for data structures and their operations. Operations are single object methods and we suppose that all operations in a history are operations on the same data structure (so the example above cannot be extended to allow set union and intersection operations, which combine objects). An operation op_t is called *moot* if $C_t = C_{t-1}$. We classify each operation op_t as either *successful* or *unsuccessful* depending on its response, and this classification is specified as part of the suite of operations for a data structure. In this classification, all unsuccessful operations are moot, but the converse need not hold.

The intuition for "unsuccessful" classification is that, although the operation is guaranteed to be moot, the response of the operation may not be trustworthy if the history begins with a data structure damaged by a transient fault. We formalize this below in the definitions of availability and stabilization. For the set example above, only the response *full* could be classified as an unsuccessful operation; if *full* is an unsuccessful response, then intuitively, in a damaged state, an `insert` operation that responds *full* is allowed to do so even though the set contains fewer than K items.

A *well-formed* history is one where the responses of all operations agree with the definition of content for the data structure. A detailed specification of what is means for a history to be well-formed depends on the semantics for the data structure operations, their responses, and the definition of content. A *legitimate* history is a well-formed history so that the operation running times are within the bounds, as a function of content, given by the implementation for that data structure. For example, suppose the set data structure has an implementation where the running time for an operation on a set of size m is linear in m. An example of an illegitimate history is one where an `insert`(x) operation fails at some point t even though $|C_t| < K$. Another example of illegitimacy is a `remove`(x) operation that has 2^m running time at point t although $|C_t| = m$. Observe that if all moot operations are removed from a legitimate history, the result is either a legitimate history or a (finite) prefix of a legitimate history.

Given history H, let H_t denote the suffix of H following point t. Let $U(I)$, for a history I or segment I of a history, be the sequence obtained by removing all unsuccessful operations from I. A history H is *available* if there exists a point t and a sequence of operation invocations P such that $P \circ U(H_t)$ is a well-formed history or a (possibly empty) finite segment of a well-formed history ($\circ$ is the sequence catenation operator); also, the running time of any operation in H is no more than the worst-case running time of any operation, taken over all legitimate histories (usually this worst-case running time is obtained for an operation on a full data structure). Because unsuccessful operations are moot, it follows that legitimate histories are available histories. Examples of histories that are available but not legitimate include histories with operations whose running times larger than one would expect for the content, and operations that return unsuccessful responses in unexpected cases (e.g. an `insert`(x) responds *full* even though the content has less than the data structure's capacity number of items). An implementation of a data structure is available iff all its histories are available. A trivial implementation of a set to guarantee availability would be to have any `remove` do nothing to the data structure and respond *ack* in $O(1)$ time, and to have any `insert` do nothing to the data structure and respond *full* in $O(1)$ time. To see that this implementation is available, take any history H, let t be the initial point of H, let P be empty, and choose C_t to be the empty set. Then $P \circ U(H_t)$ is a legitimate history since it has no `insert` operations and `remove` operations are moot.

A history H is *stabilizing* if there exists a point t and a sequence of operation invocations P such that $P \circ H_t$ is a legitimate history. In the definition of a

stabilizing history, there can be many choices for the point t and prefix P to make $P \circ H_t$ a legitimate history; call t the *stabilization point* if there exists no point s preceding t such that $Q \circ H_s$ is a legitimate history for any prefix Q. An implementation of a data structure is stabilizing iff all its histories are stabilizing. The *stabilization time* of a stabilizing history H is the number of operations preceding its stabilization point. The stabilization time of an implementation of a data structure is the maximum stabilization time of all of its histories (if there is no maximum, then the stabilization time is infinite).

3 Available and Stabilizing Components

Before presentation of the composite data structure in Section 4, we review first the constituent data structures used to build the composite. Some constituents are trivially available and stabilizing: for instance, we may regard an atomic variable (word of memory) to be a data structure supporting `read` and `write` operations. It is simple to show that such variables are available and stabilizing. Interesting data structures such as heaps and search trees require nontrivial constructions to satisfy availability and stabilization. Between the trivial case of a variable and more advanced data structures, we consider first in this section two elementary data structures, a queue and a type of stack, to illustrate some challenges of implementing availability and stabilization.

3.1 Queues, Stacks, and Conservative Implementations

Two elementary examples of available and stabilizing data structures are described here, a queue and a type of stack. We then show two variations on the queue that do not have available and stabilizing implementations. These variations on the queue are somewhat artificial, but they are useful to illustrate difficulties that arise later in the presentation of a composite data structure. An important concept introduced in this section is a *conservative* implementation of a data structure. Informally, an implementation is conservative if operations cannot substitute values for missing data.

A *K-queue* is a queue with a capacity of K items supporting `enqueue` and `dequeue` operations. The response to an `enqueue`(x) operation is either *ack* or *full*, and the response to a `dequeue` is either an item or *empty*. Only the *full* response is classified as an unsuccessful response. The content C_t at any point t can be defined from the sequence of successful `enqueue` operations up to point t and the sequence of `dequeue` operations that return items up to point t (we omit the formal definition). In any legitimate history, the running time of an operation is $O(1)$. In any well-formed history, responses have the following properties. An `enqueue` operation at point t responds *full* iff $|C_t| \geq K$, and otherwise responds *ack*; a `dequeue` at point t responds *empty* iff $|C_t| = 0$ and otherwise responds with the initial item of sequence C_t.

The conventional implementation of a bounded queue by a circular array is an available and stabilizing K-queue. Let $\mathtt{A}[0..(K-1)]$ be an array of K

items and let `head` and `tail` be two integer variables. The K-queue is empty if (`head` mod K) = (`tail` mod K), and full if ((`tail` + 1) mod K) = (`head` mod K). An `enqueue`(x) succeeds iff the K-queue is non-full by writing x to `A`[(`tail`+ 1) mod K] and then assigning `tail` $\leftarrow$ (`tail` + 1) mod K (the capacity of this queue representation is thus $K-1$ items). The `dequeue` is similarly defined, with the result that the state of the K-queue is well-defined for all possible values of `head`, `tail`, and `A`. By straightforward expansion of the definitions, availability and stabilization hold for this implementation and the stabilization time is zero.

The composite data structure described in Section 5 makes use of a relative of the K-queue. A *lossy K-stack* is a stack with a capacity of K items and `push` and `pop` operations. A `push` operation for the lossy K-stack always succeeds and a `pop` operation always returns an item. The K-queue can be implemented by an array of K elements and a `top` variable to contain an index for the top of the stack, which is incremented modulo K by `push` and decremented modulo K by `pop`. This stack is "lossy" because, for $K+1$ consecutive `push` operations, the last `push` writes over the oldest item on the stack. We omit the straightforward details of the formal definition of this data structure and verification of its availability and stabilization properties.

The elementary examples of queue and stack become more challenging when there are domain restrictions on the items contained in the data structure. Consider a queue that may only contain (pointers to) prime numbers as items. The `enqueue` operation can return a new response *err*: the response to `enqueue`(x) is *err* for any nonprime x, and the operation is moot. The running time of `enqueue` is no longer constant, because `enqueue` tests its argument for primality. The `dequeue` operation has constant running time, either returning (a pointer to) an item or the *empty* response. Now suppose we have an implementation of this data structure and let σ be the state for some queue of prime numbers. It is possible for a transient fault to transform σ into some σ' by changing some (pointers to) items in the queue from prime to nonprime values. This is a problem because the `dequeue` operation can now return a nonprime number. Returning a nonprime number is not possible in a legitimate history and because an item in response to `dequeue` is a successful operation, there is a conflict with availability if `dequeue` is allowed to return a nonprime number. The only resolution is for `dequeue` to check an item for primality before responding. We do not know of any method to test primality in constant time, so `dequeue` will require nonconstant running time for faulty states such as σ'. If the implementation is also stabilizing, then every history has a suffix where all `dequeue` operations have constant running time. Yet it is impossible for `dequeue` to distinguish between correct and faulty states in constant time unless there is a constant-time primality test. Therefore, every `dequeue` operation on a nonempty queue checks for primality, which conflicts with the running time constraint for a stabilizing implementation. We have thus sketched the proof of the following.

Lemma 1. No implementation of the prime queue is available and stabilizing.

Another difficulty with a domain-restricted queue is shown by a queue that may only items of the form *a:b* with a an even number. Again, execution of

`enqueue`(*x:y*) reponds with *err* for odd x, but this can be checked in constant time. The `dequeue` operation can also verify that the head item has an even first component in constant time, so all operations run in constant time for a legitimate history. After a transient fault, there could be queue items with odd first components, so the question arises, what should `dequeue` do if it detects that the head is odd? One choice would be to respond with *0:z* in place of any item *r:z* with odd r found at the head of the queue. This choice would satisfy availability, because there is a legitimate history where zero is enqueued in place of any odd value (notice this method cannot be used for the prime number queue because of running time constraints). If there are further domain restrictions on items, say a relation between a and b in a pair *a:b*, or perhaps relationships with other variables outside the queue, then the substitution of *0:z* for *r:z* would not be valid.

We call an implementation *conservative* if no operation returning a data structure item is allowed to invent the item — a conservative implementation can only return items that are present in the data structure, and cannot coerce invalid values to legal ones. This is a key decision in our presentation, and differs from classical work on self-stabilizing control structures. For classical problems such as mutual exclusion, it does not matter how illegitimate states are converted into legitimate ones, but for the stabilization of data structures, we prefer to limit techniques to conservative measures where data items are not created to satisfy a domain constraint. The motivation for conservative operations is not just to make a theoretical problem nontrivial: in practice, stabilizing algorithms that limit the effect of faults are preferable to those that do not enforce any such limit [4, 6, 7]. While it is true that a transient fault could inject apparently legal values never actually inserted in a data structure, such transient faults are uncontrolled, whereas operation implementation can be designed to avoid the injection of artificial values (moreover, practical techniques such as error detecting codes can decrease the probability of legal values injected by faults).

Lemma 2. No conservative implementation of the *x:y* queue is available and stabilizing.

The lemma can be shown by an adversary argument: each `dequeue` has to distinguish, in $O(1)$ time, whether the queue is empty or has an item meeting the domain constraint, and for whatever strategy `dequeue` uses to examine the queue, an initial state can be constructed to defeat that strategy.

3.2 Heap and Search Tree Review

The heap and search tree data structures are more complicated than the queue, and items in the heap and search tree have domain restrictions, unlike a simple queue. In the heap, an item's value is the least of a subtree of values, and in a search tree, item keys are ordered. There is, however, a crucial way in which these domain restrictions are simpler than the example of the prime number queue: there is an implementation so that all operations access items via a path

from the tree's root. This fact generalizes to the observation that, for any values of items in a such a tree, there is a maximal subtree for which the domain restrictions hold. For the heap this is called the *active heap* and for the search tree this is called the *active tree*.

The available and stabilizing search tree given in [9] is a 2-3 tree. Its active tree is defined as the maximal subtree so that the distance from root to leaf is the same for all leaves, all keys are in order, and several other validity conditions hold for child and parent pointers. For the available and stabilizing heap [8], the active heap is taken to be the maximal rooted subtree so that each node's value is a lower bound on the values of its children. The table below shows the signatures and responses for the 2-3 tree and heap structures. The method to ensure availability is straightforward: all data structure operations are implemented with respect to the active structure. A consequence of this method is that the active structure can be unbalanced, so that operations on the active tree may have no longer have running that is logarithmic in the number of active items. The remaining implementation task is stabilization, which entails balancing the active structure. Informally, the balancing of the active structure is a "background activity" similar to garbage collection in memory allocation schemes. This background activity is also needed to repair pointers, repair free storage chains, and correct various other internal variables of the data structure. Because no actual background process is assumed by the model of data structures, each operation on the data structure invokes a limited amount of background processing. The running time of any call to background processing is $O(\lg K)$ in [8, 9] in an arbitrary state and $O(\lg n)$ after the stabilization point, where K is the capacity of the data structure and n is the number of items in the active structure. These running times are the same as the operation complexities, since in the worst case, an active heap or search tree encounters a path of length $O(\lg K)$, and when the structure is balanced, all paths are $O(\lg n)$.

signature	successful	unsuccessful	illegitimate	legitimate
`STinsert`(k, d)	*ack*	*full*	$O(\lg K)$	$O(\lg n)$
`STdelete`(k)	*ack*		$O(\lg K)$	$O(\lg n)$
`STfind`(k)	$(k, d)/$*missing*		$O(\lg K)$	$O(\lg n)$
`Hinsert`(v, e)	*ack*	*full*	$O(\lg K)$	$O(\lg n)$
`Hdeletemin`$(\ )$	$(v, e)/$*empty*		$O(\lg K)$	$O(\lg n)$
`Hdelete`(p)	*ack*		$O(\lg K)$	$O(\lg n)$

4 Composite Data Structure

The composite data structure presented here is called the *heap-search* tree. Definitions of the heap-search operations are explained in this section; the construction of the heap-search tree is presented in Section 5. The table below presents our initial table of the signatures and responses of the operations; later in this section we revise this table, after presenting an impossibility result for a conservative implementation.

signature	successful	unsuccessful	illegitimate	legitimate
`insert`(k, v)	*ack*	*full*	$O(\lg K)$	$O(\lg n)$
`delete`(k)	*ack*		$O(\lg K)$	$O(\lg n)$
`find`(k)	(k, v)/*missing*		$O(\lg K)$	$O(\lg n)$
`deletemin`( )	(k, v)/*empty*		$O(\lg K)$	$O(\lg n)$

Each item of the heap-search tree is a pair (k, v), and the content of the heap-search tree at any point is a multiset (bag) of such pairs. Below we use union ($\cup$) and subtraction ($\setminus$) for multiset operations, so $C \setminus \{(k, v)\}$ removes at most one copy of (k, v) from multiset C. Let s and t be consecutive points in a legitimate history and let the operation and response occur between s and t. Operation `insert`(k, v) is moot and responds *full* if $|C_s| \geq K$; otherwise $|C_s| < K$ and `insert`(k, v) responds *ack*, with $C_t = C_s \cup \{(k, v)\}$. Operation `delete`$(k)$ responds with *ack*; the operation is moot if there exists no b satisfying $(k, v) \in C_s$, otherwise $C_t = C_s \setminus \{(k, v)\}$ for some v satisfying $(k, v) \in C_s$. Operation `find`(k) is always moot, and either returns a pair (k, v) for some b satisfying $(k, v) \in C_s$, or returns *missing* if no such pair exists. Operation `deletemin`( ) returns *empty* and is moot if $|C_s| = 0$, otherwise the response is a pair (k, v) such that v is the minimal value for any pair's second component, and $C_t = C_s \setminus \{(k, v)\}$ in this case.

Before making statements about heap-search tree implementations, we first formalize what it means for an implementation of this data structure to be a composite of the heap and 2-3 tree. Informally, the implementation is a composite if it is a construction made by assembling one heap and one 2-3 tree, so that any pair $(k, v) \in C_s$ is represented by the item k in the 2-3 tree at point s and v in the heap at point s, and no other data structures are used in the storage of items. More formally, the implementation of the heap-search tree satisfies the following three constraints: (*i*) the composite heap-search tree implementation has one 2-3 tree S, one heap T, and possibly other data structures used for background processing; (*ii*) at any point s in a history, $(k, v) \in C_s$ iff $(k, d) \in S_s$ and $(v, e) \in T_s$, where S_s and T_s respectively denote the contents of the 2-3 tree and heap at point s, with d and e being associated data as defined by operations; and (*iii*) for any $(k, v) \in C_s$, the key k is contained only in the 2-3 tree — keys are not contained in any structure other than the 2-3 tree; similarly, the value v is not contained in any other structure than the heap. Note that (*iii*) could be ambiguous for a structure with integer keys as well as integer heap values, because a key could coincidentally reproduce a heap value. To resolve this ambiguity, assume that keys and heap values are taken from different types (say key and value) for purposes of defining constraint (*iii*).

Lemma 3. No conservative implementation of the composite heap-search tree using the heap and 2-3 tree is available and stabilizing.

The conclusion we draw from Lemma 3 is that either we should settle for an implementation that is not conservative, or the operations should be modified. The proof of Lemma 3 (omitted from this extended abstract) points out that

`deletemin` is responsible for the difficulty, so we propose the following change: let `deletemin` have a new response *err*, to be returned if the value returned by `Hdeletemin` does not have a corresponding key in the 2-3 tree. This change satisfies availability by classifying any `deletemin` with an *err* response as an unsuccessful operation. However to satisfy stabilization, we shall require that no `deletemin` be unsuccessful after the stabilization point, since the *err* response does not occur in a legitimate history. For application purposes, *err* is useful because it informs the `deletemin` caller that something is incorrect in the heap-search data structure, but by returning within $O(\lg K)$ time, quickly gives the application the choice of repeating `deletemin` (and progressing toward stabilization) or using some other type of recovery. Lemma 3 can also be proved using the `find` operation instead of `deletemin`, because the search tree may contain duplicate keys in an initial state; the `delete` has another similar difficulty. Therefore, for the remainder of the paper, let `delete`, `find`, and `deletemin` return *err* if the operation is unsuccessful.

5 Heap and 2-3 Tree Composite Construction

5.1 Variables, Constituent Structures, and Pointers

The composite data structure is composed of two binary variables `STbit` and `Hbit`, a variable `curcolor` with domain {0,1,2}, a K-lossy stack, a heap, and a 2-3 tree. As explained earlier, all of these constituents (variables and structures) are available and stabilizing components. Throughout the remainder of the paper, K is the capacity of the heap-search tree and also the capacity of the heap and 2-3 tree. For convenience, we use the term `nextcolor` to mean $(1 + \texttt{curcolor})$ mod 3 and the term `prevcolor` to mean $(2 + \texttt{curcolor})$ mod 3.

Our construction uses pointers to connect heap items and 2-3 tree items. The use of pointers in conventional random access memory is challenging because damaged pointers can lead to further damage in data structures (for instance, modifying data accessed by a damaged pointer). We make some assumptions concerning how data, especially the heap and 2-3 tree, are arranged in memory. We assume that a procedure `Hpointer`(p) evaluates p and in $O(1)$ time returns *true* if p could be an item of the heap, and *false* otherwise. An implementation of `Hpointer`(p) could check that p is an address within a range $[Hstart, Hend]$ defined for heap items, and also check that p is a properly aligned address ($Hstart$ and $Hend$ are program constants not subject to transient fault damage). Similarly, we assume there is a procedure `STpointer`(p) to determine whether p can be a 2-3 tree item.

The next level of pointer checking is to determine whether or not a given p is a pointer to an item in the active structure. Let `STintree`(p) respond *true* if p is a pointer to an item in the active 2-3 tree, and *false* otherwise. The running time of `STintree`(p) is $O(h)$ where h is the height of the active 2-3 tree. The implementation of `STintree`(p) could be similar to one described in [9]; after using `STpointer`(p) to validate candidacy of p, the procedure follows parent pointers of tree items to verify that the tree's root is an ancestor, and also

verifies properties of keys in items are such that all items in the path are in the active 2-3 tree. A similar procedure `Hinheap`(p) determines whether p is an item of the active heap. `Hinheap`(p) can be implemented by tracing the parentage of p back to the heap's root. Unlike `STintree`(p), which consumes $O(h)$ time, the complexity of following p's parentage for arbitrary p satisfying `Hpointer`(p) is $O(\lg K)$ — the running time is $O(h)$ if p is an item of the active heap. In some cases, the time bound for `Hinheap`(p) should be constrained, even if the result is inaccurate. Let `Hinheap`(p, t) be an implementation that limits the parentage trace to t iterations, and if t iterations do not suffice to reach the heap's root, `Hinheap`(p, t) returns *false*.

Each item of the K-lossy stack is a pointer. Items of the heap and 2-3 tree provide for a data field in the respective `insert` operations (see (k, d) and (v, e) in the table defining operations), and we use these data fields for pointers. For an item (k, d) of the 2-3 tree, d is a pointer. For an item (v, e) of the heap, $e = (q, c)$ with q being a pointer and c being a "color" in the range $\{0,1,2\}$. Our convention is to refer to d as the pointer associated with the 2-3 tree item (k, d), to call q the pointer associated with heap item (v, e), and to call c the color associated with (v, e). We also use the notation x.`color` to refer the color of an item x.

