Distributed Algorithms

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

486

J. van Leeuwen N. Santoro (Eds.)

Distributed Algorithms

4th International Workshop
Bari, Italy, September 24–26, 1990
Proceedings

Springer-Verlag
Berlin Heidelberg New York London Paris
Tokyo Hong Kong Barcelona Budapest

Volume Editors
Jan van Leeuwen
Department of Computer Science, University of Utrecht
Padualaan 14, P.O. Box 80.089
3508 TB Utrecht, The Netherlands

Nicola Santoro
School of Computer Science, Carleton University
Ottawa, Canada K1S 5B6

CR Subject Classification (1991): C.2.2, C.2.4, D.4.1, D.4.4–5, F.1.1, F.2.2

ISBN 3-540-54099-7 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-54099-7 Springer-Verlag New York Berlin Heidelberg

Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
2145/3140-543210 – Printed on acid-free paper

Preface

The 4th International Workshop on Distributed Algorithms (WDAG 90) was organized by the Istituto di Scienze dell' Informazione of the University of Bari (Italy) and held in Serra Alimini (near Otranto), some 200 kilometers southeast of Bari along the beautiful Adriatic coast, September 24-26, 1990. The workshop was intended as a forum for researchers, students and other interested persons to discuss the recent results and trends in the design and analysis of distributed algorithms for communication networks (i.e., on graphs) and decentralized systems. The earlier workshops on Distributed Algorithms were held in Ottawa (1985, proceedings published by Carleton University Press), Amsterdam (1987, see Lecture Notes in Computer Science, Vol. 312), and Nice (1989, see Lecture Notes in Computer Science, Vol. 392).

Papers were solicited describing original results in all areas of distributed algorithms and their applications including, e.g. distributed combinatorial algorithms, distributed optimization algorithms, distributed algorithms on graphs, distributed algorithms for decentralized systems, distributed data structures, distributed algorithms for control and communication, design of network protocols, routing algorithms, fail-safe and fault-tolerant distributed algorithms, distributed database techniques, algorithms for transaction management, replica-control algorithms, and other related fields. The program committee for the workshop consisted of

D. Dolev	(IBM Almaden Research Center and Hebrew University),
F. Mattern	(University of Kaiserslautern),
N. Santoro	(Carleton University, Ottawa), *co-chairman*
P. Spirakis	(Computer Technology Institute, Patras, and New York University),
R.B. Tan	(University of Oklahoma, Chickasha, and University of Utrecht),
S. Toueg	(Cornell University, Ithaca),
J. van Leeuwen	(University of Utrecht), *co-chairman*
P.M.B. Vitanyi	(CWI and University of Amsterdam),
S. Zaks	(Technion, Haifa).

Close to sixty papers were submitted out of which the program committee selected twenty-eight papers for presentation in the workshop. The selection reflects several current directions of research in the area of distributed algorithms, although certainly not all aspects of the field could be covered in the three-day workshop. The present volume contains the revised version of all papers presented in the workshop. The revised versions are based on the comments and suggestions received by the authors during and after the workshop. Several papers are in the form of preliminary reports on

continuing research. The papers in this volume should give a good impression of the current work in distributed algorithms and stimulate further research. The sessions at the workshop were chaired by: Friedemann Mattern, Nicola Santoro, Paul Spirakis, Richard Tan, Sam Toueg/Gil Neiger, Jan van Leeuwen, Paul Vitanyi, and Shmuel Zaks.

We are grateful to the Centre for Parallel and Distributed Computing (*PARADISE*) of Carleton University (Ottawa, Canada) and the Department of Computer Science of the University of Utrecht (Utrecht, the Netherlands) for supporting the organization of the workshop, and to the Centro Internazionale Congressi S.r.l. and its staff for the superb local arrangements. The workshop was partly supported by IBM (Italy), Siemens (Italy) and by the Banca Vallone/Gruppo Ambrosiano. We also thank Annerie Deckers (Department of Computer Science, University of Utrecht) for her invaluable assistance during the preparation of the workshop.

