Distrib Comput (1993) 7: 3–16

DISTRIBUTED COMPUTING

Self-stabilization of dynamic systems assuming only read/write atomicity*

Shlomi Dolev[1], **Amos Israeli**[2], **Shlomo Moran**[1]

[1] Department of Computer Science, Technion – Israel Institute of Technology, Haifa 32000, Israel
[2] Department of Electrical Engineering, Technion – Israel Institute of Technology, Haifa 32000, Israel

Received February 1990 / Accepted April 1993

Shlomi Dolev received his B.Sc. in Civil Engineering and B.A. in Computer Science in 1984 and 1985, and his M.Sc. and Ph.D. in computer Science in 1989 and 1992 from the Technion Israel Institute of Technology. He is currently a postdoctoral fellow in the Department of Computer Science at Texas A & M University. His current research interests include the theoretical aspects of distributed computing and communication networks.

Amos Israeli received his B.Sc. in Mathematics and Physics from Hebrew University in 1976, and his M.Sc. and D.Sc. in Computer Science from the Weizmann Institute in 1980 and the Technion in 1985, respectively. Currently he is a senior lecturer at the Electrical Engineering Department at the Technion. Prior to this he was a postdoctoral fellow at the Aiken Computation Laboratory at Harvard. His research interests are in Parallel and Distributed Computing and in Robotics. In particular he has worked on the design and analysis of Wait-Free and Self-Stabilizing distributed protocols.

Shlomo Moran received his B.Sc. and D.Sc. degrees in mathematics from Technion, Israel Institute of Technology, Haifa, in 1975 and 1979, respectively. From 1979 to 1981 he was assistant professor and a visiting research specialist at the University of Minnesota, Minneapolis. From 1981 to 1985 he was a senior lecturer at the Department of Computer Science, Technion, and from 1985 to 1986 he visited at IBM Thomas J. Watson Research Center, Yorktown Heights. From 1986 to 1993 he was an associate professor at the Department of Computer Science, Technion. In 1992-3 he visited at AT & T Bell Labs at Murray Hill and at Centrum voor Wiskunde en Informatica, Amsterdam. From 1993 he is a full professor at the Department of Computer Science, Technion. His research interests include distributed algorithms, computational complexity, combinatorics and graph theory.

Summary. Three self-stabilizing protocols for distributed systems in the shared memory model are presented. The first protocol is a mutual-exclusion protocol for tree structured systems. The second protocol is a spanning tree protocol for systems with any connected communication graph. The third protocol is obtained by use of *fair protocol combination*, a simple technique which enables the combination of two self-stabilizing dynamic protocols. The result protocol is a self-stabilizing, mutual-exclusion protocol for dynamic systems with a general (connected) communication graph. The presented protocols 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 any connected network and even for dynamic networks, in which the topology of the network may change during the execution.

Key words: Self-stabilization – Read/write atomicity – Protocol combination

* 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

Correspondence to: A. Israeli

1 Introduction

A *self-stabilizing* system which is started from an arbitrary initial configuration, regains its consistency and demonstrates legal behavior by itself, without any outside intervention. Consequently, a self-stabilizing system need not be initialized to any particular configuration, and 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. In this paper we present self-stabilizing protocols for mutual-exclusion and for constructing a spanning tree. The presented protocols work on connected networks of arbitrary topology which can change dynamically during execution. Communication among neighboring processors is carried out by use of *communication registers* (called *registers* throughout this paper). The atomic operations that these registers support are *read* and *write*.

We model distributed self-stabilizing systems as a set of state machines called *processors*. Each processor can communicate with some subset of the processors called its *neighbors*. The system's *communication graph* is the graph formed by representing each processor as a node an by drawing an edge between every two neighbors. A *protocol* is a parameterized family of systems where the parameters can vary over the number of different state machines used by the protocol, the various families of communication graphs, the set of atomic operations supported by the communications registers, etc. A processor's *degree* is equal to the number of its neighbors. A protocol is *uniform* if all processors of the same degree are identical. If all processors of the same degree are identical except a single processor in the entire system, then the protocol is *semi-uniform*. An *atomic step* is the "largest" step that is guaranteed to be executed uninterruptedly. A protocol uses *composite* atomicity if some atomic step contains (at least) a read operation and a write operation. A processor uses *read/write* atomicity if each atomic step contains either a single read operation or a single write operation but not both. The behavior of the system is modeled by the interleaving model in which processors are activiated by a scheduler. Whenever an enabled processor is activated, it executes a single atomic step. To ensure the correctness of a protocol, the scheduler is regarded as an adversary and the protocol is required to be correct in all possible executions. The common schedulers are the *central demon* which activates processors one by one and the *distributed demon* which activates subsets of processors.

The class of self-stabilizing protocols was defined by Dijkstra in his pioneering paper [4]. In that paper Dijkstra presents three semi-uniform, self-stabilizing, mutual-exclusion protocols for rings. Protocols in the same setup but under the distributed demon are presented by Brown et al. in [1], and by Burns in [3]. Burns and Pachl in [2] present a uniform, self-stabilizing, mutual-exclusion protocol for rings with a prime numer of processors. A semi-uniform, self-stabilizing protocol for some variant of the mutual-exclusion problem which runs on tree systems is presented by Kruijer in [11]. A self-stabilizing, mutual-exclusion protocol for systems with arbitrary communication graphs is presented by Tchuente in [16]. Unlike the aforementioned protocols the protocol presented in [16] is not semi-uniform, in fact the program of each processor depends on the system's communication graph, and for many communication graphs all processors are distinct. Furthermore, obtaining the protocol for each individual system requires extensive programming work. All these papers use the shared-memory model. The work of Katz and Perry in [10] deals with the message-passing model which is different in some respects from the shared memory model. In [10], Katz and Perry present a general method for converting arbitrary programs in the message-passing model to equivalent self-stabilizing programs in the same model.

All previous self-stabilizing protocols use composite atomicity. In the work of Loui and Abu-Amara in [12], it was shown that while there exists no consensus protocol for systems that use read/write atomicity, the consensus task is solvable for systems that use composite atomicity. Since any system under composite atomicity can trivially emulate an equivalent system that uses read/write atomicity, composite atomicity is strictly stronger than read/write atomicity.

A protocol is *dynamic* if it tolerates changes in the communication graph during execution as long as the communication graph remains connected. The changes we allow are processor addition or removal and link addition or removal. Every self-stabilizing, uniform protocol that works on every communication graph is dynamic, since it stabilizes after any topology change. A semi-uniform protocol that works on any communication graph is dynamic as long as the (single) special processor is not removed from the system. In [4], Dijkstra used symmetry considerations and showed that for rings of composite size, there exists no uniform, self-stabilizing, mutual-exclusion protocol. Thus, if one opts for dynamic, self-stabilizing, mutual-exclusion protocols then the best that can be achieved are semi-uniform protocols.

Most previous works assumed that one-way communication from P_1 to P_2 is carried out by P_1 changing its state which is observable by P_2. This mode of communication is equivalent to the use of a single communication register in which P_1 writes and from which all processor to which it can communicate read. It is not hard to show that under this communication mode, there exists no semi-uniform, self-stabilizing, mutual-exclusion protocol in many systems, including systems with very simple communication graphs. There are two possible ways to remedy this problem: The first one, which was chosen by Tchuente in [16], is to give up uniformly altogether and program each processor individually. Since in this method each processor is programmed individually, it cannot yield dynamic protocols. The alternative way, which we choose in this work, is to allow each processor to break the symmetry among its neighbors locally. This is done by introducing a *link* between every pair of neighbors. Each link is composed of two registers and supports two-way communication. One neighbor writes in the first register and reads from the second, the other neighbor reads from the second register and writes in the first. Each register is serializable (atomic) with respect to read and write actions.

We present two semi-uniform, self-stabilizing protocols: The first protocol is a mutual-exclusion protocol for tree structured systems. The second protocol constructs a spanning tree of the system's communication graph; both protocols are correct under read/write atomicity. We then present *fair protocol combination* as a technique for combining self-stabilizing protocols into another self-stabilizing protocol. The presentation is completed by combining the two aforementioned protocols into a semi-uniform, self-stabilizing, mutual-exclusion protocol for systems with any connected communication graph using fair protocol combination. The combined protocol, like both its building blocks, is correct under read/write atomicity. Using this final protocol we show that any protocol which is self-stabilizing under composite atomicity can be executed in a self-stabilizing fashion in the presence of read/write atomicity.