At any point in a legitimate history, the pointers associated with items in the active heap and active 2-3 tree should bind a pair (k, v), which means that the pointer d associated with k in the 2-3 tree refers to an item with value v in the heap, and in turn the pointer q associated with v refers to the item (k, d). Let `STcross`(p), for p a pointer to a 2-3 tree item (k, d), be a boolean function returning *true* only if `Hpointer`(d) is *true* there is a pointer q associated with the heap item of d, and $q = p$. A similar function `Hcross` is defined for heap items, and we generically use the term *crosscheck relation* to mean that the pointer associated with an item, heap or 2-3 tree, refers to an item in the other structure that has the expected back pointer; we say that two items crosscheck if they satisfy the crosscheck relation. Note that the crosscheck relation can be checked in $O(1)$ time, but satisfying the crosscheck relation does not imply membership in the active heap or 2-3 tree.

5.2 Modifications to Constituent Operations

We change the operations of the heap and 2-3 tree only by adding a some extra steps to look after the pointers and the color field associated with a heap item. The reason that pointers are an issue is that items can change location inside a data structure as a result of insertions or deletions. For the 2-3 tree, this can occur by node splitting or merging. For the heap, this occurs by item swaps as part of "heapify" routines to restore the heap property after an item is removed or added.

The change relating to pointers is: whenever an operation moves an item x, the operation validates x by a crosscheck, and if the crosscheck holds, then the operation adjusts pointers so that the crosscheck relation will be hold after the item moves. Conversely, if the crosscheck relation does not hold for x prior to the move, then the pointer is forcibly invalidated so that crosscheck will not hold

(accidentally) as a result of moving x. This change applies to both heap and 2-3 tree operations, since procedures of both can move their respective items within the active structures. Background activities that move items are also changed to attend to item movement. In particular, the background heap activity includes an `balance` routine that deletes an item of maximum depth, reinserting it at minimum possible depth; the movement of this item by `balance` requires pointer adjustment so that subsequent crosschecks are valid.

The change relating to colors only applies to heap operations and background activities. Whenever an operation or background routine examines an active heap item with color c, it pushes a pointer to that item on the K-lossy stack if $c =$ `nextcolor`, and then assigns the item's color to be the value of `curcolor`. A single operation or background routine for the heap may examine many nodes (for instance, examining all the nodes along a path from root to a leaf), so an operation can push many pointers on the stack as a result of this change. However, since checking the color field and stack pushes are $O(1)$ time steps, the operation complexities of the heap are unchanged.

Two background activities of the constituent structures have new duties in the composite data structure. First, we review terminology for the existing background operations. The heap has a background operation called `Hscan` and the 2-3 tree has a similar operation called `STscan`. One intent of these background operations is the same for both structures, which is to examine a path of nodes from root to leaf, possibly truncating nodes that are not part of the active structure, correcting variables within nodes, and in the case of the 2-3 tree, possibly merging nodes. One `Hscan` invocation can examine up to $\lg K$ items, since each node in the active heap contains an item. One `STscan` invocation examines two items by looking at two paths from root to leaf (only the leaves of the active 2-3 tree contain items). In any sequence of operations on a structure the paths chosen by `Hscan` or `STscan` advance through the tree in a standard "left to right" order. The path chosen in an initial state is unpredictable, but after examining the rightmost path, the next invocation returns to the leftmost path, so that all nodes will be examined in any sequence of (sufficiently many) operations.

For the composite data structure, we add a step to `Hscan` in only one occasion, which is immediately after the rightmost path of the heap is examined: the assignment $\texttt{Hbit} \leftarrow f_{\texttt{H}}(\texttt{Hbit}, \texttt{STbit})$ is executed, for

$$f_{\texttt{H}}(x,y) \;=\; \begin{cases} x & \text{if} \quad x = y \\ 1 - x & \text{if} \quad x \neq y \end{cases}$$

If $\texttt{Hbit} = 1 \;\wedge\; \texttt{STbit} = 1$ as a result of this assignment, then the assignment $\texttt{curcolor} \leftarrow \texttt{nextcolor}$ is executed.

One change to `STscan` resembles the change to `Hscan`: immediately after `STscan` examines the rightmost path of the 2-3 tree, the assignment $\texttt{STbit} \leftarrow f_{\texttt{ST}}(\texttt{Hbit}, \texttt{STbit})$ is executed, where

$$f_{\texttt{ST}}(x,y) \;=\; \begin{cases} y & \text{if} \quad x \neq y \\ 1 - y & \text{if} \quad x = y \end{cases}$$

The remaining change to `STscan` adds extra work in the examination of a 2-3 tree item. When `STscan` examines an item (k, d), the procedure crosschecks (k, d); if the crosscheck indicates that k does not have a corresponding heap value, then `STscan` removes (k, d) from the active 2-3 tree. If the crosscheck test passes, `STscan` then evaluates `Hinheap`(d), and removes (k, d) from the active 2-3 tree if `Hinheap`(d) is *false*. Finally, if (k, d) passes crosscheck and `Hinheap` tests, then `STscan` assigns $c \leftarrow$ `curcolor` where c is the color variable of the heap item associated with k.

5.3 New Background Activities

Each invocation of a heap operation contains a call to `Hscan`. Each invocation of a 2-3 tree operation contains a call to `STscan`. The modifications of these background activities described in Section 5.2 supply most of the effort needed to stabilize the composite structure; only one additional, new procedure is needed. We call this new routine `trimheap`. Procedure `trimheap` is called once in the execution of any of the composite operations, and consists of steps shown in Figure 1. By reasoning about the running times of `Hpointer`, *crosscheck*, `STintree`, `Hinheap`, and `Hdelete`, it follows that any call to `trimheap` requires at most $O(\lg K)$ time at an arbitrary state and $O(\lg m)$ time at a legitimate state for a heap-search tree containing m items (at a legitimate state, the heap's root has an accurate *height* field, which then is used to limit the running time of `Hinheap`).

We also suppose that each of the composite operations include calls to `STscan` and `Hscan`. Such calls will already be included whenever an operation invokes one of the appropriate constituent operations, however not all composite operations invoke constituent operations on both heap and 2-3 tree structures. The `find` operation, for instance, invokes `STfind` but does not include any heap operation; and although `deletemin` invokes both `Hdeletemin` and `STdelete` in a legitimate history, it may not do so before the stabilization point, as we show in Section 5.4. Therefore we suppose that `trimheap` includes calls to `Hscan` and `STscan` as needed, to ensure these constituent background activities execute with each operation in any history.

5.4 Operations of the Composite

The four operations of the heap-search tree have psuedo-code listed in Figure 1. We suppose in this figure that if `STfind`(k) returns an item (a, b), then a pointer to item (a, b) in the 2-3 tree component is available or can be obtained in $O(1)$ time. Also, for the sake of brevity, we do not provide details for how to deal with duplicate key values; we assume that a `STdelete`(k) will remove the item return by a `STfind`(k) immediately preceding, and suppose similar behavior for insertions.

The usage of *crosscheck* in the code for `deletemin` bends our earlier explanation of checking item pointers. This *crosscheck*(p) should check that the 2-3 tree item referred to by p is an item (k, d) such that d points to the root of the

```
trimheap
    q ← pop( )
    if (¬Hpointer(q) ∨ ¬crosscheck(q)) return
    p ← search tree item for q
    if ¬STintree(p) return
    t ← height field of heap's root
    if Hinheap(q, t) then Hdelete(q)
```

```
find(k)
  if (STfind(k) = missing)
      return missing
  (a, b) ← STfind(k)   // a = k
  p ← address of item (a, b)
  if (STcross(p) ∧ Hinheap(b))
      (v, e) ← heap item via pointer b
      return (k, v)
  STdelete(k)   // delete invalid key
  return err
```

```
delete(k)
  if (STfind(k) = missing) return ack
  (a, b) ← STfind(k)   // a = k
  p ← address of item (a, b)
  if (STcross(p) ∧ Hinheap(b))
      Hdelete(b)
      STdelete(k)
      return ack
  STdelete(k)   // delete invalid key
  return err
```

```
deletemin( )
  t ← Hdeletemin( )
  if (t = empty) return empty
  (v, (p, c)) ← t
  if (crosscheck(p) ∧ STintree(p))
      (k, d) ← 2-3 tree item via pointer p
      STdelete(k)   // delete (k, d)
      return (k, v)
  return err
```

```
insert(k, v)
  d ← temporary value
  if (STinsert(k, d) = full)
    return full
  // locate ST item just inserted
  (a, b) ← STfind(k)
  t ← address of item (a, b)
  e ← (t, curcolor)
  if (Hinsert(v, e) = full)
    STdelete(k)   // backout
    return full
  replace b within 2-3 item by
    b ← address of (v,e) in heap
  return ack
```

Fig. 1. `trimheap` procedure and composite data structure operations.

heap — since the heap item t returned by `Hdeletemin` was located at the root of the heap prior to being removed.

5.5 Verification

For a given point t in a history of operations on the heap-search tree, let $\mathtt{H}_t$ be the bag of items in the active heap and let $\mathtt{ST}_t$ be the bag of items in the active search tree. Let A_t denote the *active multiset* at point t, defined by $A_t = \{\ (k,v) \mid k \in \mathtt{ST}_t \ \wedge\ v \in \mathtt{H}_t \wedge (k,v)$ satisfy the crosscheck relation $\}$. The expressions $|\mathtt{H}_t|$ and $|\mathtt{ST}_t|$ the number of items in the (respective) active structures. A state of the composite data structure is *ST-legitimate* if the search tree component of the state is legitimate as defined in [9]. A state of the composite

data structure is *H-legitimate* if the heap component of the state is legitimate as defined in [8]. A state is *ST/H-legitimate* if it is both ST-legitimate and H-legitimate. A state is *legitimate* if (*i*) it is ST/H-legitimate, (*ii*) for each item x of the active search tree, there exists an item y of the active heap such that x and y are related by the crosscheck relation, and (conversely) (*iii*) for each item a of the active heap, there exists an item b of the active search tree such that a and b are crosscheck related. The availability and closure properties, based on these definitions, have simpler proofs of correctness than the proof of convergence. To show convergence, we define a segment of a history to be a *color phase* if the `curcolor` value is the same at each point in the segment. Define *colorsafe* to be the weakest predicate closed under heap-search operations such that for any point s where *colorsafe* holds, the state of the data structure is ST/H-legitimate and $(\forall x : x \in A_s : x.\texttt{color} \neq \texttt{nextcolor})$.

Lemma 4. Let s be an ST/H-legitimate point in a history and let $m_s = \max(|\texttt{H}_s|, |\texttt{ST}_s|)$. The color phase containing point s terminates at some point w after at most $15m_s$ operations and $\max(|\texttt{H}_w|, |\texttt{ST}_w|) \leq 16m_s$. Following any point s at which ST/H-legitimacy holds and $m_s = \max(|\texttt{H}_s|, |\texttt{ST}_s|)$, there occurs a point t within $O(m)$ operations of any history such that the data structure at point t is *colorsafe* and $m_t = \max(|\texttt{H}_s|, |\texttt{ST}_s|) = O(m)$.

For any point s in a history of operations on the composite data structure, let $depth_s(j)$ for $j \in \texttt{H}_s$ be the number of `pop` operations required to obtain a pointer to j from the lossy stack ($depth_s(j) = \infty$ if there is no pointer to item j on the lossy stack). Let $\overline{\texttt{H}}_s$ denote the bag of active heap items that for which the crosscheck relation does not hold, that is, $\overline{\texttt{H}}_s$ contains those active heap items that do not correspond to any active multiset item. Define *stacksafe*(m) to be the weakest closed predicate such that any point s where *stacksafe*(m) holds, the state of the data structure is *colorsafe* and $(\forall j : j \in \overline{\texttt{H}}_s : depth_s(j) < 3m)$.

Lemma 5. Following any *colorsafe* point s with $m_s = \max(|\texttt{H}_s|, |\texttt{ST}_s|)$, there occurs a point t within $O(m_s)$ operations of any history such that the data structure at point t is *stacksafe*(m_s) and $m_t = \max(|\texttt{H}_s|, |\texttt{ST}_s|) = O(m_s)$.

Theorem 1. Within $O(m_s)$ operations of any history the state of the composite data structure is legitimate, where $m_s = \max(|\texttt{H}_s|, |\texttt{ST}_s|)$ for the initial point s of the history.

6 Conclusion

There are likely simpler ways to construct a stabilizing available structure with the signatures of the composite presented in Section 4 (two search trees, an augmented tree, etc), however our aim was to investigate the composition of a data structure from given components. Although we present a construction, our results concerning the larger question are somewhat negative: not everything is achievable given constraints of (conservative) availability and stabilization (the

possibility of conflict between availability and fault tolerance was observed long ago [1]).

The reader may wonder whether the case of sequential data structure operations, without distributed implementation or concurrency, merits any investigation: after all, the very title of [2] is "self-stabilization in spite of distributed control" (recent definitions of self-stabilization refer only to behavior [14] and care not whether the system is distributed or sequential). Here, the difficulties to overcome have to do with two aspects of availability during convergence, namely the integrity of operation responses and the time bounds of operations. Both aspects have practical motivation: users of data structures and system designers of forward error recovery value the integrity of operation responses; and having guaranteed time bounds on operations is useful for the design of responsive applications in synchronous environments. It could be interesting to reconsider our constraint on the time bound, perhaps allowing some polynomial extra time during convergence.

References

1. SB Davidson, H Garcia-Molina, D Skeen. Consistency in partitioned networks. *ACM Computing Surveys* 17(3):341-370, 1985.
2. EW Dijkstra. EWD391 Self-stabilization in spite of distributed control. In *Selected Writings on Computing: A Personal Perspective,* pages 41–46, Springer-Verlag, 1982. EWD391's original date is 1973.
3. S Dolev. *Self-Stabilization.* MIT Press, 2000.
4. S Dolev and T Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 3(4), 1997.
5. A Fekete, D Gupta, V Luchangco, N Lynch, A Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220:113-156, 1999.
6. S Ghosh, A Gupta, T Herman, and SV Pemmaraju. Fault-containing self-stabilizing algorithms. In *PODC'96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 45–54, 1996.
7. T Herman. Superstabilizing mutual exclusion. *Distributed Computing*, 13(1):1–17, 2000.
8. T Herman, T Masuzawa. Available stabilizing heaps. *Information Processing Letters* 77:115-121, 2001.
9. T Herman, T Masuzawa. A stabilizing search tree with availability properties. *Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems (ISADS'01)*, pp. 398-405, March 2001.
10. JH Hoepman, M Papatriantafilou, P Tsigas. Self-stabilization of wait-free shared memory objects. In *Distributed Algorithms, 9th International Workshop Proceedings (WDAG'95)*, Springer-Verlag LNCS:972, pp. 273-287, 1995.
11. B Lampson. How to build a aighly available system using consensus. In *10th International Workshop on Distributed Algorithms (WDAG'96)*, Springer-Verlag LNCS 1151, pp. 1-17, 1996.
12. M Li, PMB Vitanyi. Optimality of wait-free atomic multiwriter variables. *Information Processing Letters*, 43:107-112, 1992.
13. M Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5):9-22, 1990.
14. G Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

Stabilizing Causal Deterministic Merge*

Sandeep S. Kulkarni and Ravikant

Software Engineering and Networks Laboratory
Department of Computer Science and Engineering
Michigan State University, East Lansing MI 48824 USA
{sandeep,ravikant}@cse.msu.edu

Abstract. We present a causal deterministic merge program for publish-subscribe systems. Our program ensures that if two subscribers receive two messages then they receive them in the same order. Also, it guarantees that the order in which a subscriber receives messages is a linearization of the causal order among those messages. To develop our program, we expect two guarantees from the underlying system: the first guarantee deals with the difference between physical clocks and the second guarantee deals with message delays. While $O(n^2)$ space is required for causal delivery of unicast messages in asynchronous systems, our program only uses $O(log\ n)$ space. We also show how our program can be made stabilizing while using only a bounded space. And, the recovery time for our program is proportional to the guarantees made by the underlying system.

1 Introduction

We focus our attention on the problem of causal deterministic merge in a publish-subscribe network. A publish-subscribe network consists of a set of *publishers* that publish messages and a set of *subscribers* that subscribe to a subset of those messages. We assume that the subscribers wish to receive messages in a uniform total order that conforms to the causal order. The uniform total order requires that if two subscribers receive two messages, then they receive them in the same order. The causal order requires that if a subscriber receives messages m_1 and m_2 such that m_2 causally depends on m_1 then that subscriber delivers m_1 before delivering m_2. The causal dependency may occur through published messages or through internal communication among publishers. Such causal order delivery is desirable in several applications. For example, consider a scenario where a publisher *publishes* a question and another publisher *publishes* its answer. It is desirable that the subscriber receives the question before receiving the answer.

A uniform total order that conforms to the causal order can be obtained by using a centralized solution. However, a centralized solution lacks scalability

* This work is partially sponsored by NSF CAREER 0092724, DARPA contract F33615-01-C-1901, ONR grant N00014-01-1-0744, and a grant from Michigan State University.

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 183–199, 2001.

and reliability. To redress this, we use replicated mergers (as proposed in [1]). Each merger is associated with a set of subscribers. All published messages are sent to the (relevant) mergers. The mergers reorder the messages so as to obtain a causal deterministic order. The deterministic order guarantees that if two mergers receive overlapping messages, they order those overlapping messages consistently. Moreover, the causal order guarantees that if m_2 causally depends on m_1 then m_1 will be ordered before m_2. The reordered messages are then sent to the subscribers.

We build these mergers by extracting two simple guarantees expected of the underlying distributed system on which the publish-subscribe network is built. The first guarantee is related to the clocks: we require that the system provide a bound, say ϵ, such that the clock drift between different processes (that implement the publishers, subscribers and mergers) in the system is bounded by ϵ. This guarantee can be met by using GPS clocks, network time protocol, atomic clocks or clock synchronization programs. The second guarantee relates to the message delays: we require that messages that reach their destination do so within some bound, say δ. This guarantee can be met by using protocols that characterize messages as being timely or late.

Our solution uses bounded space and provides stabilizing fault-tolerance in the presence of faults. It uses $O(\epsilon log\ n\ +\ log\ \delta)$ space and recovers in $O(\epsilon+\delta)$ time from an arbitrary state (that could be reached in the presence of faults such as message corruption, transients, temporary violation of system guarantees). However, for a given system, the space used by our program is bounded in that it does not grow as the computation progresses.

We decompose the problem of causal deterministic merge into two parts: (1) how to design logical timestamps that capture causality among messages, and (2) how to use the logical timestamps to obtain a uniform total order that conforms to the causal order. Specifically, in the first part, we develop bounded-space and stabilizing logical timestamps. In the second part, we use the logical timestamps to order messages in a uniform total order that conforms to the causal order.

To develop our solution for logical timestamps, we begin with the following observation: *Had the underlying system provided a global clock ($\epsilon=0$), the physical clock alone would have been sufficient to implement the logical timestamps.* We, therefore, develop logical timestamps that consist of the physical clock value and some additional information that is proportional to ϵ and δ. Also, the guarantees given by the distributed system are used to add stabilizing tolerance; the resulting implementation, thus, ensures that even if the logical timestamp values are perturbed, eventually they are restored so that the causality is tracked correctly.

Contributions of the Paper. The contributions of this paper are as follows: (1) We present a bounded and stabilizing solution to logical timestamps. The space cost for our program is $O(\epsilon log\ n\ +\ log\ \delta)$ and the recovery time for our program is $O(\epsilon+\delta)$. (2) Using the logical timestamps, we present a bounded and stabilizing implementation of causal deterministic merge. If we concentrate only on one merger, the causal deterministic merge program also serves as a causal delivery program. The state space for our program grows logarithmically

in the number of processes. By contrast, the space cost for previous programs (e.g., [2, 3]) is quadratic in nature. Our solution guarantees that if a message arrives at its destination within δ time, it will be delivered within $\delta+3\epsilon$. Thus, the resulting system is one where the clocks differ by ϵ and the messages arrive within time $\delta+3\epsilon$ or are lost. Note that these assumptions are similar to the assumptions of the underlying system (with δ replaced by $\delta+3\epsilon$). We find that this observation simplifies the task of designing causal deterministic merge in hierarchical systems. (3) We present extensions of our program where we identify simple conditions to reduce the space complexity further. With these conditions, the space requirement is independent of the number of processes.

Organization of the Paper. In Section 2, we formally specify our system model, guarantees expected from the underlying system, and the types of faults. Then, in Section 3, we present our solution to logical timestamps. We use the solution in Section 3 to develop our causal deterministic merge program in Section 4. In Section 5, we discuss extensions of our program, and identify a simple condition under which the message size could be made independent of the number of processes. In Section 6, we present the role of our model in each of these solutions. Finally, we make concluding remarks and point out future directions in Section 7. For reasons of space, we refer the reader to [4] for proofs.