Utrecht and Ottawa/Bari
March 1991

Jan van Leeuwen and Nicola Santoro
Co-Chairmen WDAG 90

CONTENTS

Self-Stabilizing Ring Orientation

Amos Israeli *
Dept. of Electrical Engineering
Technion — Israel

Marc Jalfon ‡
Dept. of Computer Science
Technion — Israel
and Intel — Haifa, Israel

Abstract

A *self-stabilizing* system is a distributed system which can be started in any *possible* global state. Once started the system regains its consistency by itself, without any kind of an outside intervention. A *ring* is a distributed system in which all processors are connected in a ring. A ring is *oriented* if all processors in the ring agree on common right and left directions. A protocol is *uniform* if all processors use the same program.

In this paper we answer the following question: *Does a uniform self stabilizing protocol for ring orientation exist?* We begin the presentation by answering this question negatively for deterministic protocols. Then we present a randomized uniform self stabilizing protocol for *ring orientation*. When the protocol stabilizes all processors agree upon a "right" (privileged) direction. The protocol works for a ring of any size and even tolerates dynamic additions and removals of processors as long as the ring topology is preserved. The number of states of each processor is $O(1)$, and its stabilization time is $O(n^2)$, where n is the number of processors in the system.

1 Introduction

A *self-stabilizing* system is a distributed system which can be started in any *possible* global state. Once started the system regains its consistency by itself, without any kind of an outside intervention. Two advantages of self stabilizing systems are:

- Self stabilizing systems need not be initialized globally. Each component can be started separately and in an arbitrary state. The system will self-stabilize into a legitimate configuration.
- The self stabilization property makes the system tolerant to *transient bugs*, bugs in which the state of a component is changed spontaneously while the component is still correct.

A *ring* is a distributed system in which all processors are connected in a circle. A ring is *oriented* if all processors in the ring agree on common right and left directions. A protocol is *uniform* if all processors use the same program. A common problem in distributed protocols is symmetry breaking.

*Partially supported by Technion VPR Funds - Japan TS Research Fund and B. & G. Greenberg Research Fund (Ottawa).
‡Partially supported by a Gutwirth fellowship.

Almost all known self-stabilizing protocols for rings, e.g. [Di-74], [BGW-87], [Bu-87], present self stabilizing protocols for mutual exclusion. In order to avoid dealing with symmetry breaking they make two strong assumptions:

non-uniformity: There is a single "special" processor whose program is different from the program of the rest of the processors.

orientation: The ring is oriented.

The work of of [DIM-89] introduces the use of *registers* for communication between processors. Using registers they design a non uniform self stabilizing protocol for mutual exclusion for general graphs and in particular, for unoriented rings. In [Di-74], Dijkstra has observed that no deterministic uniform self stabilizing protocol for mutual exclusion for rings exists. In [IJ-89] we use randomization to break symmetry and present a randomized uniform self stabilizing mutual exclusion protocol for oriented rings. Thus orientation can be given up in the presence of non-uniformity; while for randomized protocols non-uniformity can be given up in the presence of orientation. It is natural to ask whether both assumptions can be dispensed simultaneously.

In this paper we answer the following question: *Does a uniform self stabilizing protocol for ring orientation exist?* We begin the presentation by answering this question negatively for deterministic protocols. Then we present a randomized self stabilizing uniform protocol for *ring orientation.* When the protocol stabilizes, all processors agree upon a "right" (privileged) direction. The protocol works for a ring of any size and even tolerates dynamic additions and removals of processors as long as the ring topology is preserved. The number of states of each processor is $O(1)$. The expected stabilization time is $O(n^2)$, where n is the number of processors in the system.

The protocol is composed of two "levels", each level consists of a self stabilizing protocol. On the lower level all edges of the ring are directed, not necessarily in a consistent way. Each edge is directed separately by a *randomized* protocol which is run by the processors at its endpoints. The higher level of the protocol is a *deterministic* protocol for orientation of a directed ring (that is a ring whose edges are directed). The final protocol is obtained by using the technique of fair combination of self stabilizing protocols due to [DIM-89].

Recently the works of [BP-88] and [IJ-89] presented uniform self stabilizing protocols for oriented rings. The protocol of [BP-88] works on prime rings in which no symmetric global states exist. The work of [IJ-89] uses *randomization* to break symmetry. The present protocol enables execution of any uniform self stabilizing protocol for oriented rings on any ring. This work together with [IJ-89] are the first to introduce randomization to self stabilizing protocols in which processors communicate using shared memory. A randomized self stabilizing version of the alternating bit protocol appears in [AB-89]. The protocol of [AB-89] uses message passing for communication.

