TECHNICAL REPORT 90-02

Understanding Self-Stabilization in Distributed Systems Part I

RESEARCH REPORT

March 1990

by

Sukumar Ghosh

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF IOWA • IOWA CITY

THE UNIVERSITY OF IOWA
DEPARTMENT OF COMPUTER SCIENCE

List of Technical Reports 1975-

75-01* R. J. Baron and A. Critcher, *STRINGS^2: Some FORTRAN-callable string processing routines.*
75-02 H. Rich, *SPITBOL GROPE: A collection of SPITBOL functions for working with graph and list structures.*
75-03 A. C. Fleck, *Formal languages and iterated functions with an application to pattern representations.*
75-04 A. Mukhopadhyay, *Two-processor schedules for independent task systems with bounded execution.*
75-05* D. Leinbaugh, *Finite array schemata.*
75-06* T. Sjoerdsma, *An interactive pseudo-assembler and its use in a basic computer science course.*
75-07* J. Lowther, *The non-existence of optimizers and subrecursive languages.*
76-01 A. Deb, *Parallel numerical computation.*
76-02* D. Buehrer, *Natural resolution: A human-oriented logic based on the resolution.*
76-03* C. Yang, *A class of hybrid list file organization.*
76-04 C. Yang, *Avoid redundant record accesses in unsorted multilist file organizations.*
77-01 A. Mukhopadhyay, *A fast algorithm for the longest common subsequence problem.*
77-02* D. Alton and D. Eckstein, *Parallel searching of non-sparse graphs.*
78-01 C. Yang and G. Salton, *Best-match querying in general database systems.*
78-02 C. Yang and R. Hartman, *Extended semantics to the entity-relationship model.*
78-03 A. Mukhopadhyay and A. Hurson, *ASL—an associative search language for data base management and its hardware implementation.*
78-04 D. Riley, *The design and applications of a computer architecture utilizing a single control processor and an expandable number of distributed network processors.*
78-05 A. Critcher, *Function schemata.*
79-01* D. Willard, *Polygon retrieval.*
79-02 S. R. Seidel, *Language recognition and the synchronization of cellular automata.*
79-03* R. J. Baron, *Mechanisms of human facial recognition.*
79-04 R. J. Baron, *Information storage in the brain.*
80-01 D. E. Willard, *K-d trees in a dynamic environment.*
80-02 D. M. Dungan, *Bibliography on data types.*
80-03* D. M. Dungan, *Variations on data type equivalence.*
80-04* R. Baron and S. Zimmerman, *BODIES: A collection of FORTRAN IV procedures for creating and manipulating body representations.*
81-01 R. F. Ford, Jr., *Design of abstract structures to facilitate storage structure selection.*
81-02 K. V. S. Bhat, *A graph theoretic approach to switching function minimization.*
81-03* K. V. S. Bhat, *On the notion of fuzzy consensus.*
81-04* D. W. Jones, *The systematic design of a protection mechanism to support a high level language.*
81-05* J. T. O'Donnell, *A systolic associative LISP computer architecture with incremental parallel storage management.*
81-06 K. V. S. Bhat, *An efficient approach for fault diagnosis in a Boolean n-cube array of microprocessors.*
81-07 K. V. S. Bhat, *On "Fault diagnosis in a Boolean n-cube array of microprocessors."*
82-01 K. V. S. Bhat, *Algorithms for finding diagnosability level and t-diagnosis in a network of processors.*
82-02* R. K. Shultz, *A performance analysis of database computers.*
82-03* D. W. Jones, *Machine independent SMAL: A symbolic macro assembly language.*
82-04 C. T. Haynes, *A theory of data type representation independence.*
82-05 P. G. Gyllstrom, *Fault-tolerant synchronization in distributed computer systems.*
83-01 C. Denbaum, *A demand-driven, coroutine-based implementation of a nonprocedural language.*
83-02 A. C. Fleck, *A proposal for comparison of types in Pascal and associated models.*
83-03 S. Yang. *A string pattern matching algorithm for pattern equation systems with reversal.*
83-04* K. W. Miller, *Programming in vision research using pixelspaces, a data abstraction.*
83-05 C. Marlin, *A methodical approach to the design of programming languages and its application to the design of a coroutine language.*

* no longer available.

Continued on inside back cover

Understanding Self-Stabilization in Distributed Systems

Part 1

Sukumar Ghosh

Department of Computer Science
The University of Iowa
Iowa City, IA 52242
(319)-335-0738
email: ghosh@herky.cs.uiowa.edu

Abstract

A self-stabilizing system is a network of machines, which starts from an arbitrary initial state and always converges to a legitimate configuration in a finite number of steps. Dijkstra, in his 1974 CACM paper, first demonstrated the feasibility of designing self-stabilizing systems. This report analyzes two of his three protocols, and evolves a methodology for designing non-trivial self-stabilizing systems. To demonstrate the feasibility of this methodology, Dijkstra's solutions have been extended to graph topologies. Finally, the importance of self-stabilizing systems have been highlighted with possible potential applications in different areas.

Understanding Self-Stabilization in Distributed Systems

Sukumar Ghosh

Abstract

Dijkstra [7] [10] introduced the problem of self stabilization in distributed systems as an interesting exercise for achieving global convergence through local actions. In [7], he suggested three solutions to a specific version of the self-stabilization problem, one of which was proved in [8]. Ever since this paper was written, the understanding as well as the design of self-stabilizing systems remained a challenging exercise. This paper analyzes the self-stabilization mechanism, and discusses how such algorithms can be synthesized. In addition to the synthesis of Dijkstra's algorithms, this paper considers generalizations of these algorithms on graph topologies of distributed system, and highlights self-stabilization as a new paradigm for designing distributed algorithms.

Categories and Subject Descriptors: C.2.4 [Computer-Communication Network]: Distributed Systems - distributed applications; D.4.1 [Operating Systems]: Process Management - synchronization.

General Terms: Theory, Algorithms.

Additional Keywords and Phrases: Self-stabilization, distributed algorithm, synthesis.

1 Introduction

A distributed system traditionally consists of a set of loosely connected processes which do not share a global memory. Depending on the connectivity of such a system, each component process gets a partial view of the global state. Due to the finite propagation delay of the signals, the notion of a global state is not simple, and its computation is a nontrivial task, which has been demonstrated in [5].

Notwithstanding these difficulties, there is a great demand for the design of distributed algorithms. Each such algorithm essentially transforms an initial global state to a final global state in a finite number of steps, without taking into consideration the actual amount of time required by a process to complete an action or by a signal to reach its destination. The actions mentioned herein are all local actions, yet a correct formulation of an algorithm must always ensure that the global

state eventually converges to a specific goal. This amply highlights the importance of studying suitable methodologies for achieving global convergence through local actions.

In [7], Dijkstra described an interesting exercise for achieving global convergence through local actions. His system consisted of n processes $(0, 1, \ldots, n-1)$ connected in the form of a ring (Fig. 1). Each process is a finite state machine which can read (i) its own state, (ii) the state of its left neighbor, and (iii) the state of its right neighbor. Depending on some predicate(s) defined over these states, a process can update its own state. A process enjoys a *privilege* when such a predicate is true, and a privileged process may update its own state, which is called a *move*. Furthermore, when more than one process enjoys privileges, the choice of the process which is entitled to make a move is determined arbitrarily by *a central demon*.

Based on some predefined postcondition, the set of possible global states of such a system can be divided into two classes: *legitimate* and *illegitimate*. Dijkstra defined the following global criteria for the legitimate states:

LS1: In each legitimate state, one or more privileges must be present.

LS2: In each legitimate state, each possible move would again bring the system in a legitimate state. Thus during an infinite run, the system must eventually cycle through a finite number of legitimate states.

LS3: Each privilege must be present in at least one legitimate state.

LS4: For any pair of legitimate states, there must exist a sequence of moves transferring the system from the one to the other.

A system was called *self-stabilizing*, if and only if regardless of the initial state, and regardless of the privilege selected each time for the next move, at least one privilege would always be present, and the system is guaranteed to find itself in a legitimate state after a finite number of moves.

In all his solutions, Dijkstra considered a specific version of LS1 to characterize a legitimate state – he considered a legitimate state to be one in which exactly one machine enjoys a privilege.

In [7], Dijkstra presented without proof three different solutions to the self-stabilization problem:

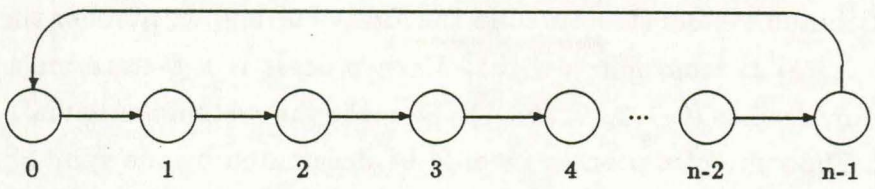

Figure 1: A ring-structured distributed system.

(a) Solution with k-state machines ($k \geq n$);
(b) Solution with 3-state machines;
(c) Solution with 4-state machines.