2 System Model

A distributed system consists of a finite set of processes which communicate via message passing. A process is used to implement a publisher, subscriber or a merger. Each process j has access to a physical clock $rt.j$. We do not assume any relation between $rt.j$ and the auxiliary global time, i.e., we do not assume that any process is aware of the 'real' time.

We assume that in the absence of faults, the distributed system satisfies the following two guarantees, and in the presence of faults, these guarantees may be violated only temporarily. (We treat the above guarantees as assumptions that system users can make. Hence, depending upon the context, we use the word assumption instead of guarantee.)

Guarantees of the Distributed System.

*G*1**.** The value of $rt.j$ is non-decreasing, and at any time, the difference between the clock values of any two processes is bounded by ϵ. In other words,

$$\forall j, k :: |rt.j - rt.k| \leq \epsilon$$

*G*2**.** Let m_j be a message sent by process j to k. Also, let st_m denote the clock value of j when j sent m_j, and let rd_m denote the clock value of j when k received m_j. We require that k should receive m_j within time δ unless m_j is lost. In other words,

$$\forall m :: ((rd_m \leq (st_m+\delta)) \quad \vee \quad rd_m = \infty)$$

Notation. A distributed system instantiated with parameters ϵ and δ is denoted as $ds(\epsilon, \delta)$.

Remark. We assume that time is discrete and the value of ϵ is an integer; ϵ equals *ceiling*(max clock drift/minimum time difference between events). The issue of fine-tuning ϵ, δ is discussed in Section 6.

Execution of a process consists of a sequence of events; an event can be a local event, a send event, or a receive event. In a local event, the process neither receives a message nor sends a message. In a send event, the process sends one or more messages, and in a receive event, the process receives one or more messages. For simplicity, we assume that, for one clock tick of j, at most one event is created at process j. (Note that this assumption can be easily weakened so that at most K events are generated for each clock tick, where K is any constant.)

Notation. In this paper, we use i, j, k, and l to denote processes. We use e and f to denote events. Where needed, events are subscripted with the process at which they occur, thus, e_j is an event at j. We use m to denote messages. Messages are subscripted by the sender process, thus, m_j is a message sent by j.

Fault Model. In our fault-model, messages can be corrupted, lost, or duplicated. Moreover, processes could be improperly initialized, and channels may contain garbage messages in the initial state. The state of processes could be transiently (and arbitrarily) corrupted at any time. Also, the guarantees made by the system ($G1$ and $G2$) may be temporarily violated. We assume that the number of fault occurrences is finite. Our solutions are stabilizing fault-tolerant in the presence of these faults, where

Definition. A program is *stabilizing fault-tolerant* iff starting from an arbitrary state, it eventually recovers to a state from where its specification is satisfied.

3 Logical Timestamps

In this section, we present our bounded-state, stabilizing solution for logical timestamps. Towards this end, we first precisely define the problem of logical timestamps in Section 3.1. Then, we provide an implementation of the logical timestamps in Section 3.2. Finally, we show how that implementation can be made stabilizing in Section 3.3.

3.1 Problem Statement

Towards defining the problem of logical timestamps, we first define the causal relation [5], $\longrightarrow$, among events. Then we introduce a definition which will be used in the problem statement.

Happened-Before. The happened before relation [5] is the smallest transitive relation that satisfies, for any two events e, f, $e \longrightarrow f$ if (1) e and f are events

on the same process and e occurred before f, or (2) e is a send event in one process and f is the corresponding receive event in another process. □

Notation. Let ts be a type, and let $ts.e$ and $ts.f$ be values of type ts. Then, $less(ts, ts)$ is a function that takes two arguments of type ts and returns a boolean.

Definition. Let ts be a type. A function $less(ts, ts)$ is well-formed iff it is irreflexive and asymmetric.

The problem of logical timestamps can now be defined as follows.

Specification of Logical Timestamps. Identify a type ts, a well-formed relation $less(ts, ts)$, and assign a timestamp of type ts to each event in the given program computation such that for any two events e and f with timestamps $ts.e$ and $ts.f$ the following condition is true:

- $e \longrightarrow f \quad \Rightarrow \quad less(ts.e, ts.f)$ □

Remark. In Lamport's scalar clock implementation, ts is instantiated to be *integer*, and $less(ts.e, ts.f)$ iff $ts.e < ts.f$. The vector clock implementation by Fidge [6] and Mattern [7] can also be viewed as solving the logical timestamp problem provided we instantiate ts to be an array of integers, and define $less(ts.e, ts.f)$ to be true iff $((\forall k :: ts.e[k] \leq ts.f[k]) \wedge (\exists k :: ts.e[k] < ts.f[k]))$. Note that $less$ is not a total relation in that for events e and f, it is possible that both $less(ts.e, ts.f)$ and $less(ts.f, ts.e)$ are false. However, both of them cannot be true.

3.2 Solution to Logical Timestamps

We propose that the timestamp of event e_j be of the form $\langle r.e_j, c.e_j, kn.e_j \rangle$ where $r.e_j$ denotes the physical clock value of j when e_j was created. The variable $c.e_j$ is used to capture the knowledge that j had about the maximum clock value in the system when e_j was created. Specifically, $c.e_j$ equals the difference between the maximum clock value in the system that j is aware of when e_j was created and $r.e_j$. The variable $kn.e_j$ is an array of size 2ϵ. The variable $kn.e_j[t]$, $-\epsilon \leq t < \epsilon$, is used to capture the knowledge about the number of events f such that $r.f = r.e_j + t$ and $f \longrightarrow e_j$. (We maintain $kn.j[t]$ only for $t < \epsilon$ because j cannot learn of events whose timestamp is at least $rt.j + \epsilon$. To see this, observe that when $rt.j = x$, the maximum clock value in the system is $x + \epsilon$. However, any message sent with timestamp $x + \epsilon$ can be received at j only when $rt.j$ is incremented.)

Our logical timestamp program is as follows: Each process j maintains $rt.j$, $r.j$, $c.j$ and the array $kn.j$. The variable $rt.j$ is the physical time at j, and $\langle r.j, c.j, kn.j \rangle$ is the timestamp of the last event on j. We assume that the first event is created on each process when its physical clock value is 0. Hence, we initialize $rt.j$, $r.j$, $c.j$ to be 0. Also, we initialize $kn.j[0]$ to be equal to 1 and all other elements in $kn.j$ to be 0. The variable $rt.j$ is updated by the underlying system that ensures that $G1$ is satisfied. The logical timestamp protocol can only read $rt.j$.

We ensure that $r.j+c.j$ equals the knowledge that j has about the maximum clock value in the system. Consider the case where j creates a local event e_j at time $rt.j$. Note that $r.j+c.j$ equals the maximum clock value that j was aware of when it created the last event. If $r.j+c.j \geq rt.j$ then it implies that $r.j+c.j$ is still the maximum clock value that j is aware of. In that case, we set $c.j$ to be equal to $r.j+c.j-rt.j$. If $r.j+c.j \leq rt.j$ then it implies that the maximum clock value that j is aware of is the same as $rt.j$. Hence, we set $c.j$ to 0. We update kn based on the previous value of $kn.j$. Then, we increase $kn.j[0]$ to capture the fact that j is aware of one extra event at time $rt.j+0$. In the send event, $r.e_j$, $c.e_j$ and $kn.e_j$ are updated in the same way and the message carries the timestamp $\langle r.e_j, c.e_j, kn.e_j \rangle$. When j receives message(s), it updates r, c and kn in the same way except that it does the update based on the previous event at j and the event(s) corresponding to the sending of (those) message(s). Thus, the program is as shown in Figure 1. (While presenting the program, for simplicity of presentation, we assume that $kn.j[t]$ equals 0 if $t < -\epsilon$ or $t \geq \epsilon$.)

Initially:
$rt.j, r.j, c.j = 0$, $\forall t : t \not= 0$ $kn.j[t] = 0, kn.j[0] = 1$

Local event e_j/Send event e_j (message being sent is m_j)
$c.j := max(0, r.j + c.j - rt.j)$
$\forall t : -\epsilon \leq t < \epsilon : kn.j[t] := kn.j[t + rt.j - r.j]$
$kn.j[0] := kn.j[0] + 1$
$r.j := rt.j$
$r.e_j, c.e_j, kn.e_j := r.j, c.j, kn.j$
if e_j is a send event then $r.m_j, c.m_j, kn.m_j := r.j, c.j, kn.j$

Receive event e_j (message m received with timestamp $\langle r.m, c.m, kn.m \rangle$)
$c.j := max(0, r.j + c.j - rt.j, r.m + c.m - rt.j)$
$\forall t : -\epsilon \leq t < \epsilon : kn.j[t] := max(0, kn.j[t + rt.j - r.j], kn.m[t + rt.j - r.m])$
$kn.j[0] := kn.j[0] + 1$
$r.j := rt.j$
$r.e_j, c.e_j, kn.e_j := r.j, c.j, kn.j$

Fig. 1. Logical Timestamp Program

For the program in Figure 1, the following lemmas are true in the absence of faults.

Lemma 3.1.

$\forall e :: kn.e[c.e] > 0$
$\forall m :: kn.m[c.m] > 0$
$\forall e, t : c.e < t < \epsilon : kn.e[t] = 0$
$\forall m, t : c.m < t < \epsilon : kn.m[t] = 0$ □

Lemma 3.2. $\forall e :: c.e < \epsilon$. □

Lemma 3.3. $\forall e,t : -\epsilon \leq t < \epsilon : kn.e[t] \leq |\{e\} \cup \{f : f \longrightarrow e \wedge r.f = r.e+t\}|$. □

Lemma 3.4. Given a system with n processes, $\forall e,t : : -\epsilon \leq t < \epsilon : kn.e[t] \leq n$ □

Remark. We have assumed that the minimum delay in message transmission is 0 and, hence, messages could be received instantaneously. However, if the underlying system ensures that there is a minimum delay δ_{min} in transmission of messages, we need to maintain $kn.j[t]$ only for $-(\epsilon - \delta_{min}) \leq t < (\epsilon - \delta_{min})$. In this case, the number of elements in the array $kn.j$ are reduced to $2(\epsilon - \delta_{min})$. Finally, when $\delta_{min} \geq \epsilon$, there is no need to maintain the array. (We would like to point out that Lamport had made the observation for the case where $\delta_{min} \geq \epsilon$ in [5].)

Comparing Timestamps. Given two events e_j and f_k with timestamps $\langle r.e_j,\ c.e_j,\ kn.e_j\rangle$ and $\langle r.f_k, c.f_k, kn.f_k\rangle$, we first use the r and c values to determine the truth value of $less(\langle r.e_j, c.e_j, kn.e_j\rangle, \langle r.f_k, c.f_k, kn.f_k\rangle)$. As mentioned above, $r.e_j + c.e_j$ captures the maximum clock value that j was aware of when e_j was created. Thus, if $rt.e_j + c.e_j < rt.f_k + c.f_k$, we can safely decide $less(\langle r.e_j, c.e_j, kn.e_j\rangle, \langle r.f_k, c.f_k, kn.f_k\rangle)$ to be true. Finally, if $rt.e_j + c.e_j = rt.f_k + c.f_k$ then we use the array kn to determine the truth value of $less$ $(\langle r.e_j, c.e_j, kn.e_j\rangle, \langle r.f_k, c.f_k, kn.f_k\rangle)$. Towards this end, we use a lexicographical comparison. First, we compare $kn.e[c.e]$ with $kn.f[c.f]$. If these two values are unequal then they determine the truth value of $less(\langle r.e_j, c.e_j, kn.e_j\rangle$, $\langle r.f_k, c.f_k, kn.f_k\rangle)$. If $kn.e[c.e]$ equals $kn.f[c.f]$ then we compare $kn.e[(c.e)-1]$ and $kn.f[(c.f)-1]$, and so on. Moreover, as shown in Theorem 3.6, comparing only ϵ elements in kn is sufficient. Thus, we define relation $less$ as follows:

$$less(\langle r.e, c.e, kn.e\rangle, \langle r.f, c.f, kn.f\rangle)$$

iff

$$(r.e + c.e < r.f + c.f) \quad \vee$$
$$((r.e + c.e = r.f + c.f) \wedge lexgraph(kn.e, c.e, kn.f, c.f, \epsilon))$$

where

$$\begin{aligned} lexgraph(kn1, c1, kn2, c2, n) = \text{if} \quad & (n = 0) && \text{then } false \\ \text{elseif} \quad & kn1[c1] < kn2[c2] && \text{then } true \\ \text{elseif} \quad & kn1[c1] > kn2[c2] && \text{then } false \\ \text{else} \quad & lexgraph(kn1, c1-1, kn2, c2-1, n-1) \end{aligned}$$

If for two events e_j and f_k, $\neg less(\langle r.e_j, c.e_j, kn.e_j\rangle, \langle r.f_k, c.f_k, kn.f_k\rangle)$ and $\neg less(\langle r.f_k, c.f_k, kn.f_k\rangle, \langle r.e_j, c.e_j, kn.e_j\rangle)$, then the events are ordered based on their process ID.

Note that kn values are compared only when $r.e + c.e$ equals $r.f + c.f$. Thus, $kn.f[c.f]$ is compared with $kn.e[r.f + c.f - r.e]$ $(= kn.e[c.e])$. More generally, $kn.f[t]$ is compared with $kn.e[r.f + t - r.e]$. This is due to the fact that $kn.f[t]$ denotes the knowledge about events at $r.f + t$. Likewise, $kn.e[r.f + t - r.e]$ denotes the knowledge about events at $r.e + r.f + t - r.e (= r.f + t)$.

For the above relation, we first make the following observation and then prove that the program in Figure 1 satisfies the specification of logical timestamps.

Observation 3.5. The *less* relation given above is transitive and well-formed. □

Theorem 3.6. $\forall e, f :: e \longrightarrow f \Rightarrow less(\langle r.e, c.e, kn.e\rangle, \langle r.f, c.f, kn.f\rangle)$

Proof. We prove this by induction. Initially, for any two events, e, f, $e \longrightarrow f$ is false. Hence, the theorem is trivially true. Now, consider the case where a new event, say f, is created at some process, say j.

1. f is a send/local event:
   Let e be the event that occurred at j just before f. From the assignment to $c.f$, we have: $r.e+c.e \leq r.f+c.f$. If $r.e+c.e < r.f+c.f$ then it follows that $less(\langle r.e, c.e, kn.e\rangle, \langle r.f, c.f, kn.f\rangle)$ is true. Hence, we consider the case where (a) $r.e+c.e = r.f+c.f$. In this case, based on how the program updates $kn.f$, we have (b) $\forall t : -\epsilon \leq t < \epsilon \wedge \ -\epsilon \leq t+r.f-r.e < \epsilon : \ kn.e[t+r.f-r.e] \leq kn.f[t]$. Also, since $kn.f[0]$ is increased by the program, we have (c) $kn.e[r.f - r.e] < kn.f[0]$.
   In evaluating $less(\langle r.e, c.e, kn.e\rangle, \langle r.f, c.f, kn.f\rangle)$, we first compare $kn.f[c.f]$ with $kn.e[c.e]$. Then, we compare $kn.f[c.f-1]$ with $kn.e[c.e-1]$, and so on. When we have compared $c.f+1$ elements, the truth value of *less* will be determined since $kn.f[c.f-c.f]$ is greater than $kn.e[c.e-c.f]$ $(= kn.e[r.f-r.e])$. If the truth value of $less(\langle r.e, c.e, kn.e\rangle, \langle r.f, c.f, kn.f\rangle)$ is determined before we compare $kn.f[c.f-c.f]$ with $kn.e[c.e-c.f]$, from (b), $less(\langle r.e, c.e, kn.e\rangle, \langle r.f, c.f, kn.f\rangle)$ must be true. Moreover, since $c.f$ is less than ϵ at most ϵ elements in kn are compared. Finally, from (a), (b) and (c), it follows that $less(\langle r.e, c.e, kn.e\rangle, \langle r.f, c.f, kn.f\rangle)$ is true. Now for any event a, $a \neq e$, $a \longrightarrow e$ iff $a \longrightarrow f$. Moreover, if $a \longrightarrow e$ then $less(\langle r.a, c.a, kn.a\rangle, \langle r.e, c.e, kn.e\rangle)$ is true by induction. Hence by transitivity, $less(\langle r.a, c.a, kn.a\rangle, \langle r.f, c.f, kn.f\rangle)$ is also true.
2. f is a receive event.
   This proof is similar to case 1 except that we need to consider two events e_1, the event on j just before f, and e_2, the corresponding send event. As in the previous, we show that $less(\langle r.e_1, c.e_1, kn.e_1\rangle, \langle r.f, c.f, kn.f\rangle)$ and $less(\langle r.e_2, c.e_2, kn.e_2\rangle, \langle r.f, c.f, kn.f\rangle)$ are true.
   Once again, for any event a, $(a \neq e_1 \wedge \ a \neq e_2), a \longrightarrow f$ iff $(a \longrightarrow e_1 \vee a \longrightarrow e_2)$. Hence, by transitivity, $less(\langle r.a, c.a, kn.a\rangle, \langle r.f, c.f, kn.f\rangle)$ □

Bounding the State Space of Logical Timestamps. From Lemmas 3.2 and 3.4, it follows that the c and kn values are bounded. Also, rt values can be bounded by letting j maintain only $brt.j = (rt.j \ mod \ B)$ where B is a large enough constant. More specifically, B should be large enough so that when j receives a message m that carries $brt.m$ (instead of $rt.m$), j should be able to update its logical timestamp appropriately. When j receives message m_l from l, from $G1$, $rt.l$ is in the range $[rt.j-\epsilon..rt.j+\epsilon]$. From $G2$, when m was sent, $rt.l$ was in the range $[rt.j-\epsilon-\delta..rt.j+\epsilon]$. Hence, the vector $kn.m_l$ carries information about events that occurred at time $[rt.j-\epsilon-\delta-\epsilon..rt.j+\epsilon+\epsilon-1]$. It follows that if B is greater than or equal to $4\epsilon+\delta+1$, j would be able to update the logical timestamp appropriately. Hence, to bound the space used by j, we maintain $brt.j = rt.j \ mod \ B$ where $B \geq 4\epsilon+\delta+1$.

3.3 Bounding and Stabilizing Logical Timestamps

If the logical timestamps are perturbed by faults such as failure and repair of process, corrupted timestamps, temporary violation of system guarantees, we ensure that the program reaches a state from where causality is tracked correctly. The stabilization proceeds in four steps. First, if the system guarantees are violated then they are restored. Several approaches may suffice for this purpose, including clock synchronization programs, GPS clocks, network time protocol, etc. The particular approach used is not relevant for our purpose; any approach may be used.

In the second step, we satisfy the local properties in Lemmas 3.1, 3.2 and 3.4. Towards this end, for any process j, if $c.j$ is ever assigned a value that is greater than or equal to ϵ, $kn.j[t]$ is assigned a value that is greater than n, $kn.j[c.j]$ is zero or for some t, $t > c.j$, $kn.j[t]$ is nonzero, we set $c.j$ to 0, $kn.j[0]$ to 1 and all other elements of $kn.j$ to 0. Likewise, whenever a message is received we ensure that Lemmas 3.1, 3.2 and 3.4 are satisfied for that message before processing that message.

In the third step, messages with corrupt timestamps are either delivered or are lost. Consider a message m whose timestamp is corrupted after it was sent. After δ time has passed, m is delivered or lost (from $G2$). Since reception of m does not affect rt values, $G1$ continues to be true. Also, if r, c, kn values are changed in a way that local invariants are violated, step 2 restes them. Thus, the correction achieved by steps 1 and 2 is not affected and the number of messages whose timestamp is corrupted while in transit, is reduced by 1. It follows that eventually for any message m_k in transit, the timestamp of m_k is the same as the timestamp of k when m_k was sent.