The rest of this paper is organized as follows: in Section 2 the formal model and requirements for self-stabilization and ring orientation are presented. Some impossibility results are brought in Section 3. In Section 4 we present a uniform self-stabilizing protocol for orienting a ring. Concluding remarks appear in Section 5.

2 Model and Requirements

A *uniform ring* consists of n *identical processors*, denoted by $P_0, P_1, ..., P_{n-1}$. Each processor is a *randomized finite state machine.* Processors are *anonymous*, they do not have identities. The subscripts $0, 1, \ldots, n-1$ are used for ease of notation only. Each processor resides on a node of the system's *communication ring.* Two processors residing on neighboring nodes are called *neighbors.*

Neighbors communicate using *registers*. If e is an edge of the ring between processors P_i and P_j then e is realized by two registers r_{ij} in which P_i writes and from which P_j reads, and r_{ji} in which P_j writes and from which P_i reads. Registers in which a processor P writes are referred to as P's registers. When all edges of the ring are undirected (directed) we call this ring *undirected* (*directed*). The edges of the ring can be either directed or undirected. When an edge of the ring is directed the two processors at its endpoints agree on its direction, one is designated as the edge's "head" while the other is the edge's "tail". An undirected edge is symmetric with respect to its endpoints. In this work the direction of edges is used solely for symmetry breaking; communication is allowed in both directions of an edge regardless of whether it is directed or undirected.

A processor is a finite state machine with *state set* M and transition function δ. The arguments of δ are the state of the processor and the values it reads from the registers of its neighbors. Whenever a processor is activated it executes a single *atomic step* which is determined by the transition function δ. In a single atomic step a processor may read the registers of some of its neighbors, write in some of its registers and then move to its new state. The "size" of each step depends on the type of the adversary and will be discussed below. The transition function δ can be *randomized*; in this case it may enable more than one transition. The transition which is actually executed is chosen with equal probability. A *uniform protocol* is a triplet $< \mathcal{G}, M, \delta >$ where $\mathcal{G}$ is a family of communication graphs, M and δ are the state set and transition function, common to all processors, respectively. If the function δ is randomized the protocol is *randomized.*

The global state of a uniform n processor system is described by its configuration. A configuration of a system is an n-tuple $c = (q_0, q_1, \ldots, q_{n-1})$ where $q_i \in M$ for $0 \leq i \leq n-1$. A processor P is *enabled* in configuration c if $\delta(q, r_1, r_2, \ldots) \neq q$, where q is the state of P in c and $r_1, r_2, \ldots$ are the values read by P from the registers of its neighbors (for randomized protocols we require $\{\delta(q, r_1, r_2, \ldots)\} \neq \{q\}$). If P is not enabled, it is said to be *disabled.* A configuration c is a *deadlock* configuration if no processor is enabled in c.

The behavior of real life distributed systems is modeled by the interleaving model. Processor activity is managed by a *scheduler*. To ensure correctness of the systems, we regard the scheduler as an adversary. We let the scheduler choose its activated processors *on line* using processor states as its input. We do not allow the scheduler to use the results of *future* coin tosses, as this may nullify the extra strength added to the system by randomization. Whenever the adversary activates a processor, the processor executes a single atomic step. The more freedom the adversary has in choosing its activated processors and the smaller the atomic step is, the stronger the adversary is. We hereafter list the most common types of adversaries used in the literature:

(a) The weakest adversary activates processors in sequence, one after the other. Whenever a processor is activated it reads the registers of *all* its neighbors and then it moves to a new state while writing in *all* its registers. This adversary is known as *Central Demon.*

(b) A stronger adversary which is also known as *Distributed Demon* can activate any subset of the system's processors together. Whenever a set of processors is activated by the adversary, all activated processors simultaneously read all the registers of their neighbors. Subsequently all activated processors move to their new state while writing in all their registers.

(c) An even stronger adversary is the *Read/Write Demon.* In a single atomic step it activates a single processor which either reads from a single register, or writes into a single register (but not both).

An adversary is *proper* if it activates only enabled processors as long as the system is not in a deadlock configuration. In this paper we use the **proper distributed demon** as the adversary. An *execution* of a system is a list of configurations $c^1, c^2, \ldots$ where each configuration, c^{i+1}, is obtained

from the previous configuration, c^i, by a single activation of some set of processors. The list of subsets of processors activated by the scheduler constitutes a *schedule*.