In [7], Dijkstra presented a proof of his second solution. Another proof for this solution was furnished by Kessels [11]. Kruijer [12] extended Dijkstra's solution to the tree topology. In a modified framework, Brown et.al. [2] suggested three solutions for a self-stabilizing token system. In all these approaches, it was found that self-stabilizing systems could not be designed, if all the component processes were identical. As an interesting exception, recently Burns [4] showed that it is possible to have a symmetric solution to the self stabilization problem when n is prime.

Dijkstra's proof [8] provides useful insight to the mechanism of self-stabilization. However, with the exception of [4] none of these papers suggests any methodology for designing a self-stabilization algorithm. This paper is an attempt towards evolving such a methodology. The paper is organized as follows: section 2 deals with the basic concepts of privileges and moves, section 3 analyzes the unidirectional protocol for self-stabilization, section 4 discusses the bidirectional protocol for cyclic graph topologies, and section 5 deals with extensions of this protocol to acyclic topologies. Finally, section 6 contains some concluding remarks and discusses the importance of studying self-stabilization. The reader is expected to be familiar with Dijkstra's original work reported in [7].

2 Basic Concepts

2.1 Notations

In a distributed system structured in the form of a ring, we number the processes as $(0, 1, 2, \ldots, n-1)$ from left to right. Each process is a k-state machine whose states are numbered as $(0, 1, 2, \ldots, k-1)$. To make the system nontrivial, we assume that $n > 2$. The state of a process i would be designated by the symbol $s[i]$.

In a self-stabilizing system, the lifecycle of a process is as follows:

repeat
    if *privilege*[1] **then** *move*[1] **fi**;
    if *privilege*[2] **then** *move*[2] **fi**;
    ... ...
    if *privilege*[m] **then** *move*[m] **fi**;
forever.

For a process i, a privilege is a boolean function of $(s[i-1]), s[i], s[i+1])$. Note that each - and + operation is a **mod n** operation, however, for the simplicity of writing, this detail would be omitted from the remaining part of this paper. A *move* by process i simply modifies the value of $s[i]$.

2.2 Privileges and Moves

To keep our discussions simple, we start with a simple version of a privilege, where it is a boolean function p of $(s[i-1], s[i])$ only. We represent the privilege for a process i by the symbol $p[i]$. Note that for Dijkstra's self-stabilizing systems, exactly one machine can have a privilege in the legitimate state. However, in an arbitrary initial state, an arbitrary number of machines may enjoy privileges. Therefore, when $p[i]$ is true, the corresponding move by process i must try to change $p[i]$ to false. Obviously there would be a danger of deadlock when no machine enjoys privilege in the initial state, or the only privilege existing in the system is also killed by such a move. However, this would be taken up later. Examples of privileges and moves for a process i are:

(a) **if** $s[i-1] = s[i] + 1$ **mod** k **then** $s[i] := s[i-1]$ **fi**;

(b) **if** $s[i-1] = s[i]$ **then** $s[i] := s[i-1] + 1$ **mod** k **fi**;

As a general tool, a move by process i may negate $p[i]$ in a finite number of steps. An example is

if $s[i-1] \neq s[i]$ **then** $s[i] := s[i] + 1 \bmod k$ **fi**;

Here, depending on the relative values of $s[i-1]$ and $s[i]$, several consecutive moves may be needed by process i to kill the privilege. However, there is no apparent reason for adopting such a phased approach for killing a privilege.

These observations lead to the following axiom:

Axiom 1.1: *Each privilege for a process i must be negated in a finite number of moves by that process.*

Although so far we argued in favor of killing the privileges, it is necessary to sustain the last privilege, otherwise the system would be deadlocked. It is therefore necessary to design the privileges and the moves in such a way that *at least one privilege always exists* in every possible global state of the system. With symmetric machines, when (k, n) are relatively prime, this can be safeguarded using the following privilege and move:

if $s[i] \neq s[i-1] + 1 \bmod k$ **then** $s[i] := s[i-1] + 1 \bmod k$;

However, if one allows at least one machine to be an exception, then the presence of at least one privilege can be ensured by designing a suitable privilege for the exceptional machine. This leads to the following invariant for the global state:

Axiom 1.2: $p[0] \vee p[1] \vee p[2] \ldots \vee p[\text{n-1}] = \mathit{true}$.

Having ensured that at least one process would have a privilege, it is necessary to examine the effects of killing a privilege.

We use the following notation to represent the effect of a move by a process:

precondition $\rightarrow$ *postcondition*

In order to satisfy LS2, when a privilege of a process i is killed by a move, a new privilege must be generated for the process $(i+1)$. As long as axiom 1.2 is satisfied, there is no special concern about it, and this should be an obvious outcome. Note that we are now dealing with legitimate states. So, when process i has a privilege, obviously process $(i+1)$ would not have any privilege. At this point, we are not at all concerned about what would happen if both the processes i and $(i+1)$ have privileges. This would be studied later. However, our observations about legitimate states lead to the following axiom:

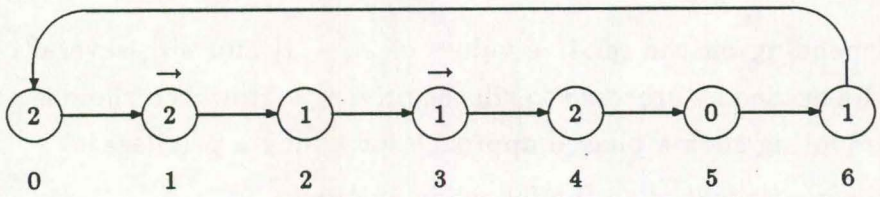

Figure 2: An example of a system which may not stabilize. Each machine has three states (0,1,2). Each $\rightarrow$ indicates a privilege. If the moves are made sequentially by the processes 1,3,2,4,3,5,4,6,5,0,... then the privileges never collide.

Axiom 1.3: $p[i] \wedge \neg\, p[i+1] \rightarrow \neg\, p[i] \wedge p[i+1]$.

2.3 Global Convergence

Axiom 1.1 is not enough to achieve global convergence. Since axiom 1.3 requires that each privilege propagates to the right neighbor, a malicious demon may choose the processes in such a way that multiple privileges continue to exist in the ring forever. Figure 2 illustrates such a situation, where the only privileges and moves for the different machines are

$$\textbf{if } s[i] \neq s[i-1] + 1 \bmod 3 \textbf{ then } s[i] := s[i-1] + 1 \bmod 3$$

Global convergence thus calls for some extra effort by which the number of privileges are bound to decrease until this number comes down to 1. Apparently, there may be different ways to ensure this.

2.3.1 Collision of privileges

We represent a privilege by a ($\rightarrow$) marked on the node having the privilege (Fig. 2). When two privileges *collide* (which means that two adjacent processes i and $(i+1)$ have privileges) a move by process i would definitely reduce the number of privileges by at least 1. However, since the privileges propagate at unpredictable speeds, the collision of two nonadjacent privileges may not be inevitable during an infinite run, as has been illustrated in Figure 2. Therefore, without further refinements, collision of privileges is not an acceptable methodology for global convergence.

2.3.2 Absorption of privileges

If one of the machines is considered to be an exceptional machine, then it might be possible to design the privilege for this exceptional machine in such a way that the exceptional machine absorbs some of the privileges without generating new ones.

2.3.3 Neutralization of privileges

So far, we considered only one type of privilege for a process i. This privilege was defined as a boolean function of $s[i]$ and $s[i-1]$. A system which achieves self-stabilization using only this type of privilege is said to follow a *unidirectional protocol.* It is equally possible to consider another type of privilege which would be defined as a boolean function of $s[i]$ and $s[i+1]$. The latter type of privilege would be represented by drawing a $(\leftarrow)$ on the node having that privilege. It is easy to visualize that all the arguments presented in the earlier sections are equally valid with the latter type of privilege also, with the sole exception that this privilege would propagate in the opposite direction. It is trivial to show that every unidirectional protocol defined with $(\rightarrow)$ can also be defined in terms of $(\leftarrow)$. However, there are interesting possibilities of achieving global converegence when both these types of privileges are considered together. Such a protocol would be called a *bidirectional protocol.* The strength of the bidirectional protocol lies in the fact that it is easier to kill a privilege by arranging a collision with a privilege of the opposite type, since the two types of privileges propagate in opposite directions along the ring.

3 A Unidirectional Protocol

3.1 Dijkstra's First Algorithm

As discussed earlier, in this model, we define the privilege of a machine i in terms of its own state $s[i]$ and the state $s[i-1]$ of its left neighbor only. We begin with a description of Dijkstra's first algorithm discussed in [7] [10]. This algorithm assumes that node 0 is an exceptional node[1], whose privileges and moves are different from the remaining $n-1$ nodes in the systems. Also note that this algorithm works only when $k \geq n$.

For the sake of adopting a uniform notation in describing self-stabilization algorithms, we henceforth use the symbol $s.self$ to designate the own state of a process. The states of the predecessor and successor nodes would be represented by the symbols $s.pre$ and $s.succ$ respectively.

Algorithm 1.1: *Dijkstra's first algorithm.*

{For every machine 1..n−1}

if $s.self \neq s.pre$ **then** $s.self := s.pre$ fi;

{For the exceptional machine 0}

if $s.self = s.pre$ **then** $s.self := s.self + 1$ **mod** k fi;

Theorem 1.1: *Algorithm 1.1 guarantees self-stabilization when* $k \geq n$.

Proof: The privileges in this algorithm satisfy axiom 1.2, so there is no risk of deadlock.

Without any loss of generality, we can assume that the state of machine 0 is initially 0. We can also assume that machine 0 enjoys a privilege, so that $s[n-1] = 0$.

When $k \geq n$, there may be at most $n-1$ machines having *distinct* states from the set $\{0..k-1\}$. This also implies that even if we want to assign *distinct* values of initial states to the machines $1..(n-1)$, *at least* one element of the set {0.. k-1}

[1]The need for an exceptional node has been emphasized in [1][10].

must be left out[2]. Note that with such an assignment of initial states, all the n machines have privileges.

We define the cycle of a machine as the state sequence $0, 1, 2, \ldots k-1, 0$. Every move by machine 0 creates a privilege for machine 1 in case it does not have one. Also, if machine i makes x moves, then it is implied that machine $i-1$ also must have made at least x moves. Therefore, if machine 0 completes a cycle by making k moves, then machine $n-1$ must have made at least k moves.

Can machine 0 complete a cycle if machine $(n-1)$ makes exactly k moves ? Clearly, this is possible if the system is in the *legitimate* state. However, this is impossible if the system is in an illegitimate state.

To show this impossibility, note that machine $(n-1)$ also needs to complete a cycle in the first k moves. However, with $k \geq n$, since at least one element of {0.. k-1} is always missing in the array of states of the machines $1..n-1$, there must be at least one machine i $(1 \leq i \leq n-1)$ for which $s.self \neq s.pre$ or $s.pre + 1$. Since with every move $s.self$ is changed to $s.pre$, machine i cannot complete a cycle in the first k moves. Accordingly, no machine in $i+1 \cdots n-1$ can complete a cycle in k moves.

Therefore, machine $n-1$ has to make *more than k* moves in order to enable machine 0 to complete one cycle. These k moves by machine 0 can trigger *at most* k subsequent moves for machine $n-1$. Since the total number of privileges finite, the number of privileges would steadily drop down until it reaches one. It is trivial to show that if the system has exactly one privilege, then that privilege is sustained for ever. Thus the system is eventually stabilized.

□

Note that when $k < n$ the assertion *"machine $(n-1)$ must move more than k times in order that machine 0 can make k moves"* is no longer valid. An example of a system with ($k = 2$ and $n = 4$) which may not stabilize is shown in Figure 3. If the machines fire in the sequence $0, 3, 2, 1, 0, 3, 2, 1$ then the system goes back to the initial state without ever entering a legitimate state.

[2]When k different eggs are to be put in less than k baskets, at least one egg must be left out.

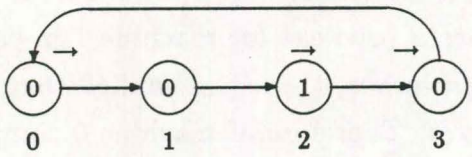

Figure 3: A distributed system which may not self-stabilize.

3.2 Performance Issues

The proof of theorem 1.1 shows that for every k move by machine 0, machine $n-1$ must have moved at least $k+1$ times. Thus every k moves by machine 0 must reduce the number of privileges by at least one. Since in the worst case, initially all n machines can have privileges, to reduce the number of privileges to one, one would need at most $k(n-1)$ moves by machine 0. With n machines, the total number of moves made by all the machines in the system would be $O(kn^2)$. For a better analysis, see Chang [6]. Tchunte [13] presents another algorithm which converges twice as fast as Dijkstra's algorithm.

In [7] [10], it has been mentioned that the above algorithm achieves self-stabilization when $k \geq n$. Note that as discussed in [7], the strict lower bound for k becomes $(n-1)$ instead of n when the *central demon* is replaced by a *distributed demon.* This is because, with distributed demons, the individual machines can simaltaneously read the states of the neighbors, and make simultaneous moves.

3.3 Self-stabilization on a graph

Regarding the generalization of the unidirectional protocol for application to arbitrary graph topologies, Dijkstra [7] refers to a private communication from Scholten. To rediscover such an algorithm, we consider the different cycles which form the graph. To prevent the different privileges in a cycle from chasing each other, we consider an exceptional machine in *each cycle of the graph.* Our first attempt is to use an algorithm in the same style as Dijkstra's first algorithm. To illustrate our approach, we consider an example graph shown in Figure 4. There are two directed cycles in this graph, and since machine 0 belongs to both these cycles, it would be

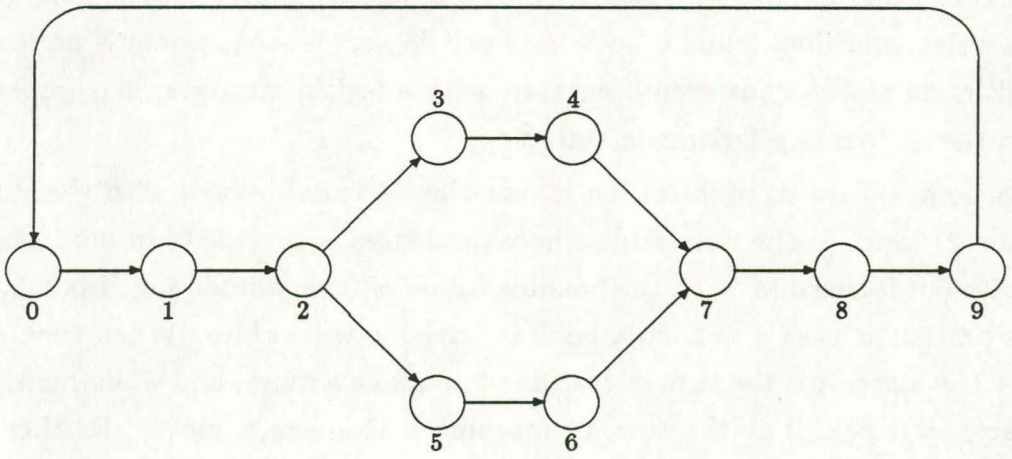

Figure 4: A graph on which self-stabilization needs to be achieved.

treated as an exceptional machine. The algorithm is as follows:

Algorithm 1.2: *First Attempt for self stabilization on a graph.*

{For the exceptional machine 0}

if $s.self = s.pre$ **then** $s.self := s.self + 1$ **fi**;

{For all other machines}

for *each node* $\in$ *pre* {predecessor nodes}
if $s.self \neq s.pre$ **then** $s.self := s.pre$ **fi**;

The correctness of the above algorithm depends on the definition of the legitimate state. If the global condition corresponding to the legitimate state requires that the number of privileges in each cycle of the graph is exactly one, then the above algorithm is correct as long as $k \geq n$. If however, it is desired that in the legitimate state, exactly one privilege must be present in the entire graph, then algorithm 1.2 is not applicable.

To see why this algorithm is not applicable, consider the states of the different machines as shown in Figure 5. Here, only one machine 0 has a privilege. As machines 0, 1 and 2 move successively, the state of machine 2 changes from 0 to 1. At this point, machines 3 and 5 both have privileges, and the system is no more in the legitimate state. Thus even if we start with a legitimate state, it is impossible to keep the system in a legitimate state.[3]

As a remedy to this problem, let us associate a variable *turn* with the state of the node 2, which is the only node whose outdegree is greater than one. For any node with outdegree d ($d > 1$), the possible values of *turn* would range from $0..d-1$. In this particular case $d = 2$, so a boolean variable will suffice. When $turn.pre = 0 \wedge s.self \neq s.pre$, it is the turn of machine 3 to make a move, and when $turn.pre = 1 \wedge s.self \neq s.pre$, it is the turn of machine 5 to make a move. Furthermore, to give equal opportunity to the nodes 3 and 5, *turn* must be incremented (mod d) between successive moves. Therefore, in the example graph, when machine 2 makes a move, it should also execute $turn := \neg turn$. With these modification, the algorithm looks as follows:

Algorithm 1.3: *Second Attempt for self stabilization on a graph.*

{State of node 2 is (s, turn), where turn $\in$ (0,1).
k indicates the number of possible values of s and k $\geq$ n.}

{For the exceptional machine 0}

if $s.self = s.pre$ **then** $s.self := s.self + 1$ **fi**;

{For machine 2 with outdegree greater than 1}

if $s.self \neq s.pre$ **then** $s.self = s.pre$; $turn.self := \neg\ turn.self$ **fi**;

{For machine 3 which is a successor of machine 2}

if $s.self \neq s.pre \wedge turn.pre = 0$ **then** $s.self = s.pre$ **fi**;

{For machine 5 which is the other successor of machine 2}

if $s.self \neq s.pre \wedge turn.pre = 1$ **then** $s.self = s.pre$ **fi**;

{For all other nodes}

[3] Kruijer [12] however uses a modified definition of the legitimate state.

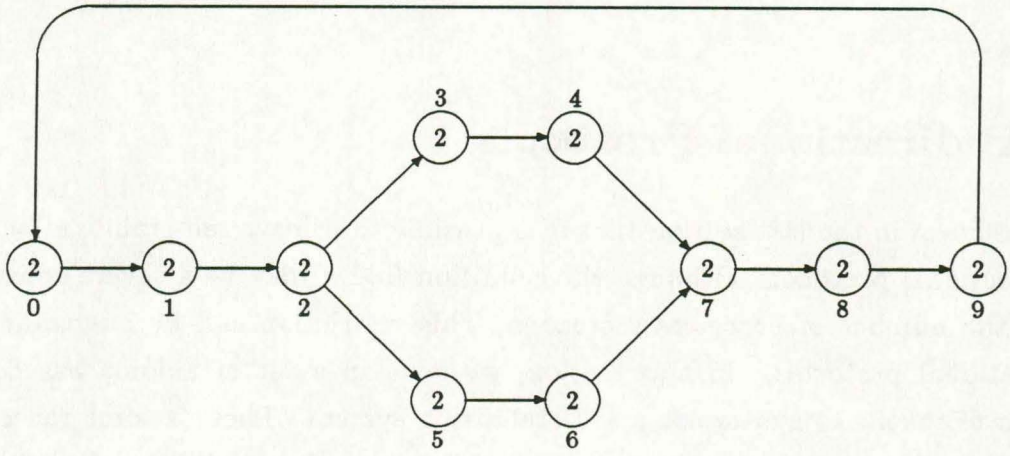

Figure 5: Illustration of an incorrect unidirectional algorithm. each machine has three states 0,1,2 and the system is in a legitimate state. But after machines $0, 1, 2$ move, both 3 and 5 can have privileges.

for *each predecessor node pre*
if *s.self* $\neq$ *s.pre* **then** *s.self* := *s.pre* **fi**;

Theorem 1.2: *Algorithm 1.3 guarantees self-stabilization for the graph in Figure 4.*

Proof: The arguments in the proof would be similar to the arguments used in proving theorem 1. Note that with the privileges of algorithm 1.3, deadlock is impossible.

Assume that initially more than one machine have privileges. Since $k \geq n$, in order that the exceptional node 0 make k consecutive moves, the node 9 must move *more than* k times. However, k moves by node 0 can lead to *at most* k moves by node 9. Thus the number of privileges would steadily decrease.

Once in the legitimate state, the one privilege remains the only one in the entire graph. Thus the system is eventually self-stabilized.

Furthermore, in an infinite sequence of moves, node 2 is able to move infinitely

often. Since each move negates the value of *turn*, nodes 3 and 5 would also be able to move infinitely often. Thus, starvation is prevented.

□

4 Bidirectional Protocols

It was shown in the last section that it is possible to achieve self-stabilization with unidirectional protocols, although the condition $k \geq n$ may be a severe constraint when the number of processes increases. This restriction can be overcome with bidirectional protocols. In this section, we develop a set of axioms based on a system of tokens to synthesize a self-stabilizing system. These axioms reflect one of the possible strategies only, and do not rule out the feasibility of alternative strategies.

In the bidirectional case, two different types of privileges of a machine can be defined separately in terms of $(s[i], s[i-1])$ and $(s[i], s[i+1])$. From now onwards, we would represent the privileges $p(s[i], s[i-1])$ and $p(s[i], s[i+1])$ for a process i by $pL[i]$ and $pR[i]$ respectively.

In designing bidirectional protocols, we first adapt axiom 1.3 to the bidirectional case. This leads to the following axiom:

Axiom 4.1:

(a) $pL[i] \wedge \neg pR[i] \wedge \neg pL[i+1] \rightarrow \neg pL[i] \wedge pL[i+1]$.
(b) $pR[i] \wedge \neg pL[i] \wedge \neg pR[i-1] \rightarrow \neg pR[i] \wedge pR[i-1]$

Axiom 4.1 implies that the privilege pL can only propagate towards the right neighbor, whereas the privilege pR can only propagate towards the left neighbor.

To ensure that in any state at least one machine has a privilege, we rewrite axiom 1.2 with bidirectional privileges.

Axiom 4.2: *In any state, there must be at least one machine i such that $pL[i]$ or $pR[i]$ is true.*

Our next axiom is a a vital one, and is related to the cancellation of privileges when a machine i for which both $pL[i]$ and $pR[i]$ are true makes a move.

Axiom 4.3: $pL[i] \wedge pR[i] \rightarrow \neg pL[i] \wedge \neg pR[i]$.

At this point, it is not possible to fully justify this axiom, except for an early disclosure of the fact that it is through this kind of cancellation mechanism outlined in section 2.3 that the number of privileges in the system would be reduced until this number drops down to one. However, this may not be the only conceivable strategy for systematically reducing the number of privileges – there may be other possible strategies.

As another important step towards implementing global convergence, the following condition must hold while defining pL and pR:

Axiom 4.4: $\forall$ i $\in 1..n$ $pL[i] \wedge pR[i-1] = \mathit{false}$

To see why this is important, assume that $pL[i] = \mathit{true}$ and $pR[i-1] = \mathit{true}$. Depending on the mood of the central demon, either i or $(i-1)$ may be chosen to make a move (Figure 6). If i makes a move, $pL[i+1]$ is true, which may imply $pR[i]$ is true. Thus i can again be selected for making the next move, which makes $pR[i-1]$ true. This may again imply that $pL[i]$ is true, and we come back to the starting point ! With more than one machine enjoying privileges, such a biased action by the central demon cannot be ruled out, which would lead to a *locally confined oscillations* of the privileges without making any real progress towards global convergence.

4.1 Choosing Privileges and Moves

Following axioms 4.1,4.3 and 4.4, there can be many possible choices of privileges and moves – this is really a coding problem. Axiom 4.4 indicates that for *every state* $s[i]$ of a machine i, there should be a state $s[i-1]$ for its neighboring machine $(i-1)$, corresponding to each of the following three conditions:

(a) $pL[i] \wedge \neg pR[i-1]$
(b) $\neg pL[i] \wedge pR[i-1]$
(c) $\neg pL[i] \wedge \neg pR[i-1]$

The possible states of the pair $(s[i], s[i-1])$ and the corresponding state transitions are shown in Figure 7. As examples of choosing privileges and moves, we consider the following two cases:

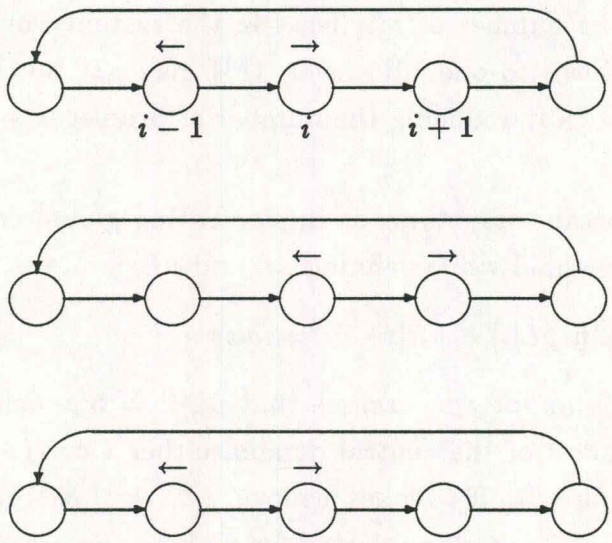

Figure 6: An example of a confined oscillation discussed in the proof of Axiom 4.4

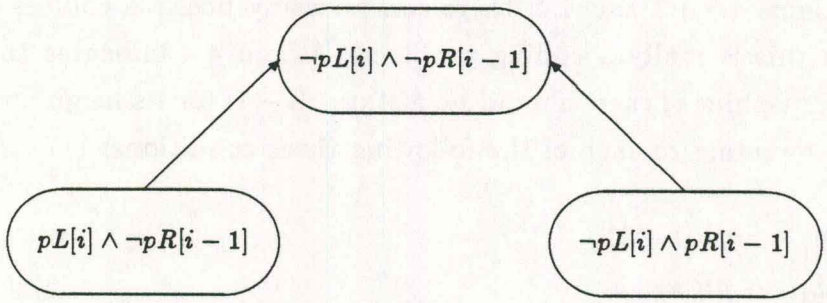

Figure 7: The possible states of $(s[i], s[i-1])$ and the state transitions.

Example: 3-state machine

For a three state machine, the states can be represented by the integers (0,1,2), and the privileges and the moves can be coded as follows:

$pL[i] :: s[i-1] = s[i] + 1 \bmod 3$
move corresponding to $pL[i] :: s[i] := s[i-1]$
$pR[i] :: s[i+1] = s[i] + 1 \bmod 3$
move corresponding to $pR[i] :: s[i] := s[i+1]$

Note that in the above example, the choice of the privileges and the moves satisfies axioms 4.1, 4.3 and 4.4.

4.2 Reflector and Generator Nodes

Having selected the privileges and the moves, it is now necessary to concentrate on the overall mechanism of self-stabilization. When the system starts from an arbitrary initial state, pL and pR may be true for any number of machines. The possible cases are as follows:

(a) More than one machine have privileges;
(b) Only one machine has a privilege.

Let us consider case (b) first. As long as axiom 4.2 is satisfied, the condition "no machine has a privilege" is impossible. In general, this requires the use of at least one exceptional machine, as pointed out in [1] [10]. Also, it is ensured that the number of privileges never increases,so the system is stabilized. Case (b) is therefore a trivial one.

Since case (b) is a trivial one, we start with case (a). For algorithmic convergence to a legitimate state, we treat the machine 0 to be an exceptional machine, and call it a *reflector node.* A reflector node has only one privilege pR, and the corresponding move follows axiom 4.5 below:

Axiom 4.5: *For the reflector node* 0, $pR[0] \wedge \neg pL[1] \rightarrow \neg pR[0] \wedge pL[1]$.

To see why this is necessary, consider Figure 8. Every pR propagates to the left, changes it to a pL at the reflector node, and starts moving towards the right. This

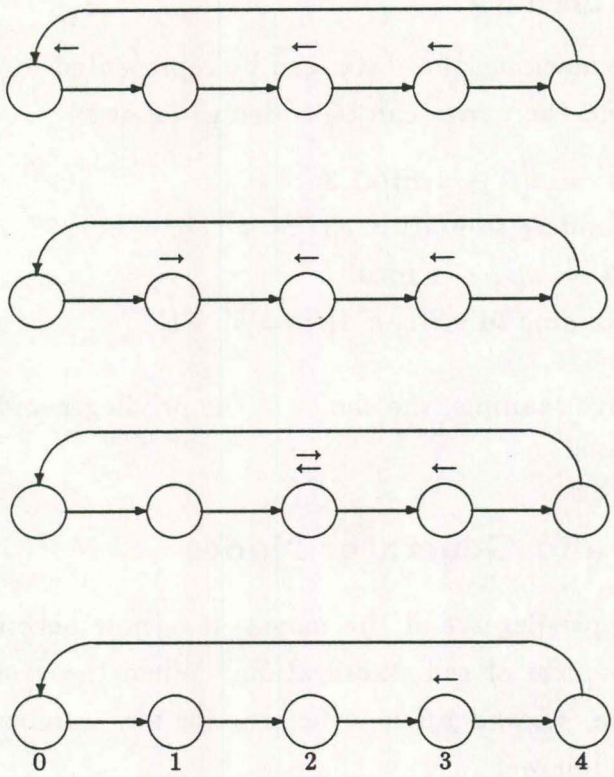

Figure 8: The role of the reflector node in reducing the number of privileges.

makes the collision with a pR in another machine (if there exists one) inevitable. Axiom 4.3 indicates that this leads to the elimination of two privileges.

It appears that with an arbitrary number of pL's and pR's in initial state, convergence to the legitimate state can be achieved if machine $(n-1)$ is also made to behave like a reflector node. However, this may not be the case if the number of privileges in the initial state is even (including zero)[4]

Machine $(n-1)$ therefore would require special attention, and it will be called a *generator node.* To understand the behavior of the generator node, it is necessary to explain the *pseudo-deadlock condition.*

Consider the situation when in the initial state none of the machines $0..n-2$ has a privilege, and recall how axiom 4.2 safeguards this situation. The condition "no other machine has a privilege" should be considered as a privilege for machine $(n-1)$, which would *detect* this condition and make a move to *generate* a privilege

[4]Note that although we have pointed out the need for axiom 4.2, we have not done anything to enforce it so far.

for its neighboring machine $(n-2)$.

How can the generator node detect that there is no privilege for any other machine $0..(n-2)$ in the system ? Apparently, it is not detectable without the help of a central demon. But, depending on how the condition $(\neg pL[i] \wedge \neg pR[i-1])$ has been encoded for a machine i and its left neighbor $(i-1)$, it is possible for machine $n-1$ to *guess* this possibility by examining the states of the machines 0 and $(n-2)$. For example, if the condition $(\neg pL[i] \wedge \neg pR[i-1])$ for machine i corresponds to $(s[i] = s[i-1])$, then a *possible indication* for the condition "no other machine has a privilege" might be $(s[0] = s[n-2] \wedge \neg pR[n-2])$. Note that, this does not uniquely correspond to the condition "no other machine has a privilege", since this condition may also be satisfied if a number of machines *has* privileges. We would refer to this condition as a *pseudo-deadlock condition*, since it is the *best guess* for a possible deadlock in the system considering locally available parameters. We would treat the pseudo-deadlock condition as a privilege for machine $(n-1)$, and permit it to make a move so that $pR[n-2]$ becomes true after such a move. This would also satisfy axiom 4.2. We would shortly find in theorems 4.1 – 4.3 that this indeed leads to self-stabilization. This leads to axiom 4.6:

Axiom 4.6: *If the generator node* $(n-1)$ *detects a pseudo-deadlock condition, then it must make a move to generate a privilege* pR *for machine* $(n-2)$.

4.3 The Algorithm

Based on the axioms discussed so far, it is now possible to write a generic version of the self-stabilization algorithm.

Algorithm 4.1: *A generic self stabilization algorithm.*

{For a machine i $\in$ (1.. n−2)}

$pL[i] \rightarrow \neg pL[i]$;
$pR[i] \rightarrow \neg pR[i]$;

{For the reflector node 0}

$pR[0] \rightarrow pL[1]$;

{For the generator node n−1}

pseudodeadlock $\rightarrow pR[n-2]$;

To prove the correctness of the generic algorithm, it is first necessary to establish the following three lemmas:

Lemma 4.1: *If only one machine has a privilege (pL or pR), then the system is stabilized.*

Proof: A move by a privileged machine can lead to exactly one privilege by a neighboring machine. Therefore, once there is a single privilege in the whole system, in all future moments, exactly one machine would have a privilege. So the system is stabilized.

□

Lemma 4.2: *Between two consecutive moves by the generator node, the reflector node must move at least once.*

Proof: The generator node makes a move when it detects a pseudodeadlock condition. Let s_0, s_{n-2} and s_{n-1} be the states of the nodes 0, $n-2$ and $n-1$ respectively when such a condition is detected. Accordingly, node $n-1$ makes a move which changes s_{n-1} to s^1_{n-1}, and the pseudodeadlock condition becomes false. This creates a privilege for machine $n-2$, which makes a move and changes its state to s^1_{n-2} and the pseudodeadlock condition still remains false. To reassert the pseudodeadlock condition, machine 0 now must change its state, which requires at least one move.

□

Lemma 4.3: *Between two consecutive moves by the reflector node, the generator node can move at most once.*

Proof: Suppose that after the first move by the reflector node 0, the pseudodeadlock condition is satisfied, and the generator node makes a move. According to lemma 4.2, before the generator node makes another move, the reflector node must move at least once. This shows that between two consecutive moves by the reflector node, the generator node can move at most once.

□

Theorem 4.1: *Algorithm 4.1 guarantees self-stabilization when in the initial state, the only type of privileges present in the system is pL.*

Proof: Assume that initially there are x pL's ($\rightarrow$) in the system. $x = 1$ is the trivial case, so let us consider the case where $x > 1$.

Each pL ($\rightarrow$) eventually propagates to the right. Every time node $(n-2)$ makes a move, the number of pL's decreases by 1. As x gradually decreases, the generator node can make *at most one move*, since the detection of the pseudodeadlock condition depends on the state of node 0 which does not change. The only move by the generator node generates a pR ($\leftarrow$) for the node $(n-2)$, which would neutralize a pL propagating from the left (axiom 4.3), and reduce by 1. Otherwise, everytime node $n-2$ moves, x reduces by 1. This would continue until there remains only one pL in the system.

□

Theorem 4.2: *Algorithm 4.1 guarantees self-stabilization when in the initial state, the only type of privileges present in the system is pR.*

Proof: Assume that initially there are y pR's ($\leftarrow$) in the system. $y = 1$ is the trivial case, so let us consider the case where $y > 1$.

Each pR ($\leftarrow$) eventually propagates to the left. Every time node 0 makes a move, a $pR[0]$ is converted to a $pL[1]$. This must neutralize another pR ($\leftarrow$) propagating from right to left (axiom 4.3). Thus, before node 0 makes a second move, the number of pR's ($\leftarrow$) would be $(y-2)$. However, in the mean time, the generator node $(n-1)$ can potentially make one move (lemma 4.3). Taking into account the generated pR, the number of pR's becomes $(y-1)$. Thus, the number of privileges decreases until it reaches one, and the system is stabilized.

□

Theorem 4.3: *Algorithm 4.1 guarantees self-stabilization when the system starts from an arbitrary initial state.*

Proof: Three different cases would be considered:

(a) In the initial state, no machine in $0..n-2$ has a privilege.
(b) In the initial state, exactly one machine has a privilege.
(c) In the initial state, there are x $(x > 0)$ machines having pL's and y $(y > 0)$ machines having pR's.

In case (a), a pseudo-deadlock condition is detected by the generator node, which immediately makes a move so that $pR[n-2]$ holds. According to the rules of propagation of the privileges, this is the only privilege which remains in the system, so self-stabilization is achieved.

Case (b) is a trivial case, since the system starts from a legitimate state. Lemma 4.1 shows that in this case, the system always remains in the legitimate state.

In case (c), assume that node 0 has just made a move, which asserted the pseudodeadlock condition for the generator node $n-1$. When node $n-1$ makes a move after sensing the pseudo-deadlock condition, the number of pR's increases to $y+1$, since a new pR has been created for the node $n-2$. At the same time, the pseudodeadlock condition has been invalidated. Before the generator node makes a second move by sensing a second pseudodeadlock condition, the following events must happen in an arbitrary order (Figure 9):

- node $(n-2)$ must make a move. Unless $pL[n-2]$ is true, this would keep the number of pL's and pR's unchanged to $(x, y+1)$.
- node 0 must make a move (lemma 4.2). The earlier move created a pL for node 1. Before node 0 makes this move, the previous pL must have been neutralized by a pR (axiom 4.3) propagating from the right (unless that is the only privilege remaining in the system), leading to the elimination of two privileges (a pL and a pR), and leaving at most $(x-1)$ $pL's$ and y $pR's$. Thus *after* node 0 makes a move, the number of pL's and pR's in the system must be at most $(x, y-1)$.

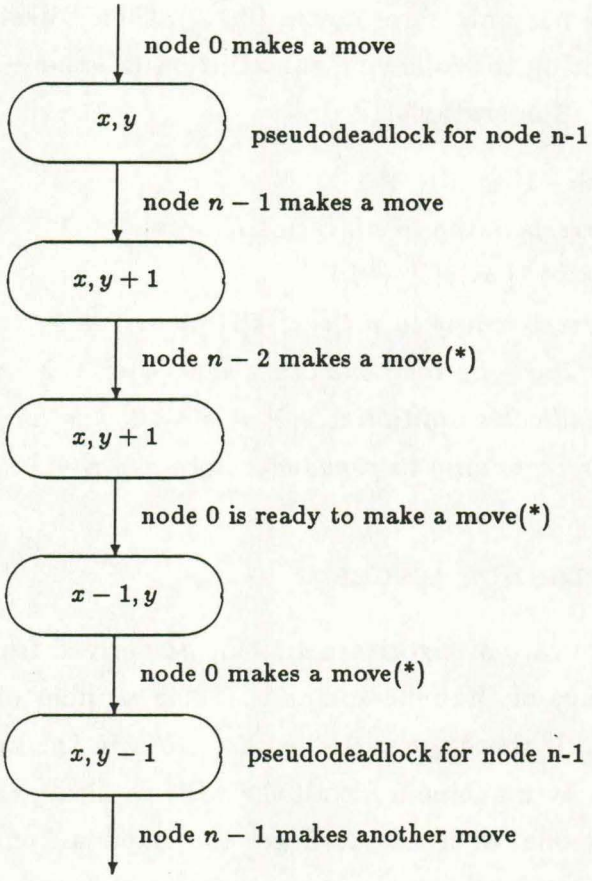

Figure 9: The stabilization mechanism discussed in the proof of theorem 4.3. The events marked with (*) may take place in an arbitrary order.

This shows that the number of pR's is bound to decrease continuously during an infinite run. If the system is left with only pL's, then theorem 4.1 shows that the system is eventually stabilized.

□

4.4 Derivation of Dijkstra's 3-state algorithm

If every machine has only three states (0,1,2), then Dijkstra's 3-state algorithm [6] can be derived using the following substitution (all the + operations in s[] + 1 and s[] + 2 are **mod 3** operations):

$pL[i] :: s[i-1] = s[i] + 1$
Move corresponding to $pL[i] :: s[i] := s[i] + 1$
$pR[i] :: s[i+1] = s[i] + 1$
Move corresponding to $pR[i] :: s[i] := s[i] + 1$
Move for the reflector node $0 :: s[i] := s[i] + 2$
pseudodeadlock condition $:: s[0] = s[n-2] \wedge s[n-1] \neq s[n-2] + 1$
Move corresponding to pseudodeadlock $:: s[n-1] := s[n-2] + 1$

4.5 Performance Issues

The convergence rate of algorithm 4.1 can be derived from Figure 9. Between two consecutive firings of the reflector node 0, the number of privileges must decrease by at least one. If there are x pL's and y pR's in the system, then it would take at most y steps by machine 0 for all the pR's to disappear. Theorem 4.1 indicates that for all (but one) of the x privileges to disappear, one might need at most $x/2$ moves by machine 0. Thus the maximum number of moves by machine 0 could be $(x/2 + y)$. Since the maximum value of x or y could be $(n-1)$, the maximum number of steps required for self-stabilization could be $(n-1)$. With n machines in the system, the total number of steps required for self-stabilization would be $O(n^2)$.

What is the minimum number of states that a machine should have for which the self-stabilization algorithm can be applied ? To answer this question, let us take a look at Figure 7. In order that axioms 4.1, 4.3, and 4.4 be satisfied, it is clear that for every state of a machine i, its neighboring machine should have the following three states:

(a) $pL[i] \wedge \neg pR[i-1]$
(b) $\neg pL[i] \wedge pR[i-1]$
(c) $\neg pL[i] \wedge \neg pR[i-1]$

This points to the fact that we need at least 3-state machines. Can we somehow satisfy the axioms with 2-state machines ? To explore this possibility, we try to define pL and pR for a machine i in terms of all the three states $(s[i-1], s[i], s[i+1])$. There can be at most four choices for the privileges (including both pL and pR), which are as follows:

(i) s[i] = s[i-1] $\wedge$ s[i] = s[i+1]
(ii) s[i] = s[i-1] $\wedge$ s[i] $\neq$ s[i+1]
(iii) s[i] $\neq$ s[i-1] $\wedge$ s[i] = s[i+1]
(iv) s[i] $\neq$ s[i-1] $\wedge$ s[i] $\neq$ s[i+1]

Assume that (iii) corresponds to the condition $pL[i]$. Then, it is impossible to find another condition for $pR[i]$ from the above list, since every choice of $pR[i]$ would violate either axiom 4.1 or axiom 4.4.

This leads to the theorem:

Theorem 4.4: *The minimum number of states that a machine should have in a self-stabilizing system of ring topology is three.*

4.6 Self-stabilization on a graph

In the next step, we extend Dijkstra's 3-state algorithm to arbitrary graphs of cyclic topology. For the purpose of illustration, we use the same graph as shown in Figure 4.

Our first approach is to define exceptional machines on *every cycle* of the graph. For the example graph of Figure 4, this can be achieved if node 0 is considered as the reflector node, and node 9 is treated as the generator node.

Secondly, we introduce the concept of *coloring the nodes* and *coloring the privileges*. Since there are two distinct cycles (0,1,2,3,4,7,8,9,0) and (0,1,2,5,6,7,8,9,0) in the graph, we consider only two colors (*red*, *blue*), and define the state of each machine as (*red*, *blue*), where $red \in \{0,1,2\}$ and $blue \in \{0,1,2\}$. The machines 3 and

4 are *always red* ($blue = 0$), and the machines 5 and 6 are *always blue* ($red = 0$), so that these are essentially 3-state machines, whereas all others (excepting the generator node) are 9-state machines.

When a machine i computes a privilege by examining its own substate $red.self$ and the substate $red.pre$ of its left neighbor, it computes a colored version of pL ($pL.red$). Depending on the value of this privilege, the machine i then decides whether it would update $red.self$. In a similar manner, it computes $pL.blue$, $pR.red$, and $pR.blue$. The updation of the two substates $red.self$ and $blue.self$ are mutually noninterfering, so that in case both $pL.red$ and $pL.blue$ are true, the corresponding updations could be taken up in any order.

Since the machines 3 and 4 are always red, they check for $pL.red$ and $pR.red$ only. Similarly, since the machines 5 and 6 are always blue, they check for $pL.blue$ and $pR.blue$ only.

Assuming all the colored privileges to satisfy the axioms 4.1, 4.3 and 4.4, one can easily visualize the state transitions in terms of the red privileges propagating up and down the path (0,1,2,3,4,7,8,9) and the blue privileges propagating up and down the path (0,1,2,5,6,7,8,9). The role of the reflector node would remain the same for each colored privilege. However, in order that only one privilege finally remains in the system, the role of the generator node needs to be modified. We suggest that only when the generator node 9 detects a pseudodeadlock for *both* the colors, it makes a move to generate a privilege for machine 8, and does it for the two colors alternately. This requires that the state of the generator node be defined as $(r, b, turn)$, where $turn \in \{0, 1\}$. The generator machine would thus have 18 states.

Algorithm 4.2. *Algorithm for self stabilization on a graph.*

{The state s[i] of machine i is (r, b), where r,b $\in$ (0,1,2);
For the machines 3 and 4, b is always 0;
For the machines 5 and 6, r is always 0}

{Those machines which have b = 0 examine only the r−part of
the states of the neighbors. Similarly, those machines which
have r = 0 examine only the b−part of the states of the neighbors}

{For every machine 1..8}

for every $x \in (r,b)$ *and for every pre and succ*
if $x.self = x.pre - 1 \bmod 3$ **then** $x.self := x.pre$ **fi;**
if $x.self = x.succ - 1 \bmod 3$ **then** $x.self := x.succ$ **fi;**

{For the reflector node 0}

for every $x \in (r,b)$ *and for every succ*
if $x.self = x.succ - 1 \bmod 3$ **then** $x.self := x.succ + 1 \bmod 3$ **fi;**

{For the generator node 9}

if $\forall x \in (r,b)\ x.pre = x.succ \wedge x.self \neq x.pre + 1 \bmod 3$ **then**
    if $turn = 0$ **then** $b.self := b.pre + 1 \bmod 3$; $turn := 1$ **fi;**
    if $turn = 1$ **then** $r.self := r.pre + 1 \bmod 3$; $turn := 0$ **fi;**
fi;

Theorem 4.5: *Algorithm 4.2 guarantees self-stabilization for the graph in Figure 4.*

Proof: The proof is an extension of the proof of theorem 4.3, and can be constructed from an observation of the following sequence of events:

- Between two consecutive moves made by the generator machine 9 returning a red privilege, at least two blue privileges (a $\leftarrow$ and a $\rightarrow$) must disappear. Thus during a continuous run, for every single red privilege generated by machine 9, the number of blue privileges must decrease by at least 2.

- Between two consecutive moves made by the generator machine returning a blue privilege, at least two red privileges (a $\leftarrow$ and a $\rightarrow$) must disappear. Thus during a continuous run, for every single blue privilege generated by machine 9, the number of red privileges must decrease by at least 2.

- During an infinite run, the generator node must fire infinitely often for both the red and the blue privileges. Since the rate of extinction of the privileges exceeds their rate of generation, the number of privileges steadily comes down.

- When the number of privileges come down to 1, it remains as the only privilege in the system. A red privilege propagates upto the generator node, and

comes back as a blue privilege. Similarly, a blue privilege propagates upto the generator node, and comes back as a red privilege.

Thus, the system is stabilized in a finite number of steps.

□

This outlines our philosophy for achieving self-stabilization on a graph. The salient features are as follows:

- Find a spanning tree of the graph, and designate the root as the generator node.
- Find the minimum set of nodes whose removal converts the given graph into a directed acyclic graph, and designate these nodes as the reflector nodes.
- Find all the distinct paths from the generator node to the reflector node(s), and number these from $0..k-1$. For each node, define the state as an *array of colors*, where each color $\in \{0,1,2\}$, and the dimension of the array would range over the serial numbers of the paths passing through it.
- Adapt algorithm 4.2 to implement self-stabilization.

4.7 Replacing the Central Demon

The performance of all the algorithms discussed so far depends on the presence of a central demon, which is rather awkward to implement. It would be much nicer if these algorithms also work with distributed demons. In [4] Dijkstra shows why his first algorithm (unidirectional protocol) works with distributed demons also. This section examines whether his solution with 3-state machines (as well as our generic version of the self-stabilization algorithm shown in Algorithm 4.1) works with distributed demons. Burns [3] already showed that this is feasible.

Every action taken by a machine has two parts: $(read, move)$. With a central demon, the atomicity of such an action is automatically preserved, but with distributed demons, it is difficult to ensure since each decision is taken locally by the machines. This makes the problem of global convergence more complex.

To study the possibility of global convergence with distributed demons, note that two operations by non-neighboring machines do not have any interference at all. For two neighboring machines i and $(i+1)$ axiom 4.4 holds — so if $pR[i]$ is true, then $pL[i+1]$ must be false If this were not so, it could be possible that machines i and $(i+1)$ read each other's state, and then make moves so that $pR[i]$ and $pL[i+1]$ still remain true – thus leading to a confined oscillation of the privileges in the ring.

With distributed demons, due to the arbitrary interleaving of the operations *read* and *move*, it is however possible to find a chain of machines $i..i+k$, all of whom find that pR (or pL) exists, make moves, and still find that pR (or pL) is true ! Can the algorithm guarantee global convergence in presence of distributed demons ?

To guarantee global convergence, it is sufficient to show that even in such adverse conditions, the number of privileges eventually decreases. The clue lies with the exceptional nodes 0 and $(n-1)$ which have been termed as the reflector and the generator nodes. Note that the reflector node 0 does not have any pL, all that it can have is a pR. Similarly, a generator node have only one privilege p (corresponding to the pseudodeadlock condition).

Now, consider a chain of machines $i..i+k$ for all of which pR is true, $i+k$ being the highest numbered node for which it holds. If $(i+k) < (n-2)$, then since $pR[i+k] \wedge pL[i+k+1]$ is false, (and $pR[i+k+1]$ is also false since $i+k$ is the highest numbered node in the chain for which pR is true), there must be a process $(i+k+1)$ which would have no privilege at all. Therefore, if all of them first discover that pR is true and then make moves, the value of $pR[i+k]$ would definitely become false after the move. Even when $(i+k) = (n-2)$, the same argument remains valid if $pL(i+k+1]$ is replaced by $p[n-1]$. Thus in the worst case, at least one process will definitely change its pR to false.

Similarly, consider a chain of machines $i..i-k$ for all of which pL is true, $i-k$ being the lowest numbered node for which it holds. If $(i-k) \geq 1$, then since $pL[i-k] \wedge pR[i-k-1]$ is false, (and $pL[i-k-1]$ is also false since $i-k$ is the lowest numbered node in the chain for which pL is true), there must be a process $(i-k-1)$ which would have no privilege at all. Therefore, if all of them first discover that pL is true and then make moves, the value of $pL[i-k]$ would definitely become false after the move. Thus in the worst case, at least one process will definitely change its pL to false after the moves.

With these observations, it is however necessary to modify axiom 4.3 as follows:

Axiom 4.7: $pL[i] \wedge pR[i] \rightarrow \neg pL[j] \wedge \neg pR[k]$ $(j \leq i, k \geq i)$.

However, the fact remains that when two privileges collide at a machine i and that machine makes a move, two privileges (a pL and a pR) disappear somewhere in the system. Since this was the foundation of the proof of algorithm 4.1 (theorems 4.1, 4.2 and 4.3), the algorithm is valid with distributed demons also. This leads us to the following theorem:

Theorem 4.6: *Algorithm 4.1 guarantees self-stabilization with distributed demons.*

5 Conclusion

5.1 General Remarks

This paper illustrates our partial understanding of Dijkstra's work [7] on self-stabilizing systems. Much of the contents were influenced by Dijkstra's proof of the 3-state systems which appeared in [8]. The purpose of this research was to evolve a method for synthesizing nontrivial self-stabilizing systems. It is shown how the study of self-stabilization could be abstracted to the study of a token system based on a set of axioms, and the coding problem could be separated from the converegence analysis. This abstraction was found useful in writing self-stabilization algorithms on graph topologies.

5.2 Relaxation Algorithms

What is the importance of studying self-stabilization ? Once we are convinced that self-stabilization algorithms work with distributed demons, we have a powerful concurrent programming tool for achieving global convergence through local actions. Depending on how the legitimate state or the stable behavior of the system is defined, it should be possible to modify Dijkstra's algorithms to design new distributed algorithms for specific applications, where the programmer need no more be concerned about global states, but only need to concentrate on local actions.

To illustrate this viewpoint, it is necessary to consider a computation as a journey (in the state space) from some initial state to a final state satisfying a definite

postcondition. A *privilege* is a *local measure* of the distance of the current state from the final state. In an arbitrary initial state, an arbitrary number of machines may enjoy privileges, but in the final state, no machine should have a privilege. A self-stabilization algorithm should be able to ensure that when each machine computes its privilege locally and makes appropriate moves to kill that privilege, it is possible to kill all the privileges in the system in a finite number of moves and the final state is reached. Such algorithms are known as relaxation algorithms, which have been studied elsewhere in different contexts. The present work may find use in proving the converegence properies of relaxation algorithms in a concurrent programming environment. An interesting study about a cyclic relaxation problem appears in [8]. Such computations are attractive, since these are highly nondeterministic, yet the programmer need not worry about timing constraints or explicit synchronization aspects to guarantee global converegence.

5.3 Analogy of Feedback Control

It is often said that self-stabilizing systems have important implications in the design of fault-tolerant systems. Such systems can automatically recover from transient failures. Furthermore, these systems can stabilize from arbitrary initial states, so if a faulty processor is repaired, no extra effort is required to integrate it in the system. While this is certainly true, is appears more appropriate to consider a self-stabilization algorithm as a manifestation of the theory of feedback control systems in distributed algorithm design. In a typical feedback control system, we have a set of inputs $\{x_1, x_2, \ldots x_n\}$ and a set of outputs $\{y_1, y_2, \ldots, y_m\}$, and in a stable state, a predefined condition amongst the different input and the output variables must be true. The inputs could come from the sensors, whereas the outputs could be changed by the actuators. It is typical of real time systems that the inputs change due to unforeseen variations in environmental conditions or system parameters, and the system has to stabilize itself as a reaction to this change by adjusting the outputs so that predefined criteria of stability is always satisfied. Dijkstra's algorithms propose the legitimate state to be one in which exactly one machine has a privilege. Consequently, if the system is in a legitimate state, and some of the machines suddenly change their states, then momentarily multiple privileges are created, but the system reacts to this change in such a way that in a finite number of steps, all but one privilege is killed and the system again returns to a legitimate

state. These algorithms thus also provide a basis for the design of real time digital control systems in a distributed environment.

5.4 Other issues

Self-stabilizing systems are thus of fundamental importance in the design of distributed algorithms. However, in the present form, these appear to be too pure to be of direct use in practical applications. One notable achievement is to establish the feasibility of replacing the centralized demon by a distributed demon which has already been demonstrated here as well as in [4]. The other major task would be the detection of stability. If the detection of stability is done a particular machine i in the system, then we can imagine that it would have a local variable $stable[i]$ which would be set to *true* whenever the sustem reaches a stable state. However, since each machine is allowed to start from an arbitrary initial state, then for machine i $stable[i]$ may initially be true even though the system is not in a legitimate state ! This illustrates that for practical applications, a modified set of assumptions may be necessary.

6 Acknowledgement

Ackowledgements are due to Ajoy K. Datta, Weifeng Huang and Mehmet Hakaan Karaata for a careful reading of the manuscript and numerous comments which led to substantial improvements in the presentation of this report.

7 References

1. Angluin, D. "Local and Global Properties in Networks of Processors", *Proc. 12th ACM Symp. Theory of Computing, pp. 82 - 93*, 1980.

2. Brown, G.M., Gouda, M.G. & Wu, C.L. "Token Systems that Self-stabilize" *IEEE Trans. Comput. 38, No. 6, pp. 845 - 852*, June 1989.

3. Burns, J.E. "Self-stabilizing Rings without Daemons", *Tech. Report GIT-ICS-87/36, Georgia Institute of Technology*, November 1987.

4. Burns, J.E. "Uniform Self-stabilizing Rings", *ACM Trans. Prog. Lang. & Syst. 11, No. 2, pp. 330 - 344*, April 1989.

5. Chandy, K.M. & Lamport, L. "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Trans. Comput. Syst. 3, No.2, pp.63 - 75*, February 1985.

6. Chang, E.J.H., Gonnet, G.H. and Rotem, D. "On the Costs of Self-Stabilization", *Information Processing Letters 24, pp. 311 - 316*, 1987.

7. Dijkstra, E.W. "Self-stabilizing Systems in spite of Distributed Control", *Commun. ACM 17, 11, pp. 643-644*, Nov. 1974.

8. Dijkstra, E.W. "A Belated Proof of Self-stabilization", *Distributed Computing 1, No. 1, pp. 5-6*, 1986.

9. Dijkstra, E.W. "The Solution to a Cyclic Relaxation Problem", *EWD 386, in Selected Writing on Computers: A Personal Perspective, pp. 34-35*, Springer Verlag, 1982.

10. Dijkstra, E.W. "Self-stabilization in Spite of Distributed Control", *EWD 391, in Selected Writing on Computers: A Personal Perspective, pp. 41-45*, Springer Verlag, 1982.

11. Kessels, J.L.W. "An Exercise in Proving Self-stabilization with a Variant Function", *Information Processing Letters 29, No. 2, pp. 39 - 42*, 1988.

12. Kruijer, H.S.M. "Self-stabilization (inspite of distributed control) in Tree-Structured Systems", *Information Processing Letters 8, No. 2, pp. 91 - 95*, 1979.

13. Tchuente, M. "Sur l'auto-stabilisation dans un reseau d'ordinateurs", *RAIRO Informatique Theorique 15, pp. 47 - 66*, 1981.

83-06 K. V. S. Bhat, *An optimum reliable network architecture.*
83-07 R. Ford and K. Miller, *An abstract data type development and implementation methodology.*
83-08 T. Rus, *TICS system: a compiler generator.*
83-09 C. Marlin and D. Freidel, *A model for communication in programming languages with buffered message-passing.*
83-10* D. A. Eichmann, *A preprocessor approach to separate compilation in Ada.*
83-11* R. K. Shultz, *Simulation of multiprocessor computer architectures using ACL.*
84-01 D. H. Freidel, *Modelling communication and synchronization in parallel programming languages.*
84-02* T. Rus and F. B. Herr, *An algebraic directed compiler generator.*
84-03 M. E. Wagner, *Performance evaluation of abstract data type language implementations.*
84-04* R. Ford and M. Wagner, *Performance evaluation methodologies for abstract data type implementation techniques.*
84-05 K. Brinck, *The expected performance of traversal algorithms in binary trees.*
84-06 K. Brinck, *On deletion in threaded binary trees.*
84-07 K. Siu-ming Yu, *A testbed database generator.*
84-08 D. Sawmiphakdi, *A multiprocess design for an integrated programming environment.*
84-09 D. W. Jones, *Machine independent SMAL: a symbolic macro assembly language.*
84-10* R. K. Shultz, *Comparison of database operations on a multiprocessor computer architecture.*
84-11 J. H. Kingston, *A new proof of the Garsia-Wachs algorithm.*
85-01 T. Rus, *Fast pattern matching in strings.*
85-02 T. Rus, *An inductive approach for program evaluation.*
85-03 G. S. Singer, *Extensions to the Iowa logic specification language.*
85-04 A. C. Fleck, *Babble reference manual.*
85-05 K. Brinck, *Computing parent nodes in threaded binary trees.*
85-06 J. H. Kingston, *The amortized complexity of Henriksen's algorithm.*
85-07* R. Ford, M. J. Jipping, and R. Shultz, *On the performance of an optimistic concurrent tree algorithm.*
85-08* D. W. Jones, *Iowa capability architecture project ICAP programmer's preference manual.*
85-09* S. P. Miller, *Automated instrumentation of communication protocols for testing and evaluation.*
86-01 M. J. Jipping, *An information-based methodology for the design of concurrent systems.*
86-02* H. I. Mathkour, *An extended abstract data type specification mechanism.*
86-03 D. E. Glover, *Experimentation with an adaptive search strategy for solving a keyboard design/configuration problem.*
86-04 W. Pan, *Designing an operating system kernel based on concurrent garbage collection.*
86-05 B. C. Wenhardt, *A comparison of concurrency control algorithms for distributed data access.*
86-06 R. Shultz and I. Miller, *An execution cost analysis of multiple processor join methods.*
86-07 R. Shultz, *Controlling testbed database characteristics.*
86-08 R. E. Gantenbein, *Dynamic binding of separately compiled objects under program control.*
86-09 M. Pfreundschuh and R. Ford, *A model for modular system builds based on attribute grammars.*
86-10 R. Shultz and I. Miller, *Memory capacity in multiple processor joins.*
86-11* M. P. Pfreundschuh, *A model for building modular systems based on attribute grammars.*
87-01* M. J. Kean, *A communications architecture for distributed applications comprised of broadcasting sequential processes.*
87-02* B. A. Julstrom, *A model of mental image generation and manipulation.*
87-03* M. J. Jipping and R. Ford, *An information-based model for concurrency control.*
87-04* Ravi Mukkamala, *Design of partially replicated distributed database systems: an integrated methodology.*
87-05* A. C. Fleck, *A case study comparison of four declarative programming languages.*
88-01 Jing Jan, *Data abstraction in the Iowa logic specification language.*
88-02* J. P. Le Peau and T. Rus, *Interactive parser construction.*
88-03* J. Gilles, *A window oriented debugging environment for embedded real time ada systems.*
88-04 S. R. Sataluri and A. C. Fleck, *Incremental development of semantics using relational attribute grammars.*
88-05 S. R. Satulari, *Generalizing semantic rules of attribute grammars using logic programs.*
88-06 Hantao Zhang, *Reduction, superposition and induction: Automated reasoning in an equational logic.*
89-01 C. M. Gessner, *A case study in post-developmental testing.*
89-02 Ken Slonneger, *Denotional semantics in prolog.*
89-03 Deepak Kapur and Hantao Zhang, *RRL: Rewrite rule laboratory user's manual*
89-04 Hantao Zhang, *Prove ring commutativity problems by algebraic methods*
89-05 David Allen Eichmann: *Polymorphic extensions to the relational model*
89-06 In Jeong Chung, *Improved control strategy for parallel logic programming*
89-07 Po-zung Chen, *New directions on stochastic timed petri nets*
89-08 Frank William Miller, *A predictive real-time scheduling algorithm*
90-01 Teodor Rus, *Algebraic construction of a compiler*
90-02 Sukumar Ghosh, *Understanding self-stabilization in distributed systems, part I*