Consider a state, say s, obtained after steps 1-3. Let $rt.j = x$ in state s. Thus, from $G1$, we find that for any process k, $rt.k$ is in the range $[x-\epsilon..x+\epsilon]$. Moreover, from $G2$, for any message, say m_k, in transit, m_k was sent when $rt.k$ was in the range $[x-\epsilon-\delta..x+\epsilon]$. Hence, m_k can carry information of about events whose physical time is in the range $[x-\epsilon-\delta-\epsilon..x+\epsilon+\epsilon-1]$. It follows that no process (or message) has knowledge about events that occur at physical time t where $t \geq x+2\epsilon$. In other words, if j acquires knowledge about events that occur at time t, $t \geq 2\epsilon$, it must be due to actual events rather than due to faults such as transients. However, the knowledge about events in the range $[x..x+2\epsilon-1]$ may be incorrect. Now, if $rt.j$ is unbounded and $rt.j$ is advanced to $x+3\epsilon$, the $kn.j$ vector will only maintain information about events whose physical time is in the range $[x+2\epsilon..x+4\epsilon-1]$ and, as discussed above, $kn.j$ will be consistent. After all kn values become consistent, causality will be tracked correctly.

Also, note that the time required for steps 2 and 3 is δ. And, the time required for step 4 is 3ϵ. Thus, the time required for recovery is $O(\epsilon+\delta)$.

Once again, we maintain $brt.j = (rt.j \ mod \ B)$ where B is a large enough constant. For stabilization, we increase the value of B so that j can distinguish between the different rt values that may occur during the computation where $rt.j$ is advanced from x to $x+3\epsilon$. From the above discussion, the rt values en-

countered in that computation are in the range $[x-2\epsilon-\delta..x+4\epsilon-1]$. Hence, we maintain $brt.j = rt.j \ mod \ B$ where $B \geq 6\epsilon+\delta+1$. Thus, we have:

Theorem 3.7. If the timestamping program in Figure 1 is modified so that the conditions specified in Lemmas 3.1, 3.2 and 3.4 are satisfied (as given in the second step above) and a variable $brt.j = rt.j \ mod \ B$ is maintained to capture $rt.j$ then the resulting program uses bounded space and is stabilizing fault-tolerant provided $B \geq 6\epsilon+\delta+1$. □

Now, we consider the state space required to implement our logical timestamps. As discussed above, the maximum value for c and brt is $O(\epsilon+\delta)$. Thus, the space required for those variables is $O(log(\epsilon+\delta))$. Each element in kn is bounded by n and, hence, requires $O(log\ n)$ space. It follows that the maximum space required for kn is $O(\epsilon\ log\ n)$. Summing up the space requirement for brt, c and kn, we have

Theorem 3.8 The space used by the stabilizing logical timestamp program is $O(\epsilon\ log\ n \ + \ log\ \delta)$. □

4 Causal Deterministic Merge

In this section, we use the timestamp introduced in Section 3 to present our program for causal deterministic merge. We first define the problem in Section 4.1. Then, we give the bounded-space stabilizing solution in Section 4.2.

4.1 Problem Statement

The problem of causal deterministic merge requires that causal delivery is satisfied and that the messages are merged deterministically. Thus, whenever a message is received, it should be buffered until the receiving process determines the place where the message should be put while obtaining the causal deterministic merge. Whenever the appropriate place in the causal deterministic merge is determined, we *deliver* that message. Thus, the problem requires that the following two specifications are satisfied.

Note that the above problem statement requires causal deterministic merge of messages that reach the destination within δ time. In other words, messages lost in transit are ignored. Such delivery pattern is useful for applications, e.g., audio/video streams, where such message loss can be easily tolerated. Moreover, many such applications require that most of the messages be delivered in a timely fashion even though some messages are never delivered. In other words, they require that even if a message is lost, the messages that causally depend on it should not suffer excessive delay. Based on the application requirements and its ability to tolerate message loss, it is possible (cf. Section 6) to fine-tune the value of δ: increasing the value of δ decreases the number of messages lost but increases the maximum delay that messages may incur and the amount of buffer required to obtain causal deterministic merge.

Specification of causal delivery.
For messages m_1 and m_2 and for process j that satisfy
$send(m_1) \longrightarrow send(m_2)$,
Process j is included in the destination set of m_1 and m_2, and
m_1 and m_2 are received at j
the following two conditions are satisfied:
m_1 is delivered before m_2, and
m_1 and m_2 are eventually delivered

Specification of deterministic merge.
For messages m_1 and m_2 and processes j and k that satisfy
Processes j and k are included in the destination set of m_1 and m_2, and
Both m_1 and m_2 are received at j and k
the following condition is satisfied:
The order in which m_1 and m_2 are delivered at j and k is the same, and
m_1 and m_2 are eventually delivered

4.2 Solution

We first present the condition for causal ordering of two messages, m_1 and m_2, received at process j such that $m_1 \longrightarrow m_2$. Let us assume that m_2 is received first at j. We now find how long m_2 should wait at j before being delivered so that m_1 is received and delivered before that. From the timestamp comparison, we have $m_1 \longrightarrow m_2 \Rightarrow r.m_1 + c.m_1 \leq r.m_2 + c.m_2$. Moreover, since $c.j \geq 0$, it follows that $r.m_1 \leq r.m_2 + c.m_2$. In other words, when m_1 was sent the physical clock value of the sender process was at most $r.m_2 + c.m_2$. From $G2$, when m_1 is received at j, the clock value of the sender (of m_1) will be at most $r.m_2 + c.m_2 + \delta$. Moreover, from $G1$, the clock value of j will be at most $r.m_2 + c.m_2 + \delta + \epsilon$. So, m_1 will reach j before $rt.j$ equals $r.m_2 + c.m_2 + \delta + \epsilon$ or m_1 will be lost. Hence, to obtain causal delivery, it suffices that m_2 wait until $rt.j$ equals $r.m_2 + c.m_2 + \delta + \epsilon$. Thus, the delivery condition for message m is $delcond(m, j)$ where

$$delcond(m, j) = (rt.j = r.m + c.m + \delta + \epsilon)$$

Causal Delivery Program. In our causal delivery program, whenever message m is received at process j, m is buffered until $delcond(m, j)$ is satisfied. As soon as $delcond(m, j)$ is satisfied, message m is delivered. If multiple messages satisfy the delivery condition simultaneously, j determines the causal relation (if any) between them using $less$ relation: given two messages m_k and m_l if $less(\langle r.m_k, c.m_k, kn.m_k \rangle, \langle r.m_l, c.m_l, kn.m_l \rangle)$ is true we deliver m_k before m_l. If both $less(\langle r.m_k, c.m_k, kn.m_k \rangle, \langle r.m_l, c.m_l, kn.m_l \rangle)$ and $less(\langle r.m_l, c.m_l, kn.m_l \rangle, \langle r.m_k, c.m_k, kn.m_k \rangle)$ are false then we deliver them based on their process ID. Observe that there is only one way to deliver these messages.

From Lemma 3.2, the value of $c.m$ is at most ϵ. Thus, message m will be delivered before $rt.j$ reaches $r.m + \delta + 2\epsilon$. From $G1$, the clock value of the sender

when m is delivered is at most $r.m+\delta+3\epsilon$. Also, if j receives m when $rt.j$ is x then $r.m$ is at least $x-\epsilon$. Thus, m can be buffered only for time $\epsilon+\delta+\epsilon+c.m$. It follows that no message is buffered for longer than $\delta+3\epsilon$. Thus, the following theorems are true.

Lemma 4.1. If j sends message m when its physical clock value was $r.m$ then it would be delivered before the physical clock value of j reaches $r.m+\delta+3\epsilon$. □

Lemma 4.2. A process buffers a message for at most $\delta+3\epsilon$ time. □

Theorem 4.3. Given a system $ds(\epsilon, \delta)$ if our causal delivery program is used to deliver messages then the resulting system will be $ds(\epsilon, \delta+3\epsilon)$ (cf. Figure 2). □

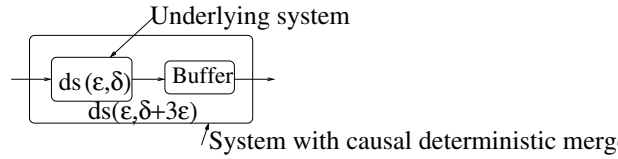


Fig. 2. Guarantees of Our Causal Delivery Program

Now, we show that the causal delivery program given above suffices for ensuring causal deterministic merge in a publish-subscribe system. Formally,

Theorem 4.4. If two messages m_1 and m_2 arrive at processes j and k then the order in which they are delivered would be same on both processes. □

Theorem 4.5. If two messages m_1 and m_2 such that $send(m_1) \longrightarrow send(m_2)$, arrive at any process j then m_1 is delivered before m_2. □

The stabilization of causal deterministic merge is similar to that of the logical timestamps. Note that even if the logical timestamps of messages are corrupted, the causal delivery program never deadlocks since every message is eventually considered for delivery and when multiple messages are considered for delivery simultaneously the *less* relation (along with the tie-breaker based on the process ID) determines the order in which they should be delivered. Once the logical timestamps are restored to their legitimate states, the subsequent computation will satisfy the requirements of causal deterministic merge. Thus, we have

Theorem 4.6. The causal delivery program given above is stabilizing fault-tolerant. □

5 Extensions

In this section, we discuss extensions to our logical timestamps presented in Section 3.2 such that the array $kn.j$ is eliminated. This is achieved at the cost of maintaining unbounded value of $c.j$. Also, for a standard *less* relation, we show that $c.j$ value cannot be bounded. Subsequently, we show that if the application

provides certain guarantees about creation of events in the system, then $c.j$ can also be bounded. In both cases, the timestamp is of the form $\langle r.j, c.j\rangle$. Each of these implementations can be used to solve stabilizing causal deterministic merge.

5.1 Eliminating $kn.j$ at the Cost of Unbounded c Value

In this case, we modify our logical timestamp to be of the form $\langle r.j, c.j\rangle$ where $r.j$ is still the physical clock value of the last event at j. The values of $r.j$ and $c.j$ are updated as shown in Figure 3.

Initially:
$rt.j, r.j, c.j = 0, 0, 0$

Local event e_j/Send event e_j (message being sent is m_j)
if $r.j+2\epsilon < rt.j$ then $c.j = 0$;
else $c.j := max(0, r.j+c.j-rt.j)$
$r.j := rt.j$
$r.e_j, c.e_j, r.m_j, c.m_j := r.j, c.j, r.j, c.j$

Receive event e_j (message m received with timestamp $\langle r.m, c.m\rangle$)
if $(r.m + 2\epsilon < rt.j \ \wedge \ r.j + 2\epsilon < rt.j)$ then $c.j = 0$
else $c.j := max(0, r.j+c.j-rt.j, r.m+c.m-rt.j+1)$
$r.j := rt.j$
$r.e_j, c.e_j := r.j, c.j$

Fig. 3. Revised Logical Timestamp Program

In the program in Figure 3, we first consider the case where the previous event was more than 2ϵ time apart. In that case, we reset $c.j$ to 0. Otherwise, for the send event the program remains unchanged. For the receive event, if $r.m+2\epsilon \not< r.f$ then we ensure that $r.m+c.m < r.f+c.f$ where f is the event that corresponds to receive of message m. (In the previous program, we had permitted $r.m+c.m \leq r.f+c.f$; this allowed us to bound c value at the cost of maintaining the array kn.)

For the program in Figure 3, we define the $less(ts, ts)$ relation for our timestamp of the form $\langle r.j, c.j\rangle$ as follows: Given two events e and f,

$less(\langle r.e, c.e\rangle, \langle r.f, c.f\rangle)$
iff
$(r.e+\epsilon < r.f) \quad \vee$
$((|r.e - r.f| \leq \epsilon) \wedge ((r.e+c.e < r.f+c.f) \vee ((r.e+c.e = r.f+c.f) \wedge r.e < r.f)))$

We note that stabilization can be added to the program in Figure 3 as discussed in Section 3.3.

Proof for Unboundedness. Now, we show that the use of the above $less$ relation requires that the value of $c.j$ is unbounded even in a system with only 3 processes, say j, k and l. We begin in a state where $rt.j = \epsilon$, $rt.k = 0$, $rt.l = 0$

and the c values for all the processes are 0. Now, let j send a message to k. This message is sent with the timestamp $\langle \epsilon, 0 \rangle$. Let this message be received when $rt.k = 1$. The receive event at k has the timestamp $\langle 1, \epsilon \rangle$. Now, let k send a message to l when $rt.k$ equals 2. This message is sent with the timestamp $\langle 2, \epsilon - 1 \rangle$. Let this message be received at l when $rt.l = 1$. The receive event at l has the timestamp $\langle 1, \epsilon + 1 \rangle$. Now, let l send a message to j when $rt.l$ equals 2. This message is sent with the timestamp $\langle 2, \epsilon \rangle$. Let this message be received at j when $rt.j = \epsilon + 1$. The receive event at j has the timestamp $\langle \epsilon + 1, 2 \rangle$. Now, we repeat the scenario with j sending a message with timestamp $\langle \epsilon + 2, 1 \rangle$ as shown in Figure 4. It is straightforward to verify that the c value is unbounded.

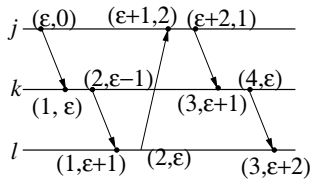


Fig. 4. Eliminating $kn.j$ at the cost of unbounded c value for Program in Figure 3

5.2 Bounding c Value Using Application Guarantee

In the logical timestamp program in Figure 3, the c value is unbounded. However, if an application still requires a bounded solution, it can do so by satisfying a simple guarantee about creation of events. More specifically, if the application periodically provides a window of size 2ϵ such that no events are created on any process in that window then $c.j$ can be bounded. The application can easily satisfy this guarantee if each process ensures that no events are created when the physical clock of that process is in the interval $[\alpha x..\alpha x + 2\epsilon]$ where x is the period and α is a natural number. In this case, the bound on c will be proportional to the period x.

6 Discussion

In this section, we discuss the role that our assumptions played in our program, their fine-tuning and their generality. We also discuss other interpretations of $G1$ and $G2$.

Role of $G1$ and $G2$. In presenting our programs for logical timestamps and causal deterministic merge, we expected two guarantees from the underlying systems. Both these guarantees were necessary to obtain bounded stabilizing solutions. In this sense, the guarantees our solution expects are minimal. It is obvious that if only guarantee $G1$ were available, we could not derive a bounded

stabilizing solution; a message that is delayed for a long enough time, so that a message with a similar timestamp can be generated in the meanwhile, can violate the requirements of logical timestamps (and causal deterministic merge). If only guarantee $G2$ were available, it would not be possible to bound the number of elements in kn; we used $G1$ to determine the number of elements that are maintained in kn.

Fine-Tuning $G1$ and $G2$. In a given system, one needs to determine the values of ϵ and δ. It may also be possible to fine-tune these values depending upon other application requirements. The value of ϵ depends upon the closeness of clock values and their precision. The closeness of clock values will depend upon the clock synchronization algorithm used to correct them. If we provide higher priority for the process that corrects the clock on each processor, provide higher priority for messages sent by the clock synchronization algorithm, and reduce the non-determinism involved in the clock synchronization algorithm, we can reduce the value of ϵ (and, hence, buffer requirements, recovery time, etc). The value of ϵ can also be reduced by increasing the minimum time difference between events. Regarding δ, it is easy to see that there is a tradeoff between the value of δ and the percentage of messages lost. For example, we can use techniques such as forward-error-correction and send parity messages to reduce the message loss. However, in that case, the value of δ will be high as we need to wait for these parity messages to arrive at the destination.

Generality of $G1$ and $G2$. The guarantees expected in our model are satisfied by most existing systems. These guarantees are, however, stronger than that in [8]. Specifically, $G2$ can be easily obtained using the fail-aware datagram service. However, to the best of our knowledge, it is not known whether one can implement $G1$ and ensure that in the presence of faults, $G1$ is established in that model.

Other Interpretations of $G1$ and $G2$. In our model, the value of $rt.j$ may not be related to the auxiliary global time. We have permitted this explicitly to allow for the case where $rt.j$ denotes some other progress measure for the program. For example, in [9], $rt.j$ could denote the number of reset operations that j has performed. Or, in a checkpointing and recovery program, $rt.j$ could denote the incarnation number of j. (Of course, in this case, we will permit K events between incrementing of $rt.j$ where K is a predetermined constant.) Thus, if a program provides the two guarantees $G1$ and $G2$ with respect to the new interpretation of $rt.j$, our solutions can also be applied.

7 Conclusion and Future Work

In this paper, we presented (in Section 4) a bounded state, stabilizing solution for causal deterministic merge. Our solution ensured that even if faults such as message corruption, improper initialization, temporary violation of system guarantees or transients occur, eventually the causal deterministic merge is provided. In our algorithm, the recovery was quick and proportional to the system guarantees. Our solution used only $O(\epsilon log\ n\ +\ log\ \delta)$ space.

To develop a solution to causal deterministic merge, we presented (in Section 3) a self-stabilizing solution for logical timestamps. In our solution, the space cost to implement logical timestamps and the time required to recover from faults is proportional to the guarantees provided by the system.

We also presented (in Section 5) variations of the above solution where we showed that the space required to implement logical timestamps can be made independent of the number of processes if the application satisfies a simple condition on how events are created. For a particular '$less$' relation, in Section 5, we showed that it is impossible to bound the size of logical timestamps without such a condition.

In developing the above solutions, we expected the underlying system to make two guarantees $G1$ and $G2$. We argued (in Section 6) that these guarantees are reasonable in that existing distributed systems satisfy them. We also pointed out how these guarantees can be fine-tuned in a given system.

We showed that given a system that satisfies $G1$ and $G2$, the system obtained after considering the buffering introduced by our causal delivery program also satisfies $G1$ and $G2$ (with slightly different parameters). We expect this property to be useful while building a hierarchical system. At the top level of this system, we will have subnetworks that consist of producers and mergers (which act as subscribers in this subnetwork). In turn, the mergers act as producers in another subnetwork and so on. In this case, the values of ϵ and δ may be different in different subnetworks.

Our solution also improves the previous Δ causal order solution in [2, 3]. Specifically, the solution in [2] assumes the existence of a global clock, and requires $O(n^2)$ space that grows unbounded as the computation progresses. The solution in [3] allows the clock values to differ. However, they also require $O(n^2)$ unbounded space, and can miss some causal dependencies. By contrast, we do not assume the existence of a global clock and use only $O(log\ n)$ bounded space for our timestamps. In [2], a message received within time Δ will be delivered within time Δ. If we assumed the existence of global clocks ($\epsilon = 0$) then our solution will also provide the exact same guarantee (in addition to stabilizing fault-tolerance and bounded logarithmic space).

Regarding work on deterministic merge, Aguilera and Strom [1] have presented a deterministic merge program. In their program, the authors assume that the expected message rate of all producers is known and that the producers send dummy messages if they do not produce the data at the given rate. Also, if the producers produce messages at a faster rate than expected then the message delay grows unbounded. And, the order in which the messages are delivered in their program is not related to the causal order between them. By contrast, we provide causal delivery, do not assume the knowledge about the rate of production of messages, and limit the amount of time for which a message needs to be buffered. However, unlike the program in [1], our program requires that $G1$ and $G2$ are satisfied.

Our causal delivery algorithm is useful in multimedia real-time applications and group-ware real-time applications. In these applications, buffering is limited and the data is valuable only if it is received within some limited time. Our

protocol allows precomputation of buffering requirements and expected delays. Moreover, as discussed in Section 6, we can fine-tune the suitable values for ϵ and δ based upon available buffers and maximum permitted delay. It follows that it would be possible to exploit system level guarantees to further provide guarantees about the flow of the multimedia real-time data.

Our work suggests several directions for future work. Regarding logical timestamps and causal deterministic merge, future work includes identifying the lower bound to solve these problems with $G1$ and $G2$. In [10], we have presented a causal delivery program whose complexity is $O(nlog(\epsilon+\delta))$ whereas the complexity of the program presented in Section 3 is $O(\epsilon\ logn\ +\ log\ \delta)$. It would be interesting to determine if we can reduce the complexity further so that it is logarithmic in both ϵ and n.

References

1. M. Aguilera and R. Strom. Efficient atomic broadcast using deterministic merge. *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 209–218, 2000.
2. R. Baldoni, M. Mostefaoui, and M. Raynal. Causal deliveries in unreliable networks with real-time delivery constraints. *Journal of Real-Time Systems*, 10(3):1–18, 1996.
3. F. Adelstein and M. Singhal. Real-time causal message ordering in multimedia systems. *International Conference on Distributed Computing Systems*, pages 36–43, 1995.
4. S. S. Kulkarni and Ravikant. Tracking causality with physical clocks. Technical Report MSU-CSE-01-20, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, July 2001.
5. L. Lamport. Time, clocks, and the ordering of events in a disributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
6. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, Feb 1988.
7. F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.
8. F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.
9. A. Arora, S. Kulkarni, and M. Demirbas. Resettable vector clocks. *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 269–278, 2000.
10. S. S. Kulkarni. Loosely synchronized timed model. Technical Report MSU-CSE-01-4, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, January 2001.