We proceed by defining the self-stabilization requirements for randomized distributed systems. The set of all possible configurations is denoted by C. Let $L \subset C$ be a set of configurations of a given distributed system. L is called the set of *legitimate* configurations of the system. A system is *self-stabilizing* with respect to L if the following requirements are satisfied:

deadlock - Every deadlock configuration of the system is in L.

closure - For every $c \in L$, and for every $c' \in C$, if c' is reached from c by a single transition then $c' \in L$. (Once the system reaches a legitimate configuration, it will always remain in a legitimate configuration).

randomized no livelock - There is a function f from the natural numbers to the interval $[0,1]$ satisfying $\lim_{k\to\infty} f(k) = 0$ such that for every initial configuration and for every proper scheduler, the probability that the k-th configuration reached is in L is $1 - f(k)$.

Note: In this definition two common requirements are amended. The **no deadlock** requirement is replaced by the **deadlock** requirement which allows legitimate deadlock configurations. The (deterministic) **no livelock** requirement is replaced by the slightly relaxed **randomized no livelock** requirement.

In ring systems each processor has two neighbors. The processor can internally distinguish between a *first* neighbor and a *second* neighbor. Deciding which neighbor is the first and which one is the second is done by the hardware, the correctness and complexity of a protocol should not be affected by any possible assignment of neighbors as first and second. We arbitrarily choose to number processors clockwise (although in unoriented rings processors do not "know" what "clockwise" is). For any processor P_i, P_{i+1} (P_{i-1}) denotes the clockwise successor (predecessor) of P_i, whose subscript is actually $i+1 \pmod n$ ($i-1 \pmod n$).

An *orientation* of a state of a processor in a ring system is a choice of one of the neighbors of that processor. Let R be a ring, a self stabilizing protocol for R in which each processor has an orientation is an *orienting* protocol, if in each legitimate configuration all orientations form a directed cycle and if whenever a legitimate configuration is reached the chosen orientation never changes.

3 Impossibility Results for Ring Orientation

In this section we prove that for some uniform and nonuniform rings, no deterministic orienting protocol exists.

Lemma 1: Let R be a ring of even size in which all processors but one are identical. If each processor communicates with both its neighbors using a single register then R has no self stabilizing orienting protocol under central demon.

Proof: Let $2n$ be the size of the ring. Neighbor ordering is as follows: P_0 is the first neighbor of P_1, P_1 is the first neighbor of P_2,..., and P_{n-2} is the first neighbor of P_{n-1}. P_0 is the first neighbor of P_{2n-1}, P_{2n-1} is the first neighbor of P_{2n-2},..., and P_{n+2} is the first neighbor of P_{n+1} (see Fig 1). There is no requirement on the neighbor ordering of P_0 and P_n. Let c be an initial configuration such that:

$$c_1 = c_{2n-1}\ ,\ c_2 = c_{2n-2}\ ,\ \ldots,\ c_{n-1} = c_{n+1}$$

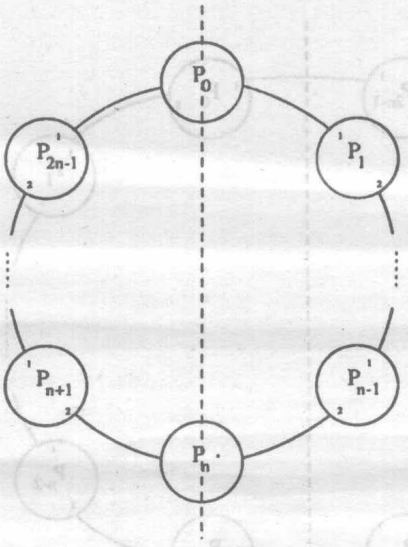

Figure 1: Symmetric neighbor ordering

Thus, we have: $c = c_0 w c_n w^r$, and c has a symmetry axis as shown in Fig 1. It is easy to see that a configuration having such symmetry axis cannot be legal, as edges (P_1, P_2) and (P_{2n-1}, P_{2n-1}) have opposite orientations. Note that for all $i \neq 0$, the state of P_i's first (second) neighbor equals the state of P_{2n-i}'s first (second) neighbor. Thus if we activate P_i and then P_{2n-i}, both will enter the same state, and the configuration will have the same symmetry axis as c. Clearly, activating P_0 or P_n does not break the symmetry. Thus, if the demon activates processors with the following schedule:

$$(P_0\, P_n\, P_1\, P_{2n-1}\, P_2\, P_{2n-2} \ldots P_{n-1}\, P_{n+1})^{\infty}$$

the system will never reach a legal configuration. □

Lemma 2: Let R be a ring of even size. The ring R has no uniform deterministic orienting protocol which is self stabilizing in the presence of distributed demon.

