SELF STABILIZATION OF DYNAMIC SYSTEMS
ASSUMING ONLY READ/WRITE ATOMICITY

by

S. Dolev, A. Israeli and S. Moran

Technical Report #590

November 1989
Self Stabilization of Dynamic Systems
Assuming Only Read/Write Atomicity

October 31, 1989

Shlomi Dolev† Amos Israeli‡ Shlomo Moran†

ABSTRACT

Self stabilizing protocols for distributed systems in the shared memory model are defined and studied. The protocols presented here improve upon previous protocols in two ways: First, it is assumed that the only atomic operations are either read or write to the shared memory. Second, our protocols work for dynamic networks, in which the topology of the network may change during the execution.

Three dynamic self-stabilizing protocols are presented. The first protocol is a spanning tree protocol for systems with any connected communication graph. The second protocol is a mutual exclusion protocol for tree structured systems. The third protocol is a self stabilizing protocol for mutual exclusion, for systems with a general (connected) communication graph. This last protocol is obtained by combining the previous two protocols. The combination employs a simple technique called fair protocol composition, which is enabled by both the self-stability and by the flexibility of dynamic protocols.

† Department of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, ISRAEL. Part of this research was supported in part by Technion V.P.R. Funds - Wellner Research Fund, and by the Foundation for Research in Electronics, Computers and Communications, administrated by the Israel Academy of Sciences and Humanities.

‡ Department of Electrical Engineering, Technion - Israel Institute of Technology, Haifa 32000, ISRAEL.
1. Introduction

Self Stabilizing protocols add fault tolerance power to distributed systems. A self stabilizing system which is started from any arbitrary initial configuration regains its consistency and demonstrates legal behavior by itself, without any kind of an outside intervention. Two important advantages of self stabilizing systems are:

1. A self stabilizing system need not be initialized to any particular configuration. When the system is started from an arbitrary configuration it self stabilizes into a legal configuration.

2. A self stabilizing system can recover from transient bugs, bugs which change the state of one or more components of the system but keep those components in working order. When a transient bug occurs the system looses its consistency and enter a somewhat arbitrary configuration, while all its components are still correct. Following the occurrence of a transient bug a self stabilizing system will reach a legal configuration and continue execution.

In most papers on self stabilization, distributed systems are modeled by a set of state machines. Each state machine resides on a node of the system’s Communication Graph. Its transition function depends on its own state, and on the states of its neighbors which it can observe. The terms system and protocol are used interchangeably to denote a set of some specific state machines which operate on some communication graph.

In his pioneering paper [Di74], Dijkstra defined the class of self stabilizing protocols and presented three self-stabilizing protocols for the mutual exclusion problem. Dijkstra’s protocols work for a system whose communication graph is a directed ring. A processor in such a system knows which edge enters its node and which edge is leaving it, communication along such an edge goes in both directions. Following Dijkstra several more works were done in this area. Self stabilizing protocols for mutual exclusion on a directed ring (or a directed chain) are presented in [BGW87], [BP87], and [Bu87]. A self stabilizing protocol for some variant of the mutual exclusion problem on tree structured systems is presented in [Kr79]. A self stabilizing mutual exclusion protocol for systems with arbitrary communication graphs is presented in [Tc81]. This protocol requires an extensive programming work for every individual system since it is recursively composed of protocols defined over covering subgraphs of the original communication graph.

All self stabilizing protocols in the references above are variants of mutual exclusion protocols. This trend is so strong that many times, mutual exclusion is presented as an inherent requirement for self stabilization. In this
paper we redefine the concept of self stabilizing protocol. In the new definition the requirements for self stabilization of a protocol are defined in a general way and are separated from its task (see [KP89], [BGM89] for other such definitions). Furthermore, we present a self stabilizing protocol whose task is to find a spanning tree of the system’s communication graph.

To model the behavior of a real distributed system we use the interleaving model. In this model processors are activated by a scheduler according to some rules. Whenever a processor is activated by the scheduler it executes a single atomic step which is the "smallest" step that a processor executes uninterrupted. To ensure correctness of our protocols we regard the scheduler as an adversary and require the protocol to be correct in all possible executions. The nature of the rules by which the scheduler activates processors and the atomicity level it allows, determines its power. We hereby list three common types of schedulers which are used in the literature and describe their properties.

**Central Demon:**

This scheduler is defined by [Di74]. In each activation it activates a single processor. The activated processor executes an atomic step which is composed of reading the states of all its neighbors, deciding whether to change its state, and then move to its new state. During execution of this step no other processor is allowed any action.

Such an atomicity may pose difficult hardware problems and may cause delays in execution. Subsequent protocols were designed to avoid those implementation difficulties, by assuming a weaker type of scheduler, called:

**Distributed Demon:**

This scheduler defined in [BGW87] and further studied in [Bu87]. In each activation the distributed demon activates a subset of the system’s processors. All activated processors execute a single atomic step simultaneously. That is, they all read the states of all their neighbors, decide whether to change their state, and then move to their new state.

In this work we further relax the limitations on the scheduler. We assume the most basic atomicity level by defining the more realistic:

**Read/Write Demon:**

In this model a processor has no direct access to the state of its neighbors. The only way to pass information
from one processor to another is via the use of shared registers. The only atomicity requirement is: All shared registers are serializable with respect to read/write operations. This fine level of atomicity allows us to assume, without loss of generality, that at any given time exactly one atomic step is executed in the entire system. Hence the read/write demon always activates a single processor. The activated processor executes some internal computation followed by either a write operation or by a read operation, but not by both. The present work is the first time in which read/write demon is used in the context of self stabilization.

The use of read/write demon introduces a new kind of difficulties in implementing protocols, due to the possibility that a processor's move may be affected by outdated values. In fact, we will show that protocols that are self stabilizing in a trivial sense under distributed demon, are not self stabilizing in the presence of read/write demon.

The use of shared registers enables us to considerably extend the applicability of self stabilizing protocols, which is another point we address here. As will be shown, the previous models are quite limited. In particular, in these models, impossibility of symmetry breaking prevents the existence of self stabilizing mutual exclusion protocols in certain topologies (e.g. undirected rings), even in the presence of the central demon. The use of a separate register for each edge in our system, enables us to break symmetry by letting the processors write different values to (registers read by) different neighbors. Moreover, in our model it is possible to design self stabilizing protocols in which the program of each processor depends only on the number of links connected to it (and their directions, if they are directed). Using this property, dynamic protocols in which a processor's program accommodates the changes in its local topology (the number and direction of its links) can be designed. Such protocols can be applied to dynamic systems, whose topology may change during execution. No reinitialization is needed following any such change in topology.

Using read/write demon, we present three self stabilizing dynamic protocols. The task of the first protocol is to construct a rooted spanning tree of the system's communication graph. This protocol does not assume any special communication graph. The second protocol achieves mutual exclusion on tree structured systems. Both these protocols assume the most basic atomicity level, that is read and write atomicity only. We then proceed to introduce Fair Protocol Composition as a basic design tool for self stabilizing protocols. This work is concluded by using fair protocol composition to combine the two aforementioned protocols into a single self stabilizing dynamic protocol for mutual exclusion in general graphs. This last protocol enjoys the same atomicity and dynamic properties as its two subprotocols.
The rest of this paper is organized as follows: in Section 2 the formal model and formal requirements for self stabilization are presented. In Section 3 we discuss the relation between our model and previous models which were used for self stabilizing systems, and give some evidence to justify our choice of model. In sections 4 and 5 self stabilizing spanning tree protocol and a mutual exclusion on a dynamic tree protocol are presented. In Section 6 the two protocols are combined to obtain the final protocol. Concluding remarks are brought in Section 7.

2. Model and Requirements

2.1. Static Systems and Protocols

A distributed system consists of \( n \) processors, denoted by \( P_1, P_2, \ldots, P_n \). Each processor resides on a node of the system's Communication Graph. Some (or all) links of the communication graph may be directed, meaning that for each such link the two processors on its endpoints agree on its direction: one is designated as the link's head and the other as the link's tail. Two processors which reside on two neighboring nodes of the communication graph are called neighbors. Neighbors communicate between themselves by the use of shared communication registers. Every communication register is atomic (serializable) with respect to read and write operations, that is all read and write actions to the same register can be serialized in time. For the rest of this paper we use the term register for communication register. For every processor \( P \), the registers of \( P \) are those registers to which \( P \) writes. Each processor is a RAM (as defined in first section of [AHU74]). The processors are anonymous, they do not have identities; the subscripts 1, \( \ldots \), \( n \) are used for ease of notation only. The program of each processor is composed of atomic steps. An atomic step of a processor consists of an internal computation followed by a read or a write action, but not both. Each processor has an internal register called program counter (pc). The pc of each processor always points to the next atomic step to be executed by that processor.