Fast Self-Stabilizing Depth-First Token Circulation*

Franck Petit

LaRIA, Université de Picardie Jules Verne
5, rue de Moulin Neuf, 80000 Amiens, France

Abstract. We present a simple deterministic distributed depth-first token circulation (DFTC) protocol for arbitrary rooted network. This protocol does not require processors to have identifiers, but assumes the existence of a distinguished processor, called the root of the network. The protocol is self-stabilizing, meaning that starting from an arbitrary state (in response to an arbitrary perturbation modifying the memory state), it is guaranteed to converge to the intended behavior in finite time. The proposed protocol stabilizes in $O(n)$ time units, i.e., no more than the time for the token to visit all the processors (in the depth-first search order). It compares very favorably with all previously published DFTC algorithms for arbitrary rooted networks—they all stabilize in $O(n \times D)$ times, where D is the diameter of the network.

1 Introduction

Modern distributed systems have the inherent problem of faults. The quality of a distributed system design depends on its tolerance to faults that may occur at various components of the system. Many fault tolerant schemes have been proposed and implemented, but the most general technique to design a system that tolerates arbitrary transient faults is self-stabilization [8]. A *self-stabilizing* protocol guarantees that, starting from an arbitrary initial state, the system converges to a desirable state in a finite time.

The *depth-first token circulation problem* (DFTC) is to implement a token circulation scheme where the token is passed from one processor to another in the depth-first order such that every processor gets the token at least once in every token circulation cycle. This scheme has many applications in distributed systems. The solution to this problem can be used to solve the mutual exclusion, spanning tree construction, synchronization, finding the biconnected components, and many other important tasks.

Related Work. Dijkstra introduced the property of self-stabilization in distributed systems by applying it to algorithms for mutual exclusion on a ring [8]. Several other deterministic self-stabilizing token passing algorithms for rooted ring networks and linear arrays of processors have been proposed in the literature, e.g., [3, 4, 11, 12, 20]. Dolev, Israeli, and Moran [9] gave a self-stabilizing

* Supported in part by the Pole of Modelization of Picardie.

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 200–215, 2001.

mutual exclusion protocol by circulating a token in the depth-first search order on a tree network under the model whose actions only allow read/write atomicity. Deterministic self-stabilizing DFTC algorithms for tree networks can be found in [16, 18, 19]. In [9], the authors provide a token circulation scheme for arbitrary networks by constructing a spanning tree and implementing the depth-first token circulation scheme on the constructed spanning tree. There exists many self-stabilizing spanning tree construction algorithms in the literature, e.g., [1, 2, 5]. The algorithms proposed in [16, 18, 19] can be combined with any spanning tree construction providing a token circulation scheme for arbitrary networks. Note that the resulting protocol is not necessary a depth-first token circulation for general networks, but it depends on the structure of the constructed tree on which the DFTC works.

Self-stabilizing depth-first token circulation for arbitrary rooted networks (without pre-computing a rooted spanning tree) was first considered by Huang and Chen [13]. The solution of [13] needs processors having $O(n)$ states ($O(\log n)$ bits). Subsequently, several protocols to solve the self-stabilizing DFTC were proposed [7, 14, 15, 17]. All these papers attempted to reduce the space complexity to $O(\Delta)$, where Δ is the degree of the network. The algorithm presented in [7] offers the best space complexity.

All self-stabilizing DFTC algorithms for general networks in the current literature [7, 13, 14, 15, 17] use two colors to distinguish two consecutive token circulation cycles. To make sure that each cycle reaches all the processors of the network, each cycle must start from a configuration where all the processors are uniformly colored. For all the above solutions, in any configuration where an error exists, it is required *(i)* to correct the abnormal token circulations (token circulations not initiated by the root), *(ii)* to correct processors abnormally colored. The first case (Case *(i)*) is solved in $O(n)$ time units in [13] and $O(n \times D)$ in [7, 14, 15, 17], where D is the diameter of the network. But, all the solutions in [7, 13, 14, 15, 17] need $O(n \times D)$ time units to correct processors abnormally colored (Case *(ii)*). So, the stabilization time for all the existing solutions [7, 13, 14, 15, 17] is $O(n \times D)$.

Contributions. In this paper, our goal is to reduce the stabilization time complexity of the DFTC scheme. We present a self-stabilizing DFTC algorithm (called Algorithm $f\mathcal{DFTC}$) for general networks with a distinguished root. Algorithm $f\mathcal{DFTC}$ needs processors having $O(n)$ states, but it requires no colors to distinguish the token circulation cycles. Our solution stabilizes in only $O(n)$ time units. Note that our stabilization time complexity and the time for the token to complete one DFTC cycle are of the same order.

The rest of the paper is organized as follows: In Section 2, we describe the distributed systems and the model in which our token circulation scheme is written, and give a formal statement of the token passing problem solved in this paper. In Section 3, we present the token passing protocol, and in the following section (Section 4), we give the proof of stabilization of the protocol. In Section 5, the space and the stabilization time complexity of the protocol are given. Finally, we make concluding remarks in Section 6.

2 Preliminaries

In this section, we define the distributed systems and programs considered in this paper, and state what it means for a protocol to be self-stabilizing. We then present the statement of the DFTC problem and its properties.

2.1 Self-Stabilizing System

A *distributed system* is an undirected connected graph, $S = (V, E)$, where V is a set of processors ($|V| = n$) and E is the set of bidirectional communication links. We consider networks which are *asynchronous* and *rooted*, i.e., all processors, except the root are *anonymous*. We denote the root processor by R. A communication link (p, q) exists iff p and q are neighbors. Every processor p can distinguish all its links. To simplify the presentation, we refer to a link (p, q) of processor p simply by the *label* q. We assume that the labels, stored in the set N_p, are arranged in some arbitrary order $\succ_p$ ($\forall q_1, q_2 \in N_p :: (q_1 \succ_p q_2) \wedge (q_2 \succ_p q_1) \iff (q_1 = q_2)$). We assume that N_p is a constant and is maintained by an underlying protocol.

Each processor executes the same program except R. The program consists of a set of *shared variables* (henceforth referred to as variables) and a finite set of actions. A processor can only write to its own variables and can only read its own variables and variables owned by the neighboring processors. So, the variables of p can be accessed by p and its neighbors.

Each action is uniquely identified by a label and is of the following form: $< label >:: < guard > \longrightarrow < statement >$. The guard of an action in the program of p is a boolean expression involving the variables of p and its neighbors. The statement of an action of p updates variables of p. An action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed: the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step. The atomic execution of an action of p is called a *step* of p.

The *state* of a processor is defined by the values of its variables. The *state* of a system is a product of the states of all processors ($\in V$). In the sequel, we refer to the state of a processor and system as a (*local*) *state* and *configuration*, respectively. Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\mapsto$ on $\mathcal{C}$, the set of all possible configurations of the system. A *computation* of a protocol $\mathcal{P}$ is a *maximal* sequence of configurations $e = (\gamma_0, \gamma_1, ..., \gamma_i, \gamma_{i+1}, ...)$, such that for $i \geq 0, \gamma_i \mapsto \gamma_{i+1}$ (a single *computation step*) if γ_{i+1} exists, or γ_i is a terminal configuration. *Maximality* means that the sequence is either infinite, or it is finite and no action of $\mathcal{P}$ is enabled in the final configuration. All computations considered in this paper are assumed to be maximal. The set of computations of a protocol $\mathcal{P}$ in system S starting with a particular configuration $\alpha \in \mathcal{C}$ is denoted by $\mathcal{E}_\alpha$. The set of all possible computations of $\mathcal{P}$ in system S is denoted as $\mathcal{E}$.

A processor p is said to be *enabled* in γ ($\gamma \in \mathcal{C}$) if there exists at least an action A such that the guard of A is true in γ. When there is no ambiguity,

we will omit γ. We consider that any processor p executes a *disable action* in the configuration step $\gamma_i \mapsto \gamma_{i+1}$ if p is enabled in γ_i and not enabled in γ_{i+1}, but does not execute any action between these two configurations. (The disable action represents the following situation: At least one neighbor of p changes its state between γ_i and γ_{i+1}, and this change effectively made the guard of all actions of p false.) Similarly, an action A is said to be enabled (in γ) at p if the guard of A is true at p (in γ). We assume a *weakly fair and distributed daemon.* The *weak fairness* means that if a processor p is continuously enabled, then p is eventually chosen by the daemon to execute an action. The *distributed* daemon implies that during a computation step, if one or more processors are enabled, then the daemon chooses at least one (possibly more) of these enabled processors to execute an action.

In order to compute the time complexity, we use the definition of *round* [10]. This definition captures the execution rate of the slowest processor in any computation. Given a computation e ($e \in \mathcal{E}$), the *first round* of e (let us call it e') is the minimal prefix of e containing the execution of one action (an action of the protocol or the disable action) of every continuously enabled processor from the first configuration. Let e'' be the suffix of e, i.e., $e = e'e''$. Then *second round* of e is the first round of e'', and so on.

Let $\mathcal{X}$ be a set. $x \vdash P$ means that an element $x \in \mathcal{X}$ satisfies the predicate P defined on the set $\mathcal{X}$. A predicate is non-empty if there exists at least one element that satisfies the predicate. We define a special predicate true as follows: *for any* $x \in \mathcal{X}$, $x \vdash \mathsf{true}$.

We use the following term, *attractor* in the definition of self-stabilization.

Definition 1 (Attractor). *Let X and Y be two predicates of a protocol $\mathcal{P}$ defined on $\mathcal{C}$ of system $\mathcal{S}$. Y is an attractor for X if and only if the following condition is true:* $\forall \alpha \vdash X : \ \forall e \in \mathcal{E}_\alpha : \ e = (\gamma_0, \gamma_1, ...) :: \ \exists i \geq 0, \forall j \geq i, \gamma_j \vdash Y$. *We denote this relation as* $X \triangleright Y$.

Definition 2 (Self-Stabilization). *The protocol $\mathcal{P}$ is self-stabilizing for the specification $\mathcal{SP}_\mathcal{P}$ on $\mathcal{E}$ if and only if there exists a predicate $\mathcal{L}_\mathcal{P}$ (called the legitimacy predicate) defined on $\mathcal{C}$ such that the following conditions hold:*
1. $\forall \alpha \vdash \mathcal{L}_\mathcal{P} : \ \forall e \in \mathcal{E}_\alpha :: \ e \vdash \mathcal{SP}_\mathcal{P}$ *(correctness),*
2. $\mathit{true} \triangleright \mathcal{L}_\mathcal{P}$ *(closure and convergence).*

2.2 Specification of the Depth-First Token Circulation Protocol

Definition 3 (*DFTC-cycle*).

A finite computation $e \in \mathcal{E}$ is called a DFTC-cycle if the following conditions are true:

[S] *Exactly one processor holds a token in any configuration;*
[L1] *The root initiates the DFTC-cycle by sending out exactly one token;*
[L2] *When a processor p receives a token, p sends the token to a processor following the depth-first search order.*

We consider a computation e of $f\mathcal{DFTC}$ to satisfy $\mathcal{SP}_{DFTC}$ iff e is an infinite repetition of depth-first circulation cycles (as in Definition 3).

3 Depth-First Token Circulation Algorithm

We borrow the following term from [6]: The *first DFS tree* of the graph G is defined as the DFS spanning tree rooted at R, created by traversing the graph in the DFS manner and visiting the adjacent edges of every processor p in the order induced by $\succ_p$. The depth-first token circulation proposed in this section is designed in such a way that the token is passed among the processors following the first DFS tree. We first present the data structure used by the processors. We then explain the normal behavior of the algorithm, followed by the description of the error correction mechanism.

3.1 Data Structures

The self-stabilizing depth-first token circulation algorithm (Algorithm $f\mathcal{DFTC}$) is shown in Algorithm 1 for the root (R) and in Algorithm 2 for the other processors. The macros are not variables and are dynamically evaluated. The predicates are used to describe the guards of the actions in Algorithm $f\mathcal{DFTC}$.

Algorithm 1 ($f\mathcal{DFTC}$) For Processor R ($p = \mathtt{R}$)

Constants: N_p : set of (locally ordered) neighbors; $L_p = 0$;
Variables: $S_p \in N_p \cup \{done\}$;
Macros:
$Next_p \equiv (q = \min_{\succ_p}\{q' \in N_p :: (q' \succ_p S_p) \wedge (S_{q'} = idle)\})$ **if** q **exists**, $done$ **otherwise**;
Predicates:
$Forward(p) \equiv (S_p = done)$
$Backward(p) \equiv (S_p = q :: q \in N_p \wedge S_q = done)$
$Locked(p) \equiv (\exists q \in N_p :: S_q \not= idle)$
. . . . . .
Actions:
$F :: Forward(p) \wedge \neg Locked(p) \rightarrow S_p := Next_p;$
$B :: Backward(p) \rightarrow S_p := Next_p;$

Each processor p maintains two variables, S_p and L_p. If $p = \mathtt{R}$, then $S_{\mathtt{R}} \in N_{\mathtt{R}} \cup \{done\}$. Otherwise, ($p \neq \mathtt{R}$), $S_p \in N_p \cup \{idle, done, wait\}$. For every processor p, if there exists $q \in N_p$ such that $S_p = q$, then p (resp., q) is said to be a *predecessor* of q (resp., the *successor* of p). In other words, when $S_p \notin \{done, idle, wait\}$, S_p plays the role of a pointer pointing to the neighbor to which p sent the token. Variable L_p contains the length of the path followed by the token from the root to p. Since the length from the root to itself is 0, $L_{\mathtt{R}}$ must be equal to zero, and hence, is shown as a constant in Algorithm 1.

Algorithm 2 ($f\mathcal{DFTC}$) For other processors ($p \neq \mathtt{R}$)

Constants: N_p : set of (locally ordered) neighbors; $L_{max} = n - 1$;
Variables: $S_p \in N_p \cup \{idle, done, wait\}$; $L_p \in \{1, \ldots, L_{max}\}$;
Macros:

$Next_p \equiv (q = \min_{\succ_p}\{q' \in N_p :: (q' \succ_p S_p) \wedge (S_{q'} = idle)\})$ **if** q **exists**, $done$ **otherwise**;
$Pred_p \equiv \{q \in N_p :: S_q = p\}$;
$RealPred_p \equiv \{q \in Pred_p :: L_q = L_p - 1\}$;

Predicates:

$ForwardI(p) \equiv (|Pred_p| = 1) \wedge (S_p = idle)$
$ForwardW(p) \equiv (|Pred_p| = 1) \wedge (S_p = wait)$
$Forward(p) \equiv ForwardI(p) \vee ForwardW(p)$
$Backward(p) \equiv (|RealPred_p| = 1) \wedge (S_p = q :: q \in N_p \wedge S_q = done)$
$Locked(p) \equiv (\exists q \in N_p :: S_q = done) \vee (\exists q \in Pred_p :: L_q \geq L_{max})$
$Clean(p) \equiv (S_p = done) \wedge ((\forall q \in N_p :: S_q \in \{idle, done\}) \vee (\exists q \in N_p :: S_q = wait))$
$CleanW(p) \equiv (S_p = wait) \wedge (|Pred_p| = 0))$
$EDetect(p) \equiv (S_p = q :: q \in N_p \wedge ((L_q = 0) \vee (|RealPred_p\}| = 0)))$

Actions:

$F :: Forward(p) \wedge \neg Locked(p) \rightarrow S_p := Next_p;\ L_p := L_{q:\{q\}=Pred_p} + 1;$
$B :: Backward(p) \rightarrow S_p := Next_p;$
$W :: ForwardI(p) \wedge Locked(p) \rightarrow S_p := wait;$
$C :: Clean(p) \rightarrow S_p := idle;$
$Error :: EDetect(p) \vee CleanW(p) \rightarrow S_p := idle;$

3.2 Normal Behavior of Algorithm $f\mathcal{DFTC}$

In this subsection, we first ignore the parts of the algorithm where Variable L_p is involved (e.g., Macro $RealPred_p$, the right part of Predicate $Locked$, etc.), because Variable L_p is used to handle the correction of abnormal situations only. Once stabilized, the system must contain only one token which circulates following the first DFS tree. In such a configuration, a processor can make a move only if it holds the token. Holding the token means either $Forward(p)$ or $Backward(p)$ is true. Formally, $Token(p) \equiv Forward(p) \vee Backward(p)$. A processor p ($p \neq \mathtt{R}$) is said to be $idle$ ($S_p = idle$) when p is ready to participate in the next DFTC cycle. Similarly, a processor p is in state $done$ ($S_p = done$) when p has completed the current DFTC cycle. Let us omit the state $wait$ ($S_p = wait$) for a while.

Macro $Pred_p$ (Algorithm 2) is the subset of the neighbors of p which took p as successor. When Algorithm $f\mathcal{DFTC}$ behaves accordingly to its specification, $|Pred_p| = 1$. Macro $Next_p$ is used to choose the next successor of p.

Let us explain the normal behavior of Algorithm $f\mathcal{DFTC}$ following the example shown in Figure 1. For a better presentation, in the example, we assume that the daemon chooses all enabled processors to execute an action (i.e., the daemon behaves as a synchronous daemon). In Configuration (i), only Action F

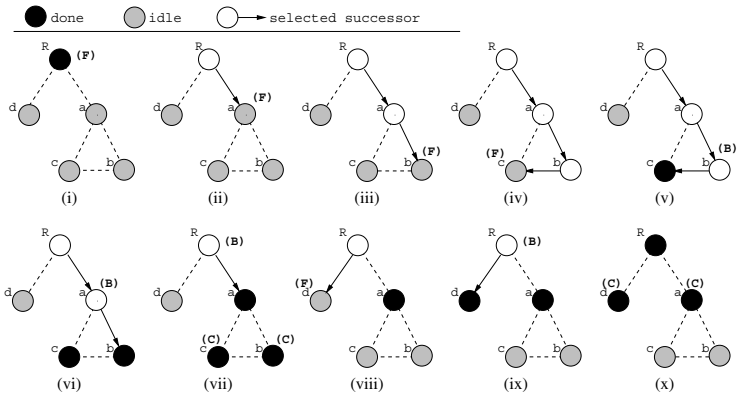


Fig. 1. An example showing the normal behavior of Algorithm $f\mathcal{DFTC}$.

is enabled at R. This means that the root is ready to start a new DFTC cycle. The root chooses Processor a as the successor (Macro $Next$). This is shown in Configuration (ii). Similarly, Processor a chooses a successor (Configuration (iii)) executing Action F. This process of extending the path continues until c executes Action F. Processor c does not have any neighbor to choose from. So, c executes $S_c := done$ (see Macro $Next$). This indicates to its predecessor b that the token has traversed all processors reachable from c in the first DFS tree (Configuration (v)). Now, $Backward(b)$ becomes true and b can execute Action B. Since b has no more unvisited neighbors, S_b becomes equal to *done* (Configuration (vi)). Next, $Backward(a)$ also becomes true leading a to execute $S_a := done$ (Configuration (vii)). Note that when Processor a is done, Processors b and c can *clean* their state by changing their state from *done* to *idle*. So, Actions F and C (or B and C) can run concurrently. Actions F and B are repeated until all processors are visited by the token (Configurations $(viii)$ to (x)). The configuration following Configuration (x) is Configuration (i) when Processors a and d clean their state. Then, R starts a new DFTC cycle.

As we mentioned before, the example shown in Figure 1 assumed a synchronous daemon working synchronously. But, our algorithm works in an asynchronous environment as well. Due to asynchrony in the network, some processors may be involved in a DFTC cycle whereas the others are still cleaning their state following the previous DFTC cycle. But, we need to make sure that these two cycles do not confuse each other. We solve this problem by using the *cleaner* [20]. The cleaner is a tool adding two preventives in the algorithm. (i) A processor p is allowed to change its state from *done* to *idle* (Action C) only when all its neighbors are *done* or *idle* (Predicate $Clean(p)$). (ii) A processor p such that $Forward(p)$ is true can select its successor when each of its neighbors which is not its predecessor is *idle* ($Locked(p)$ must be false). Thus, the processors which are slow to execute their Action C, are protected from the next DFTC cycle.