Proof: Let neighbor ordering be as follows: For all $0 < i \leq n$, P_i is the first neighbor of P_{i+1}, and for all $n < i \leq 2n$, P_i is the first neighbor of P_{i-1}. Let c be an initial configuration of the form $c = ww^r$; c has a symmetry axis as shown in Fig 2. The protocol is deterministic and all processors are identical, hence if P_i and P_{2n-i-1} are concurrently activated, then a configuration with the same symmetry will be reached, and therefore the following is a livelock schedule:

$$(\ \{P_0, P_{2n-1}\}\ \{P_1, P_{2n-2}\} \ldots \{P_n, P_{n+1}\}\ )^{\infty}$$

□

4 The Protocol

In this section we present a self stabilizing protocol for orientation of undirected rings, prove its correctness and analyze its complexity. The protocol is composed of two "levels". Each level consists

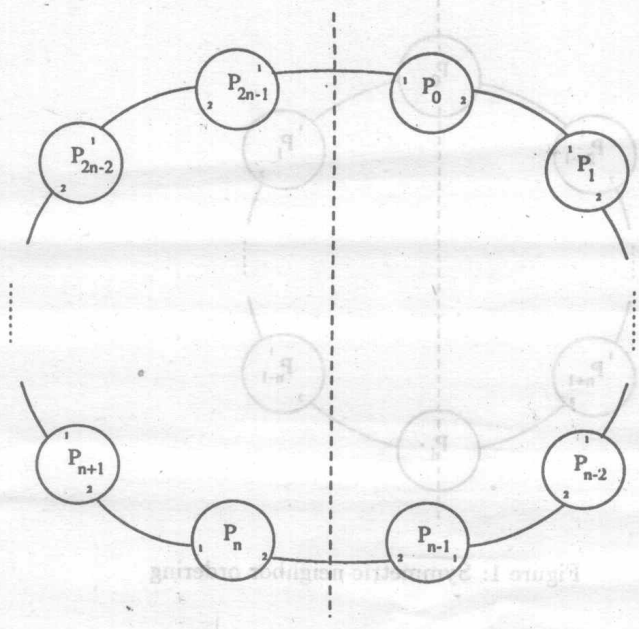

Figure 2: Symmetric neighbor ordering

of a self stabilizing protocol. The lower level directs all edges of the ring, where each edge is directed by a *randomized* protocol which is run by the processors on its endpoints. The higher level of the protocol is a *deterministic* protocol for orientation of a directed ring. The final protocol is obtained as a combination of the two by using the technique of fair combination of self stabilizing protocols presented in [DIM-89].

4.1 Directing a Ring

The n edges of a ring are directed by running n separate copies of a randomized edge directing protocol. The protocol for each edge is executed by its two endpoints. An edge e connecting processors P and Q is implemented by two registers: $P.dir$ which is written by P and read by Q, and $Q.dir$ which is written by Q and read by P. This situation is depicted in Figure 3.

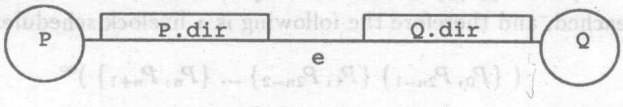

Figure 3: Two processors and edge registers between them

The state set of each processor is $M = Z_3$. The set of system configurations consists of all pairs of numbers from Z_3. We say that the edge e is directed from P to Q if $P.dir + 1 = Q.dir \pmod 3$. The set of legitimate configurations contains all pairs of the form: $\{ (P.dir, Q.dir) \mid P.dir \neq Q.dir\} = \{(0,1), (0,2), (1,0), (1,2), (2,0), (2,1)\}$. Both processors are enabled if the values of the dir fields are equal. When a processor is activated, it executes the following statement:

$$\textbf{if} \quad P.dir = Q.dir \quad \textbf{then} \quad P.dir = \text{random}(Z_3 - \{P.dir\})$$

Or in words "if your neighbor's value is the same as yours, then pick something else". Random(A) means choose at random (uniformly) an element from set A. In our protocol, we always have $|A| = 2$.