A protocol for a system is a collection of programs for all processors in it. When no ambiguity occurs, we identify processors with their programs and refer by \( P_i \) to the \( i \)-th processor together with its program. One can look at such a processor as a state-machine. A state, \( s_j \), of a processor, \( P_i \), is defined by the contents of its memory, including the pc. In some cases, like in the protocol presented in section 5, the processor's states are not all possible memory configurations, but only a subset of these configurations that satisfy certain properties. In such cases one should prove that these properties are invariants of the protocol. Denote by \( S_i \) the set of states of \( P_i \). Given a system we associate with each shared register \( r \) the set of symbols \( \Sigma_r \) that can be stored in \( r \) (the set \( \Sigma_r \) is not necessarily
finite). A configuration of the system is the vector of states of all processors and the contents of all registers. Denote by $C = (S_1 \times S_2 \times \cdots \times S_m \times \Sigma \times \cdots \times \Sigma)$ (where $m$ is the number of shared registers in the system), the set of all possible configurations of the system.

Processor activity is managed by the read/write demon. In any given configuration the demon activates a single processor which executes a single atomic step. Let $c_1$ and $c_2$ be two configurations of the system where $c_2$ is reached from $c_1$ by a single atomic step of a single processor. We denote this fact by $c_1 \rightarrow c_2$. An execution $E$ of the system is an infinite sequence of configurations $E = c_1, c_2, \cdots$ where $c_i \rightarrow c_{i+1}$ for $i = 1, 2, \cdots$. A schedule is a sequence of processor subscripts. For any execution $E$, we define the schedule of $E$ as the sequence of processor subscripts, which correspond to the order of the atomic steps taken by the processors during $E$. Note that an execution is uniquely determined by its initial configuration and by its schedule. For example if $E$ is started by an action of $P_j$ then the subscript $j$ is first in its corresponding schedule. An (infinite) schedule $S$ is fair if every processor subscript appears in $S$ infinitely often. A fair execution is an execution whose schedule is fair.

2.2. Self Stabilization

In this section we define the requirements from a self stabilizing distributed protocol. The new definition includes the one in [Di74], but appears to be more general. A self stabilizing system is characterized by its capability of demonstrating behavior sometimes after it is started from any arbitrary configuration. A natural way to model legal behavior of a system is by specifying a set of legal sequences of configurations. Each execution of the system should eventually demonstrate legal behavior by having some legal sequence as its suffix. Hence we define a task $LS$ to be a set of sequences of system configurations which are called legal sequences. The set of legal sequences which constitute some task depends both on the aim of the protocol and on the system. For instance, the mutual exclusion task for some system is identified with the set of sequences of system configurations which satisfy: In each configuration there is at most one processor in its critical section, and every processor is in its critical section in infinitely many configurations. In this spirit we define:

[Self Stabilization]

Let $LS$ be a task, that is a set of sequences of system configurations. A protocol for the system is self stabilizing with respect to $LS$ if every infinite fair execution of the protocol has a suffix in $LS$. 
This definition is not necessarily the strongest definition for self stabilization. A stronger one uses the notion of a safe configuration: A configuration $c$ of a system is safe with respect to a task $LS$ if any fair execution of the protocol, starting from $c$ belongs to $LS$. Our definition does not exclude from $LS$ sequences in which no safe configuration is ever reached. In executions which do not include a safe configuration there is always a danger that the system "deviates" to a "bad" configuration. In any fair execution the number of such deviations is guaranteed to be finite but the system is never "safe". We choose to adopt this definition since it appears to make the presentation clearer. However, as we shall point out, our protocols are self stabilizing also according to the following stronger definition:

[Strong Self Stabilization]
A protocol is strongly self stabilizing if starting from any system configuration, it eventually reaches a safe configuration.

A distinction between two different kinds of self stabilizations appears first in [BGM89]. There, the two kinds are called Pseudo Self Stabilization and Self Stabilization. The distinction between self stabilization and strong self stabilization in our definitions is similar to that of [BGM89].

2.3. Dynamic Systems

A system is dynamic if its communication graph may change during execution. The changes we allow are any kind of addition or deletion of a link or a node of the system's communication graph. Changes to a self stabilizing system should be regarded as just another kind of transient bugs. In other words, the self stabilizing properties of the system should be strong enough to allow the system to stabilize itself some time after the last topology change is done. A topology change should be a very local matter. We would not wish, for instance, that due to a topology change in one part of the system many processors in other parts of the system would change their programs. Also we would not wish the program of a processor to depend on its location in the network. Therefore, it is very natural to require that the program of a processor depends only on the number and directions of its attached links.

Dijkstra [Di74] has noticed that when all processors are identical, there are topologies for which it is impossible to break symmetry. To overcome this difficulty we adopt the approach of [Di74] and subsequent works, and assume that exactly one of the processors in the system is a special processor, the rest of the processors are identical. Our protocols should be correct for any possible choice of special processor. These considerations are summed up in the following Uniformity condition:
[Uniformity]

(a) Exactly one processor in the system is a special processor.
(b) The program of each processor depends only on its being or not being a special processor, and the number and directions of its incident links. (i.e., the number and directions of incident links are considered as a parameter of the program of each processor).

3. Discussion of the Model

A mutual exclusion protocol for a distributed system can be defined as follows: Each processor in the system has a critical section of code (which is, in general, not a part of the protocol). It is required that, during execution of the protocol, at any given configuration there is at most one processor in its critical section, and that every processor that wishes to enter its critical section, eventually enters it. A processor which can either enter its critical section or which is already in its critical section is called privileged. This definition implies that a processor can decide from its local variables whether it is privileged or not. Furthermore, the same definition also implies that a privileged processor cannot lose its privilege by any action of another processor. The only way in which a privileged processor may become unprivileged is by changing the values written in some of its registers, since this is the only way in which a processor can inform the outside world that it is not in its critical section any more.

In most of the previous works on self stabilizing protocols for mutual exclusion (e.g. [Di74, Kr79, BGW87, Bu87]), the following three assumptions are used (sometimes implicitly):

(a) All processors but one are identical.
(b) Each processor can write to a unique register, called its state. This register is accessible for reading by the processor’s neighbors.
(c) Either a central demon or a distributed demon is used. In both models a processor may read its neighbors’ states and move to a new state in a single atomic operation. For this reason these models do not require the use of internal variables to store the values read by a processor.

Assumption (a) above was mentioned in the previous section, and is also used in our model. Our model does not use assumption (b). Instead it assumes that communication along each link is conducted by two registers. Each of these registers is written to by one processor and read from by the other. The reason for this change is demonstrated by the following lemma, which shows that assumption (b) severely restricts the topologies for which mutual
Lemma 3.1: Under assumption (a) - (b) above, there is no self stabilizing mutual exclusion protocol for undirected rings, or for rings whose edges are directed but not necessarily in a consistent way.

Proof: Consider a four processor ring as described in Figure 3.1 where \( P_2 \) and \( P_3 \) are identical processors. Assume that \( P_2 \) and \( P_3 \) "think" that \( P_1 \) is their left neighbor and \( P_4 \) is their right neighbor. Starting the system with \( P_2 \) and \( P_3 \) in the same state it is easy to see that in the (fair) schedule \( (P_1,P_4,P_2,P_3) \) every step taken by \( P_2 \) is followed by the same step taken by \( P_3 \). Hence \( P_2 \) and \( P_3 \) either enter the critical section together infinitely often, or are starved. In both cases the system never demonstrates a normal behavior or in other words: For this system there exists no self stabilizing protocol for mutual exclusion. Similarly, in case of a directed tree system which is not a directed path (e.g. the system depicted in Figure 3.2) no self stabilizing mutual exclusion protocol exists. For this reason in [Kr79], where these systems were studied, did not solve the mutual exclusion problem but some variant of it.

The impossibility result of Lemma 3.1 holds even in the presence of a central demon. In contrast, using a separate register for each neighbor enables us to present such protocols for every (in fact, for dynamic) topology, assuming only read/write atomicity.