Our protocols improve upon all previous protocols in two important aspects:

- **Atomicity:** All previous self-stabilizing protocols use composite atomicity. Our protocols use read/write atomicity, hence they subsume all aforementioned self-stabilizing protocols.
- **Topology:** Almost all previous self-stabilizing protocols work only on restricted families of communication graphs. In this respect our protocols improve upon all previous protocols except the protocol of [16], since they work in systems with arbitrary connected communication graphs. Furthermore, our protocols are semi-uniform hence, they are also dynamic and superior to the protocol of [16].

The rest of this paper is organized as follows: in Sect. 2 the computational model and the requirements for self-stabilization are discussed and formally defined. In Sect. 3 we present a simple self-stabilizing protocol called the *balance-unbalance* protocol for mutual-exclusion in a two processor system and show how to adapt it to read/write atomicity. In Sect. 4 we present a self-stabilizing, mutual-exclusion protocol for tree-structured systems which uses the balance-unbalance protocol as a building block. In Sect. 5 we present a self-stabilizing protocol for finding a spanning tree of the system's communication graph. We proceed by presenting fair protocol combination. Combining the spanning tree protocol with the mutual-exclusion protocol yields the final protocol. Section 6 contains some concluding remarks.

2 Model and requirements

2.1 The model

A distributed system consists of n processors, denoted by $P_1, P_2, \ldots, P_n$. Each processor is a (possibly infinite) state machine. Processor P_1 is distinguished as a *special* processor. All other processors are called *normal*. Normal processors have no distinct identities, the subscripts $2, \ldots, n$ are used for notation only. Neighbors P_i and P_j communicate with each other by using two shared registers, r_{ij} in which P_i writes and from which P_j reads, and $r_{j,i}$ in which P_j writes and from which P_i reads. All links incident to each processor P_i are ordered by some arbitrary ordering α_i which induces in a natural way an ordering of the neighbors of P_i. The collection of all these orderings is denoted by $\alpha = (\alpha_1, \ldots, \alpha_n)$.

Every register r is associated with the set Σ_r of *permitted values* which can be stored in r (the set Σ_r is not necessarily finite). Each register r has a *writer* - a processor that can write in r, and a *reader* - a processor that can read from r. A *write* operation to r stores a value from Σ in r. A *read* operation retrieves the value (from Σ_r) stored in r. Each register is *serializable* with respect to read and write operations. The registers in which processor P can write are called the registers of P. We choose to look at a processor and its registers as a single entity, thus the state of a processor fully describes the values stored in its registers. Denote by S_i the set of states of P_i. A *configuration* of the system is the vector of states of all processors. Denote by $\mathsf{C} = (S_1 \times S_2 \times \cdots \times S_n)$ the set of all possible configurations of the system.

An atomic step of a processor consists of an internal computation followed by either a read or a write operation, but not both. Processor activity is managed by a scheduler, which is also called the *central demon*. In any given configuration the demon activates a single processor which executes a single atomic step. An *execution* E, of the system, is an infinite sequence of configurations $E = C_1, C_2, \cdots$ where for every $i > 0$, C_{i+1} is reached from C_i by a single atomic step of a single processor. An infinite execution is *fair* if every processor executes steps infinitely often.

2.2 Task specification and self-stabilizing protocols

A self-stabilizing system demonstrates *legitimate behavior* some time after it is started from an arbitrary configuration. A natural way to specify a behavior in an abstract way is by a set of sequences of configurations. We define *tasks* as sets of *legitimate-sequences*. The semantics of any specific task is expressed by requirements on its sequences. Intuitively each legitimate sequence can be thought of as an execution of a protocol but we do not require it formally. For instance, the mutual-exclusion task is defined as the set of sequences of configurations which satisfy: Each processor has a subset of its states called the *critical section*; in each configuration, at most one processor is in its critical section, and every processor is in its critical section in infinitely many configurations.

To formally define a task T, one should specify for each possible system ST, a set of *legitimate sequences* for ST. The *task* T is defined as the union of the legitimate sequence sets over all possible systems. A configuration C of a system is *safe* with respect to a task T and a protocol Pr if any fair execution of Pr starting from C belongs to T. A protocol and a scheduler determine the set of all possible executions of the protocol under this scheduler. In the non-self-stabilizing model, a protocol implements a task if all its executions belong to the set of sequences which constitutes the task. In the self-stabilizing model this requirement is relaxed, and a protocol is defined to be self-stabilizing with respect to a task T if the following definition holds:

[Self-stabilization]

A protocol is *self-stabilizing* if starting from any system configuration, it eventually reaches a safe configuration.

This definition separates the specific task which the protocol implements from the general requirements for self-stabilization and allows self-stabilizing protocols for any task. It is natural (though not necessary) to require that a task is closed under the suffix operation. When this requirement is adopted, any configuration which is reachable from a safe configuration is also safe, therefore the set of safe configurations for task T with respect to protocol Pr is closed under executions of Pr.

2.3 Protocol description

A semi-uniform protocol is specified by describing two types of processors: A special processor and a normal processor. A processor is entirely determined by its type and by the number of its neighbors. For convenience we choose to represent each of our processors as a RAM executing a program. Since the system is dynamic the number of neighbors of each processor may change during execution. This is modeled by assuming that each processor has access to a local constant called *nu_neighbors* in which the number of the processor's neighbors is stored. This constant is assumed to be updated by the hardware whenever the number of neighbors is changed. (Later we discuss the technique of protocol combination. In a combined protocol *nu_neighbors* can be updated by a lower level protocol).

The program of each processor is partioned to distinct atomic steps. It is assumed that each program is executed step by step where each step is executed uninterruptedly. Each processor is assumed to be equipped with a *program counter (pc)* whose value indicates the next atomic step to be executed. The partition of the program into atomic steps is straightforward: Each atomic step consists of a sequence of internal operations which ends either with a write operation or with a read operation. The state of a processor is determined by the internal state of the RAM and the contents of its registers. The internal state of the RAM is fully described by the values stored in its internal variables and by its next step (the *pc*). Each internal variable has a set of *permitted values*, a permitted state of a processor is any assignment of permitted values to its internal variables and to its registers.

3 The balance-unbalance protocol

3.1 The basic protocol

A processor is *enabled* if it can execute a state-transition. A mutual-exclusion protocol under composite atomicity is designed so that in each legitimate configuration there exists a single enabled processor. The enabled processor is *privileged*, it has the right to enter its critical section. Presumably the enabled processor finds out that is is enabled by reading its neighbors' registers, executes its critical section, and then passes the privilege to one of its neighbors by executing a state transition which includes writing new values in some of its registers. The composite atomicity ensures that this extended atomic step is executed uninterruptedly. The *balance-unbalance* protocoll is probably the simplest protocol for mutual-exclusion under composite atomicity. It is designed for a system of two processors, which are connected by a link. The two processors are the *unbalancing* processor UB and the *balancing* processor BA. 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 UB and BA respectively. Thus, the system has four possible configurations: $(0,0)$, $(1,0)$, $(1,1)$, and $(0,1)$. Processor UB is enabled when the link is balanced. Its transition function *unbalances* the link by transfering $(0,0)$ to $(1,0)$ and $(1,1)$ to $(0,1)$. Analogously, BA is enabled when the link is unbalanced. Its transition function *balances* the link by transfering $(0,1)$ to $(0,0)$ and $(1,0)$ to $(1,1)$.

Consider an execution of the protocol under composite atomicity. In any possible configuration, exactly one processor is enabled (and privileged); the enabled processor passes the privilege to the other processor by changing its state. Thus, starting with any configuration of the system, and regardless of the specific behavior of the demon, the system configuration is changed repeatedly according to the following cycle: $((0,0)$, [UB writes], $(1,0)$, [BA writes], $(1,1)$, [UB writes], $(0,1)$, [BA 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, which is a suffix of every possible execution of the protocol. In a way, this protocol is well known and is a simplified version of protocols presented in [3, 4, 11, 13].

Under read/write atomicity an atomic step includes either a read action or a write action. When an atomic step ends by a read action, the read value may affect the next transition. Therefore the state of each processor should reflect the last value it read from the (register of the) other processor. We use here the term *state* to denote the full information describing the processor behavior, while the balance-unbalance bits are called *colors*. The state of UB is described by the following components: {the color of UB, the last color of BA read by UB, the next action to be executed by UB}. The state of BA is described analogously as follows: {the last color of UB read by BA, the color of BA, the next action to be executed by BA}. A configuration of the systems is a pair of the processor states.

Lemma 3.1. *Under read/write atomicity the balance-unbalance protocol is not a self-stabilizing, mutual-exclusion protocol.*

Proof. Consider configuration $C = (\{0, 0, \mathbf{write}\}, \{1, 0, \mathbf{write}\})$, in which the colors of UB and BA are both 0, but as a result of a transient bug, BA "thinks" that the color of UB is 1. In C both processors are enabled (and hence privileged).

In Fig. 3.1 we depict a prefix of an execution, starting and ending with configuration C. In this prefix each pro-

({0, 0, write}{1, 0, write}) [*BA* **writes**],
({0, 0, write}{1, 1, read}) [*BA* **reads**],
({0, 0, write}{0, 1, write}) [*UB* **writes**],
({1, 0, read}{0, 1, write}) [*UB* **reads**],
({1, 1, write}{0, 1, write}) [*BA* **writes**],
({1, 1, write}{0, 0, read}) [*BA* **reads**],
({1, 1, write}{1, 0, write}) [*UB* **writes**],
({0, 1, read}{1, 0, write}) [*UB* **reads**],
({0, 0, write}{1, 0, write}) [*BA* **writes**],

Fig. 3.1. A prefix of a non-stabilizing execution

cessor is activated and both processor are simultaneously privileged. Since this prefix starts and terminates with the same configuration it can be duplicated infinitely often to obtain an infinite fair execution. In half of the configurations of this infinite execution *UB* and *BA* are both privileged, hence the system does not stabilize. □

3.2 Adaption for read/write atomicity

In this section we modify the balance-unbalance protocol to be correct under read/write atomicity. The registers of *UB* and *BA* are called r_{ub} and r_{ba} respectively. Processor *UB* has two internal variables called *my_color* and *ba_color*. These variables constitute the *view* of *UB*, the values it "thinks" r_{ub} and r_{ba} have. In case the colors of r_{ub} and r_{ba} are equal to the values of variables *my_color* and *ba_color* respectively, we say that *ub* has a *correct view*. Analogously, processor *BA* has two internal variables called *my_color* and *ub_color* which constitute *BA*'s view. In case the colors of r_{ba} and r_{ub} are equal to the values of *my_color* and *ub_color* respectively, we say that *BA* has a *correct view*.

The problem depicted in Fig. 3.1 is caused by the nature of read/write atomicity. A processor may read the color of the other processor, then the second processor may write and change its color. After that, the first processor may use the color it read, which is already outdated at this point, and enter its critical section. Consequently mutual-exclusion might be violated. In order to overcome this problem, some additional synchronization between the processors is required. For this purpose *UB* is allowed to *close* the link for *BA* using the binary *close* field with which r_{ub} is augmented. Whenever *BA* reads *UB*'s register it considers the value it reads only if the link is open, that is if *ub.close* = 0. In this way *UB* controls the number of times *BA* executes its loop between every consecutive executions of *UB*'s loop.

The code for the modified protocol appears in Fig. 3.2. The program for each processor in the modified protocol consists of a loop which is executed repeatedly. The loops for both processors have a similar structure. Each loop consists of two blocks: a *refresh* block and a *main* block. In the refresh block each processor unconditionally copies its internal variable *my_color* to its register. In this way the processor ensures that the color of its register is equal to the processor's "belief". The unconditional write is called the *refreshing* write. Following the first refreshing

```
1  UB: repeat forever
         r_ub := write(my_color, 0)        refresh
2        ba_color := read(r_ba)
         if my_color = ba_color            link seems balanced
         then begin                        main loop
           CRITICAL SECTION
3          r_ub := write(my_color, 1)      close link
4          ba_color := read(r_ba)          reread ba_color
           my_color := 1 − ba_color        complement your color
5          r_ub := write(my_color, 0)      unbalanced and open link
         end

   BA: repeat forever
6        r_ba := write(my_color)           refresh
         repeat
7          (ub_color, close)
            := read(r_ub)
         until close = 0
         if ub_color ≠ my_color            link seems open and un-
                                                           balanced
         then begin                        main loop
           CRITICAL SECTION
           my_color := ub_color            complement ba_color
8          r_ba := write(my_color)         balance link
         end
```

Fig. 3.2. The modified balance-unbalance protocol

write the value of the register can always be inferred from the values of the processor's internal variables. Hence, the only refreshing write which may change the value of a register is the first write in an execution. After refreshing its register each processor proceeds to its main block which includes the processor's critical section. Unlike the refresh block, execution of the main block is conditional. The first atomic step in the main block is a read action. The value read in this action is used as a guard for the rest of the main block in which the critical section is executed.

A system configuration is specified by the values of the varaibles *my_color* (of both processors), the values of *ba_color* and *ub_color*, the values of the registers and the next step each processor is about to execute (the *pc* of both processors). After both registers are refreshed the values of the *my_color* variables are the same as the values of the correesponding registers. In this situation and when the values of the *pc*-s are implied by the context, we describe a configuration by a 5-tuple (*ba_color*, r_{ub}.*color*, r_{ub}.*close*$\rightarrow r_{ba}$, *ub_color*) which is called the *link descriptor*. The arrow in the link descriptor stands for the link that separates the variables and register values of *UB* (at the tail) from those of *BA*.

Under composite atomicity a non enabled processor which is activated by the demon does not execute any state transition. The situation is different under read/write atomicity where a processor can always execute some action. This action may be a read action after which the processor may find that it cannot execute any write action. Another possibility is that the processor finds out that it can execute a write action, but the written value is equal to the value which is stored in the register before the write action is executed. In order to accommodate these situations we define some segment of a processor's

$(0,0,0 \longrightarrow 0,0)$
$[UB$ **writes**$], (0,0,1 \longrightarrow 0,0)$,
$[UB$ **reads**$], (0,0,1 \longrightarrow 0,0)$,
$[UB$ **writes**$], (0,1,0 \longrightarrow 0,0)$,
$[BA$ **reads**$], (0,1,0 \longrightarrow 0,1)$,
$[BA$ **writes**$], (0,1,0 \longrightarrow 1,1)$,
$[UB$ **reads**$], (1,1,0 \longrightarrow 1,1)$,
$[UB$ **writes**$], (1,1,1 \longrightarrow 1,1)$,
$[UB$ **reads**$], (1,1,1 \longrightarrow 1,1)$,
$[UB$ **writes**$], (1,0,0 \longrightarrow 1,1)$,
$[BA$ **reads**$], (1,0,0 \longrightarrow 1,0)$,
$[BA$ **writes**$], (1,0,0 \longrightarrow 0,0)$,
$,[UB$ **reads**$], (0,0,0 \longrightarrow 0,0)$,

Fig. 3.3. The legitimate cycle

execution as a *stuttering section* if the only changes in the processor's state are in its *pc*. A stuttering section that starts and ends with the same system configuration is called a *cyclic* stuttering section. A cyclic stuttering section might be removed from an execution, except the first (or last) configuration, and the resulting sequence is also an execution. We say that two executions are *equivalent up to stuttering* if when all cyclic stuttering sections are removed the resulting system executions are equal.

Consider a configuration C in which the registers are refreshed, the value of r_{ub} is $(0,0)$ and the value of r_{ba} is 0. The link descriptor of C is $(0,0,0 \rightarrow 0,0)$. In such a configuration, the only possible subsequent changes in the value of the link descriptor are given by the *legitimate cycle* which appears in Fig. 3.3: (In the legitimate cycle the refreshing writes are omitted).

Define BUB' to be the set containing the sequence l obtained by a repeated execution of the *legitimate cycle* and all suffixes of l. The task BUB is now defined as the set of all sequences which are equivalent to some task in BUB' up to stuttering. Note that BUB is a subtask of the mutual-exclusion task. By this definition, each configuration in the legitimate cycle is safe for the set BUB. Observe that when the system is in the legitimate cycle the processors access their critical section in a mutually exclusive (and fair) fashion. In the following lemmas we prove that the protocol is self-stabilizing by showing that in every fair execution some configuration in the legitimate cycle is reached.

Lemma 3.2. *In every fair execution E of the protocol in which the color of no register is changed, there is a configuration C_t after which the link is always open.*

Proof. Execution E is fair, therefore UB is activated infinitely often, in particular r_{ub} is refreshed infinitely often. Consider configuration C_t right after r_{ub} is refreshed for the first time. In C_t the link is open since it was opened in the refreshing write of UB. In its next activation UB reads r_{ba}; since during E the color of no register changes, after this read action UB has a correct view which is constant throughout E. In case the link is unbalanced in this constant view, UB does not enter its loop and therefore never closes the link. In case the link is balanced in the constant view, UB executes the loop and unbalances the link, in contradiction to the assumption that no color is changed during E. □

Lemma 3.3. *In every fair execution of the protocol both registers are refreshed infinitely often.*

Proof. The Lemma holds trivially for r_{ub}. Assume that E is a fair execution in which r_{ba} is never refreshed. This is possible only if whenever BA executes step 7, it finds that the link is closed. By Lemma 3.2 this implies that the content of some register is changed infinitely often. If register r_{ba} is changed infinitely often then we are done, so it must be the case that only the content of register r_{ub} is changed infinitely often. In particular, it implies that there is a suffix E' of E in which the content of r_{ub} is changed infinitely often, but the content of r_{ba} is never changed. We complete the proof by showing that this latter scenario is impossible.

Consider two successive changes of r_{ub} in E', which are done after r_{ub} is refreshed. The first change is done after UB executes step 4, and learns that the link is balanced. Since BA never writes in E', after UB writes (in step 5), the link becomes unbalanced. The next time UB executes step 2 it finds that the link is unbalanced, and hence it does not change the value of r_{ub} anymore – a contradiction. □

Corollary 3.4. *In every fair execution E, UB closes the link, by executing step 3, infinitely often.*

Proof. let E' be a suffix of E in which both registers are refreshed, as guaranteed by Lemma 3.3. Assume that the link is never closed during E'. In particular this implies that UB never changes the color of r_{ub} during E', and also that infinitely often during E', the color of r_{ba} is not equal to the (constant) color of r_{ub}. This implies that eventually, BA balances the link by changing the color of r_{ba}. Once BA have done this, it does not change the color again, unless the link becomes unbalanced. This means that the next time UB executes step 2, it will find out that the link is balanced, and hence it executes step 3 and closes the link – a contradiction. □

Lemma 3.5. *In every fair execution E of the protocol, the system reaches a configuration in the legitimate cycle.*

Proof. By Lemmas 3.3 and 3.4, there is a suffix E' of E during which both processors are refreshed, and during E' UB closes the link. After closing the link, UB reads the value of r_{ba}, changes the color of r_{ub} (if necessary), and opens r_{ub} for reading. Call the sequence of configurations during which UB executes these operations the *closed period* of r_{ub}. Consider the behavior of the link during the closed period of r_{ub}. Whenever BA reads r_{ub} during the closed period, it repeats executing line 7 until r_{ub} is opened. Therefore BA can change the color of r_{ba} (by executing step 8) at most once, during the closed period, since after any such change, BA reads r_{ub}. During the closed period UB reads r_{ba} once. The following three cases sum up the possible ways in which the link descriptor might be changed by BA, during the closed period of r_{ub}:

Case 1. BA does not change r_{ba} during the closed period.

Case 2. BA changes r_{ba} before UB reads from it.

Case 3. BA changes r_{ba} after UB reads from it.

In the first two cases UB reads the updated value of r_{ba} and unbalances the link (if it is not already unbalanced) hence the link descriptor in the configuration that immediately follows the closed period is equal to either $(1,0,0 \longrightarrow 1,?)$ or $(0,1,0 \longrightarrow 0,?)$ (the question mark stands for either 0 or 1), which are all in the legitimate cycle. The third case starts as follows: UB reads r_{ba}, then BA changes the color of r_{ba} to be equal to *ub_color.* At this stage the link descriptor is either $(0,?,1 \longrightarrow 1,1)$ or $(1,?,1 \longrightarrow 0,0)$. At the end of the closed period, UB tries to unbalance the link using the last (not updated) color it read from r_{ba}. Thus the value of the link descriptor in the configuration that immediately follows the closed period is either $(0,1,0 \longrightarrow 1,1)$ or $(1,0,0 \longrightarrow 0,0)$ which are both in the legitimate cycle. □

4 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 tree directed from the (special) root processor to the leaves. The protocol is dynamic as long as the topology changes preserve the tree structure. The tool by which the privilege is passed along links is the balance-unbalance protocol. Each link e, is regarded as directed from UB to BA. The registers of e are called the *unbalance* register of e and the *balance* register of e, respectively. Thus in the tree protocol a processor with *nu_sons* sons plays the role of UB *nu_sons* times, and each normal processor plays the role of BA once. A processor is privileged in the tree protocol if it is privileged in all the balance-unbalance protocols in which it participates, that is when all its outgoing links are balanced and its incoming link (for a non-root processor) is unbalanced. A picture of a node and the registers on its links appears in Fig. 4.1. The code of the protocol, for the root and for a normal processor appears in Fig. 4.2.

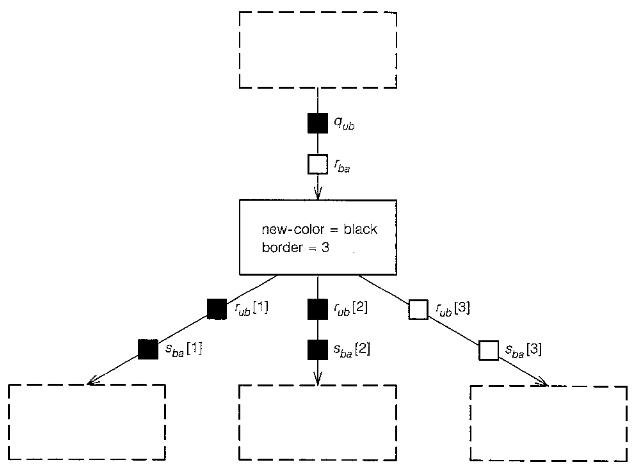

Fig. 4.1. A pictorial description of a node in the system

After stabilizing, an execution of the protocol proceeds in phases. Execution of a phase corresponds to a DFS tour of the whole tree, where the DFS's "center of activity" is at the node of the privileged processor. 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. Processor P that becomes privileged for the first time in some phase, passes the privilege to its leftmost

```
   Root: repeat forever
1            REFRESH
2            for m := 1 to nu_sons
             do ba_color[m] := read (s_ba[m])
3            if (all_out_links_balanced)  {you are privileged}
             then
                CRITICAL SECTION
                if (border = 0)
                then
                   new_color := 1 − new_color
                   border := 1
                end
4               unbalance_border_link
             end

   Other: repeat forever
5            REFRESH
             repeat
6               ⟨ub_color, close⟩ := read (q_ub)
                                          {read unbalance register}
             until close = 0
             for m := 1 to nu_sons
             do ba_color[m] := read (s_ba[m])
7            if (all_out_links_balanced)
             and (in_link_unbalanced)     {you are privileged}
             then
                CRITICAL SECTION
                if (border = 0)
                then
8                  r_ba := write (new_color)
                   new_color := 1 − new_color
                   if (not_leaf) then border := 1
                else
9                  unbalance_border_link
                end
             end

   Procedure unbalance_border_link
                            {pass privilege to your border son}
10    r_ub[border] := write (⟨1 − new_color, 1⟩)
11    ba_color[border] := read (s_ba[border])
      if (border_link_balanced)
      then
12       r_ub[border] := write (⟨new_color, 0⟩)
         border := border + 1 mod (nu_sons + 1)
      else
13       r_ub[border] := write (⟨1 − new_color, 0⟩)
      end

   Procedure REFRESH                {refresh register values}
      if not_leaf
      then
14       for m := 1 to border − 1 do r_ub[m] := write (⟨new_color, 0⟩)
15       for m := border to nu_sons
                            do r_ub[m] := write (⟨1 − new_color, 0⟩)
      end
16    if not_root then r_ba := write (1 − new_color)
```

Fig. 4.2. The mutual-exclusion protocol for dynamic tree systems

son. Once the privilege is passed to all processors in the subtree rooted at this son it is returned to P. Subsequently the privilege is passed to P's second son from the left, and so on. The phase ends when the rightmost son of the root returns the privilege to the root itself. In each phase the privilege is passed twice along each edge, once in each direction. Consequently, each processor becomes privileged d times where d is the processor's degree.

At the beginning of each phase all registers in the tree are colored by one color (say 0), during the phase the tree is recolored by the complementing color (in this case 1). For each edge e, its unbalance register is recolored whenever the privilege is passed through e, and its balance register is recolored when the privilege is returned back through e. Thus in every intermediate configuration the tree is partitioned by a path of unbalanced edges that goes from the root to the DFS's "center of activity", which is the node of the privileged processor. Every unbalance register on this path is colored with the new color (1), while all balance registers on this path are colored with the old color (0). All edges left of this path are colored with the new color (1) while all edges right of the path are colored by the old color (0).

Each processor P has two internal variables called *new_color* and *border*. Variable *new_color* is binary, its value indicates the color in which the tree is recolored. The value of *border* indicates the index of the next son of P to whom the privilege should be passed. This is also the index of the next outgoing link of P to be recolored by *new_color*. When the value of *border* is 0 the privilege should be returned to P's father in the next pass. Therefore the value of *border* ranges between 0 and *nu_sons*. In addition to *new_color* and *border*, each processor has the internal variables needed for all instances of the balance-unbalance protocol in which it participates. The state set of P_i, S_i, contains every possible assignment of permitted values to the variables of P_i.

The protocol for the root and for a normal processor is written for processor P with *nu_sons* sons. The unbalance register of P's incoming link (which is written by P's father and which is read by P) is denoted by q_{ub}, the balance register of P's incoming link (which is written by P and which is read by P's father) is denoted by r_{ba}. The registers of P's outgoing links are denoted as follows: for each outgoing edge e_m, $1 \leqq m \leqq nu\_sons$, the unbalance register of e_m, (which is written by P and which are read by P's son) is denoted by $r_{ub}[m]$ and the balance register of e_m (which is written by P's son and which is read by P) is denoted by $s_{ba}[m]$.

The program for a processor is a loop which is executed forever. Analogous to the balance-unbalance protocol the loop is divided to a refresh block and a main block. The refresh block consists of a subroutine called *REFRESH* in which all the processor's registers are refreshed. Similar to the individual balance-unbalance protocol it can be proved formally (though we do not bother to do it) that *REFRESH* may change a color of a register only until the end of its first complete execution. After *REFRESH* is executed once from its beginning to its end the processor is *refreshed* and the colors in all its registers can be deduced by the values of *border* and *new_color*.

After a processor is refreshed it proceeds to execute its main block. Processor P starts its main block by reading all the balance registers of its sons. In addition a non-root processor repeatedly reads the unbalance register of its father until its incoming link is *open*. Analogous to the balance-unbalance protocol a processor proceeds to execute its main block only if it is privileged. A processor checks that it is privileged by use of two predicates called *all_out_links_balanced* and *in_link_unbalanced* (the root only checks predicate *all_out_links_balanced*). Both predicates are checked using *local values* only, no additional read actions are required. Predicate *all_out_links_balanced* holds if the processor's internal variables indicate that all its outgoing links are balanced, for leaf nodes that have no outgoing links this predicate always holds. Predicate *in_link_unbalanced* holds if $ub\_color = 1 - new\_color$. Since the processor only checks its internal variables it may get an erroneous indication. In the correctness proof below we show that this situation may happen only finitely many times.

In order to prove that the protocol is self-stabilizing we first define the set ME of legitimate sequences of system configurations. Any sequences $s \in ME$ 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.

A subtree T is *uniformly colored* in some configuration C, if in C, all the processors of T are refreshed, all registers of T have the same color and all links of T are open. To get some intuition on how the protocol stabilizes note that after a processor is refreshed, it is privilegd only if all its outgoing links are balanced. When the processor is privileged it "assumes" that all its subtrees left of the border link are colored with *new_color* and that the border subtree itself and all the subtrees right of the border link are colored with the complementing color. When any of its outgoing links is not balanced the processor "waits" until its son balances the link. As the execution proceeds larger subtrees become uniformly colored until the entire tree is uniformly colored. In the sequel we prove that any configuration in which the entire tree is uniformly colored is a safe configuration for the protocol.

Lemma 4.1. *Let E be an arbitrary fair execution. If during E the colors of all registers in the system are constant then there exists a configuration C_t in E such that for every subsequent configuration C_u $(u \geqq t)$ all the system links are open.*

Proof. We prove the lemma by showing that for every processor P_i in the system there is an index $t(i)$ ($t(i)$ depends on E) such that starting in $C_{t(i)}$ and subsequently throughout E all the outgoing links of P_i are open. The proof proceeds by induction on d, the distance of P_i from the root.

Base Case. $d = 0$. In this case P_i is the root processor. Since E is fair the root refreshes its registers infinitely often. Let $C_{t(i)}$ be the configuration reached by the system after the root executes *REFRESH* entirely for the first time. (A processor may start in the middle of *REFRESH*, in this case we only consider the second time in which *REFRESH* is executed.) In $C_{t(i)}$ all the root's outgoing links are open. Following $C_{t(i)}$ the root reads all the balance registers of its sons. Since in E all colors are constant, by the time this read step is over the root has a correct view. The proof proceeds now by assuming that the root closes a link and by applying Lemma 3.2 to show that whenever the root closes a link it subsequently changes the color of the link's unbalance register, a contradiction.

Induction Step. We assume correctness of the lemma for all processors at distance d from the root. Let P_i be an arbitrary processor at distance $d+1$ from the root. We show the existence of a configuration $C_{t(i)}$ after which all P_i's outgoing links are open. Let P_f be the "father" of P_i. The distance of P_f from the root is d. By the induction hypothesis there exists a configuration $C_{t(f)}$ after which P_i's incoming link is open throughout E. Therefore following $C_{t(f)}$ P_i's behavior is similar to the root's behavior and the same proof applies. □

Lemma 4.2. *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 4.1 there exists a configuration C_t $(t \geq 0)$ in E such that for every configuration C_u $(u \geq t)$ all the system links are open.

Case 1. In C_t all the links of the root are balanced. Following C_t and after the root is refreshed, it reads the balance registers of all its sons, discovers that it is privileged and subsequently changes the color of its border link, a contradiction.

Case 2. In C_t at least one of the root's outgoing links is not balanced.

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 $(P_f \longrightarrow P_s)$ of maximal distance from the root. The incoming link of P_s is unbalanced and all the outgoing links of P_s are balanced in any configuration of E. Following C_t and after P_s is refreshed it reads the unbalance register of its incoming link and the balance registers of its outgoing links (if it has any) and discovers that it is privileged. Subsequently P_s unbalances its border link or, if $border = 0$, P_s rebalances its incoming link, a contradiction. □

Corollary 4.3. *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 4.2. □

Lemma 4.4. *If a processor P changes the color of one of its registers infinitely often, then P executes its loop infinitely often.*

Proof. By the observation that during one execution of its loop, P may change the color of each of its registers at most twice (at most once after the first time P completes its loop). □

Lemma 4.5. *Let P be arbitrary processor, let e_1 be P's incoming link and let e_2 be an arbitrary outgoing link of P. Let r^1_{ub} and r^2_{ub} be the unbalance registers of e_1 and e_2 respectively, and let r^1_{ba} and r^2_{ba} be the balance registers of e_1 and e_2 respectively. The following claims hold:*
(a) *If the color of r^1_{ba} is changed infinitely often, so is the color of r^2_{ub}.*
(b) *If the color of r^1_{ba} is changed infinitely often, so is the color of r^1_{ub}.*
(c) *If the color of r^2_{ub} is changed infinitely often, so is the color of r^2_{ba}.*
(d) *If the color of r^2_{ub} is changed infinitely often, so is the color of r^1_{ba}.*

Proof. By Lemma 4.4 each of the conditions in items a–d implies that P executes its loop infinitely often and in particular P is eventually refreshed in E.
(a) Once P is refreshed and throughout E it holds that right before the color of r^1_{ba} changes, the color of r^1_{ba} is not equal to the color of r^2_{ub}. This last assertion implies that between any two successive changes of the color of r^1_{ba}, the color of r^2_{ub} is changed too.
(b) If r^1_{ba} is changed infinitely often, then eventually it is changed only after P executes line 7 and finds that the incoming link e_1 is unbalanced. Assume for contradiction that r^1_{ub} is never changed from some point on. After this point, whenever P writes to r^1_{ba} the link becomes balanced, and it remains so until the next time P checks the condition in line 7. Since the link remains balanced, this condition does not hold. Hence, P never changes r^1_{ba} again, a contradiction.
(c) The proof is similar to the proof of (b).
(d) The proof is identical to the proof of (a). □

Lemma 4.6. *In every fair execution of the protocol the color of every register is changed infinitely often.*

Proof. By Corollary 4.3, the color of some register is changed infinitely often. By a repeated application of (b) and (d) of Lemma 4.5, this implies that the color of a register of the root processor is changed infinitely often. Assume that the root has *nu_sons* sons. Whenever the root colors a register, the value of *border* is increased by 1 (mod *nu_sons* + 1). 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 4.5. □

Since the color of every register in the system changes infinitely often we can now use Lemma 3.5. For this we consider a fair execution of the tree protocol, E, and regard the color changes on the registers of any link during E, as a separate execution of the balance-unbalance protocol. Using this method we get:

Corollary 4.7. *In every fair execution every link descriptor reaches a configuration in the legitimate cycle (of Fig. 3.3).*

We proceed by combining the above results in order to show that eventually the system converges to a configuration in which the entire system tree is uniformly colored.

Lemma 4.8. *There are configurations in E in which the entire system tree is uniformly colored.*

Proof. Let E' be a suffix of E in which all registers are refreshed and all link descriptors follow the legitimate cycle. Consider an arbitrary link $e=(P_f \longrightarrow P_s)$: In E' the predicate *all_out_links_balanced* holds for P_s only when all its outgoing links are indeed balanced. Thus in E', after e is unbalanced, P_s rebalances e (by changing the color of its balance register) only when e is open and after P_s's outgoing links are balanced and colored by the color of the unbalance register of e. Using this fact, we prove the following:

Claim. *Eventually, for each link $e=(P_f \longrightarrow P_s)$, P_s writes* 0 *in the balance register of e only when the subtree rooted at P_s is uniformly colored by* 0.

Proof of claim. The proof is by induction on h, the height of the subtree rooted at P_s. The claim holds trivially if P_s is a leaf. Assume it holds for any processor P such that the height of the subtree rooted at P is smaller then h. Let the height of the subtree rooted at P_s be h. Consider a configuration in which P_f writes 0 into the unbalance register of e, setting the link descriptor to $(1, 0, 0 \longrightarrow 1, 1)$. Since all processors are already refreshed, any additional execution of *REFRESH* does not change the color of the unbalance register of e. Since the link descriptor of e is in the legitimate cycle the unbalance register of e is not changed before P_s rebalances the link by recoloring the balance register of e with 0. It was already proved that P_s rebalances the link only after the registers of all its outgoing links are balanced and colored 0. Since the link descriptors are changed according to the legitimate cycle, those outgoing links are balanced by the sons of P_s. By the induction hypothesis, each son of P_s rebalances its incoming link only after its subtree is uniformly colored with 0, hence when all links outgoing from P_s are balanced with 0 the entire subtree rooted at P_s is colored with 0. 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 4.6), and all the link descriptors are changed according to the legitimate cycle. There is a configuration C' following C in which all the root's outgoing links are balanced to 0 (otherwise the root never colors 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 subtrees rooted at the root's sons are uniformly colored by 0, and hence the entire tree is uniformly colored. □

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

Lemma 4.9. *Eventually there is a single privileged processor in every configuration.*

Proof. Recall that a privileged processor in our protocol is a processor P which is about to execute either line 3 or line 7 of the code and for which the predicates *in_link_unbalanced* and *all_out_link_balanced* are true. Once every link descriptor is in the legitimate cycle, those predicates are true for P only if indeed the incoming link of P is unbalanced and its outgoing links are balanced. To prove the lemma 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 unbalanced. 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 it is the root), and all its outgoing links are balanced. Moreover, all the links in the system are balanced, except the links on the path from the root to P.*

Proof of Claim. By induction on the order of configurations in E. The induction base is a configuration C_t in which all registers and variables are colored 0. By Lemma 4.8 this configuration, which satisfies the claim, is reached in every fair execution. Assume that the claim holds for a certain configuration $C_u (u \geqq t)$. We have to show that the claim holds for C_{u+1}.

Let P be the only privileged processor in C_u. If the atomic step between C_u and C_{u+1} does not change the value of any register then we are done. If the atomic step changes the value of some register say r, then r is a register of P (since P is the only privileged processor, and only a privileged processor may write). The fact that every link descriptor is in the legitimate cycle, implies that if r is the unbalance register of an outgoing link then P unbalances the link, and the claim holds for P's son. If r is the balance register of P's incoming link, then P balances the link, and the claim holds for P's father. This completes the proof of the claim, and hence of the Lemma. □

Corollary 4.10. *The protocol is self-stabilizing relative to the set ME.*

Proof. By Lemma 4.9 at most one privileged processor exist hence the **[exclusion]** requirement holds.

By Lemma 4.6 each processor is privileged infinitely often hence the **[fairness]** requirement holds. □

5 A mutual-exclusion protocol for dynamic networks

In this section we present a mutual-exclusion protocol for dynamic networks. Consider a semi-uniform system in which each link is augmented by two *read only* registers called the *tree-registers*. The tree registers encode a spanning tree of the communication graph rooted at the special processor, as follows: The registers for each link should specify whether the link belongs to the spanning tree; in case the link belongs to the spanning tree its registers should specify its direction; these parameters can be obtained by the processors on the link's endpoints. In this case a self-stabilizing, mutual-exclusion protocol for a system with any (static) communication graph can be derived simply by pre-computing a spanning tree for this system's communication graph and encoding it into the tree registers. The tree registers can be hardwired and therefore constant throughout any execution. The protocol is obtained by executing the mutual-exclusion protocol for tree systems where links which are not in the spanning tree are ignored.

This however falls short of our aim in this paper since the resulting protocol is not dynamic and it requires some pre-computing. On the other hand this protocol motivates the following ideas:

(1) A rooted spanning tree of the communication graph will be computed by a self-stabilizing protocol whose registers are eventually constant.
(2) The registers of the spanning tree protocol will be used as tree registers for the mutual-exclusion protocol.
(3) Both protocols will be combined to achieve a dynamic, self-stabilizing, semi-uniform protocol for mutual-exclusion on general graphs.

In the rest of this section we show how these ideas are implemented.

5.1 A rooted spanning tree protocol

The spanning tree protocol produces a *BFS* tree of the system's communication graph. Let $G(V, E)$ be a graph with orderings $\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_n)$ of the edges incident to each node $v_i \in V$. Define the *First BFS Tree* of G relative to v_1 and α 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 a single neighbor of distance d from v_1, v_i is connected to its first neighbor, according to α_i, 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 node of the special processor and to the (arbitrary) orderings $\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_n)$ in which the neighbors of all processors are ordered. The special processor is called the *root* processor.

Essentially the protocol is a distributed *BFS* protocol. Each processor continuously tries to compute its distance from the root and reports it to all its neighbors by writing it in its registers. At the beginning of an arbitrary execution the only processor which is guaranteed to compute the right distance is the root itself. Once this distance is written in all the root's registers, the value stored in these registers will never change. Once all processors of distance d from the root have completed computing their distance from the root correctly and write it in all their registers, their registers remain constant throughout the execution and processors of distance $d+1$ from the root are ready to compute their own distance from the root and so forth. The spanning tree protocol bears some similarity to the ARPANET routing protocol [14, 15] (that latter protocol assumes a different model).

The output tree is encoded by means of the registers as follows: Each register r_{ij}, in which P_i writes and from which P_j reads, contains a binary *father* field denoted by

```
Root:  do forever
           for $m := 1$ to nu_neighbors do write $t_{im} := \langle 0, 0 \rangle$;
       od

Other: do forever
           for $m := 1$ to nu_neighbors do $ir_{mi} :=$ read $(r_{mi})$;
           first_found := FALSE;
(*)        $dist := \min(ir_{mi}.dist) + 1$;
           for $m := 1$ to nu_neighbors
           do
(**)           if not first_found and $ir_{mi}.distance = dist - 1$
               then
                   write $t_{im} := \langle 1, dist \rangle$;
                   first_found := TRUE;
               else
                   write $t_{im} := \langle 0, dist \rangle$;
           od
       od
```

Fig. 5.1. The spanning tree protocol for P_i

r_{ij}.*father*. If P_j is the father of P_i in the output *BFS* tree then the value of r_{ij}.*father* is 1, otherwise the value of r_{ij}.*father* is 0. In addition each register r_{ij} has a *distance* field, denoted by r_{ij}.*distance* which holds the distance from the root to P_i.

The code of the protocol, for the root and for the other processors, appears in Fig. 5.1. In this code the number of the processor's neighbors is given by the parameter *nu_neighbors*. The program for the root is very simple: It keeps "telling" all its neighbors that it is the root by repeadly writing the values $\langle 0, 0 \rangle$ in all its registers. The first 0 tells each neighbor that it is not the father of the root, the second 0 is the distance from the root to itself. The program for a normal processor consists of a single loop. In this loop the processor reads all the registers of its neighbors. Processor P_i which has *nu_neighbors* neighbors keeps *nu_neighbors* internal variables corresponding to the *nu_neighbors* registers from which P_i reads. The internal variable corresponding to register r_{ji} is denoted by ir_{ji}. This variable stores the last value of r_{ji} read by P_i. Its two fields are denoted by ir_{ji}.*father* and ir_{ji}.*distance* respectively. Once all these registers are read, P_i computes a value for the variable *dist* which represents P_i's current idea of its distance from the root. 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. The minimum in line (*) is taken over m, $1 \le m \le nu\_neighbors$.

The task ST is defined as the set of all configuration sequences in which every configuration encodes the first *BFS* tree of the communication graph. In the following lemma we characterize the set of the safe configurations for the protocol:

Lemma 5.1. *Let C be a system configuration which satisfies:*
(a) *The processors registers encode the first BFS tree of the system's communication graph rooted at the root.*
(b) *For each normal processor P_i and for each of its neighbors P_j, $ir_{ij} = r_{ij}$.*
(c) *For each normal processor P_i, the local variable dist stores the distance of P_i from the root and the value of the local variable first_found satisfies: If the value of the local*

variable m is m_0 and P_i has a neighbor of distance $dist-1$ from the root whose rank according to $\alpha_i < m_0$ then first_found = TRUE, otherwise first_found = FALSE.

Under the above conditions, configuration C is safe for the protocol.

Proof. Every configuration C that satisfies the above three requirements is reachable under some schedule when the system is initialized from the "natural" initial configuration. It is not hard to verify that following C the value of no register ever changes hence all subsequent configurations encode the first *BFS* tree of the system's communication graph as required by the definition of the set *ST*. □

The reader may suspect that requirement (c) is superfluous and that (a) and (b) are enough to ensure that a safe configuration is reached. This possibility is refuted by configuration C' in which (a) and (b) hold but (for example) all *dist* variables are set to 0 and the program counter of each normal processor points to the atomic instruction that starts in line (**). Each normal processor writes $(1,0)$ in the register in which it communicates with its first neighbor in its first atomic step past C'. In its next $d-1$ atomic steps it will write $(0,0)$ in the registers with which it communicates with its other neighbors. This obviously results in a non safe configuration. Furthermore, tampering with *first_found* while keeping the rest of the requirements, (i.e. (a), (b), and half of (c)) may even cause a loss of the tree structure. To prevent these problems we need requirement (c) that guarantees that the internal state of the processor agrees with the values of its registers. This is another demonstration of the deceiving nature of read/write atomicity in the self-stabilizing paradigm which made our job in this algorithm so difficult.

Lemma 5.2. *For every integer $d \geq 0$ there exists an integer t_d such that for every $t > t_d$ the stabilized diameter of the system is at least d. This means that for every pair of neighbors P_i and P_j where the distance of P_i from the root is l, the following hold:*
(a) If $l \leq d$ then r_{ij}.distance $= l$.
(b) If $l \leq d$ then r_{ij}.father has the "right" value. That is: if P_j is the first neighbor of P_i (using α_i) of distance $l-1$ from the root then r_{ij}.father $=1$, and otherwise r_{ij}.father $=0$.
(c) If $l < d$ then $ir_{ji} = r_{ji}$.
(d) If $l > d$ then r_{ij}.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. (Proof for $d=0$) The only node of distance 0 from the root is the root itself. Assume that the root has *nu_neighbors* neighbors. After *nu_neighbors* activiations of the root, all its register store the value $\langle 0,0\rangle$. 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 implied by the root's code 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. For each normal processor P_i assertion (d) is satisfied after P_i executes line (*) once and then completes the outer loop of the protocol since computed value of |fl*dist* is always positive.

Induction Step. (Assume Let t_d be an integer such that for every $t \geq t_d$, configuration C_t satisfies assertions (a)–(d) for some integer $d, d \geq 0$. 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 asserton (a) of the induction hypothesis, for every P_k of distance d from the root, it holds that the value stored in r_{ki}*.distance* in C_{t_d} and all subsequent configurations is d. By assertion (d) of the induction hypothesis, for every P_l of distance $> d$ from the root, it holds that the value stored in r_{li}*.distance* in C_{t_d} and all subsequent configurations is $> d$. Therefore, whenever P_i executes line (*) after C_{t_d}, the value assigned to the variable *dist* is exactly $d+1$. Once this value is written in 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. 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_i be an arbitrary processor of distance $> d+1$ from the root. The neighbors of P_i are all of distance $\geq d+1$ from the root. By assertion (d) of the induction hypothesis starting from C_{t_d} and onwards each neighbor P_j of P_i satisfies r_{ji}*.distance* $> d$. Therefore, whenever P_i executes line (*) after C_{t_d} the value assigned to the variable *dist* is $> d+1$. Once this value is written to all registers of P_i assertion (d) is satisfied for P_i. 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 it satisfies assertion (d) for $d+1$. Let $C_{t_{d+1}}$ be the later configuration among C_2 and C_3. It is easy to see that indeed every configuration C following $C_{t_{d+1}}$ satisfies assertions (a)–(d) for $d+1$. □

Corollary 5.3. *The protocol presented above is self-stabilizing relative to the set ST.*

5.2 Fair combination of self-stabilizing protocols

A self-stabilizing, mutual-exclusion protocol for general graphs can be obtained by combining the self-stabilizing rooted spanning tree protocol, presented in Subsect. 5.1, with the self-stabilizing, mutual-exclusion protocol for dynamic tree-structured systems, presented in Sect. 4, as follows: The combined protocol runs both protocols alternately, such that the latter protocol uses the tree encoded by the tree registers written by the former protocol. By the correctness of the spanning-tree protocol, the tree registers eventually encode a spanning tree, and are subsequently constant throughout the execution. By the correctness of the mutual-exclusion protocol, it eventually converges to a legitimate execution of mutual-exclusion on this spanning tree, and hence on the entire graph.

We now formalize and extend this idea to a general technique of *fair protocol combination.* In this technique two simple protocols, called a "slave" protocol and a "master" protocol, are combined to obtain a more complex protocol. Our dynamic, self-stabilizing, mutual-exclusion protocol is obtained by combining the spanning tree protocol as the slave protocol with the mutual-exclusion protocol presented in the previous section as the master protocol. In the formal definition we assume that both protocols are shared memory protocols but we do not impose any restriction on either the exact model (register or non-register) or on the specific communication graph or on the protocols' atomicity level. Using proper definitions one can also eliminate the shared memory assumption.

Assume that the "slave" protocol is called Pr_1, for a task T_1 and that the "master" protocol is called Pr_2 for a task T_2. The state set of a processor P_i in the combined protocol is $S_i = A_i \times B_i$, where A_i is the state set of Pr_1 and $A_i \times B_i$ is the state set of Pr_2 but we assume that Pr_2 modifies only the B_i components. The state transition function of the slave protocol Pr_1 for processor P_i is a function $f: A_i \longrightarrow A_i$, while the state transition of the master protocol Pr_2 for P_i is a funcion $g: A_i \times B_i \longrightarrow B_i$. These transition functions are extended to functions over S_i as follows: For $(a,b) \in S$, $f((a,b))$ is $(f(a), b)$ and $g((a,b))$ is $(a, g(a,b))$. In the combined mutual-exclusion protocol, Pr_1 is the spanning tree protocol and the A_i's are the states modified by this protocol, including the tree registers; Pr_2 is the version of the mutual-exclusion protocol which uses the tree registers to encode the tree edges on which it operates.

The next definitions formalize the concept of an execution of the master protocol which assumes a self-stabilized execution of the slave protocol. Let S_i, A_i, B_i and Pr_2 be as above, and let T_1 be a task in which the states of processor P_i are in A_i. Assume that T_1 is closed under stuttering (i.e., for each sequence L in T_1, the sequence obtained from L by duplicating each entry finitely many times is also in T_1). For configuration C, $C \in S_1 \times \cdots \times S_n$ define the *A-projection* of C as the configuration $(a_1, \ldots, a_n) \in A_1 \times \cdots \times A_n$. For a sequence of configurations $L = (C_1, C_2, \cdots)$, the A-projection of L is the sequence $(A_1, A_2, \cdots)$, where A_i is the A-projection of C_i. A *fair execution of* Pr_2 *given* T_1 is a sequence of configurations $E = (C_1, C_2, \cdots)$ such that

(a) For every two consecutive configurations $C_i = (A_i, B_i)$ and $C_{i+1} = (A_{i+1}, B_{i+1})$, either $A_i = A_{i+1}$ or $B_i = B_{i+1}$.

(b) If $B_i \neq B_{i+1}$ then the transition from C_i to C_{i+1} is a transition of Pr_2, and the sequence of these transitions is fair (i.e., each processor is activated in it infinitely often).

(c) The A-projection of E belongs to T_1.

Condition (b) says that the modifications of the states in the B_i's are done by Pr_2 in a fair manner, while condition (c) says that the sequence of states in the A_i's forms a legitimate sequence of task T_1.

We say that *protocol* Pr_2 *is self-stabilizing for task* T_2 *given task* T_1 if any fair execution of Pr_2 given T_1 has a suffix in T_2. Finally, a protocol Pr is a *fair combination* of Pr_1 and Pr_2 if in Pr every processor executes steps of Pr_1 and Pr_2 alternately. Note that for an execution E of Pr, the A-projection of E is a sub-execution of E corresponding to a fair execution of the slave protocol Pr_1.

The following theorem gives sufficient conditions under which the combination of two self-stabilizing protocols is also self-stabilizing:

Theorem 5.4. *Assume that* Pr_2 *is self-stabilizing for a task* T_2 *given task* T_1. *If* Pr_1 *is self-stabilizing for* T_1, *then the fair combination of* Pr_1 *and* Pr_2 *is self-stabilizing for* T_2.

Proof. Consider any execution E of Pr, the fair combination of Pr_1 and Pr_2. By the self-stabilizion of Pr_1, E has a suffix E' such that the A-projection of E' is in T_1. By the assumption that Pr_2 is self-stabilizing given T_1, E' has a suffix in T_2. □

Theorem 5.4 provides a general methodology to construct self-stabilizing protocols for complex tasks: Given a task T_2 for which we wish to construct the protocol, first define a task T_1 and construct a protocol Pr_2 which is self-stabilizing for T_2 given T_1, and then construct a protocol Pr_1 which is self-stabilizing for T_1. The fair combination of Pr_1 and Pr_2 is the desired protocol. Note that this methodology does not require that the protocol Pr_1 reaches a "steady state", in which the communication registers (or any other component in the state a_i of processor P_i) are never changed.

Corollary 5.5. *The fair combination of the spanning tree protocol with the mutual-exclusion protocol is a mutual-exclusion protocol on systems with an arbitrary dynamic communcation graph. This protocol is self-stabilizing in the presence of the distributed demon under read/write atomicity.*

We conclude this section by observing that the notion of fair combination of protocols can be further extended, by allowing the protocol Pr to interleave the executions of the protocols Pr_1 and Pr_2 in an arbitrary way, and not necessarily in alternating manner. In this more general setting, each processor switches from executing protocol Pr_1 to executing protocol Pr_2 and vice versa according to some internal conditions, which should guarantee fair execution of both protocols. This adds extra flexibility to the way by which one can achieve composite protocols by combining simpler ones. We demonstrate this by the following general theorem, concerning the fair combination of our mutual-exclusion protocol with an arbitrary self-stabilizing protocol. In this theorem we assume the register model used in the previous sections.

Theorem 5.6. *For any semi-uniform protocol Pr_2 which is self-stabilizing under composite atomicity there is a semi-uniform protocol Pr which is self-stabilizing (for the same task) under read/write atomicity.*

Proof. We describe a protocol *Pr* which simulates Pr_2 under read/write atomicity. *Pr* is a fair combination of Pr_2 as the master protocol and the mutual-exclusion protocol presented above, as the slave protocol Pr_1. We describe *Pr* by describing the rules by which it switches from executing Pr_2 to executing Pr_1 and vice-versa.

Each state transition of Pr_2 under composite atomicity can be written as a sequence of atomic steps under read/write atomicity. Whenever a processor P_i is scheduled to operate, it first checks if it is in its critical section according to Pr_1. If not, then P_i executes a step of Pr_1. If P_i enters its critical section in Pr_1, it stops executing Pr_1 and executes steps of Pr_2, until it completes one state transition of Pr_2 under the composite atomicity (this may take many atomic steps under the read/write atomicity). Once this is done, P_i transfers the privilege to one of its neighbors, according to Pr_1, and so on and so forth. The mutual-exclusion property ensures that as long as P_i does not complete its state transition in Pr_2, no other processor (and in particular no neighbor of P_i) executes any state transition of its own. The fair combination ensures that each processor enters its critical section infinitely often. Thus, each execution of *Pr* has a suffix which is equivalent to a fair execution of Pr_2 under composite atomicty. □

6 Concluding remarks

A semi-uniform, dynamic, self-stabilizing, mutual-exclusion protocol for systems with an arbitrary communication graph was presented. The protocol is correct in the presence of read/write atomicity under the distributed demon. (For protocols that use read/write atomicity the distributed demon and the central are equivalent.) Using this protocol we showed that any self-stabilizing protocol which is correct under composite atomicity can be executed under read/write atomicity in a self-stabilizing fashion.

Although this paper does not concern itself with complexity measures it is worth mentioning that when time is measured by some appropriately defined round complexity, the stabilization time of the spanning tree protocol is $O(D)$, where D is the diameter of the system's communication graph. The stabilization time of the mutual-exclusion protocol is $O(nD)$.

Acknowledgement. We thank three anonymous referees for their constructive remarks on an earlier of this paper. The help of Frank Stomp is also appreciated.

References

1. Brown GM, Gouda MG, Wu CL: A self-stabilizing token system. In: Proc 20th Annual Hawaii International Conference on System sciences, pp 218–223, 1987
2. Burns JE, Pachl J: Uniform self-stabilizing rings, ACM Trans Program Lang Syst 1(2): 330–344 (1989)
3. Burns JE: Self-stabilizing rings without demons. Tech Rep GIT-ICS-87/36, Georgia Institute of Technology, 1987
4. Dijkstra, EW: Self-stabilizing systems in spite of distributed control. Commun ACM 17(11): 643–644 (1974)
5. Dijkstra EW: Self-stabilizing systems in spite of distributed control (EWD 391). Reprinted in: Selected writing on computing: a personal perspective. Springer, Berlin Heidelberg New York 1982, pp 41–46
6. Dijkstra EW: A belated proof of self-stabilizion. Distrib Comput 1(1): 5–6 (1986)
7. Dolev S, Israeli A, Moran S: Self-stabilization of dynamic systems assuming only read/write atomicity (preliminary version) Proc MCC Workshop on Self-Stabilization, Austin, Texas, November 1989. Also in: Proc 9th Annual ACM Symposium on Principles of Distributed Computing, pp 103–117, 1990
8. Israeli A, Jalfon M: Token management schemes and random walks yield self-stabilizing mutual exclusion. In: Proc 9th Annual ACM Symposium on Principles of Distributed Computing, pp 119–131, 1990
9. Israeli A, Jalfon M: Uniform self-stabilizing ring orientation. Inf Comput 104: 175–196 (1993). Also in: Van Leeuwen J, Santoro N (eds) Distributed Algorithms (Proceedings of the Fourth International Workshop on Distributed Algorithms, Bari, Italy, September 1990). Lect Notes Comput Sci, vol 486. Springer, Berlin Heidelberg New York 1991, pp 1–14
10. Katz S, Perry KJ: Self-stabilizing extensions for message-passing systems. Distrib Comput 7: 17–26 (1993). Also in: Proc 9th Annual ACM Symposium on Principles of Distributed Computing, pp 91–101, 1990
11. Kruijer HSM: Self-stabilizion (in spite of distributed control) in tree-structured systems. Inf Process Lett 8(2): 91–95 (1979)
12. Loui MC, Abu-Amara HH: Memory requirements for agreement among unreliable asynchronous processes. In: Preparata FP (ed) Advances in computing research. JAI Press 1987, pp 163–183
13. Peterson GL, Fischer MJ: Economical solutions for the critical section problem in a distributed system. In: Proc ACM Symposium on Theory of Computing, pp 91–97, 1977
14. Tajibnapis WP: A correctness proof of a topology information maintenance protocol for a distributed computer network. Commun ACM 20(7): 477–485 (1977)
15. Tanenbaum AS: Computer networks. Prentice-Hall, 1981, pp 205–231
16. Tchuente M: Sur l'auto-stabilisation dans un r'eseau d'ordinateurs, RAIRO Inf Theor 15: 47–66 (1981)