We now prove that the edge directing protocol is self stabilizing in the presence of a proper distributed demon. The **deadlock** and **closure** requirements are satisfied by the simple observation that all deadlock configurations are legitimate configurations and that all legitimate configurations are deadlock configurations. The **randomized no-livelock** requirement is proved by the following lemma.

Lemma 3:

(a) For any nonlegitimate configuration c, the probability that the system reaches a legitimate configuration within a single transition is $\geq 1/2$.

(b) For any nonlegitimate configuration c, the probability that the system does not reach a legitimate configuration within ℓ transitions is $\leq 2^{-\ell}$.

Proof: In a nonlegitimate configuration, $P.dir = Q.dir$, and both processors are enabled. If the adversary activates a single processor then this processor changes the value of its *dir* field, and a legitimate configuration is reached. If both processors are activated concurrently, they change the value of their *dir* field, and the system reaches either a legitimate configuration or another configuration in which $P.dir = Q.dir$ with equal probability. Thus, the probability that the ℓ-th configuration reached is not legitimate is $\leq 2^{-\ell}$. □

The expected stabilization time of the protocol is $\sum_{l=1}^{\infty} \ell 2^{-\ell} = 2$. To direct a ring of n processors (and n edges) we run n copies of the edge directing protocol, a copy for each edge. Each processor has two *dir* fields, a field for each of its neighbors (there is no requirement on the relative values of these two fields). The total expected number of activations before all edges are directed is $O(n)$.

4.2 The Orienting Protocol

We now introduce a deterministic protocol for orientation of directed rings. The protocol achieves its goal by using tokens. Tokens are not part of the model but an abstract concept which is helpful in protocol design and verification. Each token has a *direction* which is never changed. A token is passed from its holder to the neighbor of the holder which is in the token's direction. This neighbor in its turn passes the token to its other neighbor and so on. Each processor keeps trace of the direction to which the most recent token was passed. This direction constitutes the current orientation of the processor. Whenever two tokens meet one of them is eliminated. Stabilization is achieved when all remaining tokens have the same direction and each processor is visited by at least one of these remaining tokens.

The state of a processor in this protocol consists of two fields: *token* and *orient*. In each state of processor P the values of of these fields are stored in the registers of P. The *token* fields are used to enable token passing, their possible values are I, S or R, which stand for *Idle*, *Sending token* and *Receiving token*, respectively. The binary *orient* fields in P's registers give the current orientation for P. While the values stored in both *token* fields of P are always equal, the values stored in the two *orient* fields of P are never equal. We will usually represent fields *token* and *orient* by a single letter with an arrow over it, $\vec{S}\,\overleftarrow{I}$ representing the situation depicted in Figure 4.

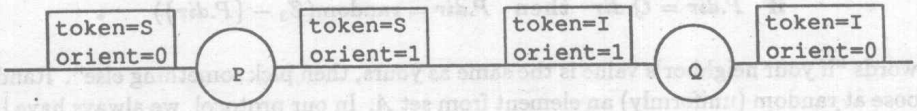

Figure 4: $\vec{S}\,\overleftarrow{I}$

The orienting protocol is defined by the transition table which appears in Figure 5. When a processor P is activated, it matches its state and the state of its first neighbor Q (which is read from Q's register) against each entry of the table. If a matching entry is found then P executes the transition associated with that entry. The transition specifies the new state of P which is written in its two registers, and may depend on the direction of the edge connecting P and Q. If no match is found, the matching is tried again with Q standing for P's second neighbor. Choosing to start the matching from the first neighbor is an arbitrary decision. The only possible multiple match is in transition 1, preferring the first neighbor over the second may affect the final orientation but does not harm the correctness of the protocol or its stabilization time.

transition #	Q	P		P
1)	$\vec{S}$	I	$\longrightarrow$	$\vec{R}$
2)	$\overleftarrow{R}$	$\overleftarrow{S}$	$\longrightarrow$	$\overleftarrow{I}$
3)	$\neg\,\vec{S}$	$\vec{R}$	$\longrightarrow$	$\vec{S}$
4)	$\vec{S}$	$\overleftarrow{S}$	$\longrightarrow$	if $e=(P,Q)$ is directed from P to Q **then** $\vec{R}$
5)	$\vec{I}$	$\overleftarrow{I}$	$\longrightarrow$	if $e=(P,Q)$ is directed from P to Q **then** $\overleftarrow{S}$