Consider now assumption (c) above. Two types of schedulers were assumed in previous works: The central demon and the distributed demon. Brown, Gouda and Wu [BGW87], and Burns [Bu87], designed protocols that have the following non-interfering property: Once a processor can move from one state to another, it will make that move regardless of any further steps taken by other processors. In [BGW89] it is proved that a self-stabilizing non-interfering protocol is guaranteed to stabilize under distributed demon.

It should be noted that one step of the distributed demon in which \( m \) processors move simultaneously can be simulated by our model: First let each of the \( m \) processors read all its neighbors' states, and then let them all move to their new states. Using this simulation, it is not hard to show that every mutual exclusion protocol that is self stabilizing for read/write demon is also self stabilizing for distributed demon. The opposite is not true, since read/write demon can generate more general schedules: For example a schedule of two processors \( P_1 \) and \( P_2 \), starting with \( P_1 \) reading, then \( P_2 \) writing and reading, and then \( P_1 \) writing. We demonstrate below that a repeated application of
such a schedule may prevent stabilization from a protocol that is trivially self stabilizing when assuming distributed (or central) demon. We denote the protocol used for this demonstration as the balance-unbalance protocol. This is probably the simplest possible protocol for mutual exclusion. It is designed for a system of two processors, $P_1$ and $P_2$, which are connected by a link. Each processor has two states, denoted by 0 and 1. The configuration of a system is defined by the states $(s_1,s_2)$ of $P_1$ and $P_2$ respectively. Thus, the system has four possible configurations: $(0,0),(1,0),(1,1),(0,1)$. The transition function of $P_1$ unbalances the system i.e. transfers $(0,0)$ to $(1,0)$ and $(1,1)$ to $(0,1)$, the transitions of $P_2$ balance the system i.e. transfer $(0,1)$ to $(0,0)$ and $(1,0)$ to $(1,1)$. $P_1$ ($P_2$) is privileged iff the link is balanced (unbalanced).

Insert Figure 3.3 here

Consider now an execution of the protocol, assuming distributed (or central) demon. In any possible configuration, exactly one of the processors can change its state - the privileged processor. By changing its state, it passes the privilege to its neighbor. Thus, starting with any configuration of the system, and regardless the specific behavior of the demon, the system will repeatedly change its configuration according to the cycle $((0,0), [P_1 \text{ writes}], (1,0), [P_2 \text{ writes}], (1,1), [P_1 \text{ writes}], (0,1), [P_2 \text{ writes}], (0,0))$. Therefore, this protocol is a self stabilizing mutual exclusion protocol in the strongest possible sense: There is a unique legitimate sequence of configurations, such that every possible execution of the protocol is a suffix of this sequence. In a way, this protocol is well known and is a simplified version of protocols presented in [Di74], [Kr79], [PF77], [Bu87].

Consider now executions of this protocol under the read/write demon. The behavior of each processor is now determined not only by its state and its neighbor state. It is determined also by the next atomic step to be executed, i.e. read or write. Moreover, in order to determine the value that a processor is going to write, one must also know the last value read by it.

To avoid confusion, we use here the term status to denote the full information describing the processor behavior, while reserving the term state to denote the values 0 or 1 written in its register. The status of processor $P_1$ is described by the following components: (the state of $P_1$, the last state of $P_2$ read by $P_1$, the next action to be executed by $P_1$). A sample execution of the balance-unbalance protocol is demonstrated in Figure 3.3. In this figure each processor is represented by a circle; the state of a processor is written in the box adjacent to that circle; the last value read by the processor and the next atomic operation it is going to perform are written in that circle.
A configuration of the system is a pair of the processors statuses (shown in Figure 3.3 as a horizontal pair of processors). It should be noted that now it is possible to have statuses in which both processors are privileged, e.g. it is possible to have the status \((0,0,\text{write}) (0,1,\text{write})\), meaning that the actual states of \(P_1\) and \(P_2\) are both \(0\), but as a result of a transient bug, \(P_2\) "thinks" that the state of \(P_1\) is \(1\) (this status is depicted in the uppermost pair of Figure 3.3). Note that in this configuration both \(P_1\) and \(P_2\) are privileged, and hence the system is not stabilized. We bring here a fair execution, starting from the above configuration, in which the system never stabilizes. In fact during this execution the two processors are simultaneously privileged infinitely often.

\[
\begin{align*}
&=(0,0,\text{write}) (0,1,\text{write})][P_2 \text{ writes}],
&(0,0,\text{write}) (1,1,\text{read})[P_2 \text{ reads}],
&(0,0,\text{write}) (1,0,\text{write})][P_1 \text{ writes}],
&(0,1,\text{read}) (1,0,\text{write})[P_1 \text{ reads}],
&(1,1,\text{write}) (1,0,\text{write})[P_2 \text{ writes}],
&(1,1,\text{write}) (0,0,\text{read})[P_2 \text{ reads}],
&(1,1,\text{write}) (0,1,\text{write})[P_1 \text{ writes}],
&(0,1,\text{read}) (0,1,\text{write})[P_1 \text{ reads}],
&(0,0,\text{write}) (0,1,\text{write})[P_2 \text{ writes}],
\end{align*}
\]

The first four operations of this execution are depicted in Figure 3.3. In this figure privileged processors are double circled, the processor which is scheduled to operate is pointed to by an arrow.

The part of the execution presented above starts and terminates with the same configuration and can be duplicated infinitely often to achieve an infinite fair execution. In half of the configuration in this infinite execution \(P_1\) and \(P_2\) are about to change their state simultaneously, and hence are both privileged, and may enter their critical section together.

Our self stabilizing protocol for mutual exclusion in directed trees, presented in Section 5, is based on a corrected version of the balance-unbalance protocol.

4. A Spanning Tree Protocol

In this protocol every pair of neighbors, \(P_i\) and \(P_j\) communicate by using two shared registers, \(r_{ij}\) and \(r_{ji}\). Processor \(P_i\) \((P_j)\) writes into \(r_{ij}\) \((r_{ji})\) and reads from \(r_{ji}\) \((r_{ij})\). All links incident to each processor \(P_i\) are ordered by some arbitrary ordering \(\alpha_i\). The collection of all these orderings is denoted by \(\alpha = (\alpha_1, \cdots, \alpha_n)\). The ordering \(\alpha_i\) induces in a natural way an ordering of the neighbors of \(P_i\).

The spanning tree protocol is a distributed BFS protocol. The output spanning tree is a BFS tree of the communication graph of the system. This spanning tree is rooted at the special processor of the system, which is called the root processor.
A graph may have more than a single BFS tree rooted at the same node. Let \( G(V, E) \) be a graph with orderings \( \alpha=(\alpha_1, \alpha_2, \ldots, \alpha_n) \) of the neighbors of each node \( v \in V \) and a root \( v_1 \in V \). Define the First BFS Tree of \( G \) relative to \( v_1 \) and \( \alpha \) to be a BFS tree, rooted at \( v_1 \). In case a node, \( v_i \) of distance \( d+1 \) from \( v_1 \) has more than one neighbor of distance \( d \) from \( v_1 \), \( v_i \) is connected to its first neighbor, according to \( \alpha_i \), among all its neighbors whose distance from \( v_1 \) is \( d \). The protocol always produces the First BFS Tree of the system’s communication graph, with respect to the root processor and to the (arbitrary) orderings \( \alpha \) of the neighbors of every processor.

Each register \( r_y \) is composed of two fields. The \( r_y.father \) field, which is a binary field, and the \( r_y.distance \) field which stores a positive integer. Once the output tree is constructed it is encoded by means of the registers as follows: If the value of \( r_y.father \) is 1 then (the node of) \( P_j \) is the "father" of (the node of) \( P_i \) in the tree. At the same time \( r_y.distance \) holds the distance between \( P_i \) and the root processor.

Each processor \( P_i \) keeps an internal variable for each neighbor. The internal variable, corresponding to the neighbor \( P_j \) is denoted by \( ir_{ij} \). This variable stores the last value of \( r_{ij} \) that was read by \( P_i \). The two fields of \( ir_{ij} \) are denoted by \( ir_{ij}.father \) and \( ir_{ij}.distance \) respectively. The code of the protocol, for the root and for the other processors, appears in Figure 4.1. In writing the code for processor \( P_i \) we assume that it has \( k \) neighbors, ordered according to \( \alpha_i \). The purpose of the boolean variable \( first\_found \) is to make sure by the end of each pass of the loop that each processor has a single father.