Finally, consider the configuration in which $Forward(p)$ is true on processor p and p is waiting for some of its neighbors being slow to execute Action C. Note that in such a configuration, p can execute Action W (both $ForwardI(p)$ and $Locked(p)$ are true). By executing Action W, p changes its state S_p from *idle* to *wait*. This is called the *wait* mechanism. The wait mechanism enforces each neighbor q of p which is slow to change its state from *done* to *idle*, whatever the state of the other neighbors of q (see Predicate $Clean()$). This does not slow down the progress of the token because it is executed concurrently with the cleaning phase (of slow processors). However, it is clear that this mechanism is not necessary to make sure that Algorithm $f\mathcal{DFTC}$ behaves correctly (according to its specification). We show in the next subsection (Subsection 3.3) that the wait mechanism ensures the liveness of Algorithm $f\mathcal{DFTC}$ starting from an abnormal configuration.

3.3 Error Correction

We now consider the abnormal situations due to the unpredictable initial configurations and transient errors. An example showing an illegitimate configuration is shown in Figure 2.

Informally, a configuration is an "*abnormal*" configuration if one of the following situation occurs:

[AC1] Some neighbors of the root have the root as a successor (see Processor d in Figure 2);

[AC2] Some processors are involved in some successor chains not rooted in R (in Figure 2, all processors other than the root and h, i, n, and o are in such a configuration);

[AC3] Some processors are in the state *wait* without predecessor (Processor h in Figure 2 is in this configuration).

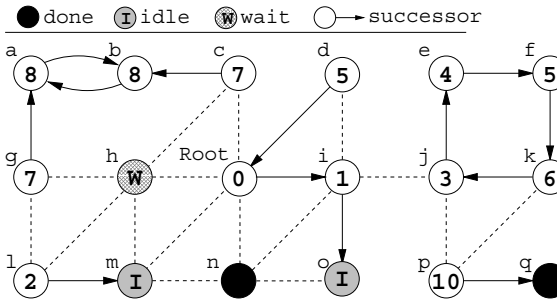


Fig. 2. An example showing an abnormal configuration.

The system may contain chains of successors forming cycles (possibly including R) or "alive" chains, i.e., chains of successors not rooted in R which causes

several tokens circulating in the network. In order to break the cycles, we use the length of the path followed by the token to p, i.e., Variable L_p [13]. When $S_p \in \{idle, done, wait\}$, the value of Variable L_p is ignored. Any processor p sets its variable L_p to the value of its predecessor by executing Action F ($L_{\mathtt{R}}$ is fixed to 0). (In Figure 2, Variable L_p is shown for processors having a successor.) Thus, if some processors p are involved in a cycle, some of them must have their level different from their predecessor q in the cycle, i.e., $L_p \neq L_q + 1$. In Figure 2, Processors a, b, and j are such processors. Every predecessor q of p such that $L_q = L_p - 1$ is called a *real* predecessor of p (see Macro $RealPred_p$ of Algorithm 2). A processor p without a real predecessor but having a successor are called *worm bottom*. The successor chain starting in p is called a *worm*.

Using L_p, a neighbor of the root can also detect that it is a predecessor of $\mathtt{R}$ because the level of its successor is 0. Variable L_p also allows to stop the progress of worms. A processor p ready to forward a token ($Forward(p)$ is true) will be allowed to do so if the level value of its predecessors is lower than $L_{max} = n - 1$ (see Predicate $Locked(p)$ in Algorithm 2). Action *Error* of Algorithm 2 is used to handle all the abnormal configurations described above.

So far, we did not explain the use of the wait mechanism in dealing with the errors. Consider the configuration where $Forward(p)$ is enable at a processor $p \neq \mathtt{R}$. Due to the unpredictable initial configuration, some neighbors of p can be in the state *done*. An example of such a configuration is shown in Figure 2: Processor o is waiting for Processor n to execute Action C, but Processor n is waiting for all its neighbors (in particular $\mathtt{R}$ and Processor i) to be *done* or *idle*. Note that the two features of the cleaner prevents this kind of configurations to occur during the normal token circulation (see Subsection 3.2). Now, assume that the network contains no worm, and no processor has $\mathtt{R}$ as a successor. Then, only one token exists in the network ($Token(p)$ is true). The system seems to be in a deadlock configuration. We introduce the wait mechanism to unlock such a configuration. By executing Action W, the processor holding the token enforces its neighbors to change their state from *done* to *idle* (see Predicate $Clean()$). In Figure 2, once Processor o executed Action W, Action C is enabled at Processor n which eventually executes Action C by fairness. But, the state *wait* introduces the third abnormal situation we mentioned above (Case [$AC3$]): some processors are in state *wait* with no predecessor. Predicate $CleanW()$ and Action *Error* handle the correction of this abnormal situation.

4 Stabilization Proof of Algorithm $f\mathcal{DFTC}$

A *DFTC-cycle* starts in a configuration which satisfies some special properties in terms of the state variable values. We now define an equivalence class of configurations which satisfies these properties.

Let us define the set SC (starting configuration set) such that

$$SC = \left\{ \gamma \in \mathcal{C} :: \ \gamma \vdash \begin{pmatrix} S_{\mathtt{R}} = done \ \wedge (\forall p \in N_{\mathtt{R}} :: \ S_p = idle) \wedge \\ (\forall p \in V \setminus (\{\mathtt{R}\} \cup N_{\mathtt{R}}) :: \ S_p \in \{idle, done\}) \end{pmatrix} \right\}$$

We define two configurations γ and $\gamma\prime$ as *equivalent* with respect to the set SC if the following condition is true: $\gamma \Leftrightarrow_{SC} \gamma\prime$ iff $(\gamma \in SC) \wedge (\gamma\prime \in SC)$.

Definition 4. *Let $\mathcal{L}_{DFTC}$ be the legitimacy predicate such that a configuration γ satisfies $\mathcal{L}_{DFTC}$ ($\gamma \vdash \mathcal{L}_{DFTC}$) if the following holds: $\forall \gamma_0 \in SC$, $\exists e = \gamma_0, \gamma_1, \ldots \in \mathcal{E}_{\gamma_0}$, $\exists i \geq 0 :: \ \gamma = \gamma_i$.*

We exhibit a finite sequence of state predicates $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_m$, of Protocol $f\mathcal{DFTC}$ such that the following conditions hold: (*i*) $\mathcal{A}_0 \equiv \mathsf{true}$ (meaning any arbitrary state); (*ii*) $\forall j : \ 0 \leq j < m :: \ \mathcal{A}_j \triangleright \mathcal{A}_{j+1}$ (*iii*) $\mathcal{A}_m \Rightarrow \mathcal{L}_{DFTC}$.

Definition 5 (Successor Path). *For any node p such that $S_p \in N_p$, the successor path $\overrightarrow{p}$ is the unique path $p_1, p_2, \ldots, p_k$ such that (i) $p = p_1$, (ii) $k \geq 2$, (iii) $\forall i$, $1 \leq i \leq k-1$, $S_{p_i} = p_{i+1}$, and (iv) $S_{p_k} \notin \{idle, done, wait\} \Rightarrow S_{p_k} \in \{p_1, p_2, \ldots, p_{k-1}\}$. $\forall i \geq 1$, p_i is said to belong to $\overrightarrow{p}$ and is denoted as $p_i \in \overrightarrow{p}$. k is called the length of $\overrightarrow{p}$.*

Let us define the following predicates :

$LevelError(p) \equiv (S_p = q : q \in N_p \wedge (S_q \neq done \Rightarrow (S_q = q' : q' \in N_q \wedge L_q \neq L_p + 1)))$

$Top(p) \equiv \quad (p = \mathtt{R} \wedge (S_p \neq done \Rightarrow LevelError(p))) \vee$
$\qquad (p \neq \mathtt{R} \wedge ((S_p \in \{idle, wait\} \wedge Pred_p \neq \emptyset) \vee LevelError(p)))$

A processor p is called a *top* in Configuration γ if $Top(p)$ holds in γ. When there is no ambiguity, we will omit γ. In Figure 2, Processors a, b, d, k, m, o, and p are top processors.

Remark 1. Every successor path $\overrightarrow{p}$ contains at least one top processor.

Let us define another predicate :
$WBottom(p) \equiv (S_p \in N_p \wedge RealPred_p = \emptyset)$. A processor p is called a *Worm Bottom* (or simply a *Bottom*) in Configuration γ if $WBottom(p)$ holds in γ. When there is no ambiguity, we will omit γ. Note that if $Pred_p = \emptyset$, then $WBottom$ is true if $S_p \in N_p$. In Figure 2, Processors c, d, g, j, l, and p are bottom processors.

Definition 6 (Worm). *A successor path $\overrightarrow{p} = p_1, p_2, \ldots p_k$ is a worm (path) if and only if the following conditions are true:*
(1) $WBottom(p)$,
(2) $\forall i, \ 1 \leq i \leq k, \ Top(p_i) \Rightarrow i = k$.

Note that the length of a worm can be equal to 1. That is the case where the state of the successor of the worm bottom is *done*. In Figure 2, the following successor paths are worms: (c, b), (g, a), (d), (j, e, f, k), (l, m), and (p). Processors (d) and (p) form worms of length 1.

Remark 2. Every worm $\overrightarrow{p} = p_1, p_2, \ldots, p_k$ with length greater than 1 ($k > 1$) is an (elementary) path such that $\forall i, \ 1 \leq i < k-1$, $L_{p_{i+1}} = L_{p_i} + 1$.

4.1 Root without a Predecessor

Let $Pred_{\mathtt{R}}$ be the set of processors p such that $S_p = \mathtt{R}$. We define $\mathcal{A}_1 \equiv (Pred_{\mathtt{R}} = \emptyset)$. We now show that, $\mathcal{A}_1$ is an attractor.

Lemma 1. *Every processor $p \in Pred_{\mathtt{R}}$ remains in $Pred_{\mathtt{R}}$ at most one round.*

Proof. Except Action *Error*, no action is enabled on processors in $Pred_{\mathtt{R}}$. Since $L_{\mathtt{R}}$ is a constant ($L_{\mathtt{R}} = 0$), for every $p \in Pred_{\mathtt{R}}$, $EDetect(p)$ is true while p does not execute Action *Error*. So,by fairness, every $p \in Pred_{\mathtt{R}}$ eventually executes Action *Error*. □

Theorem 1 ($\mathcal{A}_0 \triangleright \mathcal{A}_1$). *In at most* 1 *round, $Pred_{\mathtt{R}} = \emptyset$, and, once this happens, $Pred_{\mathtt{R}} = \emptyset$ forever.*

Proof. Since $S_{\mathtt{R}}$ cannot be equal to *idle*, R cannot be chosen as a successor by any of its neighbors. (see in macro $Next_p$). So, $|Pred_{\mathtt{R}}|$ cannot increase. By Lemma 1, in at most 1 round $Pred_{\mathtt{R}} = \emptyset$. Once $Pred_{\mathtt{R}} = \emptyset$, again because R cannot be chosen as a successor, $Pred_{\mathtt{R}} = \emptyset$ forever. □

4.2 Worm Destruction

In the remainder, we assume that $\mathcal{A}_1$ always holds. Let W be the number of worms in the network. Define $\mathcal{A}_2 \equiv \mathcal{A}_1 \wedge (W = 0)$. We now show that $\mathcal{A}_2$ is an attractor, i.e., the network eventually contains no worm.

Lemma 2. *Every worm bottom p remains a worm bottom until it executes Action Error.*

Proof. Let $\gamma_i \mapsto \not\rightarrow \gamma_{i+1}$ be the transition of an execution e such that (1) p is a worm bottom in γ_i and (2) p does not execute Action *Error* during $\gamma_i \mapsto \not\rightarrow \gamma_{i+1}$. Since $S_p \neq idle$ in γ_i, no neighbor of p can choose p as successor during $\gamma_i \mapsto \not\rightarrow \gamma_{i+1}$ (see Macro $Next_p$). So, $|Pred_p|$ in γ_i is lower than or equal to $|Pred_p|$ in γ_{i+1}. Moreover, no $q \in Pred_p$ can change its level value (L_q) during $\gamma_i \mapsto \not\rightarrow \gamma_{i+1}$ (no rule for processors having a successor exists to do so). So, $|RealPred_p|$ in γ_i is lower than or equal to $|RealPred_p|$ in γ_{i+1}. So, p remains a worm bottom in γ_{i+1}. By induction on each transition following $\gamma_i \mapsto \not\rightarrow \gamma_{i+1}$ in e, p remains a worm bottom until it executes Action *Error*. □

Corollary 1. *A worm bottom processor p can stay as a worm bottom at most one round.*

Lemma 3. *If for any $\gamma_i \mapsto \not\rightarrow \gamma_{i+1}$, a processor p becomes a worm bottom with level L_p in γ_{i+1}, and p was not a worm bottom in γ_i, then in γ_i, $\forall q \in Pred_p$, q is a worm bottom with level $L_q = L_p - 1$.*

Proof. Assume by contradiction that there exists at least one $q \in Pred_p$ which is not a worm bottom in γ_i. So, both p and q are not worm bottom in γ_i. Consider the three following cases:

1. $S_p \in \{idle, wait\}$ in γ_i. Since p is a worm bottom in γ_{i+1}, p executes an action during $\gamma_i \mapsto \gamma_{i+1}$ such that S_p changes from $S_p \in \{idle, wait\}$ (in γ_i) to q', $q' \in N_p$, in γ_{i+1}. The only possible action that can make this happen is Action F. This implies that $S_q = p$ in γ_i, and q executes an action during $\gamma_i \mapsto \gamma_{i+1}$ such that $S_q \neq p$ or $L_q = L_p - 1$ in γ_{i+1} . The only possible action which q can execute to make this change is Action $Error$. So, q is a worm bottom in γ_i, which contradicts the assumption.
2. $S_p = done$ in γ_i. Only Action C can be executed when $S_p = done$. But, $Pred_p$ must be empty. So, p cannot become a worm bottom during $\gamma_i \mapsto \gamma_{i+1}$, which contradicts that p is a worm bottom in γ_{i+1}.
3. $S_p = q'$, $q' \in N_p$ in γ_i. Then, q can execute Action $Error$ only because $S_q \notin \{idle, wait, done\}$ prevents q to execute Actions F, W, C, and $S_p \neq done$ prevents q to execute Action B. So, q is a worm bottom in γ_i, which contradicts the assumption.

□

Corollary 2. *For any $\gamma_i \mapsto \gamma_{i+1}$, if there exists no worm bottom in γ_i, then there exists no worm bottom in γ_{i+1}.*

From Remark 2, Corollary 1 and Lemma 3 directly follows:

Lemma 4. *The highest level of any worm bottom in the network increases by at least one in one round.*

From Corollary 2, Lemma 4 , and the fact that the level of every processor is bounded by L_{max}, the following theorem holds:

Theorem 2 ($\mathcal{A}_1 \triangleright \mathcal{A}_2$). *In at most L_{max} rounds, $W = 0$ (the network contains no worm) and, once this happens, $W = 0$ (the network remains without worm) forever.*

Corollary 3. *In any configuration $\vdash \mathcal{A}_2$, there exists exactly one processor such that $Token(p)$ is true.*

4.3 Abnormal Waiting Processor Removal

In the remainder, we assume that $\mathcal{A}_2$ always holds. A processor p is said to be an *abnormal waiting* processor if $S_p = wait$ and $|Pred_p| = 0$. Let AWS be the set of abnormal waiting processors. Define $\mathcal{A}_3 \equiv \mathcal{A}_2 \wedge (|AWS| = 0)$.

Theorem 3 ($\mathcal{A}_2 \triangleright \mathcal{A}_3$). *In at most 1 round, $AWS = \emptyset$ (no abnormal waiting processor exists), and, once this happens, $AWS = \emptyset$ (no abnormal waiting processor exists) forever.*

Proof. Since no action allows any predecessor of a waiting processor to change its state, no waiting processor can become an abnormal waiting processor. (Recall that once $\mathcal{A}_2$ holds, there exists no worm). So, $|AWS|$ never increases. Moreover, no action allows a processor to choose an abnormal waiting processor as successor. So, by fairness, every abnormal waiting processor p remains an abnormal waiting processor at most one round. Thus, in at most 1 round, $AWS = \emptyset$. Once this happens, since $|AWS|$ cannot increase, $AWS = \emptyset$ forever. □

4.4 Legitimacy Predicate

In the remainder, we assume that $\mathcal{A}_3$ always holds. At this point of the proof, one can notice that even if $\overrightarrow{\mathtt{R}}$ is the only possible successor path in the network, it may follow edges which are not edges of the first DFS free. Moreover, while R does not initiate a new DFTC cycle, the token may not visit some parts of the network. So, we need to show that the token infinitely moves among the processors, and the system eventually reaches a configuration from which Algorithm $f\mathcal{DFTC}$ behaves according to its specification.

Lemma 5. *Let p ($p \neq \mathtt{R}$) be a processor such that $Clean(p)$ is true at $\gamma_i \vdash \mathcal{A}_3$. Processor p executes Action C in at most one round.*

Proof. Starting from a configuration γ_i ($(\vdash \mathcal{A}_3)$ in which Processor $p \neq \mathtt{R}$ is *done*, p can execute only Action C (unique possible action in the state *done*). So, starting from a configuration in which $Clean(p)$ is true, either p eventually executes Action C or p never executes any action ($S_p = done$ forever). Assume by contradiction that p does not change S_p to *idle* in at most one round. So, starting from a configuration γ_i ($\vdash \mathcal{A}_3$) where $Clean(p)$ is true, p never executes Action C ($S_p = done$ forever), but $Clean(p)$ does not remain true forever (otherwise, by fairness, p eventually executes Action C, which contradicts the assumption). Since $Clean(p)$ is true in γ_i, $S_p = done$ and, either $\exists q \in N_p$ such that $S_q = wait$ or $\forall q \in N_p$, $S_q \in \{idle, done\}$, at γ_i.

1. There exists $q \in N_p$, $S_q = wait$ at γ_i. Since $S_p = done$ at γ_i, $Locked(q)$ is true at γ_i. Since by assumption $S_p = done$ forever, once in the state *wait*, $Locked(q)$ is true forever. So, q is never able to execute an action. Thus, $Clean(p)$ remains true forever, which contradicts the assumption.
2. $\forall q \in N_p$, $S_q \in \{idle, done\}$ at γ_i. No neighbor q in the state *idle* can execute either Action F (because $Locked(q)$ is true) or Action W (this case would lead to Case 1). So, every q in the state *idle* remains *idle* forever. Any q in the state *done* can only change to *idle* (by executing Action C). So, either every $q \in N_p$ is eventually *idle* forever or some $q \in N_p$ remains in the state *done* forever. In both cases, $Clean(p)$ remains true forever, which contradicts the assumption.

□

Lemma 6. *Let p be a processor such that $Forward(p)$ is true at $\gamma_i \vdash \mathcal{A}_3$. Processor p executes Action F in at most two rounds.*

Proof. There are three cases:

1. $p = \mathtt{R}$ and $S_p = done$ at γ_i. From Lemma 5, $\forall q \in N_p$, $S_q = idle$ in at most one round. Next, $Premission(p)$ is continuously true and p eventually executes Action F (in one round).
2. $p \neq \mathtt{R}$ and $S_p = wait$ at γ_i. This case is similar to Case 1.
3. $p \neq \mathtt{R}$ and $S_p = idle$ at γ_i. Again, there are two cases:
   (a) $Locked(p)$ is false in γ_i. Then, p eventually executes Action F (in one round).
   (b) $Locked(p)$ is true in γ_i. Then, From Lemma 5, $\forall q \in N_p$, $S_q = idle$ in at most one round. Then, once $\forall q \in N_p$, $S_q = idle$, $Locked(p)$ is false and either S_p remains $idle$ (p does not execute Action W), or p changed its states to $wait$. In both case, in at most one extra round, p executes Action F.

□

Lemma 7. *Let p be a processor such that $Backward(p)$ is true at $\gamma_i \vdash \mathcal{A}_3$. Processor p executes Action B in at most one round.*

Proof. Let q be the neighbor of p such that $S_p = q$. Then, $S_q = done$ and no action is enabled on q. So, $Backward(p)$ remains true until p executes Action$B$. □

From Lemmas 6, Lemma 7, Theorems 1, 2, 3 and Corollary 3 directly follows:

Theorem 4 (Liveness). *The token always circulates among the processors.*

We now define the following for a configuration $\gamma \vdash \mathcal{A}_3$: $\mathcal{A}_4 \equiv \mathcal{A}_3 \wedge \mathcal{L}_{\mathcal{DFTC}}$.

Lemma 8. *Every computation starting from a configuration $\gamma_i \vdash \mathcal{A}_3$ leads to a configuration in SC in at most $3 \times n$ rounds.*

Proof. From Theorem 4, the token always circulates among the processors. For any processor p, while $S_p \in N_p$, no neighbor of p which is *done* can clean its state. So, p cannot choose a neighbor as successor twice during a token circulation cycle. Since Macro $Next_p$ is based on the order $\succ_p$ on the finite set N_p, every processor p involved in the *DFTC-cycle* is eventually *done*. So, each edge pq visited by the token is visited at most twice before reaching a configuration in SC. So, without lost of generality, p executes Action F first (to send the token to q). Next, p executes Action B (once q is done). Thus, from Lemmas 6 and 7 and Theorem 4 again, after at most $3 \times n$ rounds, the system is in a configuration in SC. □

Theorem 5 ($\mathsf{true} \triangleright \mathcal{A}_4$). *Algorithm DFTC verifies the closure and convergence property.*

Proof. Directly follows from Theorems 1, 2, 3, 4, Lemma 8, and Definition 4. □

By construction of Algorithm $f\mathcal{DFTC}$, the token circulates among the processors following the first DFS tree. So, by Definition 3, Algorithm $f\mathcal{DFTC}$, and Theorem 5 follows:

Theorem 6 (Self-Stabilization). *Protocol $f\mathcal{DFTC}$ stabilizes for Specification $\mathcal{SP}_{DFTC}$.*

5 Complexities of Algorithm $f\mathcal{DFTC}$

Each processor $p \neq \mathtt{R}$ needs $n-1 \times (\Delta_p + 3)$ states ($n-1$ states for Variable L_p, $\Delta_p + 3$ states for Variable S_p), Δ_p is the degree of p.

From Theorem 1, the root has no predecessor in at most one round. From Theorem 2, the network contains no worm in at most L_{max} rounds. From Theorem 3, the network contains no abnormal waiting processor in at most one round. Note that the fact that no neighbor of the root has the root as a successor is independent of the removal of the worm. So, they proceed in parallel, i.e., the first round of the L_{max} rounds to remove the worms is the same round where the predecessors of the root correct their state. Conversely, the round removing abnormal waiting processors can occur only after L_{max} rounds spent in removing the worms. This is due to the fact that the removal of a worm can create an abnormal waiting processor (see the proof of Theorem 3). In the worst case, when the system contains only one token and no abnormal waiting processor, the number of rounds required to reach a configuration in SC is the time to complete a DFTC cycle, i.e., $3 \times n$ (Lemma 8). Hence, by adding the number of rounds required to remove all abnormal configuration ($n+1$ rounds) to the time to complete a DFTC cycle, we get the following result:

Theorem 7. *Algorithm $f\mathcal{DFTC}$ requires $n-1 \times (\Delta_p + 3)$ states per processor. In the worst case, the stabilization time of Algorithm $f\mathcal{DFTC}$ is $(4 \times n) + 1$ rounds.*

Note that Theorem 7 shows the stabilization time for the DFTC problem. However, if Algorithm $f\mathcal{DFTC}$ is used to implement the mutual exclusion in an arbitrary graph, from Corollary 3, the mutual exclusion property is achieved when no worm exists, i.e., in at most $n-1$ rounds.

6 Conclusion

We presented a simple self-stabilizing DFTC algorithm for general rooted networks. The proposed algorithm stabilizes faster than any algorithm for general networks in the current literature. The stabilization time is $O(n)$. So, the stabilization time is similar to the time for the token to complete one DFTC cycle.

Acknowledgments

We are grateful to Vincent Villain and Ajoy Datta for the valuable discussions. Also, we would like to thank the anonymous reviewers whose comments greatly helped improve the presentation of the paper.

References

1. A Arora and MG Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:1026–1038, 1994.
2. B Awerbuch, S Kutten, Y Mansour, B Patt-Shamir, and G Varghese. Time optimal self-stabilizing synchronization. In *STOC'93 Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 652–661, 1993.
3. GM Brown, MG Gouda, and CL Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38:845–852, 1989.
4. JE Burns and J Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11:330–344, 1989.
5. NS Chen, HP Yu, and ST Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.
6. Z Collin and S Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49:297–301, 1994.
7. AK Datta, C Johnen, F Petit, and V Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distributed Computing*, 13(4):207–218, 2000.
8. EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.
9. S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.
10. S Dolev, A Israeli, and S Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997.
11. S Ghosh. An alternative solution to a problem on self-stabilization. *ACM Transactions on Programming Languages and Systems*, 15:735–742, 1993.
12. MG Gouda and FF Haddix. The stabilizing token ring in three bits. *Journal of Parallel and Distributed Computing*, 35:43–48, 1996.
13. ST Huang and NS Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7:61–66, 1993.
14. C Johnen, G Alari, J Beauquier, and AK Datta. Self-stabilizing depth-first token passing on rooted networks. In *WDAG'97 Distributed Algorithms 11th International Workshop Proceedings, Springer-Verlag LNCS:1320*, pages 260–274, 1997.
15. C Johnen and J Beauquier. Space-efficient distributed self-stabilizing depth-first token circulation. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 4.1–4.15, 1995.
16. F Petit. Highly space-efficient self-stabilizing depth-first token circulation for trees. In *OPODIS'97, International Conference On Principles Of Distributed Systems Proceedings*, pages 221–235, 1997.
17. F Petit and V Villain. Color optimal self-stabilizing depth-first token circulation. In *I-SPAN'97, Third International Symposium on Parallel Architectures, Algorithms and Networks Proceedings*, pages 317–323. IEEE Computer Society Press, 1997.
18. F Petit and V Villain. Time and space optimality of distributed depth-first token circulation algorithms. In *Proceedings of DIMACS Workshop on Distributed Data and Structures*, pages 91–106. Carleton University Press, 1999.
19. F Petit and V Villain. Optimality and self-stabilization in rooted tree networks. *Parallel Processing Letters*, 10(1):3–14, 2000.
20. V Villain. A key tool for optimality in the state model. In *Proceedings of DIMACS Workshop on Distributed Data and Structures*, pages 133–148. Carleton University Press, 1999.

On a Space-Optimal Distributed Traversal Algorithm

Sébastien Tixeuil

Laboratoire de Recherche en Informatique
UMR CNRS 8623, Université de Paris Sud, France
tixeuil@lri.fr

Abstract. A traversal algorithm is a systematic procedure for exploring a graph by examining all of its vertices and edges. A traversal is Eulerian if every edge is examined exactly once. We present a simple deterministic distributed algorithm for the Eulerian traversal problem that is space-optimal: each node has exactly d states, where d is the outgoing degree of the node, yet may require $O(m^2)$ message exchanges before it performs an Eulerian traversal, where m is the total number of edges in the network. In addition, our solution has failure tolerance properties: *(i)* messages that are exchanged may have their contents corrupted during the execution of the algorithm, and *(ii)* the initial state of the nodes may be arbitrary.
Then we discuss applications of this algorithm in the context of self-stabilizing virtual circuit construction and cut-through routing. Self-stabilization [8,9] guarantees that a system eventually satisfies its specification, regardless of the initial configuration of the system. In the cut-through routing scheme, a message must be forwarded by intermediate nodes before it has been received in its entirety. We propose a transformation of our algorithm by means of randomization so that the resulting protocol is self-stabilizing for the virtual circuit construction specification. Unlike several previous self-stabilizing virtual circuit construction algorithms, our approach has a small memory footprint, does not require central preprocessing or identifiers, and is compatible with cut-through routing.

1 Introduction

Traversal. A *traversal* algorithm is a systematic procedure for exploring a graph by examining all of its vertices and edges. Traversal algorithms are typically used to explore unknown graphs (see [7]) and build a map as the graph is visited. The case of Eulerian connected directed graphs (where every node has as many incoming edges as outgoing edges) offers best performance since a traversal may be performed by visiting each link exactly once (it is then an *Eulerian traversal*). In [7], a centralized algorithm is proposed that traverses an Eulerian graph by visiting at most $2m$ edges, where m is the overall number of directed edges; once the graph map has been built, one can easily use well-known centralized algorithms to compute an *Eulerian cycle* (a cycle that includes all edges exactly once), which in turn can be used to perform an Eulerian traversal. In [12], a distributed solution to the Eulerian cycle construction is given, that requires

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 216–228, 2001.

$2m$ message exchanges and $\Omega(d^2)$ memory states at each node, where d is the outgoing degree of the node.

Self-Stabilization. Robustness is one of the most important requirements of modern distributed systems. Various types of faults are likely to occur at various parts of the system. These systems go through the transient faults because they are exposed to constant change of their environment. One of the most inclusive approaches to fault tolerance in distributed systems is *self-stabilization* [8, 9]. Introduced by Dijkstra in [8], this technique guarantees that, regardless of the initial state, the system will eventually converge to the intended behavior or the set of *legitimate* states. Since most self-stabilizing fault-tolerant protocols are non-terminating, if the distributed system is subject to transient faults corrupting the internal node state but not its behavior, once faults cease, the protocols themselves guarantee to recover in a finite time to a safe state without the need of human intervention. This also means that the complicated task of initializing distributed systems is no longer needed, since self-stabilizing protocols regain correct behavior regardless of the initial state. Furthermore, note that in practice, the context in which we may apply self-stabilizing algorithms is fairly broad since the program code can be stored in a stable storage at each node so that it is always possible to reload the program after faults cease or after every fault detection.

Cut-Through Routing. The *cut-through* routing is used in many ring networks (including IBM *Token Ring* and *FDDI*). In this routing scheme, a node can start forwarding any portion of a message to the next node on the message's path before receiving the message in its entirety. If this message is the only traffic on the path, the total delay incurred by the message is bounded by the transmission time (calculated on the slowest link on the path) plus the propagation delay. So, the total message delay is proportional to the length of the message and to the number of links on the path. Some pieces of the same message may simultaneously be traveling on different links and some other pieces are stored at different nodes. As the first bit of the message is transmitted on the links on the message's routing path, the corresponding links are reserved, and the reservation of a link is released when the last bit of the message is transmitted on the link.

This approach removes the need of having a local memory of any node greater than the one required to store a bounded number of bits, and also reduces the message delay to a small (bounded by the buffer size of the node) value. As with the current processors, the time needed for sending/receiving bits to/from a communication medium is far greater than the time needed to perform the basic computational steps (such as integer calculations, tests, read/write from/to registers, etc.), we can assume that a given process can perform a limited number of steps between the receipt of two pieces of a message.

Our Contribution. First, we present a distributed algorithm that is state optimal relatively to the Eulerian traversal problem. At every node, exactly d memory states are needed, where d is the actual outgoing degree of the node. In addition, our algorithm makes very few assumptions about the system on which it

is run. For example, nodes need not have unique identifiers or a special distinguished leader. Node variables need not be properly initialized when the protocol is started. Moreover, our protocol remains behaving accordingly to the Eulerian traversal specification even when messages that are exchanged between nodes have their content arbitrarily corrupted from time to time, even during the execution of the algorithm. Still, when it is first started, our algorithm may exhibit a transient $O(m^2)$ time period (where m is the overall number of edges) during which it performs the first traversal of the network. That first traversal may not be Eulerian, but every subsequent traversal is and remains Eulerian.

Second, and hinted by the failure tolerance properties of our algorithm (message contents corruption, node memory initial corruption), we transform it into a self-stabilizing virtual circuit construction algorithm by means of randomization. Informally, there exist two kinds of self-stabilizing virtual circuit constructions in the literature. Some (as reported in [9]) assume bidirectional networks (which are a proper subset of Eulerian networks), and construct in a self-stabilizing way a spanning tree, then performs an Eulerian tour of this tree (which is trivially done). Others (*e.g.* [1,14]) assume only strongly connected networks (which are a proper superset of Eulerian networks), but either require some central preprocessing or unique node identifiers, have high memory footprint, and are not compatible with cut-through routing. In comparison, our virtual circuit construction only performs in directed Eulerian networks, yet does not require central preprocessing or identifiers, has small memory footprint, and is compatible with cut-through routing. Moreover, when used as a lower layer by some other algorithm using the composition scheme of *e.g.* [10], the so-constructed virtual ring permits a bijective mapping between nodes in the original Eulerian network and nodes in the virtual ring network. This bijection allows the upper layer algorithm to remain unchanged. Then, previously known self-stabilizing cut-through algorithms that perform on unidirectional rings (*e.g.* [2,6,17]) can now be run on Eulerian networks without any change in their code.

Overview. In section 2, we present the system model and definitions that will be used throughout the paper. In section 3, a distributed Eulerian traversal algorithm is presented, along with associated correctness and complexity results. Applications to self-stabilization and the cut-through routing are given in Section 4. Concluding remarks are provided in Section 5.

2 Model

A *processor* is a sequential deterministic machine that uses a local memory, a local algorithm and input/output capabilities. Such a processor executes its local algorithm, that modifies the state of the processor memory, and send/receive messages using the communication ports. An *unidirectional communication link* transmits messages from a processor o (for *origin*) to a processor d (for *destination*). The link is interacting with one input port of d and one output port of o. We assume that links do not loose, reorder or duplicate messages.

A *distributed system* is a 2-tuple $\mathcal{S} = (P, L)$ where P is the set of processors and L is the set of communication links. A distributed system is represented by a

directed graph whose nodes denote processors and whose directed edges denote communication links. The state of a processor can be reduced to the state of its local memory, the state of a communication link can be reduced to its contents, then the global system state, called a *configuration*, is the product of the states of memories of processors of P and of contents of communication links in L. The set of configurations is denoted by $\mathcal{C}$.

Our system is not fixed: it passes from a configuration to another when a processor executes an instruction of its local algorithm or when a communication link delivers a message to its destination. This sequence of reached configurations is called a *computation*, and is a maximal alternating sequence of configurations of $\mathcal{S}$ and actions. A computation is denoted by $C_1, a_1, C_2, a_2, \ldots$ and such that for any positive integer i, the transition from C_i to C_{i+1} is done through execution of action a_i. A *projection* of a computation $C_1, a_1, C_2, a_2, \ldots$ on some set of actions A is the sequence of actions $a_{i_1}, a_{i_2}, \ldots$ such that each action $a_{i_1}, a_{i_2}, \ldots$ is in A. A finite subsequence of a computation or projection is called a *factor*. Configuration C_1 is called the *initial configuration* of the computation. In the most general case, the specification of a problem is by enumerating computations that satisfy this problem. Formally, a *specification* is a set of computations. A computation E satisfies a specification $\mathcal{A}$ if it belongs to $\mathcal{A}$.

A self-stabilizing algorithm does not always satisfy its specification. However, it seeks to reach a configuration from which any computation will verify its specification. A set of configurations $B \subset \mathcal{C}$ is *closed* if for any $b \in B$, any possible computation of system $\mathcal{S}$ whose b is initial configuration only contains configurations in B. A set of configurations $B_2 \subset \mathcal{C}$ is an *attractor* for a set of configurations $B_1 \subset \mathcal{C}$ if for any $b \in B_1$ and any possible computation of $\mathcal{S}$ whose initial configuration is b, the computation contains a configuration of B_2. Then a system $\mathcal{S}$ is *self-stabilizing* for a specification $\mathcal{A}$ if there exists a non-empty set of configurations $\mathcal{L} \subset \mathcal{C}$ such that (**closure**) any computation of $\mathcal{S}$ whose initial configuration is in $\mathcal{L}$ satisfies $\mathcal{A}$ and, (**convergence**) $\mathcal{L}$ is an attractor for $\mathcal{C}$.

In this paper, we also use a weaker requirement than self-stabilization that we call node-stabilization. Informally, a system is node-stabilizing if it reaches a correct behavior independently of the initial state of the nodes, yet one may assume that the state of the communications links satisfies some global predicate. Then a system $\mathcal{S}$ is *node-stabilizing* for a specification $\mathcal{A}$ if there exists two non-empty sets of configurations $\mathcal{L} \subset \mathcal{C}$ and $\mathcal{N} \subset \mathcal{C}$ such that (**closure**) any computation of $\mathcal{S}$ whose initial configuration is in $\mathcal{L}$ satisfies $\mathcal{A}$, (**convergence**) $\mathcal{L}$ is an attractor for $\mathcal{N}$, and (**node independence**) all possible node states are in $\mathcal{N}$.

3 State-Optimal Distributed Eulerian Traversal

In this section, we present a distributed algorithm that stabilizes to an Eulerian traversal provided that it is executed starting from a configuration where a single message is present (either at a node or within a communication link). In the following, we call such a configuration a *singular configuration*.

While the time complexity of this algorithm is not optimal ($O(m^2)$ starting from the worst possible initial configuration, while [12] provides a $O(m)$ distributed algorithm, where m is the overall number of edges of the network),

it does have nice static (it is state optimal at every node) and dynamic (it is node-stabilizing) properties.

3.1 The Algorithm

We assume $\delta^-(i)$ and $\delta^+(i)$ denote respectively the incoming and outgoing degree of node i. As the communication graph is Eulerian, let $d_i = \delta^-(i) = \delta^+(i)$. Moreover, each processor P_i has a $Path_i$ variable, that takes values between 0 and $d_i - 1$. All operations on this variable are done modulo d_i. Algorithm 3.1, that is executed upon receipt of a message m, is the same for all processors in the system.

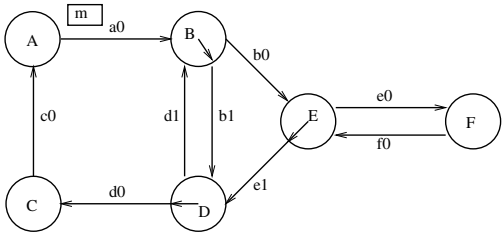
In this section, we assume that the communication between nodes is achieved through asynchronous message passing. In Section 4, we discuss the possibility of using Algorithm 3.1 in a synchronous or semi-synchronous system.

Distributed Eulerian traversal algorithm at node i
Send m using the outgoing link whose index is $Path_i$
$Path_i \leftarrow Path_i + 1$

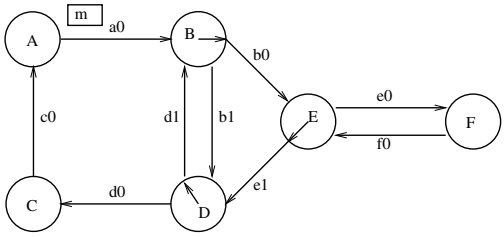
Example of Computation. Figure 1(a) presents a distributed system whose communication graph is Eulerian: processors A, C, and F each have one incoming link and one outgoing link; processors B, D and E each have two incoming links and two outgoing links. The $Path_i$ variable of each processor P_i is denoted by an arrow that points to the outgoing link on which the next message will be sent to. For example, processor A has just sent message m and processor B (which is about to receive m) will retransmit it through its outgoing link b_2. We now follow the path of message m from "initiator" A (actually the latest processor that transmitted m). Given the initial $Path_i$ variables configuration, m will go through links a_0, b_1, d_0 and c_0 before it returns to processor A. The followed path is obviously not Eulerian, since links b_0, d_1, e_0, e_1 and f_0 have not been visited by m. Nevertheless, variables $Path_B$ and $Path_D$ have changed their values during this round of message m: $Path_B$ now points to b_0 and $Path_D$ to d_1.

Figure 1(b) presents the same distributed system as Figure 1(a), but at the second round of message m. Given the configuration of the $Path_i$ variables, message m follows links a_0, b_0, e_0, d_1, b_1, d_0 then c_0 before returning to processor A. Again, the followed path is not Eulerian, since links e_1 and f_0 have not been followed by m. Yet, the $Path_E$ variable has changed value during this second round of m: it now point to e_1. At the contrary, variables $Path_B$ and $Path_D$ are back to the values they had at the beginning of second round (*i.e.* b_0 and d_1, respectively).

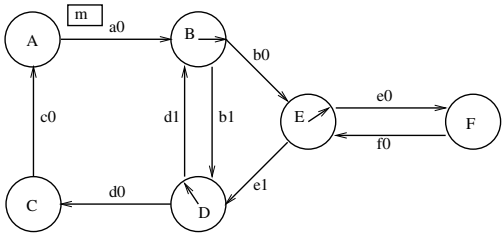
Figure 1(c) presents the same system at the beginning of third round. Given the $Path_i$ variables configuration, the message will follow links a_0, b_0, e_1, f_0, e_0, d_1, b_1, d_0 then c_0 before returning to processor A. The followed path is Eulerian, since every link is traversed exactly once, and that message m is back to the "initiator" A. Moreover, $Path_i$ variables hold the same values as at the



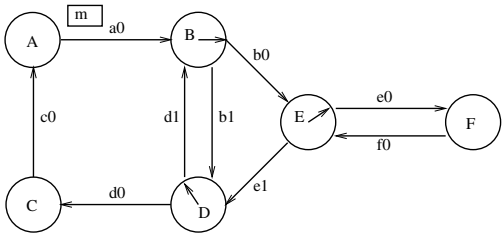
(a) The system at the beginning of first round



(b) The system at the beginning of second round



(c) The system at the beginning of third round



(d) The system at the beginning of fourth round

Fig. 1. Example of computation of Algorithm 3.1.

beginning of third round (see Figure 1(d)), which means that the fourth round will be identical to the third. Consequently, message m will follow the very same Eulerian path infinitely often.

3.2 Proof of Correctness

We wish to prove that Algorithm 3.1 is node-stabilizing for the Eulerian traversal problem. Each of the following lemmas assume that the algorithm is started from a singular configuration as defined below:

Definition 1. *A configuration C is* singular *if it contains exactly one message (either traversing a node or a communication link).*

The liveness lemma (Lemma 1) shows that Send actions through a particular link appear infinitely often in any computation, and so for any particular link. The following lemmas make use of it and consider computation factors that begin and end with the same Send action. The uniqueness lemma (Lemma 2) shows that between any two successive Send actions on the same link, no other link may be related to more than one Send action. The completeness lemma (Lemma 3) shows that after every link is related to at least one Send action, between any two successive Send action on a particular link, every other link is related to exactly one Send action. Finally, the legitimacy lemma (Lemma 4) shows that these Send actions appear always in the same order, and thus that the message performs an Eulerian traversal forever.

Lemma 1. *Starting from a singular configuration, every link is visited by the message infinitely often.*

Proof. Suppose there exists a link $c_{i\to j}$ (allowing P_i to send messages to P_j) that is not visited infinitely often starting from a singular configuration. From Algorithm 3.1, if processor P_i executes Send actions infinitely often, it does so on every outgoing link. Then, if P_i did not execute a Send action infinitely often on link $c_{i\to j}$, then P_i has executed Send action only a finite number of times in the whole computation, and thus P_i received the message only a finite number of times. Every incoming link P_i are then in the same case as $c_{i\to j}$, and have not been visited infinitely often. Applying the same reasoning again and since the network is finite and strongly connected, no link has been visited infinitely often. This contradicts the fact that Algorithm 3.1 may not deadlock starting from a singular configuration (since every receipt of a message implies an immediate Send action).

Notation 1. *For the sake of simplicity in the proof of the following lemmas, we arbitrarily number links from* 1 *to* L *(the number of links in the system) and denote by* l_j *a Send action through link number* j*. Thus* l_j *and* l_p *denote Send action on different links if* $j \neq p$ *and on the same link if* $j = p$*.*

Lemma 2. *Starting from a singular configuration, between any two Send actions on the same link, no other link is associated with a Send action twice.*

Proof. Let us consider a particular computation e of Algorithm 3.1 and its projection p on Send actions. From Lemma 1, action l_1 appears infinitely often in p. Let us study a factor f of p that starts and ends with l_1 and such that f does not contain any other l_1.

We do not discuss the trivial case $f = l_1 l_1$ where no other Send action that on the unique link is possible (in this case, the Eulerian traversal is trivially satisfied). Now assume that between the two l_1 actions, f contains some action l_k twice: $f = l_1 l_2 \dots l_k \dots l_n l_k \dots l_1$. In more details, while the n first actions of f are pairwise distinct, the second l_k is the first action that appears twice in f. We will show that the existence of l_k leads to a contradiction.

Action l_k is a Send action on link $c_{i\to j}$. If P_i performed twice a Send action involving $c_{i\to j}$ (the two occurrences of l_k), then P_i received the message $\delta^+(i)+1$

times. In turn, if P_i received $\delta^+(i)+1$ times the message, and since the graph is Eulerian, P_i received it $\delta^-(i)+1$ times and thus twice from the same incoming link. Then it follows that two Send actions occurred on this incoming link. In the writing of $f = l_1 l_2 \dots l_k \dots l_n l_k \dots l_1$, this means that some two actions between l_1 and l_n are identical, while our hypothesis claims that they are pairwise distinct. Therefore every factor f of p of length $n+1$ that starts and ends with l_1 can be written as $f = l_1 \dots l_p l_1$.

Lemma 3. *Starting from a singular configuration, and after every link has been visited by the message, between any two Send actions on the same link, every other link is associated with a Send action exactly once.*

Proof. Let us consider a computation of the algorithm. From Lemma 1, this computation contains each Send action infinitely often. It is then possible to write its projection p on Send actions as: $t_0 l_1 t_1 l_1 t_2 \dots l_1 t_n l_1 \dots$ where t_0 contains at least once each of the $l_{k\in\{1,\dots,L\}}$ and where none of the $t_{i\geq 1}$ contains l_1. From Lemma 2, it is impossible that any of the $t_{i\geq 1}$ contains the same Send action $l_{k\neq 1}$ twice. Therefore, all factors $t_{i\geq 1}$ are of length at most $L-1$.

Suppose now that the factor t_j $(j \geq 1)$ is of length strictly lower than $L-1$ and let us denote by l_p $(p \neq 1)$ the Send action that does not appear in t_j. Lemma 1 ensures that l_p appears infinitely often in the computation. Thus there exists a smallest $k > j$ such that l_p is a send action of factor t_k. Moreover, since l_p appears by definition in t_0, there also exists a greatest $m < j$ such that l_p is a Send action of factor t_m.

Consequently, the projection p has a factor $t_m l_1 t_{m+1} \dots t_j \dots t_{k-1} l_1 t_k$ where l_p $(p \neq 1)$ does not appear in any of the $t_{i\in\{m+1,\dots,k-1\}}$ but appears in t_m and in t_k:

$$\underbrace{\dots l_p \dots}_{t_m} \overbrace{l_1 t_{m+1} \dots t_j \dots t_{k-1} l_1}^{\text{no } l_p} \underbrace{\dots l_p \dots}_{t_k}$$

The Send action l_1 then appears twice between two successive Send actions l_p, which contradicts Lemma 2.

In conclusion, in the projection factor $t_0 l_1 t_1 l_1 t_2 \dots l_1 t_n l_1 \dots$, every $t_{i\geq 1}$ contains only different Send actions and is of size $L-1$. In other terms, after every link has been visited by the message (*i.e.* after t_0), between any two Send actions on the same link l_1, every other link is associated with a Send action exactly once (every $t_{i\geq 1}$ contains each $l_{k\in\{2,\dots,L\}}$ exactly once).

Lemma 4. *Starting from a singular configuration, and after every link has been visited by the message, between any two Send actions on the same link, every other link is associated with a Send action exactly once and in the same order.*

Proof. Let us consider a computation of the algorithm. From Lemma 1, this computation contains each Send action infinitely often. It is then possible to write its projection p on Send actions as: $t_0 l_1 t_1 l_1 t_2 \dots l_1 t_n l_1 \dots$ where t_0 contains at least once each of the $l_{k\in\{1,\dots,L\}}$ and where (from Lemma 3) every $t_{i\geq 1}$ contains exactly once each of the $l_{k\in\{2,\dots,L\}}$. In more details, for $t_j = l_2 l_3 \dots l_L$ we

can write t_{j+1} as $l_{\sigma(2)}l_{\sigma(3)}\dots l_{\sigma(L)}$, where σ is a permutation. Assume that there exists a smallest integer $q_1$in $\{2,\dots,L\}$ and such that $\sigma(q_1)$ /=q_1. Then there exists an integer q_2 ($q_1 < q_2 \leq L$) and such that $\sigma(q_1) = q_2$.

We are now able to rewrite a factor of the projection p as:

$$\underbrace{l_2 \dots l_{q_1-1} l_{q_1} l_{q_1+1} \dots l_{q_2-1} l_{q_2} \dots l_L}_{t_j} l_1 \underbrace{l_2 \dots l_{q_1-1} l_{q_2} l_{\sigma(q_1+1)} \dots l_{\sigma(q_2-1)} l_{q_1} \dots l_{\sigma(L)}}_{t_{j+1}} l_1$$

Then, between two l_{q_1}, Lemma 2 is contradicted. Indeed, between two successive occurrences of l_{q_1}, we find two occurrences of l_{q_2}. This contradiction permits to prove that every possible permutation σ is reduced to the identity and that the projection p can be written as $t_0(l_1 t_1)^\omega$. After every link has been visited by the message head (after t_0), between any two Send actions on the same link l_1, every other link is associated with a Send action exactly once and in the same order as in t_1.

Theorem 1. *Starting from an singular configuration, Algorithm 3.1 stabilizes to an Eulerian traversal.*

Proof. In Lemma 4, we proved that any computation has a factor of the projection p on Send actions of the form $t_0(l_1 l_2 \dots l_L)^\omega$, where t_0 is finite. Consequently, an Eulerian traversal through links 1 to L is performed infinitely often after a finite number of message exchanges.

3.3 Complexity

In order to know the outgoing link to which a message is to be sent, a processor P_i requires d_i states. Similarly, to know the incoming link by which a message was receipt, P_i requires d_i states. We show that a d_i states memory per processor is necessary for distributed Eulerian traversal.

In this section, we do not consider the space needed to manage the lower layer data link protocol. Our lower bound is for the size of the "routing table" that is needed at each node to perform an Eulerian traversal. We also assume that messages do not carry meaningful information that could be used to route them properly.

Lemma 5. *Every distributed Eulerian traversal algorithm requires d_i states at processor P_i.*

Proof. Suppose that there exists an Eulerian traversal algorithm (deterministic or probabilistic, stabilizing or non-stabilizing) such that there exists at least one processor P_i that uses at most $d_i - 1$ states. In an Eulerian network, every processor P_i has at least one incoming and one outgoing edges. Thus for any P_i, $d_i \geq 1$. If $d_i = 1$, then if P_i has less than one state, it may execute no code. Now, if $d_i \geq 2$, assume that P_i has at most $d_i - 1$ states.

In this last case, there exists at least two incoming links c_1 and c_2 of P_i by which P_i received the message and such that P_i had to perform a Send action using the same outgoing link c_3 at least twice (in case P_i's algorithm is

deterministic) or at least twice with probability $\epsilon > 0$ (in case P_i's algorithm is probabilistic). Then the message forwarding scheme is not Eulerian, since at any point in the computation, it is either certain or possible that two incoming links are not forwarded to two different outgoing links.

A direct corollary of this lemma is the following theorem.

Theorem 2. *Algorithm 3.1 is state optimal.*

For the time complexity part, a direct consequence of Lemma 2 is the following theorem.

Theorem 3. *Algorithm 3.1 performs its first Eulerian traversal within* $O(m^2)$ *message exchanges.*

4 Applications

In this section, we investigate applications of Algorithm 3.1 in the context of self-stabilization. Strictly speaking, Algorithm 3.1 is not self-stabilizing, since its correct behavior requires that it is started from a singular configuration (a configuration where a single message is present). However, it does stabilizes to an Eulerian traversal independently of nodes' initial states and messages' actual contents. Randomization allows Algorithm 3.1 to be self-stabilizing without the overkill of using a self-stabilizing mutual exclusion algorithm to guarantee uniqueness of the message; then the resulting self-stabilizing virtual circuit construction algorithm shows some interest particularly in the context of cut-through routing, where nodes must retransmit messages before they have been received entirely. Due to space constraints, all proofs in this section are only informally sketched, yet the interested reader may refer to [16].

4.1 Reaching a Singular Configuration

Since [11] showed that message-passing self-stabilizing algorithms require timeouts to handle the case where no message is initially present in the network, we concentrate here on eliminating superfluous messages in the case where the number of initially present messages is greater or equal to 2. Our solution is by giving different randomized speeds to messages so that in an infinite computation, it is possible that two messages are present at the same node at the same time. The result of that event is the node discarding every message but one.

Multiple Speeds. If the system is asynchronous (the communication time between the origin and the destination of a link may be arbitrary), we assume a random distribution on communication time, so that the speeds of the messages are actually different.

If the system is synchronous (the communication time between the origin and the destination of a link is bounded by 1), we can split every computation in *global steps* during which every message is sent and received, and assume that nodes dispose of a random Boolean variable. At each global step, every node that

receives a message consults its random variable: if the random variable returns *true*, then the node holds the message one more global step; otherwise it sends the message immediately. Note that a node may hold a message for at most one global step, and that the induced relative speeds on messages are now different and randomized. This technique has a low memory footprint since a node needs only to know if it should wait one more global step (one bit is sufficient).

Message Decreasing. Now assuming that messages do have different speeds, we show that Algorithm 3.1 stabilizes to a single message configuration. Indeed, starting from an arbitrary configuration with at least two messages, two cases may appear:

1. At least two messages follow the same circuit that goes through all links in the Eulerian graph. By the probabilistic setting, these two messages have different speeds and thus starting from any configuration, there is a positive probability that they are at the same node, which will discard all messages but one, so that the overall number of messages decreases.
2. At least two messages follow two different circuits, but from the rotating exploration nature of Algorithm 3.1, every edge of the system is visited infinitely often. Thus, these two circuits share a common node i. Then, from the probabilistic speeds of these two messages, starting from any configuration, there is a positive probability that at some point in the computation, they are at the same node, which will discard all messages but one. Then the overall number of messages decreases.

Since at any time there is a positive probability that, in a finite number of steps, the number of messages decreases if it is strictly greater that 1, then by the main theorem of [4], after finite time a single message remains in the system with probability 1.

4.2 Towards a Virtual Ring Construction

Since self-stabilization was first presented by Dijkstra in 1974 (see [8]), which provided three mutual exclusion algorithm on unidirectional ring networks, numerous works in self-stabilization were proposed on unidirectional rings (see [13] or [9]). Therefore, it is interesting to provide a scheme that permits to run such algorithms on more general networks, by constructing a *virtual ring* topology on top of which the original algorithm is run.

Many self-stabilizing solutions to the virtual ring construction problem exists for bidirectional networks (which are a proper subset of Eulerian networks); many of these works first construct a spanning tree of the graph, then perform an Eulerian tour of the spanning tree (see [9]). In general strongly connected networks (which are a proper superset of Eulerian networks), approaches by Tchuente ([14]) and Alstein *et al.* ([1]) make use of a central preprocessing of the communication graph, or assume that nodes are given unique identifiers. Two drawbacks of [1, 14] are the high memory consumption and the fact that nodes simulate several processes in the virtual ring, which may be incorrect for some applications (such as [2]). In addition, and to the best of our knowledge,

none of the aforementioned approaches can be used in non-ring networks when the cut-though routing scheme is used.

Our solution circumvents many of the previously mentioned drawbacks: the class of Eulerian graphs that we consider is intermediate between bidirectional and strongly connected graphs classes, we do not require central preprocessing nor unique node identifiers, memory consumption is low ($O(d)$ at each node, where d is the outgoing degree of the node), and efficient cut-through routing is supported.

Cut-Through Routing Compliance. The two main reasons for the cut-through routing compliance are the following: *(i)* since the underlying graph is Eulerian, each node has as many incoming links as outgoing links, so when a message arrives, it may be forwarded immediately to a free outgoing link, and *(ii)* since the message contents is unused in the forwarding scheme, no additional processing is needed before giving control to the composed cut-through algorithm (that could be any of [2, 6, 17]).

Virtual Circuit Bijection. In addition, the Eulerian property of the traversal guarantees that each link is visited exactly once at each traversal. Thus, if a node has d outgoing links, then the link that is locally labeled 0 at this node is visited exactly once at each Eulerian traversal, no matter how the local labeling on outgoing links is performed. Assume now that our Eulerian traversal algorithm is run to build a virtual circuit that is used by an upper layer application (such as [2]). If the upper layer application is activated only when a message arrives and the $Path_i$ variable equals 0, then it is guaranteed that this upper application is activated exactly once at each Eulerian traversal. This means that at the upper application level, there is a bijection between the nodes in the actual system and the nodes in the virtual ring system. This bijection is usually required for sake of correctness or service time guarantee.

5 Concluding Remarks

We presented a state-optimal distributed solution to the Eulerian traversal problem. Each node only needs d memory states, where d is the node outgoing degree. Our algorithm also presents some failure resilience properties: it is independent of the message contents and after $O(m^2)$ message exchanges, it provides an infinite Eulerian traversal whatever the initial configuration of the nodes may be (it is node-stabilizing). As such, Algorithm 3.1 proved useful to ensure mutual exclusion in uniform networks (see [3]) and mobile agent traversal for sake of self-stabilization (see [5]).

The message content independence was shown useful in the context of cut-through routing, since a node need not know the contents of a message to properly route it. The insensitivity to node initialization was extended by means of randomization so that the resulting system is self-stabilizing. This solution permits to avoid high memory consumption and preprocessing that were required by previous approaches.

Acknowledgements

I am grateful to Sylvie Delaët for her help while preparing this paper, to the attendees of [15] for their comments on a preliminary version of this work, and to the anonymous referees that helped to improve the presentation.

References

1. D. Alstein, J. H. Hoepman, B.E. Olivier , and P.I.A. van der Put. Self-stabilizing mutual exclusion on directed graphs. Technical Report CS-R9513, CWI, 1994. Published in *Computer Science in the Netherlands* (CSN 94), pp. 45–53.
2. J. Beauquier, A.K. Datta, and S. Tixeuil. Self-stabilizing Census with Cut-through Constraints. In Proceedings of the *Fourth Workshop on Self-stabilizing Systems* (WSS'99), Austin, Texas. pp. 70-77, May 1999.
3. J. Beauquier, J. Duran-Lose, M. Gradinariu, and C. Johnen. Token Based Self-Stabilizing Uniform Algorithms. LRI Research Report no. 1250, March, 2000.
4. J. Beauquier, M. Gradinariu, and C. Johnen. Memory Space Requirements for Self-stabilizing Leader Election Protocols. In Proceedings of the International Conference on *Principles of Distributed Computing* (PODC'99), Atlanta, pp. 199-208, 1999.
5. J. Beauquier, T. Herault, and E. Schiller. Easy Self-stabilization with an Agent. LRI Research Report no. 1280, April, 2001.
6. A.M. Costello and G. Varghese. The FDDI MAC meets self-stabilization. In Proceedings of the *Fourth Workshop on Self-stabilizing Systems (WSS'99)*, Austin, Texas. pp. 1-9, May 1999.
7. X. Deng and C.H. Papadimitriou. Exploring an unkown graph. In Proceedings of the 31^{th} Annual IEEE Symposium on *Foundations of Computer Science*, Vol. I, pp. 355-361, 1990.
8. E.W. Dijkstra. Self-stabilization in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643-644, 1974.
9. S. Dolev. Self-stabilization. *The MIT Press*. 2000.
10. M. G. Gouda and T. Herman. Adaptive programming. *IEEE Transactions on Software Engineering*, 17:911-921, 1991.
11. M.G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448-458, 1991.
12. M. Hadim M. and A. Bouabdallah. A distributed algorithm for constructing an Eulerian cycle in networks. In Proceedings of *International Conference on Parallel and Distributed Processing Techniques and Applications* (PDPTA'99), Las Vegas, USA, June 28^{th}-July 1^{st}, 1999.
13. T. Herman. A Comprehensive Bibliography on Self-Stabilization. A Working Paper in the *Chicago Journal of Theoretical Computer Science*. Available at `http://www.cs.uiowa.edu/ftp/selfstab/bibliography/`.
14. M. Tchuente. Sur l'auto-stabilisation dans un réseau d'ordinateurs. *RAIRO Informatique Théorique*, 15:47-66, 1981.
15. S. Tixeuil. Algorithmes Auto-stabilisants à Délai Borné. *Paris-Orsay-Cachan Seminar*, Dec. 17, 1996.
16. S. Tixeuil. Auto-stabilisation Efficace. *Ph.D. Thesis, Université de Paris Sud*, France. Jan. 2000. Available at `http://www.lri.fr/~tixeuil`.
17. S. Tixeuil and J. Beauquier. Self-stabilizing Token Ring. In Proceedings of *International Conference on System Engineering* (ICSE'96), Las Vegas, Nevada. Jul. 1996.

Author Index