Figure 5: Transition table for the orientation protocol

In transition 1, I means "P is either in state $\vec{I}$ or in state $\overleftarrow{I}$". In transition 3, $\neg\,\vec{S}$ means "Q's state is anything but $\vec{S}$". The direction of the arrow is to be understood with respect to P. For example both situations depicted in Figure 6 match transition 2, but the one in Figure 7 does not. That is, transition 2 matches a situation in which processor P is either in state $\vec{S}$ or in state $\overleftarrow{S}$, Q is the neighbor the arrow points to, and Q is in state R with its arrow pointing away from P.

A processor P_i in state $\overleftarrow{S}$ or $\vec{S}$ *holds a token* whose direction is the arrow direction. If P_{i-1} is not in state $\vec{S}$ and P_i is in state $\vec{R}$ (P_{i+1} is not in state $\overleftarrow{S}$ and P_i is in state $\overleftarrow{R}$) then P_i also *holds a token*, whose direction is the arrow direction. Tokens never change their direction.

A token held by P_i makes a *step* whenever P_i *releases* it by executing transition 2. In this case the token is *passed* to the neighbor of P_i functioning as processor Q for that transition. This could be either P_{i-1} or P_{i+1}. A token is *created* whenever a processor becomes the holder of a token where the token was not passed to it by any of its neighbors. The only transition which involves token creation is transition 5. A token is *eliminated* whenever a processor releases a token which is not passed to any of its neighbors. A processor in state R can only enter state S, so an elimination can take place only if the holder of a token leaves state S (transition 4), or if the holder of a token is in state $\vec{R}$ ($\overleftarrow{R}$) and its left (right) neighbor enters state $\vec{S}$ ($\overleftarrow{S}$) (transition 3).

The set of legitimate configurations contains all configurations in which the ring is oriented, that is all configurations in which all arrows have the same direction:

$$L = (\vec{I} + \vec{S} + \vec{R})^n + (\overleftarrow{I} + \overleftarrow{S} + \overleftarrow{R})^n$$

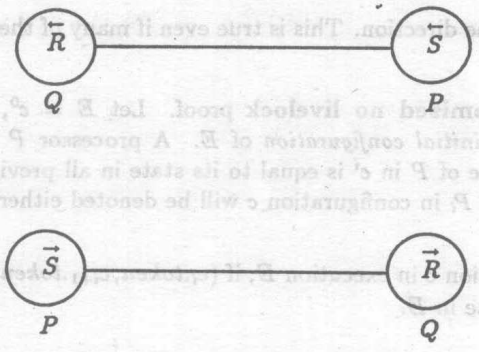

Figure 6: Two situations in which transition 2 is possible

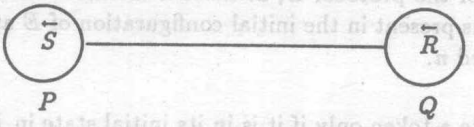

Figure 7: Transition 2 is not possible

We now prove that the orienting protocol is self stabilizing with respect to L and under a distributed demon.

Lemma 4: The **deadlock** requirement is satisfied by the protocol.

Proof: Let c be a configuration in which no processor is enabled. If any processor P is in state R, then either P or one of its neighbors is enabled (transitions 2 and 3). Otherwise, all processors are either in I or in S. Consider a maximal subvector of processors in state S in c. If there is a $\overrightarrow{S}\overleftarrow{S}$ pair in c, then one of these processors in enabled (transition 4). Otherwise the subvector is of the form $(\overleftarrow{S})^i(\overrightarrow{S})^j$. If $0 < i + j < n$, then at one end of the subvector we have a $\overrightarrow{S}\ I$ (or $I\ \overleftarrow{S}$) pair, and then the processor in state I is enabled (transition 1). Hence, either $i + j = 0$ or $i + j = n$, that is, either all processors are in state S, or they are all in state I. But neither $\overrightarrow{S}\overleftarrow{S}$ nor $\overrightarrow{I}\overleftarrow{I}$ can appear in c. Therefore, if c is a deadlock configuration, then

$$c \in \{(\overrightarrow{S})^n, (\overleftarrow{S})^n, (\overrightarrow{I})^n, (\overleftarrow{I})^n\} \subseteq L$$

□

Lemma 5: The **closure** requirement is satisfied by the protocol.