**Figure 4.1: The Spanning Tree Protocol for \( P_i \).**

```plaintext
Root: do forever
  for m:=1 to k do write \( r_{im} := <0,0> \);
  od

Other: do forever
  for m:=1 to k do ir_{im} := read (r_{mi});
  first\_found := FALSE;
  (*)
  dist := min (ir_{mi}.dist) + 1;
  for m:=1 to k do
    if not first\_found and ir_{mi}.distance = dist - 1
      then
        write \( r_{im} := <1,\text{dist}> \);
        first\_found := TRUE;
      else
        write \( r_{im} := <0,\text{dist}> \);
      od
  od
```
In order to prove that the protocol is self stabilizing we first define the task LST of legal sequences. We choose to include in this set every sequence of configurations in which every configuration encodes the first BFS tree as defined above. The following lemma essentially shows that the spanning tree protocol is self stabilizing relative to LST.

Lemma 4.1: Let \( P_i \) be an arbitrary processor whose distance from the root is \( l \) and let \( P_j \) be an arbitrary neighbor of \( P_i \). Let \( E = c_0, c_1, \cdots \) be an arbitrary fair execution of the protocol. For every integer \( d \geq 0 \) there exist \( t_d \) such that:

For every integer \( t, t \geq t_d \), the configuration \( c_t \) satisfies the following assertions:

(a) If \( l \leq d \) then \( r_{ij}.\text{distance} = l \).

(b) If \( l \leq d \) then \( r_{ij}.\text{father} \) has the "right" value. That is: if \( P_j \) is the first neighbor of \( P_i \) (using \( \omega \)) of distance \( l-1 \) from the root then \( r_{ij}.\text{father} = 1 \), and otherwise \( r_{ij}.\text{father} = 0 \).

(c) If \( l < d \) then \( r_{ij} = r_{ji} \).

(d) If \( l > d \) then \( r_{ij}.\text{distance} > d \).

Proof: We prove the theorem by induction over \( d \). In the proof we use the fact that due to the fairness of \( E \) every processor is activated in \( E \) infinitely often.

Base Case \((d=0)\): The only node of distance 0 from the root is the root itself. Assume that the root has \( k \) neighbors. After \( k \) activations of the root all its registers store the value \( <0,0> \). The values stored in the registers of the root will not be changed any more. This completes the proof of assertion (a). Assertion (b) is also satisfied since no processor is the "father" of the root. Assertion (c) holds vacuously for the base case, since there are no processors of distance <0 from the root.

To prove assertion (d) note that all values stored in the distance fields of all registers are non-negative. Therefore the value of the variable dist computed in line (*) of the protocol of all non-root processors is always \( > 0 \). In particular whenever a value is written in any register by a non-root processor its distance field is always \( \geq 1 \). Again, by the fairness of \( E \) each processor eventually writes in all its registers. Once this happens the distance fields of all registers of all non-root processors remain positive forever, hence assertion (d) is also satisfied.

Induction Step: Let \( t_d \) be an integer such that for every \( t \geq t_d \), configuration \( c_t \) satisfies assertions (a) – (d) for some integer \( d, d > 0 \). We show the existence of some integer \( t_{d+1} \) such that for every integer \( t \geq t_{d+1} \) configuration \( c_t \) satisfies assertions (a) – (d) for \( d+1 \).
If the distance of $P_i$ from the root is $d+1$ then all its neighbors are of distance $\geq d$ from the root. Moreover $P_i$ has at least one neighbor, whose distance from the root is exactly $d$. By assertion (a) of the induction hypothesis, for every $P_k$ of distance $d$ from the root, it holds that the value stored in $r_{\mu}.distance$ in $c_k$ and all subsequent configurations is $d$. By assertion (d) of the induction hypothesis, for every $P_i$ of distance $> d$ from the root, it holds that the value stored in $r_{\mu}.distance$ in $c_i$ and all subsequent configurations is $> d$. Therefore, whenever $P_i$ executes line (*) after $c_i$, the value assigned to the variable $dist$ is exactly $d+1$. Once this value is written to all registers of $P_i$, assertions (a) and (b) hold for $P_i$. The same holds for all processors of distance $d+1$ from the root. Hence there is a configuration $c_1$ reached by the system, such that for every configuration $c$ following $c_1$, assertions (a) and (b) are satisfied for all processors of distance $d+1$ from the root.

It is easy to see that from $c_1$ and onwards forever, the values stored in the registers of all processors of distance $d+1$ from the root will not be changed any more. In particular all neighbors of all processors of distance $d$ from the root will not change the values stored in their registers any more. If $P_i$ is a processor of distance $< d+1$ from the root then each read action after $c_1$ sets one of its internal variables to its final stationary value. This again is true for all processors of distance $< d+1$ from the root. Thus there is a configuration $c_2$ reached by the system, such that every configuration $c$ following $c_2$ satisfies assertion (c) for $d+1$.

Let $P_k$ be an arbitrary processor of distance $> d+1$ from the root. The neighbors of $P_k$ are all of distance $\geq d+1$ from the root. By assertion (d) of the induction hypothesis starting from $c_i$ and onwards each neighbor $P_j$ of $P_k$ satisfies $r_{\mu}.distance > d$. Therefore, whenever $P_j$ executes line (*) after $c_i$, the value assigned to the variable $dist$ is $> d+1$. Once this value written to all registers of $P_k$ assertion (d) is satisfied for $P_k$. The same holds for all processors of distance $> d+1$ from the root. Hence there is a configuration $c_3$ reached by the system, such that every configuration following $c_2$ satisfies assertion (d) for $d+1$. Let $c_{\mu_1}$ be the later configuration among $c_2$ and $c_3$. It is easy to see that indeed every configuration $c$ following $c_{\mu_1}$ satisfies assertions (a)–(d) for $d+1$.

Corollary 4.2: The protocol presented above is self stabilizing relative to the set $LST$.

A sample execution of the protocol is demonstrated in Figure 4.2. For simplicity of presentation it is assumed that this execution is under distributed demon. That is, each activation of a processor consists of reading all neighbors' registers and writing to all its registers. In this figure configurations are ordered clockwise starting from the upper left corner. Processors are drawn as framed squares, where the root lies in the lower right corner. The links of each pro-
cessor are ordered clockwise starting from its upper left corner. The numbers in the center of each processor represent the distance field of its registers. Processors which can change some register are marked by a boldface frame. The links which belong to the tree are also boldfaced. Before stabilization is reached those links do not necessarily form a spanning tree.

Insert figure 4.2

Note: The spanning tree protocol bears some similarity to the ARPANET routing protocol [Ta81] (that latter protocol assumes a different model).

5. Mutual-Exclusion Protocol for Dynamic Tree Systems

In this section we present a self stabilizing mutual exclusion protocol for systems whose communication graph is a directed tree, rooted at the (special) root processor. Let \( P_i \) be an arbitrary processor. The sons of \( P_i \) are ordered from left to right by some arbitrary order \( \alpha_i \). After stabilizing, an execution of the protocol proceeds in phases. In each phase each processor becomes privileged at least once. The first privileged processor in each phase is the root. Following its first activation the root (recursively) passes the privilege to (the subtrees rooted at) its sons in a left to right order. A processor \( P_i \) that becomes privileged passes the privilege to its leftmost son. Once the privilege is passed to all processors in the subtree rooted at this son it is returned to \( P_i \). Subsequently the privilege is passed to the second from left son of \( P_i \) and so on. The phase ends when the rightmost son of the root returns the privilege to the root itself. Hence execution of each phase corresponds to a DFS tour of the whole tree.

Let \( e=(P_f\rightarrow P_s) \) be a link of the communication tree directed from a father \( P_f \) to its son \( P_s \). The link is implemented using two registers, the instruct register, \( r_{\text{in}} \), is written to by \( P_f \) and read from by \( P_s \). The confirm register, \( r_{\text{co}} \), is written to by \( P_s \) and read from by \( P_f \). Both registers have a color field whose possible values are 0 and 1. In the sequel we refer to the value stored in the color field of a register \( r \) as the color of \( r \). In case the color of \( r_{\text{in}} \) is equal to the color of \( r_{\text{co}} \) we say that \( e \) is balanced and that its color is the common color of its two registers, otherwise \( e \) is unbalanced. The privilege is passed from \( P_f \) to \( P_s \) by unbalancing \( e \). The privilege is returned from \( P_s \) to \( P_f \) by rebalancing \( e \). At the beginning of each phase the color of all registers of the system is the same. If this value is 0 (1) we say that the tree is colored by 0 (1). Each phase recolors the tree from its current color to the opposite color. In addition to its color field each instruct register has a binary close field which is essential in assuring correctness of
the protocol in the presence of read/write atomicity.

Let \( P_i \) be an arbitrary processor. For each neighbor \( P_j, P_k \) keeps in its local memory two internal variables, \( ir_{ji} \) and \( or_{ji}' \). These internal variables function as images of the color of \( r_{ji} \) and \( r_{ji}' \) respectively. Whenever \( P_i \) reads \( r_{ji} \) it assigns the read value into \( ir_{ji} \). Similarly, whenever \( P_i \) writes into \( r_{ji}' \) it assigns \( r_{ji}' \) the contents of \( or_{ji}' \). A pictorial representation of a processor \( P_i \) together with its father \( P_f \) and one of its three sons \( P_s \) appears in Figure 5.1.

The state set of \( P_j \) is \( S_j \). Assume that \( P_j \) has \( l \) sons. The set \( S_j \) is defined by the value of \( P_j \)'s pc, by the values of the variables \( or_{11}, \ldots, or_{il} \) and \( ir_{11}, \ldots, ir_{il} \) corresponding to \( P_j \)'s outgoing links, and in case \( P_j \) is not the root, by the value of the variables \( ir_j \) and \( or_j \) corresponding to its incoming link. The states in \( S_j \) satisfy the following properties:

1. For some \( j_0, 1 \leq j_0 \leq l \), all variables \( or_{j0} \), \( 1 \leq j < j_0 \) have the same color. All variables \( or_{j}, 1 \leq j \leq l \) have the opposite color. When \( j_0 > 1 \), the registers \( r_{j_0-1} \) and \( r_{j_0} \) are called the **left border register** and **right border register**, respectively.

2. If \( P_i \) is not a root processor and if \( j_0 \) of \( \{j\} \) above is larger than 1 then the value of \( or_j \) is equal to that of \( or_{j_0} \).

3. Any combination of values of \( ir_{11}, \ldots, ir_{il} \) is allowed.

In Lemma 5.1 it will be shown that the above properties are invariants of the protocol.

The set \( C \) of system configurations consists of all possible combinations of processor states and register values.

Note that we allow configurations in which the value of the register \( r_j \) is not equal to the value of the corresponding internal variables \( or_j \) and \( ir_j \). In other words: It is not assumed that a processor has a priori knowledge of the actual value written in any register.

The code of the protocol, for the root and for the other processors, appears in Figure 5.2. The code is written for \( P_i \) with \( l \) sons. For simplicity, we denote the \( j \)th son of \( P_i \) by \( P_j \). The father of \( P_i \) is \( P_f \). A root processor is privileged if its \( pc \) points at line 22 and the predicate \( all\_out\_links\_balanced \) holds for it (note that the predicates \( in\_link\_unbalanced \) and \( not \) \( (done) \) are always true for it). A normal processor is privileged if its \( pc \) points at line 13 and the predicates \( all\_out\_links\_balanced, in\_link\_unbalanced \) and \( done \) are true, or if its \( pc \) points at line 22 and the predicates \( all\_out\_links\_balanced \) and \( in\_link\_unbalanced \) are true, and the predicate \( done \) is false for it. We shall
show that eventually, only privileged processors may change the color of their registers. In addition we define an active processor as a processor whose all outgoing links are balanced, and its incoming link is unbalanced. The root processor is active whenever all its outgoing links are balanced, a leaf processor (i.e. processor without any sons) is active whenever its incoming link is unbalanced. While the definition of a privileged processor depends only on its state and not on the value of any register, the definition of an active processor depends on the values of the registers of its links. In the sequel we will show that once the system stabilizes, every configuration has exactly one active processor and that only an active processor might be privileged.

1 Root: do forever
2 for m:=1 to l do write \( r_{im} := \langle or_{im}, 0 \rangle \);  
3 for m:=1 to l do \( ir_m := \text{read}(r_m) \);  
4 if (all_out_link_balanced) then instruct_next_son;  
5 od

6 Other: do forever
7 for m:=1 to l do write \( r_{im} := \langle or_{im}, 0 \rangle \);  
8 write \( r_{j} := \text{or}_{j} \);  
9 repeat \( \) (try to read the instruct register of your incoming link)  
10 \( ir_{j} := \text{read}(r_{j}) \);  
11 until \( ir_{j}.\text{close} = 0 \);  
12 for m:=1 to l do \( ir_{m} := \text{read}(r_{m}) \);  
13 if (all_out_links_balanced) and (in_link_unbalanced)  
14 then \( \) (you are privileged)  
15 if done then begin \( or_{j} := ir_{j}.\text{color} \); write \( r_{j} := \text{or}_{j} \) end; \( \) (balance incoming link)  
16 else instruct_next_son; \( \) (unbalance next outgoing link)  
17 od

18 Procedure instruct_next_son: \( \) (make your next son active by unbalancing its incoming link)  
19 \( \) (in this procedure, the value of \( j_0 \) is assumed to be as defined by property (p1) above)  
20 write \( r_{j_0}.\text{close} := 1 \);  
21 \( ir_{j_0} := \text{read}(r_{j_0}) \);  
22 if (all_out_link_balanced) and (in_link_unbalanced) and (not done)  
23 then begin \( or_{j_0} := \text{not} or_{j_0} \); write \( r_{j_0} := \langle or_{j_0}, 0 \rangle \); end  
24 else write \( r_{j_0}.\text{close} := 0 \);  

Figure 5.2: The Mutual-Exclusion protocol for Dynamic Tree Systems.
In the protocol several predicates are used. These predicates, which are defined over the possible values of internal variables only, are:

1. \textit{all_out_links_balanced} holds if \( or_{im} = ir_{mi} \) for \( m = 1, \ldots, l \). (this holds vacuously for a leaf, i.e. when \( l = 0 \)).
2. \textit{in_link_unbalanced} holds if \( or_y \neq ir_y.color \). (this predicate is always true for the root processor).
3. \textit{done} is true when for \( m = 1, \ldots, l \), \( or_{sm} = or_{i1} \) and \( or_y \neq or_{i1} \) (i.e., when all sons have already been instructed; this predicate is always true for a leaf, and is always false for the root).

In order to prove that the protocol is self stabilizing we first define the set \( LSM \) of legal sequences of system configurations. Any sequence \( s \in LSM \) satisfies the following:

- **[Exclusion]**: In each configuration \( c \) of \( s \) at most one processor is privileged.
- **[Fairness]**: During \( s \) each processor is privileged infinitely often.

**Lemma 5.1:** \([\text{closure of legitimate states}]\): If a processor \( P \) is at a state satisfying properties (p1) - (p3) above, then every subsequent state of \( P \) also satisfies (p1) - (p3).

**Proof:** By induction on the number of atomic steps executed by the processor \( P \). Assume that (p1) - (p3) hold after \( i-1 \) steps were executed, and consider \( P \) after executing the \( i \)th atomic step.

Clearly, the only properties that have to be taken care of are (p1) and (p2). These properties are given in terms of the values of the variables \( or_y \). Thus, if none of these variables is changed during the \( i \)th step then the claim holds by induction. Otherwise, one of these variables is updated in line 15 or line 23 of the protocol. Assume now the value of \( or_y \) is updated by executing line 15. This could happen only if the predicate \textit{done} is true before the value of \( or_y \) is changed. But this implies that after \( or_y \) is changed all the variables \( or_y \) have the same color, and hence (p1) and (p2) hold trivially. The case where one of the variables \( or_y \) is changed by executing line 23 is proved similarly. \( \square \)

In the following lemmas we use \( E \) to denote a fair arbitrary execution.

**Lemma 5.2:** If during \( E \) the colors of all registers in the system are constant then there exists a configuration \( c \) in \( E \) such that for every subsequent configuration \( c_u (u \geq i) \) the value of the \textit{close} field of all instruct registers is 0.

**Proof:** We prove the lemma by showing that for every processor \( P_i \) in the system there is an index \( o(i) \) (\( o(i) \) depends on \( E \)) such that in the configuration \( c_{o(i)} \) of \( E \), the value of the \textit{close} field of all the instruct registers of \( P_i \) is 0 and that this value is not changed in any configuration in \( E \) after \( c_{o(i)} \). The proof proceeds by induction on \( d \), the distance of \( P_i \) from the root.
Base Case: $d=0$. In this case $P_1$ is the root processor. The program for the root processor is a single loop. By the fairness of $S$ it holds that during $E$, the root executes this loop infinitely often. In particular lines 2 and 3 of the root's program are executed infinitely often. In line 2 the root sets the value of the `close` field of all its instruct registers to 0. In addition in line 2 the root sets the values of the `color` field of every instruct register $r_{im}$ to the value of its internal variable $or_{im}$. In line 3 the root reads the confirm registers of all its sons; for every register $r_{mi}$, $P_1$ sets the value of the internal variable $ir_{mi}$ to the value it read from $r_{mi}$. Let $c_{v(0)}$ be the first configuration in $E$ after the root executed lines 2,3 for the first time. In $c_{v(0)}$ the values of all `close` fields in all instruct registers of the root are 0 and for every son $P_i$ of the root $ir_{mi}=r_{mi}$ and $or_{im}=r_{im}.color$. By the assumption that no `color` field is changed during $E$, any further read action does not change the value of $ir_{mi}$. By the same assumption the value of $or_{im}$ is not changed either.

The only place where the value of the `close` field of any register is set to 1 is inside the procedure `instruct_next_son`. To complete the proof it suffices to show that following $c_{v(0)}$ the root never executes this procedure. Assume towards a contradiction that `instruct_next_son` is executed by the root starting from configuration $c_k$, ($k\geq\alpha(i)$). We reach a contradiction by showing that in this case the root also changes the color of some instruct register. Following $c_{v(0)}$ `instruct_next_son` is executed only if the predicate `all_out_links_balanced` holds. While executing `instruct_next_son` the root computes the index $j_0$ of the processor which will be "instructed". After that the root checks whether the link connecting it to $P_{j0}$ is balanced. We have already proved that in every configuration that follows $c_{v(0)}$ it holds that for all $j$, $ir_j=r_j$ and $or_j=r_j.color$, hence in $c_k$ and every subsequent configuration all out links of the root are indeed balanced. In particular the link between the root and $P_{j0}$ is balanced. In this case the root instructs $P_{j0}$ by changing the color of $r_{j0}$, contradiction.

Induction Step: We assume correctness of the lemma for all processors of distance $d$ from the root. Let $P_i$ be an arbitrary processor of distance $d+1$ from the root. We show the existence of a configuration $c_{v(d)}$ after which the value of the `close` field in every instruct register of $P_i$ is 0 forever.

The program of a non-root processor $P_i$ is composed of a main loop and an inner loop. In the inner loop $P_i$ waits until it finds that the value of all `close` fields of its father $P_f$ is 0. Once this happens $P_i$ proceeds to execute the main loop. $P_f$ is in distance $d$ from the root. By the induction hypothesis there is a configuration in $E$, $c_{v(f)}$, such that in every configuration of $E$ following $c_{v(f)}$ it holds that the value of the `close` field of the register $r_f$ is 0. By the fairness of $S$, $P_f$ is activated infinitely often after $c_{v(f)}$. Since after $c_{v(f)}$ the value of the `close` field of $r_f$ is always 0, $P_i$ does not get stuck in its inner loop after $c_{v(f)}$ and executes its main loop infinitely often. Let $c_{v(0)}$ be the configuration.
reached by the system right after \( P_i \) executes lines 7 to 12 for the first time after \( c_{e(i)} \). By an argument similar to the one used in the proof of the base case it can be shown that in \( c_{e(i)} \) the value of all close fields of \( P_i \) is 0 and that these fields will not be changed in \( E \) any more. \( \square \)

**Lemma 5.3:** Eventually, the color of at least one register in the system is changed.

**Proof:** Assume towards a contradiction that \( E \) is a fair execution during which no processor changes the color of any of its registers. By lemma 5.2 there exist a configuration \( c_t (t \geq 0) \) in \( E \) such that for every configuration \( c_u (u \geq t) \) every close field holds 0.

**Case 1:** In \( c_t \) all the links of the root are balanced

By the assumption no color field is changed. Hence in any configuration in \( E \) all outgoing links of the root are balanced. The program for the root processor consists of a single loop. By the fairness of \( S \) this loop is executed infinitely often during \( E \). In particular lines 2 and 3 are executed infinitely often. Whenever these two lines are executed the root reads the registers of all its sons. Once these registers are read the root "discovers" that all its outgoing links are balanced, and unbalances the right of border link by executing the procedure \textit{instruct\_next\_son}.

**Case 2:** In \( c_t \) there is at least one unbalanced link.

By the assumption no color field is changed. Hence any unbalanced (balanced) link in \( c_t \) remains unbalanced (balanced) during \( E \). Consider an unbalanced link \((P_f \rightarrow P_i)\) of maximal distance from the root. The incoming link of \( P_i \) is unbalanced and all the outgoing links of \( P_i \) are balanced in any configuration of \( E \). The program of the non-root processor \( P_i \) is composed of a main loop and an inner loop. After the system reaches \( c_t \) the value of \( r_i.close \) is zero and the main loop is executed infinitely often. Let \( c_u (u \geq t) \) be the first configuration after \( P_i \) executes lines 7 to 12. In its first step following \( c_u \), \( P_i \) discovers that the predicates \textit{all\_out\_links\_balanced} and \textit{in\_link\_unbalanced} hold. Once this happens, \( P_i \) either changes the color of the right of border register (in case the predicate \textit{done} does not hold) or it balances the incoming link (in case the predicate \textit{done} holds), contradiction. \( \square \)

**Corollary 5.4:** In every fair execution the color of at least one register is changed infinitely often.

**Proof:** The proof is immediate by a repeated application of lemma 5.3. \( \square \)
In the next lemmas we show that if the color of one register is changed infinitely often, then so is the color of every register in the system.

Lemma 5.5:

(a) If a processor $P_i$ changes the color of one of its registers infinitely often, then $P_i$ execute the main loop of its code (lines 2-4 or lines 7-16) infinitely often.

(b) Once a normal (root) processor $P_i$ executes lines 7-8 (line 2), it holds that in every subsequent configuration, $r_j = o_j$ for any neighbor $P_j$ of $P_i$.

Proof: (a) follows by the observation that during one execution of its main loop, a processor may change the color of each of its registers at most twice (and after the first time it completed its main loop, it may change in a single execution of the main loop the color of at most one register, at most once). (b) holds trivially immediately after executing the aforementioned lines, and since no other processor can write into $P_i$’s registers, it follows easily by observing that in any subsequent atomic step, $P_i$ changes the color of $r_j$ iff it changes the color of $o_j$.

Lemma 5.6: Let $(P_i\rightarrow P_j)$ and $(P_j\rightarrow P_i)$ be two successive links in the tree. Then

(a) If the color of $r_i$ is changed infinitely often, so is the color of $r_j$.

(b) If the color of $r_i$ is changed infinitely often, so is the color of $r_j$.

(c) If the color of $r_j$ is changed infinitely often, so is the color of $r_j$.

(d) If the color of $r_j$ is changed infinitely often, so is the color of $r_j$.

Proof: First, observe that each of the assumptions in (a) - (d) implies that the consequences of Lemma 5.5 hold for $P_i$.

(a) Each time $r_i$ is changed, the predicate done holds for $P_i$, in particular $o_i \neq o_j$, and hence, By Lemma 5.5(b), $r_j \neq r_j$. The last inequality implies that between any two successive changes of $r_i$, $r_j$ is changed too.

(b) Assume that (b) is false. Then eventually the color of $r_j$ is fixed forever. Without loss of generality assume that at that point $r_j.color = 0$. By Lemma 5.5(a) $P_i$ executes its main loop infinitely often. In particular $P_i$ reads $r_j$ infinitely often, thus eventually the color of $ir_j$ will also become 0, and stay so forever. By our assumption the color of $r_j$ changes infinitely often. By Lemma 5.5(b) the color of $o_j$ also changes infinitely often. This means that after the color of $ir_j$ becomes 0 for the last time the color of $o_j$ will become 0 too. From this point
onwards, the predicate in_link_unbalanced is false and in particular after this point, \( r_q \) cannot be changed from 0 to 1, contradiction.

(c) The proof is similar to that of (b), by observing that if (c) is false, then eventually the predicate all_out_link_balanced is violated forever.

(d) We consider two cases:
If \( j \neq 1 \), then property (p2) of the processors states guarantees that whenever the color of \( r_q \) is changed (by executing line 23 of the code), it is colored to 0 (1) iff \( or_q \) is 1 (0), and hence \( or_q \) is changed infinitely often; since eventually \( or_q = r_q \) forever, this means that also \( r_q \) changes its color infinitely often.
Assume now that \( j = 1 \), and that \( r_q \) is changed only finitely many times. Without loss of generality assume that eventually the value of \( r_q \) (and hence also of \( or_q \)) is 0 forever. This means that eventually, whenever \( P_i \) computes the predicates in line 22 of the code, and both \( j_0 = 1 \) and the color of \( r_{i1} \) (and hence of \( or_{i1} \)) is 1, the truth value computed in that line must be false. Hence, \( P_i \) will never change the color of \( r_{i1} \) from 1 to 0, a contradiction. □

Lemma 5.7:

(a) The value of every register is changed infinitely often, hence by Lemma 5.5(b), every processor executes its main loop infinitely often.

(b) Eventually \( or_i = r_i \) for all \( i \) and \( j \).

Proof: (b) follows from (a) by Lemma 5.5(b), so we only have to prove (a). By Lemma 5.3, the color of some register is changed infinitely often. By a repeated application of (b) and (d) of Lemma 5.6, this implies that the color of a register of the root processor is changed infinitely often. Assume that the root has \( l \) sons. Whenever the root colors a register, the value of \( j_0 \) defined in property (p1) is increased by 1 (mod \( l \)). This means that the color of all the root's registers is changed infinitely often. The proof is now completed by a repeated application of (a) and (c) of Lemma 5.6. □

Consider a link \((P_f \rightarrow P_s)\), whose registers are \( r_f \) and \( r_s \). In the next lemma it is shown that, eventually, whenever \( P_f \) executes line 23 it unbalances the link, and that whenever \( P_s \) executes line 15 it rebalances it. At any given configuration, the behavior of the protocol on the link \((P_f \rightarrow P_s)\) can be described by the values of the link registers \( r_s \) and \( r_f \), and the values of the local variables \( or_f, or_s, ir_f \) and \( ir_s \). By Lemma 5.7 it holds that eventually \( or_f = r_f \)
and \( c \) in any configuration. Therefore eventually in any configuration \( c \) the link \( (P \rightarrow P_s) \) is fully described by the 4-tuple \((r_{id}, r_{if}.color \rightarrow r_{id}, r_{if}.color)\), which is called the link descriptor of the link \( (P \rightarrow P_s) \) in \( c \). The two central bits are the actual colors of the link’s registers, while the leftmost (rightmost) bit is the local information of \( P \) \( (P_s) \) on the color of \( r_{id} \) \( (r_{if}) \). We use \((1 \rightarrow 0 \ 0), [P \ read], (0 \ 0 \rightarrow 0 \ 0)\) to denote the change of the link descriptor from \((1 \ 0 \rightarrow 0 \ 0)\) to \((0 \ 0 \rightarrow 0 \ 0)\) by a read operation of \( P \). A priori, the link descriptor may be any combination of binary values. In Figure ... the In the following lemma we show that the link behaves properly, according to the protocol:

**Lemma 5.8:** In every fair execution of the protocol, eventually the link descriptor of every link in the network is changed repeatedly according to the following legitimate cycle:

\[
(0 \ 0 \rightarrow 0 \ 0), [P \ write], (0 \ 1 \rightarrow 0 \ 1), [P \ read], (0 \ 1 \rightarrow 0 \ 0), [P \ write],
\]

\[
(0 \ 1 \rightarrow 1 \ 1), [P \ read], (1 \ 1 \rightarrow 1 \ 1), [P \ write], (1 \ 0 \rightarrow 1 \ 1), [P \ read],
\]

\[
(1 \ 0 \rightarrow 1 \ 0), [P \ write], (1 \ 0 \rightarrow 0 \ 0), [P \ read], (0 \ 0 \rightarrow 0 \ 0).
\]

A pictorial representation of some of the possible changes of a link descriptor appears in Figure 5.3. In this figure registers and variables whose value is 0 (1) are represented as empty (dark) squares. The legitimate cycle is drawn by continuos lines. The illegitimate cycle drawn by dashed lines is prevented by use of the close field of the link.

**Proof:** Consider a configuration in which \( r_{id} = r_{if} \) and \( r_{if} = r_{id} \) for every link \( (P \rightarrow P_s) \). In such a configuration, \( P \) may change the color of \( r_{id} \) only when \( r_{id} = ir_{id} \) (otherwise, all_out_links_balanced does not hold). Similarly, \( P_s \) may change the color of \( r_{if} \) only if \( r_{if} = ir_{if} \) (otherwise, in_link_unbalanced does not hold). Using this, it is straightforward to check that if the link descriptor is in the legitimate cycle, then it can be changed only to its successor in the legitimate cycle. Hence, it is sufficient to show that the link descriptor converges to one of the values in the legitimate cycle. We prove below that, assuming that \( or_{id} = r_{id} \) and \( or_{id} = r_{id} \), this happens after the first time \( P \) executes the function instruct_next_son with \( j_0 = s \) (recall that by Lemma 5.7 (a), this happens infinitely often).

In every execution of instruct_next_son with \( j_0 = s \), \( P \) closes \( r_{id} \) for reading, reads the value of \( r_{id} \), writes into \( r_{id} \) (if necessary), and opens \( r_{id} \) for reading (lines 20-24 of the code). Call the sequence of configurations during which \( P \) executes these operations the close period of \( r_{id} \). Consider an outgoing link of \( P \), \( (P \rightarrow P_s) \) and trace its descriptor during the close period of \( r_{id} \). During the close period \( P_s \) can change the color of \( r_{id} \) (by executing line 15)
at most once, since after any such change, \( P \) must read \( r_f \). Whenever \( P \) reads \( r_f \) during the close period of \( r_f \), it remains in its inner loop (lines 9-11) until the close period is finished. The following three cases sum up the possible ways in which the link descriptor might be changed by \( P \), during the close period of \( r_f \):

Case 1: \( P \) does not change \( r_d \) during the close period.

Case 2: \( P \) changes \( r_d \) before \( P \) reads from \( r_l \).

Case 3: \( P \) changes \( r_d \) after \( P \) reads from \( r_l \).

In the first two cases \( P \) reads the updated value of \( r_d \) and unbalances the link (if it is not already unbalanced) hence the link descriptor at the end of the close period is either \((1 \rightarrow 0 ?)\) or \((0 \rightarrow 1 ?)\) (the question mark stands for either 0 or 1), which are all in the legitimate cycle. The third case starts as follows: \( P \) reads \( r_d \), then \( P \) changes the color of \( r_d \) to be the same as \( r_l \). At this stage the link descriptor is either \((0 \rightarrow 1 1)\) or \((1 \rightarrow 0 0)\). At the end of the close period, \( P \) tries to unbalance the link using the last (not updated) color it read from \( r_d \). Thus the value of the link descriptor at the end of the close period is either \((0 \rightarrow 1 1)\) or \((1 \rightarrow 0 0)\) which are both in the legitimate cycle. \(\square\)

We proceed by combining the above results in order to show that eventually the system converges to a configuration in which all registers and internal variables have the same color.

**Lemma 5.9:** There are configurations in \( E \) in which all the registers and internal variables have the same color.

**Proof:** Consider a suffix \( E' \) of \( E \) in which all link descriptors behave according to the legitimate cycle of Lemma 5.8. When all link descriptors are in the legitimate cycle, the predicate \( \text{all_out_links_balanced} \) holds for \( P \) only when all its outgoing links are indeed balanced. Thus in \( E' \), whenever \( P \) changes the value of \( r_f \) and unbalances the link \((P_f \rightarrow P_s)\), \( P \) subsequently changes the value of \( r_d \) and rebalances \((P_f \rightarrow P_s)\) only after all its own outgoing links are balanced and are all colored by the color of \( r_f \). In other words: In \( E' \), the values of the registers \( r_f \) and \( r_d \) are changed in an alternating way. Using this fact, we prove the following:

**Claim:** Eventually, for each link \((P_f \rightarrow P_s)\), \( P \) writes 0 in \( r_d \) only when all the registers and variables in the subtree rooted at \( P \) are colored by 0.

**Proof of claim:** The proof is by induction on \( h \), the height of the subtree rooted at \( P \). It holds trivially if \( P \) is a leaf. Assume this holds for any processor \( P \) such that the height of the subtree rooted at \( P \) is smaller than \( h \). Let the height of the subtree rooted at \( P \) be \( h \). Consider a configuration in which \( P \) writes 0 into \( r_f \), setting the link descriptor to \((1
0 \rightarrow 1 1). It was already proved that $P_*$ will rebalance the link only after the registers of all its outgoing links are balanced and colored 0. Assuming that the link descriptors are changed according to the legitimate cycle, those outgoing links could be balanced only by the sons of $P_*$. Hence, by induction, all the subtrees rooted at $P_*$'s sons satisfy the induction hypothesis, and so does the subtree rooted at $P_*$. This proves the claim.

The proof of the lemma is completed by applying the claim to the links emanating from the root processor. For this, consider a configuration $c$ in which all the root registers are colored 0 (there are infinitely many such configurations in $E$ - see proof of Lemma 5.7), and all the link descriptors are changed according to the legitimate cycle. Then there is a configuration $c'$ following $c$ in which all the root's outgoing links are balanced to 0 (otherwise the root will never color any of its registers). But $c'$ could be reached only by having all the root sons balancing their incoming links to 0, which by the claim means that all the registers and variables in their subtrees are colored 0, and hence all the registers and variables in the systems are colored 0. □

We proceed by showing that eventually, in any configuration at most one processor is privileged.

Lemma 5.10: Eventually the following hold:

(a) A processor is privileged only if it is active.

(b) In every configuration there is exactly one active processor.

Proof: Recall that a privileged processor in our protocol is a processor $P$ which is in line 13 (22) of its code and the predicates $\text{in\_link\_unbalanced}$ and $\text{all\_out\_link\_balanced}$ are true, and the predicate $\text{done}$ is true (false) if it is in line 13 (22). Once every link descriptor is in the legitimate cycle, those predicates are true for $P$ only if it is active (i.e., indeed the incoming link of $P$ is unbalanced and its outgoing links are balanced). This proves (a). To prove (b), it suffices to prove that eventually there is at most one processor for which the incoming link is unbalanced (or it is the root) and all its outgoing links are balanced. We will prove the following stronger result:

Claim: Eventually, in every configuration there is at most one processor, say $P$, whose incoming link is unbalanced (or he is the root), and all its outgoing link are balanced. Moreover, all the links in the system are balanced, except the links on the path from the root to $P$.

A sample execution of this protocol is depicted in Figure 5.4. In this figure the path of unbalanced links leading from the root to the active processor and the active processor itself are represented by boldface arrows and a boldface frame respectively.
Proof of Claim: By induction on the order of the configuration in $E$. The induction base is a configuration $c_n$ in which all registers and variables are colored 0, which satisfies the claim. Assume that the claim holds for certain configuration $c_t$ ($t \geq n$). We have to show that it holds also for $c_{t+1}$.

Let $P$ be the only active processor in $c_t$. If the atomic step between $c_t$ and $c_{t+1}$ did not change the value of any register then we are done. If the atomic step changed the value of some register say $r$, then $r$ is a register of $P$ (since $P$ is the only processor in which the predicates above are true, and only such a processor may write). By using the fact that every link descriptor is in the legitimate cycle, it holds that if $r$ is a register of an outgoing link then $P$ unbalances it, and the claim holds for $P$'s son. If it is $P$'s incoming link, then $P$ balances it, and the claim holds for $P$'s father. This complete the proof of the claim, and hence of the Lemma.  

Corollary 5.11: The protocol is self stabilizing relative to the set $LSM$.

Proof: By lemma 5.10 at most one privileged processor exist hence the [exclusion] requirement holds. By Lemma 5.7 each processor executes line 15 or 23 infinitely often hence the [fairness] requirement holds.

Note: We had, in fact, proved that a configuration in which all the registers and internal variables have the same colors, and in which all the link descriptor are in the legitimate cycle, is a safe configuration (w.r.t. our protocol). Hence our protocol is self stabilizing in the strong sense (see discussion at the end of Section 2).


In this section we combine the protocols from the previous sections to obtain a mutual exclusion protocol for dynamic networks. The tool used is Fair Protocol Composition. This tool, which is described below, is extended and discussed in greater details in [DIM89b].

We start with few definitions. Let $LS$ be a task defined for a system whose communication graph is $G$. Then $LS$ is a graph specification task if the following holds: For each sequence $s \in LS$ and for each configuration $c \in s$, the value of each register of each link determines whether this link belongs to a specified graph. In case the link belongs to the specified graph, the values of the registers of this link (which should be consistent) may also specify the direction of the link. The specified graph and the directions of the links are constant throughout $s$. (Note that the specified graph is not necessarily a subgraph of $G$, since the nature of the links may be modified by the task). A protocol which realizes a graph specifying task is a graph specifying protocol. The spanning tree protocol presented in section
4 is a graph specifying protocol. Two more graph specifying protocols are presented in [U89].

Let $PR_1$ and $PR_2$ be dynamic self stabilizing protocols, such that $PR_1$ is a graph specifying protocol. The Fair Composition of $PR_1$ and $PR_2$ is a protocol $PR$ in which:

(a) every register duplicates the corresponding registers of $PR_1$ and $PR_2$.
(b) Every processor executes steps of $PR_1$ and $PR_2$ alternatingly.
(c) While executing steps of $PR_2$, only links specified by $PR_1$ are considered.

Consider an execution of a fair composition of $PR_1$ and $PR_2$, in which $PR_1$ is already stabilized. This execution defines in a natural way an execution of $PR_2$ on the graph specified by $PR_1$ in this execution. This implies the following:

**Theorem 6.1:** Let $\Delta$ and $\Gamma$ be families of graphs, and let $PR_1$ and $PR_2$ be dynamic self stabilizing protocols. Assume that $PR_1$ is a graph specifying protocol with communication graph $D\in\Delta$, which always specifies a graph $G\in\Gamma$, and that $PR_2$ is a protocol for systems with communication graph $G\in\Gamma$, for a task $LS$. The fair composition of $PR_1$ and $PR_2$ is a dynamic self stabilizing protocol with communication graph $D\in\Delta$ which realizes the task $LS$.

**Proof:** Consider a fair execution $E$ of $PR$. $PR_1$ is self stabilizing, hence there is a suffix $E'$ of $E$ which specifies a graph $G\in\Gamma$. Let $c_0$ be the first configuration in $E'$. Consider the behavior of the steps of $PR_2$ executed during $E'$. Starting from $c_0$ the graph $G$ is determined and fixed. Since $PR_2$ is dynamic and self stabilizing the sequence of these steps has a suffix in $LS$. $\square$

As a result of Theorem 6.1 we get the following corollary:

**Corollary 6.2:** The fair composition of the spanning tree protocol with the mutual exclusion protocol is a mutual exclusion protocol on systems with an arbitrary dynamic communication graph. This protocol is correct in the presence of read/write demon.

7. Concluding Remarks

Self stabilization was defined as an independent property of protocols. The viability of the new definitions was demonstrated by presenting self stabilizing protocol for mutual exclusion in dynamic systems with arbitrary communication graph. The protocol is correct with respect to the most relaxed kind of atomicity.
In some papers, self stabilization appears with no requirement from the scheduler to be fair [Di74]. The requirement of fairness is replaced there by a weaker requirement, that can be described as follows: Define a processor $P$ as *active* in a configuration $c$ if in the execution which starts from configuration $c$, and in which $P$ is the only processor that makes atomic steps, eventually $P$ changes the value of some register (in previous models, this definition of active coincide with the definition of a privileged processor). At any given configuration, the scheduler is free to choose any active processor. It can be shown that all our protocols are correct with respect to this less restricted definition of a scheduler, under an appropriate modification of the definition of "fair protocol composition".

The spanning tree protocol presented here uses registers whose size is unbounded. If an a priori bound $\delta$ on the diameter of the system is known then the size of each register is $O(\log \delta)$. In a forthcoming paper [DIM89a] we present a spanning tree protocol in which the size of the registers is constant. The fair protocol composition of this protocol with the mutual exclusion protocol for directed trees presented here, provides a self stabilizing protocol for dynamic networks in which the size of communication registers is bounded.

Composition of protocols is interesting since the correctness proof of the resulting protocol might be carried in stages. In [DIM89b] we study fair program composition in a more formal fashion.
REFERENCES


[BGM89] J. Burns, M. Gouda and G. Miller, Stabilization and Pseudostabilization a manuscript.


Figure 3.3
Figure 4.2: Execution sample of the spanning tree protocol.
Figure 5.1: Processor description.
Figure 5.3: Link descriptor transformations.
Figure 5.4: Execution sample of the mutual exclusion protocol.