Proof: Let c be a legitimate configuration. If c is a deadlock configuration then we are done. Otherwise, the only possible transitions from c are transitions 1 , 2 and 3, because transitions 4 and 5 require processors to have arrows in different directions. It is easy to see from the protocol that if transitions 1 , 2 or 3 take place when all arrows have the same direction, then after the transition

all arrows will still have the same direction. This is true even if many of these transitions take place concurrently. □

We now turn to the **randomized no livelock** proof. Let $E = c^0, c^1, \ldots$ be an execution. Configuration c^0 is called the *initial configuration* of E. A processor P is in its *initial state* in configuration c^i of E if the state of P in c^i is equal to its state in all previous configurations of E. The value of field f of processor P_i in configuration c will be denoted either by $c_i.f$ or by $P_i.f$.

Lemma 6: For any configuration c in execution E, if $(c_i.token, c_{i+1}.token) = (\vec{I}, \overleftarrow{I})$, then both P_i and P_{i+1} are in their initial state in E.

Proof: From the protocol, it is easy to see that P_i cannot enter state $\vec{I}$ if P_{i+1} is in state $\overleftarrow{I}$ and vice versa. Neither can P_i and P_{i+1} enter states $\vec{I}$ and $\overleftarrow{I}$ (respectively) concurrently. □

Lemma 7: In any execution of the protocol E_i at most n distinct tokens exist. In other words: The sum of the number of tokens present in the initial configuration of E and the number of tokens created during E does not exceed n.

Proof: A processor can create a token only if it is in its initial state in E and it is *Idle* (because token creation happens only in transition 5). Thus a processor holding a token in the initial configuration of E will not create a token during the execution, and a processor can create at most one token. It follows that in any execution, at most n distinct tokens exist. □

We now introduce some definitions for the following theorem. For any $i \geq 0$ and any configuration c, $Prefix(c,i)$ is the prefix of length i of c, i.e. the states of $P_0, \ldots, P_{i-1}$. $s \sim e$ stands for "s matches the regular expression e".

$$
\begin{aligned}
M &= \{ \vec{I}, \vec{R}, \vec{S}, \overleftarrow{I}, \overleftarrow{R}, \overleftarrow{S} \} \\
A_i &= \{ c \in M^n \,/\, prefix(c, i+1) \sim (\vec{R} + \vec{S} + \vec{I})^{i-1} (\vec{I} + \vec{R})\, \vec{R} \} \\
B_i &= \{ c \in M^n \,/\, prefix(c, i+1) \sim (\vec{R} + \vec{S} + \vec{I})^{i}\, \vec{S} \} \\
D &= \{ c \in M^n \,/\, c_0 = \vec{R} \wedge c_{n-1} \neq \vec{S} \}
\end{aligned}
$$

Theorem 8: (Main theorem) Let $E = c^0, c^1, \cdots, c^k$ be an execution of the system. If a token t which is present in c^0 makes exactly $n-1$ steps, then $c^k \in L$.

Proof: Without loss of generality assume that t is held by P_0 in c^0 and that t moves in E in a clockwise direction (represented by rightward arrows throughout this proof). Configuration c^k is reached when t makes the $(n-1)$-st step. Since we assumed that t is present in c^k we can deduce that t is not eliminated during E.

Claim: Let i_j be the number of steps t made between c^0 and c^j: the holder of t in c^j is P_{i_j}. For all $j \leq k$,

$$
\begin{aligned}
&\text{if } i_j = 0 \text{ then } c^j \in B_0 \cup D \\
&\text{if } i_j > 0 \text{ then } c^j \in A_{i_j} \cup B_{i_j}
\end{aligned}
$$

Proof: By induction on j:
Base: $j = 0$, $i_j = 0$. By our assumption t is held by P_0, so either $P_0.token = \vec{S}$ and $c^0 \in B_0$, or ($P_0.token = \vec{R} \wedge P_{n-1}.token \neq \vec{S}$) and $c^0 \in D$.

Induction Step: Suppose the claim holds for some $j \geq 1$. Consider the following three cases: