Uniform Dynamic Self-Stabilizing Leader Election

Shlomi Dolev, Amos Israeli, and Shlomo Moran

Abstract—A distributed system is *self-stabilizing* if it can be started in any *possible* global state. Once started the system regains its consistency by itself, without any kind of outside intervention. The self-stabilization property makes the system tolerant to faults in which processors exhibit a faulty behavior for a while and then recover spontaneously in an arbitrary state. When the intermediate period in between one recovery and the next faulty period is long enough, the system stabilizes. A distributed system is *uniform* if all processors with the same number of neighbors are identical. A distributed system is *dynamic* if it can tolerate addition or deletion of processors and links without reinitialization. In this work, we study uniform dynamic self-stabilizing protocols for leader election under readwrite atomicity. Our protocols use randomization to break symmetry. The leader election protocol stabilizes in $O(\Delta\mathcal{D}\log n)$ time when the number of the processors is unknown and $O(\Delta\mathcal{D})$, otherwise. Here Δ denotes the maximal degree of a node, $\mathcal{D}$ denotes the diameter of the graph and n denotes the number of processors in the graph. We introduce self-stabilizing protocols for synchronization that are used as building blocks by the leader-election algorithm. We conclude this work by presenting a simple, uniform, self-stabilizing *ranking* protocol.

Index Terms—Self-stabilizing systems, leader election, distributed algorithms, randomized distributed algorithms, synchronization.

1 Introduction

Leader-election is one of the fundamental tasks in distributed computing. Roughly speaking, a protocol that solves this task requires that when its execution terminates, a single processor is designated as a *leader* and every processor knows whether it is a leader or not. By definition, whenever a leader-election protocol terminates successfully, the system is in a nonsymmetric global state. In many cases, once a leader is elected the distributed task is solved by means of a central solution i.e., the leader controls the activity in the distributed system. A partial list of distributed tasks that can be easily realized in the presence of a leader include: consensus, resource allocation and synchronization. Therefore, it is not surprising that the leader election problem has been extensively studied (see for example, [18], [20], [23], [24], [25], [30]).

The number of processors and the sometimes noisy communication media in a distributed system impose the need for a fault tolerant design. One strong notion of fault tolerance is *self-stabilization*. Roughly speaking, a self-stabilizing protocol can cope with any kind of faults in the history. A distributed system is *self-stabilizing* if it can be started in any *possible* global state. Once started, the system runs for a while until it reaches a *legitimate* global state in which the system is consistent. The self-stabilization property makes the system tolerant to faults in which processors exhibit a faulty behavior for a while and then recover spontaneously in an arbitrary state. When the intermediate period between one recovery and the next faulty period is long enough, the system stabilizes.

Any leader-election protocol that has a symmetric initial state requires some means of symmetry breaking. In *id-based* systems each processor has a unique identifier called the processor's *id*, hence the system has no symmetric global-state. A *semiuniform* system has two kinds of processors: a unique predetermined processor of one type and all other processors are of the other type. The unique processor serves as a leader and prevents the existence of symmetric configurations. In *uniform*[1] leader-election protocols, all processors are identical, the initial state is symmetric and symmetry is broken by randomization. Such setting is very useful when the processors are fabricated in a uniform process without assigning each processor by a unique identifier. Note that even in the semiuniform setting some outside coordination is required to ensure that there exists a unique processor in the system. This coordination is specially hard in dynamic environment where processor may join and leave the system during the execution. Another motivation for the uniform system setting is the possibility of outside coordination mistakes such as assigning the same identifier to two processors. A uniform system does not rely on such outside coordination.

1.1 Previous Work

Self-stabilizing systems were introduced in the seminal paper of Dijkstra, [14]. In that paper, Dijkstra presents three semiuniform, self-stabilizing, ring protocols for mutual-exclusion. Other semiuniform, mutual-exclusion, self-stabilizing ring protocols which work under a stronger adversary, called the *distributed demon* were presented by Brown, Gouda, and Wu in [10] and by Burns in [12]. Two

- *S. Dolev is with the Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva, 84105, Israel. E-mail: dolev@cs.bgu.ac.il.*
- *A. Israeli is with Intel, Haifa, 31015 Israel.*
- *S. Moran is with the Computer Science Department, Israel Institute of Technology, Technion, Israel.*

Manuscript received May 28, 1994
For information on obtaining reprints of this article, please send e-mail to: transpds@computer.org, and reference IEEECS Log Number D95228.

1. Uniform systems are also referred to as *anonymous.*

papers considered self-stabilizing, mutual-exclusion protocols for general (connected) graphs: The first was authored by Tchuente, in [31], who presented a nonuniform protocol for that problem. A semiuniform protocol for the same problem was presented by Dolev, Israeli, and Moran in [15]. The work of [15] was the first to propose the read-write atomicity model and their protocol is the first protocol that is self-stabilizing under this model. A self-stabilizing, *id*-based protocol for mutual exclusion in complete graphs is presented by Lamport, in [27]. This protocol has exponential space complexity. Protocols for leader election in the *id*-based model for a general graph are presented by Arora and Gouda, in [5] and by Afek, Kutten, and Yung in [6]. Both protocols assume read-write atomicity.

So far, there are very few uniform self-stabilizing protocols: Burns and Pachl present a uniform, deterministic, self-stabilizing, mutual exclusion protocol for rings of prime size in [11]. Randomized, uniform, self-stabilizing protocols for mutual exclusion in a general graph and for ring orientation are presented by Israeli and Jalfon in [21] and [22] respectively. If one could run id-based and semiuniform protocols on a uniform system, the repertoire of uniform self-stabilizing protocols would be considerably enlarged. Let $\mathcal{PR}$ be an arbitrary semiuniform, self-stabilizing protocol. To run $\mathcal{PR}$ on a uniform system we employ a uniform, self-stabilizing leader-election protocol and combine it with $\mathcal{PR}$, using fair protocol composition—a technique presented in [15]. Our self-stabilizing ranking protocol assigns the processors with unique identifiers. Similarly to a semiuniform self-stabilizing protocol, it is possible to combine an id based protocol $\mathcal{PR}$ with the ranking protocol. This combined protocol can be applied to uniform systems.

Our protocol is the only protocol which solves the problem without any prior knowledge on the communication graph. Assuming some known bound on the graph's diameter solutions to the same problem have been suggested independently in [6], [4] and in [8]. The time complexity of all these solutions is inferior to the time complexity of our protocol.

There are many non-self-stabilizing distributed protocols for leader election. We now survey the most related ones: Deterministic, leader-election protocols in *id*-based systems are presented in [18], [20], [25], [24]. Uniform, randomized, leader-election protocols are presented in [23], [30], [28]. Other protocols for uniform systems appear in [2], [1].

1.2 The Current Work

In [17] we presented the computation model, a proof technique for randomized algorithms and two uniform self-stabilizing leader election protocols for complete communication graphs. In the current work we present a uniform, dynamic, self-stabilizing, leader election protocol for general graph systems.

The resources used for stabilization are execution time and memory. The following complexity measures capture the amount of resources required by our protocols:

- Stabilization time and
- Space.

The stabilization time of a self-stabilizing protocol is the maximal time (measured in asynchronous rounds which is precisely defined in the next section) it takes the system to reach a legitimate configuration where the maximum is taken over all possible executions. We consider stabilization time a very important complexity measure and carefully analyze our protocols' stabilization time. To do that, we use the *sl-game method* [17], for proving upper bounds on the time complexity of randomized distributed protocols. During the execution of our protocol each processor may extend the amount of memory it uses. The space complexity of our protocol is the expected number of extension bits per a processor. According to the above definition, the space complexity of our protocol is $O(\log n)$.

The rest of this paper is organized as follows: In Section 2, we present the formal model and requirements for uniform, self-stabilizing protocols. Section 3 presents the general graph leader election protocol, and the self-stabilizing synchronization protocols. Section 4 presents the self-stabilizing ranking protocol. Conclusions are in Section 5. The Appendix contains the notations and terms.

2 Model and Requirements

The model is identical to the one presented in [17]. A *uniform distributed system* consists of n processors denoted by $P_1, P_2, \ldots, P_n$. Processors are *anonymous*, they do not have identities. The subscript 1, 2, …, n are used for ease of notation only. Each processor communicates with all other processors using a single writer, multireader register which is serializable with respect to read and write actions. For the sake of clarity, we assume that every processor knows the exact contents of the register that it is writing to.[2]

For ease of presentation, we regard each processor as a *CPU* whose program is composed of *atomic steps*. An atomic step of a processor consists of an internal computation followed by a terminating action. The terminating actions are **read**, **write** and **coin toss**. We assume that the state of a processor fully describes its internal state and the value written in its register. Denote the set of states of P_i by S_i. A *configuration*, $c \in (S_1 \times S_2 \times \ldots S_n)$, of the system is a vector of states of all processors.

Processor activity is managed by a scheduler. In any given configuration, the scheduler activates a single processor which executes a single atomic step. To ensure correctness of the protocols, we regard the scheduler as an adversary. The scheduler is assumed to have unlimited resources, and it chooses the next activated processor *on line*, using the full information on the execution so far. An *execution* of the system is a finite or an infinite sequence of configurations $E = (c_1, c_2, \ldots)$ such that for $i = 1, 2, \ldots, c_{i+1}$ is reached from c_i by a single atomic step of some processor. A *fair execution* is an infinite execution in which every processor executes atomic steps infinitely often. A scheduler S is *fair* if, for any configuration c, with probability one, an execution starting from c in which processors are activated by S is fair.

In a distributed asynchronous system, each processor may operate at any nonconstant rate and different proces-

2. One may assume that every processor refreshes the contents of its register periodically.

sors might be slow in different parts of the execution. The following definition of round complexity attempts to give a complexity measure in which the unfair behavior of the adversary is neutralized, by capturing the rate of action of the slowest processor in any segment of the execution. Given an execution E, we define the *first round* of E to be the minimal prefix of E, E', containing atomic steps of every processor in the system. Let E'' be the suffix of E for which $E = E' \circ E''$. The second round of E is the first round of E'', and so on. For any given execution, E, the *round complexity* (which is sometimes called the execution time) of E is the number of rounds in E. Under this definition the time to complete a single round is unbounded and depends on the fairness of the adversary. Any self-stabilizing application that uses our protocol as a subroutine would probably also require fair behavior to stabilize and its complexity will be proportional to the stabilization complexity of our protocol.

We proceed by defining the self-stabilization requirements for randomized distributed systems. A behavior of a system is specified by a set of executions. Define a *task LE* to be a set of executions which are called *legitimate executions*. A configuration c is *safe* with respect to a task LE and a protocol $\mathcal{PR}$ if *any* fair execution of $\mathcal{PR}$ starting from c belongs to LE. Finally, a protocol $\mathcal{PR}$ is *randomized self-stabilizing* for a task LE, if starting with any system configuration and considering any fair scheduler, the protocol reaches a safe configuration within an expected number of rounds which is bounded by some constant C (the constant C may depend on n, the number of processors in the system).

3 Leader Election in General Graphs

3.1 Informal Description of the Protocol

In this protocol each system configuration c encodes a directed graph called the *FSG* (*father-son* relation graph) of c and denoted by $FSG(c)$. A safe configuration in this protocol is a configuration whose *FSG* is a single in-tree (called tree in the sequel) which contains all processors and for which during any execution that begins in that configuration the tree is not changed; the root of the *FSG* is the elected leader.

The protocol consists of two conceptual phases which are called *cycle elimination* and *tree fusion*. During the cycle elimination phase all cycles in the *FSG* are removed. In the tree fusion phase the number of trees in the *FSG* is reduced until it consists of a single tree. Coin tosses are used in the tree fusion phase in order to break symmetry between trees. Normal operation and completion of tree fusion depends crucially on completion of the cycle elimination phase. By the nature of self-stabilizing protocols the completion of the cycle elimination phase is undetectable locally by the processors. Hence, the cycle elimination phase does not terminate and is executed together with tree fusion phase.

The *FSG* is defined by a relation between neighbors called *father-son* relation. Each processor can either be a *root* or can have a father, which is one of its neighbors. If P_i is the father of P_j in configuration c then there is a directed edge from P_j to P_i in $FSG(c)$. Thus, in any configuration c, there are at most n edges in $FSG(c)$. Each tree of $FSG(c)$ is identified by a binary string which is called *tree-identifier*, and abbreviated *tid*. A root is the only processor which changes the tree's *tid*; this is always done by extending the *tid* with a randomly chosen bit. Each non-root processor repeatedly copies its father's *tid*. Hence in every execution, eventually all processors in a tree T, hold a prefix of the *tid* of the root of T.

To achieve cycle elimination each processor computes the *distance* to the root of its tree. Every processor computes the distance to the root of its tree by adding one to the distance of its father from the root. Whenever the processor realizes that this distance "grows" it (assumes that it is part of a cycle in *FSG* and) cuts the edge to its father and becomes a separate root. After this phase is completed *FSG* is a forest of trees.

To reduce the number of trees to one we first ensure that eventually there is a unique *tid* in the system. Each processor repeatedly scans its neighbors *tid*s. Whenever a processor P_i discovers a neighbor P_j whose *tid* is larger than its own *tid*, P_i takes P_j to be its father. If previously P_i is a root, the number of trees is reduced by one. We prove that taking a new father never introduces new cycles in $FSG(c)$. This however does not ensure that eventually there is a single tree since there might be several trees with the same *tid*.

A root processor discovers that there are other roots with the (same) maximal *tid* by repeatedly recoloring its tree using a global synchronization protocol. Each recoloring starts from the root which chooses the new color randomly. The root waits for each of its sons to confirm that every node in its subtree is recolored. Once the entire tree is recolored the root chooses a new color once more, and so on. A processor of a tree T detects the existence of another tree T' with the same *tid*, by observing that one of its neighbors is colored neither by the previous color of T, nor by its current color. In this case the processor "returns" this information to the root of T. Upon receipt of this information the root of T extends its *tid* by a random bit which is distributed again along the edges of T. At the same time T' may also extend its own *tid*. Since each extension is done randomly, symmetry is eventually broken and the system reaches a *leader configuration*. A leader configuration is a configuration with exactly one leader such that in any execution that starts with this configuration the leader is fixed. Once there exists exactly one leader in the system the protocol ensures that this leader extends his *tid* at most once (before a safe configuration is reached).

3.2 Formal Description of the Protocol

The code that appears in Fig. 1, Section 3.2, below, is written for processor P_i that has Δ neighbors. Each processor, P_i, owns a register in which it writes and all its neighbors read. The register of P_i consist of the following fields: tid_i, dis_i, f_i, $color_i$, ack_i, and ot_i.

- tid_i The field tid_i indicates the identity of the tree to which P_i belongs in the *FSG*.
- dis_i The field dis_i indicates the distance of P_i from the *root* of the tree it belongs to. In case $dis_i = 0$, P_i is a root processor.
- f_i In case $dis_i \neq 0$, the value in the field f_i indicates which of the neighbors of P_i is its father. f_i holds the index of the link of P_i that connect P_i with its father. Thus, the neighbor of P_i that is connected to P_i through this link can determine by the value of f_i

and dis_i that P_i is its son[3].

Color For the execution of the global synchronization protocol, as we describe in the sequel, we use the *color* field. *color* may contain eight colors which are denoted by integers of values 0 to 7.

Ack The *ack* field is also used for the global synchronization protocol (below). *ack* is a boolean field.

Ot The field *ot* is assigned by *true* whenever a processor notes that one of its neighbors has *color* that is different from both its own current *color* and the new *color* of its father.

Each time P_i reads the value of f_j, dis_j, tid_j, $color_j$, ack_j, ot_j of its neighbor P_j, P_i assigns those values in its internal variables $f_i[j]$, $dis_i[j]$, $tid_i[j]$, $color_i[j]$, $ack_i[j]$, and $ot_i[j]$, respectively.

The function *extend_tid* chooses a random bit and concatenates it to the tail of the current *tid*. The function *son*(*j*) executed by P_i is true only when P_j is the son of P_i, this is indicated by the value of $f_i[j]$. The function *choose_color* chooses randomly between six (out of the possible eight) colors that are not equal to the *previous_color* and to (the current) *color* and assigns *previous_color* by the value of (the current) *color*.

We define the lexicographic order between two values of *tid* fields as follows: Let tid_1 and tid_2 be two values of *tid* fields. The relation $tid_1 > tid_2$ is true when tid_2 is a prefix of tid_1, or (when neither tid_1 nor tid_2 is a prefix of the other) if tid_1' and tid_2' are derived from tid_1 and tid_2 by removing their common maximal prefix and the first bit of tid_1' is 1. Define the relation $\succ$ over pair of processors fields (*tid, dis*) as follows: $((tid_1, dis_1) \succ (tid_2, dis_2))$ if $tid_1 > tid_2$ (as defined above) or when $tid_1 = tid_2$, then $dis_1 < dis_2$. In such a case we say that the pair (tid_1, dis_1) is *greater* than (tid_2, dis_2).

3.2.1 Description of the Code

The code of the protocol appears in Fig. 1 and Fig. 2. The code consists of a single infinite do forever loop, the lines of this loop are described below.

Line 2 - P_i reads the registers of its neighbors.

Lines 3 *to* 10 - Using the values read, P_i calculates the maximal *tid* among the *tid*s of its neighbors. Then P_i finds the minimal *dis* of a neighbor among the neighbors holding the maximal *tid*. At last P_i finds the index of the first neighbor holding the above maximal *tid* and minimal *dis* and updates the local variables *F*, *C*, *A*, and *OT* accordingly.

Line 11 *to* 13 - If P_i finds that it has no neighbor P_j with $(tid_j, dis_j) \succ (tid_i, dis_i)$ then P_i becomes a root. Otherwise, P_i updates the values of its *tid* and *dis* according to the values it reads.

Line 14 - When P_i finds that for every neighbor it holds that $tid_i = tid_j$ and $|dis_i - dis_j| \leq 1$, P_i assumes that the cycle elimination is over and joins trees fusion.

3. It is assumed that P_i knows the indices given by its neighbors to every of its links.

```
1  do forever
   (* Reading *)
2  for j:=1 to Δ do (tid_i[j], dis_i[j], f_i[j], color_i[j], ack_i[j], ot_i[j])
   := read(r_j);
3  max_tid := max(tid_i[j]);
4  min_dis := min{dis_i[j] | tid_i[j] = max_tid};
5  if (tid_i[f_i], dis_i[f_i]) = (max_tid, min_dis) then
                    (F, C, A, OT) := (f_i, color_i, ack_i, ot_i)
6  else
7    begin
8          F := {first j | tid_i[j] = max_tid, dis_i[j]
                                = min_dis};
9          (C, A, OT) := (color_i[F], false, false);
10   end
   (* Cycles elimination *)
11 if (tid_i, dis_i) ≥ (max_tid, min_dis)
12   then write (dis_i, f_i) := (0, nil) (* become a root *)
13 else write (tid_i, dis_i, f_i, color_i, ack_i, ot_i)
                := (max_tid, min_dis + 1, F, C, A, OT);
14 if {∀ j | (tid_i[j] = tid_i) and (| dis_i – dis_i[j] | ≤ 1) then
             tree_fusion;
15 end
```

Fig. 1. Leader election in general graph systems.

```
16 Procedure tree_fusion
   (* Root *)
17 if (dis_i = 0) and {∀j | son(j) ⇒ ((color_i[j] = color_i) and
                                ack_i[j])} then
18   begin
19         if {∃ j | (ot_i[j] = true) or (color_i[j] ≠ color_i)}
20               then write (tid_i, ack_i)
                          := (extend_tid(tid_i, false));
21         write color_i := choose_color(previous_color,color_i) ;
22   end
23 else if (dis ≠ 0) then (* Nonroot *)
24   if (color_i ≠ color_i[f_i]) then
25         begin
26               if {∃ j | color_i[j] ∉ (color_i, color_i[f_i])}
                          then write ot_i := true
27               else write ot_i := false;
28               write (color_i, ack_i)
                          := (color_i[f_i], false) ;
29   end
30   else if not ack_i and {∀ j | son(j) ⇒ ((color_i[j] = color_i)
                          and ack_i[j])} then
31         begin
32               if {∃j | son(j) and ot_i[j] = true}
                    then ot_i := true;
33               write ack_i := true;
34         end
35 end
```

Fig. 2. Procedure tree_fusion.

Lines 17 *to* 22 - These lines consider the case in which P_i is a root and discovers that it finished coloring its tree with a single color. In this case P_i checks whether any processor of its tree detected the existence of another tree. If the existence of such a tree is detected P_i extends its *tid*. Then P_i starts coloring its tree with a new color.

Lines 23 *to* 29 - These lines consider the case in which P_i is not a root and discovers that its father is colored with a new color. If P_i finds that it has a neighbor P_j for which $color_j \notin \{color_i, color_i[f]\}$ it concludes that P_j belongs to another tree. P_i writes its conclusion in ot_i and then sets $color_i$ to the color of its father.

Lines 30 *to* 34 - These lines consider the case in which P_i is not a root and discovers that its sons finished coloring their subtrees with the color of $color_i$. P_i collects the indications on the existence of other trees and report to its father that it finished coloring its subtree.

3.3 Correctness and Complexity Proofs

Toward proving the correctness of the protocol we first describe the synchronization building blocks used by the protocol. Then we prove that the system reaches a configuration after which there is a single tree with a leader. The proof is completed by showing that this tree is fixed.

3.3.1 Synchronization

In this subsection we present two self-stabilizing synchronization protocols: a local synchronization protocol and a global-synchronization protocol. The local-synchronization protocol is designed for a two processor system. The global-synchronization protocol is an extension of the local-synchronization protocol and is designed for a tree structured system. These protocols are used as components in a protocol for leader election in general graphs. Procedure tree_fusion in Fig. 2 uses a global synchronization protocol. Nevertheless, both synchronization protocols are of an independent interest. The global-synchronization protocol, that stabilizes in $O(\Delta\mathcal{D})$ rounds, may be used as an efficient self-stabilizing synchronizer that implements self-stabilizing synchronous protocols in an asynchronous system. The global-synchronization protocol can also be used as an efficient self-stabilizing snapshot and reset protocol (see [26], [5]).

3.3.1.1 Self-Stabilizing Local-Synchronization Protocol

A local-synchronization protocol is designed for a system that consists of two processors P_f and P_s and two registers $color_f$ and $color_s$. P_f (P_s) writes in $color_f$ ($color_s$) and reads from $color_s$ ($color_f$). The color of P_f (P_s) is the value stored in $color_f$ ($color_s$), respectively.

Informally the task of the local-synchronization protocol is to ensure that P_f changes its color infinitely often and to ensure that following every time that P_f changes its color P_s changes its color to the color of P_f and only then P_f changes its color again. More precisely the task of the local-synchronization protocol is defined by a set of executions in which:

- P_f changes the value of $color_f$ infinitely often.
- Immediately before any change of the value of $color_f$ it holds that $color_f = color_s$.
- Immediately after any change of the value of $color_s$ it holds that $color_s = color_f$.

The local-synchronization protocol is defined below. During the execution of the protocol whenever P_f (P_s) reads the value of $color_s$ it assigns this value in $color_f[s]$ ($color_s[f]$, respectively). Therefore, the value of $color_f[s]$ ($color_s[f]$) is the value obtained in the last time, if any, P_f (P_s) read $color_s$ ($color_f[s]$, respectively). The protocol uses three colors (or more). P_f uses the function *choose_color* which always selects a color that is different from both the current color and the previous color of P_f. To do so P_f has an internal variable called *previous_color*. Every time the *choose_color* function is executed it chooses a color that is equal neither to *previous_color* nor to $color_f$ then assigns *previous_color* := $color_f$ and at last returns the color that it chooses.

Program of P_f—P_f repeatedly executes:

(lf1) reads $color_s$ into $color_f[s]$ and
(lf2) if $color_f[s] = color_f$ then it assigns $color_f$:= *choose_color*(*previous_color*, $color_f$).

Program of P_s—P_s repeatedly executes:

(ls1) reads $color_f$ into $color_s[f]$ and
(ls2) if $color_s[f] \neq color_s$ then it assigns $color_s := color_s[f]$.

In the following lemma we show that any execution of the local synchronization protocol stabilizes after P_s executes a constant number of atomic steps. The lemma uses the following definition which will be used throughout the paper.

DEFINITION 1. *Let Q be a program that consists of exactly one infinite do forever loop such that the first (last) line of Q is the first (last) line of this loop. A processor that executes Q completes a loop iteration during an execution E if it executes the first line of the loop during one atomic step of E and executes the last line of the loop in a later atomic step of that execution.*

Let l be a bound on the number of atomic steps executed during arbitrary execution E, that starts with an atomic step in which the processor executes the first line of the loop of Q and ends with the first successive execution of the last line of the loop of Q. By the nature of self-stabilizing protocols a processor might start with any atomic step in the loop of Q. Thus, the number of atomic steps needed to complete a loop iteration is bounded by $2l - 1$.

The next Lemma uses k (and not 2) for the number of atomic steps needed to complete a loop iteration by P_s. This choice prepares the way for future reasoning.

LEMMA 1. *If k is the number of atomic steps needed to complete a loop iteration by P_s then after at most $4k$ atomic steps of P_s the local synchronization protocol is stabilized.*

PROOF. First we observe that a configuration in which $color_f = color_s[f] = color_s$ is a safe configuration for the local synchronization protocol. Once such a configuration is reached and until P_f changes the value of $color_f$, P_s cannot change the value of $color_s[f]$ and $color_s$. When P_f changes the value of $color_f$, P_f does not change the value of $color_f$ again before the system reaches a configuration c' in which $color_f = color_s[f] = color_s$, and so forth. Thus, it is sufficient to prove that such a configuration is reached within $4k$ atomic steps of P_s.

First we assume that $color_f$ is not changed during k successive atomic steps of P_s. In this case during these k atomic steps P_s reads $color_f$ and writes this color in $color_s$. Thus, a configuration in which $color_s = color_s[f] = color_f$ is reached and the system is stabilized.

Now consider executions in which $color_f$ is changed every at most k atomic steps of P_s. Consider such execution, E, that contains 4 successive changes of $color_f$. Let $color^i$ be the color assigned to $color_f$ in the ith color change of $color_f$. Note that E contains at most $4k$ atomic steps of P_s. E contains the following steps, in the specified order:

1) P_f writes $color^1$ to $color_f$
2) P_f reads $color^1$ in $color_s$
3) P_f writes $color^2$ to $color_f$
4) P_f reads $color^2$ in $color_s$
5) P_f writes $color^3$ to $color_f$
6) P_f reads $color^3$ in $color_s$

By the programs of P_f and P_s it holds that: Between Step 2 and Step 4, P_s must write $color^2$ in $color_s$ and between Step 4 and Step 6, P_s must write $color^3$ in $color_s$. Hence, between Step 2 and Step 6, P_s must read $color^3$ in $color_f$. Since $color^1$, $color^2$, $color^3$ are distinct, that read operation of P_s must occur after Step 5 (and before Step 6). Thus, between Step 5 and Step 6, P_s reads $color^3$ in $color_f$ and then writes $color^3$ in $color_s$. Immediately after this write operation, $color_f = color_s[f] = color_s$ as required. □

3.3.1.2 Self-Stabilizing Global-Synchronization Protocol

A directed graph is an *in-tree* if the undirected underlying graph is a tree, and if every edge of the tree is directed towards a common root. For the sake of readability we use the term tree instead of in-tree. A global synchronization protocol is a protocol for a tree structured system, with a root P_r. The global-synchronization protocol uses two fields of a register for each processor. A processor, P_i, writes in two fields called $color_i$ and ack_i that are read by any of its neighbors in the tree.

Informally the task of a global-synchronization protocol is to ensure that P_r changes its color infinitely often. Following every time that P_r changes its color all the processors in the tree change their color to the color of P_r and only then P_r changes its color again. More precisely the task of a global-synchronization protocol is defined by a set of executions in which:

1. P_r changes the value of $color_r$ infinitely often.
2. For any processor P_i, immediately before any change of the value of $color_r$ it holds that $color_i = color_r$.
3. For any processor P_i, immediately after any change of the value of $color_i$ it holds that $color_i = color_r$.

The global-synchronization protocol is defined below. During the execution of the protocol whenever P_i reads the value of $color_j$ (ack_j) it assigns this value in $color_i[j]$ ($ack_i[j]$, respectively). To determine a new color P_r uses the *choose_color* function that was defined above.

Program of P_r—P_r repeatedly executes:

(gr1) Reads the fields $color_i$ and ack_i of each of its sons P_i and
(gr2) If for every son it holds that $color_r[i] = color_r$ and $ack_r[i] = true$ then it sets $color_r := choose\_color(previous\_color, color_r)$.

Program of P_i $i \neq r$—Let P_f be the father of P_i in the tree. P_i repeatedly executes:

(gi1) Reads the field $color_f$ of its father and the fields $color_s$ and ack_s of each of its sons P_s, and
(gi2) If $color_i[f] \neq color_i$ then P_i assigns $color_i := color_i[f]$ and $ack_i := false$, otherwise
(gi3) If for each of its sons, P_s, $color_i[s] = color_i$ and $ack_i[s] = true$, then P_i assigns $ack_i := true$.

LEMMA 2. *For any tree of depth $\mathcal{D}$, in which Δ is the maximal degree of a node, the global-synchronization protocol stabilizes within $O(\Delta\mathcal{D})$ rounds.*

PROOF. The program of each processor in the global-synchronization protocol consists of one infinite do forever loop. There is a constant k_1 such that a loop iteration of a processor with Δ neighbors in the tree is completed within $k_1\Delta$ atomic steps. Hence $k_1\Delta$ is an upper bound on the number of atomic steps needed for a processor to complete at least one loop iteration of the global synchronization protocol.

First we show that requirement 1 of a global synchronization protocol holds. In fact we show that P_r changes its color at least once every $2k_1\Delta\mathcal{D} + k_1\Delta$ rounds: Assume that $color_r$ is not changed during an execution E of $2k_1\Delta\mathcal{D} + k_1\Delta$ rounds. Let P_i be a neighbor of P_r. During the first $k_1\Delta$ rounds of E P_i executes (gi1) and then (gi2) at least once. Immediately after P_i executes (gi1) and (gi2) it holds that $color_i = color_i[r] = color_r$ (the color of P_i ($color_i$) is equal to the color P_i read from its father ($color_i[r]$) and equal to the color of its father ($color_r$)). Any further execution of (gi1), (gi2) or (gi3) does not change the color of $color_i$. Since P_i is an arbitrary neighbor this equation hold for every neighbor of P_r while P_r does not change its color. The same arguments holds during the second $k_1\Delta$ rounds of E for processors that are in distance two from P_r. Continuing this way, following the ith $k_1\Delta$ rounds of E all the processors whose depth is less than or equal to i read the color of P_r from their father and are colored with the color of P_r. Thus, following $k_1\Delta\mathcal{D}$ rounds the entire tree is uniformly colored.

Let E' be the suffix of E that follows the first $k_1\Delta\mathcal{D}$ rounds of E. During the first $k_1\Delta$ rounds of E' every leaf, P_i, executes (gi1) and then (gi3), hence sets $ack_i = true$. While the tree is uniformly colored any further execution of (gi1), (gi2) or (gi3) does not change the value of ack_i. Hence, after the first $k_1\Delta$ rounds of E' $ack_i = true$ for every leaf P_i. The same argument holds during the second $k_1\Delta$ rounds of E' for every processor P_i whose sons are leaves. Continuing this way, following the ith $k_1\Delta$ rounds of E' it holds that $ack_j = true$ for every processor P_j, such that the subtree rooted at P_j is of depth less than or equal to i. Thus, following $2k_1\Delta\mathcal{D}$ rounds the entire tree is uniformly colored and for every processor P_i in the tree, $ack_i = true$. Hence, in the next $k_1\Delta$ rounds P_r executes (gr1) and (gr2) and assigns a new color to $color_r$. □

Now we show that requirements 2 and 3 of a global synchronization protocol hold too. Let E be an execution starting from arbitrary configuration, c_0. Let $P_r, P_1, \ldots, P_i, \ldots, P_l$ be a path from a root of a tree P_r to a leaf in that tree, P_l. Let E_i be a suffix of E that starts immediately after $4k_1\Delta i$ rounds of E. We prove by induction on i that during E_i it holds that:

ASSUMPTION 1. Immediately before P_r changes its color it holds that $color_r = color_1 = \ldots = color_i$.

ASSUMPTION 2. In any configuration of E_i that appears after P_i changes its color for the first time (and in the same time, assigns $ack_i := false$) it holds that if $ack_i = false$ then $color_r = color_1 = \ldots = color_i$.

BASE CASE, I = 1. For every father-son pair (P_r, P_1), the executed protocol is the local-synchronization protocol: (gr1) and (gr2) includes (lf1) and (lf2), (gi1) and (gi2) includes (ls1) and (ls2). Thus, by Lemma 1 following $4k_1\Delta$ rounds it holds that $color_r = color_1$ immediately before any color change in $color_r$. Thus, Assumption 1 above is proven. Now we prove Assumption 2. By Lemma 1 during the first $4k_1\Delta$ rounds of E a configuration c in which $color_r = color_1[r] = color_1$ is reached. Following c, $color_1$ is not changed till $color_r$ is changed. Let c' be a configuration that immediately follows the first change in $color_r$ after c i.e., c' follows an atomic step during which P_r changes $color_r$ and sets $ack_r := false$. By the fact that the local-synchronization protocol is stabilized it holds that following c' the father-son pair (P_r, P_1) may repeatedly change their *color* and *ack* values only in the following order:

1) P_1 changes $color_1$ to the color of $color_r$ and sets $ack_1 := false$
2) P_1 sets $ack_1 := true$
3) P_r changes $color_r$.

Thus, following the first time P_1 executes Step 2, above, the equation $ack_1 = false$ holds only during subexecutions that start immediately after Step 1 and end immediately before the next Step 2. During these subexecutions $color_r = color_1$.

INDUCTION STEP. We first prove that following $4k_1\Delta i$ rounds the father-son pair (P_i, P_{i+1}) executes the local synchronization protocol:

LEMMA 3. *Following $4k_1\Delta i$ rounds the father-son pair (P_i, P_{i+1}) executes the local synchronization protocol.*

PROOF. To prove the lemma, it is sufficient to show that when the induction hypotheses 1 and 2 hold for i then in E_i it hold that:

1) P_{i+1} repeatedly reads $color_i$ and whenever $color_{i+1}[i] \neq color_{i+1}$, P_{i+1} assigns $color_{i+1} := color_i$.
2) between any two changes of $color_i$, P_i reads $color_{i+1}$ and finds $color_i = color_i[i + 1]$.
3) the sequence of colors that is assigned to $color_i$ is equal to the sequence of colors that is assigned to $color_r$.

(gi1) and (gi2) of the global synchronization protocol imply 1. To prove 2, we show that during E_i P_i assigns $ack_i := true$ between any two successive changes in the value of $color_i$. Whenever P_i changes its color it sets $ack_i := false$. By Assumption 2 while $ack_i = false$ it holds that $color_r = color_1 = \ldots = color_i$. Hence P_i does not change its color thereafter before it assigns $ack_i := true$. This latter operation is done only after P_i finds that $color_i[i+1] = color_i$, as required by the local synchronization protocol. This proves 2, above.

By Assumption 1, between any two changes in $color_r$, there is a configuration in which $color_i = color_r$. By Assumption 2, and the fact that whenever P_i assigns a color to $color_i$, it sets $ack_i = false$, it holds that $color_i$ is always changed to the color of $color_r$. Hence, 3. □

By Lemma 3 and by Lemma 1 it must hold that in the first $4k_1\Delta$ rounds of E_i a configuration c is reached in which $color_i = color_{i+1}[i] = color_{i+1}$. Hence, the local synchronization protocol executed by P_i and P_{i+1} is stabilized during those $4k_1\Delta$ rounds. Thus, following c the father-son pair (P_i, P_{i+1}) may repeatedly change their *color* and *ack* values only in the following cyclic order:

1) P_i changes $color_i$ and sets $ack_i := false$
2) P_{i+1} changes $color_{i+1}$ to the color of $color_i$ and sets $ack_{i+1} := false$
3) P_{i+1} sets $ack_{i+1} := true$
4) P_i sets $ack_i := true$.

We now prove that following c the induction assumptions hold for $i + 1$:

ASSUMPTION 1. We claim that following c, $color_r$ is changed only when $color_i = color_{i+1}$. Let c' be the configuration that follows c and immediately precedes the atomic step in which P_r changes the color of $color_r$. Assume toward a contradiction that in c' $color_i \neq color_{i+1}$. Then P_i changed its color at least once after c. Consider the last such change before c'. We now show that ack_i must be *false* in c'. ack_i is true only during subexecutions that start immediately after 4 and end immediately before the next 1. During these subexecutions $color_i = color_{i+1}$, and hence $ack_i = false$ in c'. Thus, by Assumption 2 $color_r$ is not changed. Hence there is a contradiction.

ASSUMPTION 2. First, we show that whenever $ack_{i+1} = false$ then $ack_i = false$, then we apply the induction assumption on ack_i: following c P_i and P_{i+1} change the value of $color_i$, $color_{i+1}$ and ack_i, ack_{i+1}, according to Steps 1 to 4, above. P_{i+1} changes its color, according to 2 above, only after P_i executes step 1 in which P_i changes its color and sets ack_i=*false*. After executing 1, P_i does not change the value of ack_i to *true* unless $color_i = color_{i+1}$ and $ack_{i+1} = true$. Thus, during any subexecution that starts immediately after 1 and ends immediately after the next 2, $ack_i = false$. $ack_{i+1} = false$ only during subexecutions that start immediately after 2 and ends immediately before the next 3. During these subexecutions $ack_i = false$ and $color_i = color_{i+1}$. Now we complete the proof by the use of the induction assumption applied to P_i: if $ack_i = false$ then $color_r = color_1 = \ldots = color_i$.

3.3.2 Single Tree With a Leader

First we prove that in every fair execution eventually *FSG* becomes a forest and once this happens *FSG* remains a forest for the rest of the execution. Then we prove that *FSG* is converted to a single tree and in the end we prove that this tree is fixed forever.

LEMMA 4. *For any processor P_i and any execution E the value of (tid_i , dis_i) does not decrease during E.*

PROOF. The value of (tid_i, dis_i) is changed only in lines 12, 13, 20 of Fig. 1 and Fig. 2. It is easy to see that when either line 12 or 20 is executed, the value of (tid_i, dis_i)

does not decrease. By the nature of read write atomicity line 13 is included in the same atomic step with the test of line 11. This test ensures that (tid_i, dis_i) never decreases.

DEFINITION 2.

1) *A processor completes a* loop iteration *during an execution if it executes line 2 and then executes lines 3 to 15 of the protocol.*
2) *A processor completes a* read iteration *during some execution of our protocol if it executes line 2 and then lines 3 to 10 of the protocol.*

As described before, an edge (P_f,P_s) belongs to $FSG(c)$ if in configuration c, f_s points to P_f and $dis_s > 0$. In this case we say that (P_f, P_s) is a *father-son pair* in c.

LEMMA 5. *Let $E' = (c_0, \ldots, c)$ be (a prefix of) an execution of the protocol in which each processor completes a loop iteration. The following hold for any father-son pair (P_f, P_s) in c:*

1) *Following the first read iteration in E', P_s executes the* ***write*** *operation in line 13 at least once.*
2) *Let c_i be the configuration reached by the system immediately after P_s executes the* ***write*** *operation of line 13 for the last time during E'. In c_i the following relation holds:*

$$(tid_f, dis_f) \succeq (tid_s[f], dis_s[f]) \succ (tid_s, dis_s)$$

PROOF. Since P_s completes a loop iteration, it executes the if statement (Line 11) at least once before c is reached. Within this if-statement P_s either declares itself a root (Line 12) or declares some processor as its father (Line 13). Consider the last time this if-statement is executed before c: In $FSG(c)$, P_s is the son of P_f, therefore in the last time line 11 was executed P_s chose to execute line 13. This proves 1.

The value of $tid_s[f]$, $dis_s[f]$ in c_i, is obtained by reading (tid_f, dis_f) (in line 2) at an earlier step. By Lemma 4 (tid_f, dis_f) does not decrease, hence the leftmost relation holds. Configuration c_i immediately follows the last execution of Line 13 by P_s, in which P_s declares P_f to be its father. This is done by P_s writing $(tid_s[f], dis_s[f] + 1)$ in (tid_s, dis_s), which implies the rightmost relation, hence 2 is proven. □

LEMMA 6. *Let $E' = (c_0, \ldots, c)$ be an execution in which each processor completes a loop iteration. For every father-son pair (P_f, P_s) in c, relation (*) holds.*

PROOF. Consider an arbitrary father-son pair (P_f, P_s). Let c_i be the configuration reached after the last time P_s executes the if statement of line 11 before c. By Lemma 5, the relation (*) holds in c_i. To complete the proof we show that the relation (*) is preserved during $E'' = (c_i, c_{i+1}, \ldots, c)$. Throughout E'', P_s is the son of P_f hence it is not a root. The only instruction in which a nonroot processor changes its (tid, dis) is the if statement of line 11. Since c_i follows the last time P_s executes this if statement, (tid_s, dis_s) is constant and P_f is the father of P_s throughout E''. To complete the proof we observe that $(tid_s[f], dis_s[f])$ is updated by P_s copying (tid_f, dis_f). By Lemma 4 this pair only grows, hence the proof. □

LEMMA 7. *If c is a configuration in which the relation (*) holds for every father-son pair then $FSG(c)$ is a forest of trees.*

PROOF. To prove the lemma we have to show that the outdegree of each node is at most 1 and the underlying graph has no cycles. Each nonroot processor has at most one father (defined by the value of f). By Lemma 6 the relation (*) holds for every father-son pair in $FSG(c)$, hence $FSG(c)$ contains no cycles. □

DEFINITION 3. Configuration, c, is a *forest* configuration if the relation (*) holds for every father-son pair (P_f, P_s) in c, and for every pair of neighbors P_i and P_j $(tid_i, dis_i) \succeq (tid_j[i], dis_j[i])$.

COROLLARY 8. *In every execution E:*

1) *A forest configuration is reached following $O(\Delta)$ rounds.*
2) *If c_i is a forest configuration and $c_i \rightarrow c_{i+1}$ then c_{i+1} is a forest configuration.*

PROOF. By Lemma 6, a forest configuration is reached in every execution after each processor has completed at least a single loop iteration. By the definition of loop iteration and by the code there exists a constant k_1 such that processor P_i with Δ neighbors takes $k_1\Delta$ steps to complete a loop iteration. Hence in any execution, the system reaches a forest configuration within $O(\Delta)$ rounds which proves 1. The proof of 2 is by arguments similar to the arguments of Lemma 6. □

DEFINITION 4. $R(c)$ is the set of the root processors in configuration c.

LEMMA 9. *For any two forest configurations, c_i, c_{i+1}, such that c_{i+1} is reached by an arbitrary atomic step a from c_i, $R(c_i) \supseteq R(c_{i+1})$.*

PROOF. Assume in contradiction that during a there is a nonroot processor P_s that assigns 0 in dis_s. Let P_f be the father of P_s just before the execution of a. Since a must contain a write operation of line 12 a starts by the execution of line 3 to 11 by the processor P_s, when P_s executes line 11 P_s finds that the condition $(tid_s, dis_s) \succeq (max\_tid, min\_dis)$ is true, and a ends with P_s executing the atomic write of line 12. During the execution of a following the execution of lines 3 to 10 it holds that $(max\_tid, min\_dis) \succeq (tid_s[f], dis_s[f])$. By the definition of forest configurations the relation (*) holds in c_i. Thus, $(max\_tid, min\_dis) \succ (tid_s, dis_s)$ and hence the write operation of line 12 is not executed, contradiction. □

Lemma 7, Corollary 8, and Lemma 9 show that if E is an execution that starts with a forest configuration, then $FSG(c)$ is a forest in every configuration c of E and no processor becomes a root during E. In the following lemmas we show that when an execution starts with a forest configuration the number of roots decreases to one in $O(\Delta\mathcal{D} \log n)$ expected number of rounds. Where $\mathcal{D}$ is the diameter of the communication graph. The following definitions are used in the sequel:

DEFINITION 5.

1) *In any configuration c, $tid_i(c)$ is the tid of P_i in c. $f_i(c)$, $dis_i(c)$ and the other value of c are defined similarly.*
2) *In any configuration c, $MTID(c) = max\{tid_i \mid 1 \le i \le n\}$ in c.*

3) *A* uniform *tid configuration c, is a configuration in which for every processor P_i $tid_i = MTID(c)$ and for every neighbor of P_i, P_j, $tid_i[j] = MTID(c)$.*

LEMMA 10. *If MTID is not changed during an execution that starts with a forest configuration and contains at least $k_1\Delta(\mathcal{D} + 1)$ rounds then a uniform tid configuration is reached.*

PROOF. Let $E = (c_0, c_1, \ldots)$ be an execution that contains at least $k_1\Delta(\mathcal{D} + 1)$ rounds throughout which *MTID* remains constant and in which c_0 is a forest configuration. Since c_0 is a forest configuration, there exists a non empty set, denoted $R_m(c_0)$, of root processors in c_0 whose *tid* is equal to $MTID(c_0)$. By Lemma 4, the *tid* of a processor never decreases, on the other hand *MTID* remains constant throughout *E*. Therefore once a processor's *tid* becomes equal to $MTID(c_0)$ it remains constant throughout *E*.

To prove the lemma we first show that during the first $k_1\Delta\mathcal{D}$ rounds of *E*, the *tid* of every processor in the system becomes equal to $MTID(c_0)$. This is proved by induction on *d*, the distance of a processor from (the closest processor in) $R_m(c_0)$; Where distance is measured on the communication graph. Denote the distance of P_i from $R_m(c_0)$ by d_i. We prove that after $k_1\Delta d$ rounds tid_i of any processor P_i in distance $d_i \le d$ is equal to $MTID(c_0)$.

INDUCTION BASE (d = 0): Clearly, the *tid* of processors in $R_m(c_0)$ is equal to $MTID(c_0)$ throughout *E*.

INDUCTION STEP: We assume that after $k_1\Delta d$ rounds, $tid_i = MTID(c_0)$ for any processor P_i in distance $d_i = d$ and we prove that following the next $k_1\Delta$ rounds $tid_j = MTID(c_0)$ for any processor P_j in distance $d_j = d + 1$. During the rounds $dk_1\Delta$ through the $(d + 1)$th $k_1\Delta$, P_j completes at least one loop iteration. In the first such loop iteration P_j reads the *tid* of a processor in distance *d* from $R_m(c_0)$ in case $tid_j < MTID(c_0)$ P_j assigns $MTID(c_0)$ to tid_j. Hence, following $k_1\Delta(d + 1)$ rounds the induction hypothesis holds for every processor P_j in distance $d_j(c_0) = d + 1$.

We proved that following the first $k_1\Delta\mathcal{D}$ rounds of *E* $tid = MTID(c_0)$ for every processor. During the following $k_1\Delta$ rounds each processor reads the *tid* of every neighbor hence for every *j* $tid_i[j] := MTID(c_0)$.

LEMMA 11. *Let $E = (c_0, \ldots, c)$ be an arbitrary execution of at least $2k_1\Delta\mathcal{D}$ rounds that is started with a uniform tid forest configuration and during which MTID is not changed. The following assertions hold for every configuration c' in the suffix of E, E', that starts following the first $2k_1\Delta D$ rounds of E:*

1) $R(c_0) = R(c')$.
2) *for any processor P_j, $dis_j(c') = d_j(c')$, where $d_j(c')$ is the distance of P_j to a closest root in c'.*
3) $f_j(c'_0) = f_j(c')$ *and the value of $f_j(c')$ is the index of a neighbor which is on a shortest path to one of the closest roots in $R(c')$.*

PROOF. By Lemma 9 and the fact that c_0 is a forest configuration, the set of roots may only be decreased following c_0. Thus, to prove Assertion 1 we only have to show that during *E* a root processor does not become nonroot. By the fact that *MTID* is not changed during *E* it holds that every configuration *c* in *E* is a uniform *tid* configuration. Thus, whenever a root, P_i, calculates *max_tid* it assigns *max_tid*: = *MTID*. A root processor, P_i, becomes nonroot only if $max\_tid > tid_i$. Since $tid_i = MTID$ throughout *E* Assertion 1 holds.

Assertions 2 and 3 are implied by the following stronger claim which is proved by induction on *d* the distance of a processor from $R(c_0)$: In any configuration c' that appears following the first $2k_1\Delta d$ rounds of *E*,

1) For any processor P_j with distance
$$d_j(c_0) \le d \quad dis_j(c') = d_j(c_0).$$
2) For any processor P_j with distance $d_j(c_0) \le d$ the value of $f_j(c')$ is not changed throughout E'. The value of $f_j(c')$ is the index of the link that connects P_j with one of P_js neighbors which is on a shortest path to one of the closest roots.
3) For any processor P_l that is in distance $d_l(c_0) \ge d$ it holds that $dis_l \ge d$.

INDUCTION BASE. $d_j(c_0) = 0$. The proof of Assertion 1 implies 1 and 2, above. By the fact that for every processor P_i $dis_i \ge 0$ Assertion 3 holds too.

INDUCTION STEP. By the induction assumption following no more than $2k_1\Delta d$ rounds of *E*, Assertions 1, 2, and 3 hold for any processor P_j in distance $d_j(c_0) \le d$. Let c_d be the configuration that immediately follows the first $2k_1\Delta d$ rounds. We now prove that 1, 2, and 3 hold for $d + 1$, $2k_1\Delta$ rounds following c_d. Every processor completes a loop iteration every $k_1\Delta$ rounds. Thus, by assertion 3, $k_1\Delta$ rounds following c_d any processor P_k that is in distance $d_k(c_0) > d + 1$ assigns dis_k by a value that is $\ge d + 1$. Following additional $k_1\Delta$ rounds any processor P_m that is in distance $d_m(c_0) = d + 1$ reads the values of the *dis* fields of all the neighbors that are in distance *d* and then (executes line 13 in which it) assigns $dis_m := d + 1$ and assigns f_m by the right value. From this point on, P_m finds that $(tid_m[f_m], dis_m[f_m]) = (max\_tid, min\_dis)$ and hence does not change the value of dis_m or f_m (see lines 3-5 and line 13 of the code) □.

DEFINITION 6. *A utff-*execution (uniform tid fixed forest execution) *is an execution in which all the configurations are uniform tid configuration and during which FSG is a constant forest. Such that, each tree in FSG is of depth less than or equal to $\mathcal{D}$.*

Let k_2 be a constant that is bigger than $k_1(3 + 1/\mathcal{D})$. By the above two lemmas any execution that starts with a forest configuration and in which MTID is constant reaches, following $3k_1\Delta\mathcal{D} + k_1\Delta = k_1\Delta\mathcal{D}(3 + 1/\mathcal{D}) < k_2\Delta$ rounds, a uniform tid fixed forest execution, abbreviated utff-execution.

To prove that a root discovers the existence of other roots with equal tid we use the global-synchronization protocol

(described in the previous section) on every tree of the fixed forest. During a utff-execution every root executes (gr1) and (gr2) of the global synchronization protocol by executing line 2 (for (gr1)), and lines 17 and 21 (for (gr2)). A nonroot processor executes (gi1), (gi2) and (gi3) of the global synchronization protocol by executing line 2 (for (gi1)), lines 24, 28 (for (gi2)), and lines 30, 33 (for (gi3)). Thus, the global synchronization protocol is executed on every tree in the fixed forest.

COROLLARY 12. *There exist constant k_3 such that if E is a utff-execution and E′ is the suffix of E that follows the first $k_3\Delta\mathcal{D}$ rounds of E, then during E′ a root processor chooses a new color only when its tree is uniformly colored.*

PROOF. During any *utff-execution FSG* defines a fixed forest of trees (with depth less than or equal to $\mathcal{D}$). Thus, during any *utff-execution* we may apply the arguments of Lemma 2 to our protocol. The arguments of Lemma 2 consider the number of rounds needed for a processor to complete at least one loop iteration of the global synchronization protocol. By the fact that any configuration of an *utff-execution* is a uniform *tid* configuration in which the value of the *dis* field of every processor is the minimal distance to its root the condition (in line 14) for executing lines 17, 21, 24, 28, 30, and 33 holds. Namely, the condition for executing the global synchronization protocol. In such an execution every processor completes a loop iteration in the global synchronization protocol every $k_1\Delta$ rounds. Hence following $4k_1\Delta\mathcal{D} = k_3\Delta\mathcal{D}$ rounds a root chooses a new color only when all its tree is uniformly colored. □

LEMMA 13. *There is a constant k_4 such that every execution that starts in a forest configuration c reaches in expected number of $k_4\Delta\mathcal{D}$ rounds a configuration c_i which is either a leader configuration or $MTID(c_i) > MTID(c)$.*[4]

PROOF. The proof is by presentation of a (7, $k\,\Delta\mathcal{D}$)-*strategy* for *luck* by the game defined by the protocol, for some constant *k*, and by the use of Theorem 5 of [17]. The strategy of *luck* is as follows:

Luck waits $(k_2 + k_3)\Delta\mathcal{D}$ rounds (the constant k_2 is defined right after Lemma 11). If a safe configuration is reached or a configuration c_i for which $MTID(c_i) > MTID(c)$ is reached then *luck* wins the *sl-game*. Otherwise, by Lemma 10, Lemma 11, and Corollary 12 a *utff-execution* is started, in which the root chooses a new color only when the tree is uniformly colored.

Furthermore, by Lemma 2, in the *utff-execution* that starts following these $(k_2 + k_3)\Delta\mathcal{D}$ rounds every root indeed chooses a new color every at most $2k_1\Delta\mathcal{D} + k_1\Delta$ rounds.

Let P_1 and P_2 be two root processors during a *utff-execution* such that there are neighbors P_i and P_j and P_i (P_j) belongs to the fixed tree of P_1 (P_2). Once *luck* waited $(k_2 + k_3)\Delta\mathcal{D}$ rounds the strategy of *luck* continues as follow: If P_1 (P_2) chooses a new color *luck* intervenes and set its new color to be in {0, 1, 2, 3} ({4, 5, 6, 7}, respectively).[5]

We now show that this strategy leads to at least one extension of the *tid* of either P_1 or P_2 within $O(\Delta\mathcal{D})$ rounds.

We start with any combination of colors for P_i and P_j. W.l.o.g let P_i be the first processor that changes its color for the fourth time i.e., changed $color_i$ from its original color to $color^1$, then to $color^2$, and to $color^3$, and, at last, to $color^4$.[6] Let $a_{l \to l+1}$ be the atomic step in which P_i changes the color of $color_i$ from $color^l$ to $color^{l+1}$. Between any two successive color changes $a_{l \to l+1}$ and $a_{l+1 \to l+2}$ P_i reads $color_j$ and set ot_i := *true* when $color_i[j] \notin \{color^1, color^2\}$. If P_i did not assign ot_i := *true* following any of those changes then it read:

- That the value of $color_j \in \{color^1, color^2\}$, between the write operations of $color^1$ in $color_i$ and $a_{1 \to 2}$
- That the value of $color_j \in \{color^2, color^3\}$, between $a_{1 \to 2}$ and $a_{2 \to 3}$
- That the value of $color_j \in \{color^3, color^4\}$, between $a_{2 \to 3}$ and $a_{3 \to 4}$

Now we show that the above yield that P_j changes its color. Assume towards a contradiction that $color_j$ is not changed: since $color^1$, $color^2$, and $color^3$ are three different colors, and since $color^2$, $color^3$, and $color^4$ are also three different colors, then by the first two reads $color_j$ must be $color^2$ however by the last two reads $color_j$ must be $color^3$, a contradiction. Thus, P_j changed its color.

By our strategy P_j changes its color to be in 4, 5, 6, 7. Thus, before the forth color change of P_i ot_i is assigned by *true*. Thus, in our *sl-game* a *tid* of a root processor is extended in $O(\Delta\mathcal{D})$ rounds and with at most seven intervention i.e., at most four intervention for P_1 and 3 for P_2. Theorem 5 of [17] implies that if *luck* has an (*f*, *r*)-*strategy* then the protocol reaches a leader configuration within at most $r2^f$ expected number of rounds. Thus, the expected number of rounds till *MTID* grows is constant number of $\Delta\mathcal{D}$ (i.e., $k_4\Delta\mathcal{D}$) rounds. □

LEMMA 14. *There is a constant k_5 such that every execution that starts in a forest configuration c reaches in $k_5\Delta\mathcal{D}$ rounds a configuration c′ in which for every root processor P_i in c′, $tid_i(c') \geq MTID(c)$.*

PROOF. The proof is similar to the proof of Lemma 10. □

COROLLARY 15. *There is a constant k_6 such that every execution that starts in a forest configuration c reaches in expected number of $k_6\Delta\mathcal{D}$ rounds a leader configuration c′ or the tid of every root processor in c′ is longer than its tid in c by at least 2 log n bits.*

PROOF. Let k_5 be the constant of Lemma 14. First we show that during the $k_5\Delta\mathcal{D}$ rounds that immediately follows the growth of *MTID*, every root P_i such that $tid_i \neq MTID$ extends its *tid* by at least one bit. Let *c* be a configuration that immediately follows a growth of *MTID*. Let P_i be the root processor that extended its *tid* to $MTID(c)$ immediately before *c*.

For any other root processor $P_j \neq P_i$ it holds that $tid_j(c) < tid_i(c)$. By Lemma 14, $k_5\Delta\mathcal{D}$ rounds after *c* the

4. It should be noted that *MTID* could grow and at the same time the length of *MTID* could become shorter i.e., assume that *MTID* = 100 and then a root processor P_i with $tid_i = 1$ extends its *tid* to hold 11.

5. Note that every such intervention is done by fixing only the most significant bit of the color.

6. Note that the other case in which P_j is the first processor to change its color for the first time is symmetric-exchange P_i with P_j and 4, 5, 6, 7, with 0, 1, 2, 3.

value of the *tid* of any root is at least the value of $MTID(c)$. Since our protocol guarantees that whenever a root processor finds a neighbor with greater *tid* it becomes nonroot processor the only way P_j could survive as a root processor is by extending its *tid*.

By Lemma 13 if the execution starts with a forest configuration then either a leader configuration is reached or *MTID* grows in expected $k_4\Delta\mathcal{D}$ rounds. Therefore, every expected $(k_4 + k_5)\Delta\mathcal{D}$ rounds either the system reaches a leader configuration or the *tid* of every root processor is extended by one bit. If a leader configuration is not reached following the first growth then during additional expected $(k_4 + k_5)\Delta\mathcal{D}$ rounds either the system reaches a leader configuration or the *tid* of every root processor is extended by 1 more bit, and so on and so forth. The proof is completed by the fact that expectation of a sum is a sum of expectations. □

The proof is completed by the following corollary:

COROLLARY 16. *The system elects a leader in expected* $O(\Delta\mathcal{D} \log n)$ *rounds.*

PROOF. By Corollary 15 there exists a constant k_6 such that whenever the system starts in a forest configuration c the system reaches in expected $k_6\Delta\mathcal{D} \log n$ rounds a configuration c' which is either a leader configuration or the *tid* of every root in c' is longer than its *tid* in c by at least 2 log n bits. Let P_i and P_j be any two arbitrary root processors in c. Let b_k (b'_k) be the kth bit of $tid_i(c')$ $(tid_j(c'))$ (where b_1, b'_1, is the most significant bit). W.l.o.g assume that $tid_i(c')$ is not longer than $tid_j(c')$. By Corollary 15 in the execution that starts with c and ends with c' at least 2 log n last bits b_l, b_{l+1}, ..., $b_{l-1+2 \log n}$ of $tid_i(c')$ were randomly chosen. By Lemma 14 if P_i and P_j survives as root processors during the next $k_5\Delta\mathcal{D}$ rounds then it must hold that $b_i = b'_i$ for every $l \geq i \geq l - 1 + 2 \log n$.

The probability that P_i choose randomly a bit, b_k, that has the same value as b'_k is 1/2. Thus, the probability that P_i choose randomly 2 log n bits, b_l, b_{l+1}, ..., $b_{l-1+2 \log n}$, that are equal to the 2 log n bits, b'_l, b'_{l+1},..., $b'_{l-1+2\log n}$ of P_j is $(1/2)^{2\log n} = 1/n^2$. The probability that there is at least one pair P_i and P_j such that those bits of their *tid* equal is less than $n^2/2 \times 1/n^2 = 1/2$ i.e., the number of all possible pairs multiplied by the probability that the value of the bits b_l, b_{l+1}, ..., $b_{l-1+2 \log n}$, and the bits b'_l, b'_{l+1},..., $b'_{l-1+2\log n}$ is equal for any given pair of root processors.

Hence, the probability to reach a safe configuration following expected $k_6\Delta\mathcal{D} \log n + k_5\Delta\mathcal{D}$ rounds is greater than 1/2. Similarly, the probability to reach a safe configuration following $l(k_6\Delta\mathcal{D} \log n) + k_5\Delta\mathcal{D}$ rounds is greater than $(1 - (1/2)^l)$. Thus, the expected number of rounds until a leader configuration is reached is less than

$$k_5\Delta\mathcal{D} + \sum_{l=1}^{\infty} l\left(k_6\Delta\mathcal{D} \log n\right)(1/2)^{l-1}(1/2)$$
$$= k_5\Delta\mathcal{D} + 2\left(k_6\Delta\mathcal{D} \log n\right)$$

rounds which is $O(\Delta D \log n)$. □

Until this stage we proved that after expected $O(\Delta\mathcal{D} \log n)$ rounds the system reaches a forest configuration with a single leader and this leader is fixed forever i.e., the system reaches a leader configuration. To complete the proof and show that the system reaches a safe configuration we will show that the (single) root does not repeatedly extend its identity (due to wrong information on the existence of other trees). In the sequel we prove that such extension could happen at most once. We show that the root executes propagation of *tid* in the system and gets feedback after the propagation is terminated.

3.3.3 Propagation of Information and Feedback

In this stage of the proof we show that our protocol executes propagation of information (propagation of the new *id* of the root) and feedback (similarly to the protocol described in [29]) in $O(\Delta\mathcal{D})$ rounds (where $\mathcal{D}$ is the diameter of the communication graph). Note that the application of the protocol described in [29], to a shared memory system, requires $O(\Delta n)$ rounds. Unlike the protocol in [29] our protocol construct a breadth first tree (*BFS* tree) during the propagation of the new identity of the root and the feedback stage. The propagation and the feedback uses the tree to ensure that the time complexity is $O(\Delta\mathcal{D})$ rounds.

Let E be an execution that starts with a leader configuration in which P_r is the single root processor. By Lemma 9 no processor becomes a root and by Lemma 7 and Corollary 8 every configuration in E define a forest, thus the single root processor is fixed throughout E. Informally, if P_r extends tid_r during E then following the first extension of the *tid*, the root floods the system with the new *tid*. Whenever a processor, P_i, assigns the new *tid* in tid_i for the first time in E, P_i simultaneously assigns the color of the root in $color_i$ and assigns *false* in ack_i and in ot_i (lines 5 to 13 of the code). We show that immediately before the root decides that the entire system is flooded with the new identifier (i.e., the condition in line 17 holds) the system reaches a leader configuration, c, that is called *BFS configuration* in which the following assertions hold:

- (*bfs*-1) $FSG(c)$ is a *BFS* tree of the entire communication graph, with a single root processor P_r, and
- (*bfs*-2) for every nonroot processor P_l with father P_f it holds:
  $tid_l(c) = tid[f](c) = tid_f(c)$,
  $dis_l(c) = dis_l[f](c) + 1 = dis_f(c) + 1$,
  $color_l(c) = color_l[f](c) = color_f(c)$, $ack_l(c) = true$,
  and
  $ot(c) = false$.
- (*bfs*-3) for every neighbor P_l of P_r it holds that $ot_r[l] = false$.

DEFINITION 7. *Define a flooding subexecution,*

$$E' = (c_0, \ldots, c_b),$$

to be subexecution that starts with a leader configuration, c_0, that immediately follows an atomic step during which P_r extends its tid (by executing line 20 of its program) and ends immediately before the next atomic step during which P_r discovers that the flooding is over (the condition in line 17 holds).

Notice that by the definition of E', during E', P_r does not extend its *tid* and does not change its color. Thus, for every configuration c in E', $MTID(c) = MTID(c_0)$. Define $MT(c)$ to be the set of processors, P_i, in the configuration c for which $tid_i(c) = MTID(c_0)$. For instance $MT(c_0)$ is the set of processors that includes P_r solely. In the sequel we show that during E', the father-son relation in $MT(c)$ induces a subtree with root P_r in $FSG(c)$ that includes all the processor with $tid = MTID(c)$.

LEMMA 17. *If during flooding subexecution, $E' = (c_0, \ldots, c_b)$, a nonroot processor P_j changes its tid to $MTID(c_0)$, then after this change and till the end of E' the following assertions hold:*

1) *The tid of any processor in the directed path from P_j to P_r in the FSG is $MTID(c_0)$.*
2) *The value of dis_j does not increase.*
3) *Every time P_j changes its father in FSG, P_j simultaneously decreases the value of dis_j.*
4) *$color_j = color_j[f] = color_r$, where P_f is the father of P_j.*
5) *$ot_j = false$.*
6) *In any configuration in which $ack_j = true$ it hold for every neighbor P_k of P_j that $tid_k = tid_j[k] = MTID(c_0)$ and $dis_k \leq dis_j[k] \leq dis_j + 1$.*
7) *Every time P_j changes the value of ack_j to false, P_j simultaneously decreases the value of dis_j.*

PROOF.

1) By Lemma 4 during flooding subexecution E', after a processor P_j that is not a root changes the value of tid_j to $MTID(c_0)$ it holds that $tid_j = MTID(c_0)$. Hence by the (*) relation of Lemma 6, the *tid* of every processor in the directed tree from P_j to the root of the tree is equal to $MTID(c_0)$.
2) By Lemma 4 when tid_j is not changed dis_j may only be decreased, hence 2.
3) The value in tid_j is not changed after the assignment of $MTID(c_0)$ in tid_j. P_j changes its father (following the execution of lines 8 to 13 of the code) only when the value of (tid_j, dis_j) grows, hence 3.
4) We prove that during E it holds for every nonroot processor in MT, P_j, with father P_f that $color_j = color_j[f] = color_r$. The proof is by induction on i, the index of the configuration in the execution. The induction base is by the fact that in c_0 the root processor is the only processor in $MT(c_0)$. We assume that the induction assumption holds for c_i and we prove that it holds for c_{i+1} where $c_i \overset{a_i+1}{\rightarrow} c_i + 1$. Clearly, the induction assumption holds in c_{i+1} when during a_{i+1} no processor in MT writes in its *color* field and no processor joins MT. If during a_{i+1} a processor in $MT(c_i)$ writes in its *color* field then it must copy its father's color that is in $MT(c_i)$ and hence by the induction assumption it holds that $color_j(c_{i+1}) = color_j[f](c_{i+1}) = color_r(c_{i+1})$. Whenever a processor joins MT the processor chooses one of the processors in $MT(c_i)$ to be its father and copies its color. Hence, also in that case the induction assumption hold in c_{i+1}.
5) Now we show that during E $ot_j = false$ for every nonroot processor in MT. When a processor joins MT the processors assigns $ot := false$. Assume for a moment that there is a processor P_j that assigns $ot_j := true$ after joining MT. Let P_j be the first processor in MT that assigns $ot_j := true$. P_j does not have sons in MT with $ot = true$, hence P_j must discover a neighbor with different color from its own color (and its father color). The condition to check the neighbors' color (line 24) is that the color of the father of P_j is different from the color of P_j, however by Assertion 4 above that condition does not hold.
6) After P_j changes the value of tid_j to hold $MTID(c_0)$ P_j assigns $ack_j := true$ only when for every neighbor P_k of P_j it holds that $tid_j[k] = MTID(c_0)$ and $dis_j[k] \leq dis_j + 1$ (line 14). Thus, all the neighbors of P_k joined MT and P_j read the value of (tid_k, dis_k) at least once after P_k joined MT. Hence every further read of P_j keeps the relation $tid_j[k] = MTID(c_0)$. Following the assignment $ack_j := true$ P_j changes the value of dis_j only if it simultaneously assigns $ack_j := false$. Thus while $ack_j = true$ the relation $dis_k \leq dis_j[k] \leq dis_j + 1$ that was true when P_j assigns $ack_j := true$ is preserved.
7) A nonroot processor P_j may assign $ack_j := false$ only if it simultaneously increases the value of (tid_j, dis_j) (line 13 of the code) or when $color_j \neq color_j[f]$ (line 28). By Assertions 1 and 4 above $color_j = color_j[f]$ and $tid_j = MTID(c_0)$ during E' and hence dis_j must decrease simultaneously with the assignment of $ack_j := false$. □

LEMMA 18. *Let $E' = (c_0, \ldots, c_b)$ be a flooding sub-execution, and c' a configuration in E', that immediately follows atomic step during which a processor P_j with $tid_j = MTID(c_0)$ changes the value of ack_j to true. Then for every son, P_s, of P_j in $FSG(c')$, $ack_s(c') = true$.*

PROOF. Let P_s be an arbitrary son of P_j in c'. Whenever P_j joins MT, P_j assigns $ack_j := false$. After each time P_j assigns $ack_j := false$ P_j does not change ack_j to *true* before it reads and finds that $tid_s = MTID(c_0)$ and $dis_s \leq dis_j + 1$.

We prove that $ack_s = true$ in c'. Let c_r be a configuration that immediately follows the last read operation of P_j from P_s before c'. In c_r $tid_s(c_r) = MTID(c_0)$ and $dis_s(c_r) \leq dis_j(c_r) + 1$. Now we show that in c_r P_s is the son of P_j and $ack_s(c_r) = true$.

First we prove that P_s is a son of P_j in c_r. Assume towards a contradiction that P_s is not a son of P_j in c_r but is the son of P_j in c'. By Lemma 17 (3) P_s changes its father after c_r only when it simultaneously decreases the value of dis_s. By the definition of c_r it holds that $dis_s(c_r) \leq dis_j(c_r) + 1$, thus if P_s

changes its father to be P_l then $dis_l(c') < dis_s(c_r) - 1 \leq dis_j(c_r) = dis_j(c')$ and hence $P_l \neq P_j$ contradiction.

Now we show that $ack_s(c') = true$ by the definition of c_r and the fact that P_s is the son of P_j in c_r it must hold that $ack_s(c_r) = true$. Assume towards contradiction that P_s assigns $ack_s := false$ during the subexecution E'' that starts with c_r and ends with c'. By lemma 17 (7) during E'' P_s changes the value of ack_s to *false* only when it simultaneously decreases the value of dis_s. Now we show that the value of dis_s does not decrease during E''. By the definition of E'' dis_j is not changed during E''. Since it holds in c_r that $dis_s(c_r) = dis_j(c_r) + 1$ and since P_s is the son of P_j in c' it holds that $dis_s(c') = dis_j(c') + 1$. Thus, the value of ack_s is not changed during E''. □

LEMMA 19. *In every flooding subexecution,* $E' = (c_0, \ldots, c_b)$, c_b *is a BFS configuration.*

PROOF. Define *legal path* in a configuration c to be a directed path in $MT(c)$, $\mathcal{P}(c) = (P_1, P_2, \ldots P_l)$ from the root $P_1 \equiv P_r$, to a leaf, P_l that have the following property: there exist $0 \leq j \leq l$ such that for every $1 \leq k \leq l$ it holds that if $k \leq j$ then $ack_k(c) = \text{false}$, and if $k > j$ then $ack_k(c) = \text{true}$.

First we prove by induction on i, the index of a configuration c in a flooding subexecution E', the following claim:

INDUCTION CLAIM. Every path $\mathcal{P}(c) = (P_1, P_2, \ldots P_l)$ from the root, $P_1 \equiv P_r$, to a leaf, P_l, of $MT(c)$ is a legal path.

INDUCTION BASE, $i = 0$. c_0 immediately follows an atomic step during which P_r extends its *tid* to $MTID(c_0)$ and in the same time assigns $ack_r := false$. It is easy to see that in c_0, the induction assumption holds.

INDUCTION STEP. we assume that the induction assumption is true till c, the ith configuration in E', and we prove that it is true in c' the $i + 1th$ configuration of E'. The only atomic steps that have influence on the induction assumption are atomic steps that includes a write operation that changes the values of *f*, *tid*, *dis* or *ack*. Changes in the values of *f*, *tid*, *dis* and *ack* of the processors in MT during $E' = (c_0, \ldots, c_b)$ can occur by the execution of lines 12, 13, 28, and 33 of the code. By the fact that c_0 is a forest configuration, line 12 is not executed during E'. By Lemma 17 (4) the color of a processor in MT is equal to the color the processor read from its father and hence by the condition of line 24, line 28 is not executed either. Thus, the only atomic steps that changes the values of *f*, *tid*, *dis*, and *ack* in MT are the atomic steps during which:

- *Case 1*: a new processor joins MT (executes line 13 of the code)
- *Case 2*: a processor in MT changes its father in the tree (executes line 13 of the code)
- *Case 3*: a processor in MT changes the value of *ack* to *true* (executes line 33 of the code)

We consider each of the above cases:

Case 1. a new processor, P_m, joins MT as the son of a processor P_f: since $tid_m(c) \neq MTID(c_0)$ then by lemma 17 (6) $ack_f(c) = false$ (i.e., P_f does not assigns $ack_f := true$ before P_m assigns $MTID(c_0)$ in tid_m). Therefore, by the induction assumption it holds in c for every processor, P_k in the path $P_1 \equiv P_r, \ldots, P_{l-1} \equiv P_f$ that $ack_k(c) = false$. Whenever P_m joins MT, P_m assigns $ack_m := false$, hence the induction assumption holds for any path that includes P_m in $MT(c')$. All the other paths in $MT(c')$ are identical to the paths in $MT(c)$, thus by the induction assumption they are legal paths in c' too.

Case 2. a processor P_m in MT changes its father in the tree to be P_f: Let P be a path in $MT(c')$. If P is a path in $MT(c)$ too then by the induction assumption P is a legal path. Otherwise, if P is a prefix of a path in $MT(c)$ then since every prefix of a legal path is a legal path the path P is legal.

We still have to check a path $\mathcal{P}$ which is a concatenation of two parts of legal paths $\mathcal{P} = \mathcal{P}_1 \circ \mathcal{P}_2$, when $\mathcal{P}_1$ is a prefix of a legal path in $MT(c)$ that is ended in P_f, and $\mathcal{P}_2$ is a suffix of a legal path in $MT(c)$, that begins in P_m. By Lemma 17 (3) it holds that such a concatenation occurs only when $dis_m(c) > dis_f(c) + 1$. In such a case, by Lemma 17 (6), $ack_f(c) = false$, and by the induction assumption for every $P_k \in \mathcal{P}_1$ it holds that $ack_k(c) = false$. $\mathcal{P}_2$ is a suffix of legal path in c that begins in P_m. Hence, in c $\mathcal{P}_2$ is a concatenation of a prefix with zero or more processors with *ack* = *false* and a suffix of zero or more processors with *ack* = *true*. Hence in c' the concatenation $\mathcal{P} = \mathcal{P}_1 \circ \mathcal{P}_2$ starts with two or more processors with *ack* = *false* that are followed by zero or more processors with *ack* = *true* and hence $\mathcal{P}$ is a legal path. Thus, the induction claim is true in c also in that case.

Case 3: a processor P_m in MT changes the value of ack_m to *true*: By Lemma 18, it holds in c' for every son, P_s, of P_m that $ack_s(c') = true$. Hence, every path that was a legal path in c is also a legal path in c', and the induction assumption is true in c'.

Now we complete the proof of the lemma: For every neighbor, P_k, of the root it holds in c_b that $tid_k(c_b) = MTID(c_0)$, $dis_k(c_b) = 1$ and $ack_k(c_b) = true$. Since, all the paths in $MT(c_b)$ are legal paths and begins with a processor with *ack* = *true* then for every processor P_i that is not a root in $MT(c_b)$, it holds that $ack_i(c_b) = true$. Thus, by Lemma 17 (6) it holds for every processor, P_i, in $MT(c_b)$ that all its neighbors are in $MT(c_b)$ too. By the definition of c_b every neighbor of the root is in $MT(c_b)$, thus $MT(c_b)$ is a spanning tree of the entire communication graph.

Now we prove that $FSG(c_b)$ is a *BFS* tree. We show that the value of dis_m of every processor, P_m, is equal to the distance d_m of P_m from P_r. Notice that by the fact that c_b is a uniform *tid* configuration and by Lemma 6 it holds that $dis_m(c_b) \geq d_m$. The proof is by induction on d_m. The induction base is due to the fact that $dis_r(c_b) = 0$. We assume that the induction assumption holds for every processor P_m for which $d_m \leq d$ and we prove that it is true for every processor P_l such that $d_l = d + 1$. By the induction assumption P_l has a neighbor P_m with $dis_m(c_b) = d = d_l - 1$. By Lemma 17 (6) $dis_l(c_b) \leq dis_m(c_b) + 1$, it holds that $dis_l(c_b) \leq d + 1$ and hence $dis_l(c_b) = d + 1$.

By the above $FSG(c_b)$ is a *BFS* tree, thus property $(bfs - 1)$ of a *BFS* configuration holds. By the facts that in c_b all the processors with $tid = MTID(c_0)$ and $FSG(c_b)$ is a *BFS* tree that includes all the processors it holds in c_b for every processor that is not the root, P_l, with father P_f, that:

$tidl(c_b) = tid[f](c_b) = tid_f(c_b)$, (due to Assertions 1 and 6 of Lemma 17), $dis_l(c_b) = dis_l[f](c_b) + 1 = dis_f(c_b) + 1$, (due to Assertion 6 of Lemma 17 and the fact that for every processor P_m $dis_m(c_b) = d_m$), $color_l(c_b) = color_l[f](c_b) = color_f(c_b)$, (due to assertion (4) of Lemma 17), $ack_l(c_b) = true$ (by the induction assumption above) and $ot_l(c_b) = false$, (due to Assertion 5 of Lemma 17). Thus, property $(bfs - 2)$ of a *BFS* configuration holds in c_b too. Since after the root extends tid_r for the first time it holds for every neighbor with $tid = tid_r = MTID(c_0)$ that $ot = false$, and by the fact that P_r reads from each of its neighbors P_m $tid_m = MTID(c_0)$ before c_b, property $(bfs - 3)$ of a *BFS* configuration holds too. □

THEOREM 20. *During any execution E, that starts with a leader configuration, the root extends its tid at most once.*

PROOF. Assume towards a contradiction that the root extends its *tid* more than once. By Lemma 19 after the first time that P_r extends its *tid* and before the second time that P_r extends its *tid* during *E*, a *BFS* configuration, c_b, is reached. In c_b it holds for every neighbor P_l of P_r that $ot_r[l] = false$. Thus, during the atomic step that follows c_b, P_r chooses a new color (line 21 of the code) and does not extend its *tid*.

Now we trace a subexecution E' that starts in a *BFS* configuration, c', (in particular, c_b) and ends just before the next atomic step of the execution during which P_r finds that the next coloring is finished (the condition in line 17 holds). We will show that during E' the *tid* of the root is not extended and E' terminates in a configuration c'_b that is a *BFS* configuration. Hence the subexecution, that begins with c'_b and ends exactly before the following atomic step in which P_r discovers that the coloring is finished, reaches a *BFS* configuration without extending the *tid* of the root. In such a way we can repeatedly use the same claims forever.

We prove that during E' the *FSG* is fixed and the global synchronization protocol is stabilized and executed in the right fashion on the fixed *FSG*. Notice that by the definition of E', during E', P_r does not extend its *tid* and does not change its color. Thus for every configuration c in E', the *tid* of a processor equals $MTID(c')$. Moreover, since c' is a *BFS* configuration the value of *dis* of every processor is the distance of the processor from the root. Thus, it holds for every two neighbors P_i and P_j that $\left|dis_j(c') - dis_i(c')\right| \leq 1$. A processor changes its father only when the processor finds that there is a neighbor P_j such that $tid_i > tid_j$, or when $tid_i = tid_j$ and $dis_i < dis_j - 1$. Since the *tid* of all the processors is equal to $MTID(c')$ and since the value of *dis* of every processor is the distance of the processor from the root no processor changes its father in the *FSG* during E'. Thus, during E' *FSG* is a fixed *BFS* tree.

Since during E' the tree that is defined by $FSG(c')$ is not changed and since for every processor P_l with father P_f in c' it holds that

$$color_l(c') = color_l[f](c') = color_f(c'),$$

the global synchronization protocol is already stabilized during E'. Therefore whenever the root finds that all its sons acknowledge that they finished the coloring the system is in configuration c'_b in which for every nonroot processor P_l with father P_f in the system it holds that $ack_l(c'_b) = true$ and

$$color_l(c'_b) = color_l[f](c'_b) = color_f(c'_b).$$

As mentioned above, during E' no processor changes its *tid*, *dis* and *f*, thus in order to prove that c'_b is indeed a *BFS* configuration it remains to show that for every nonroot processor P_l it holds that $ot_l(c'_b) = false$.

By Lemma 2 during E' the color of every processor is changed exactly once. The color is changed from $color_r(c')$ to $color_r(c'_b)$. In c' it holds for every processor that $ot(c') = false$. We now show that no processor assigns *true* in *ot* during E'. Assume towards a contradiction that there is a processor that assigns $ot := true$ during E'. Let P_l be the first processor that assigns $ot_l := true$ during E'. Again by Lemma 2 during E' P_l executes that assignment only after it finds that its fathers' color is $color_r(c'_b)$ while $color_l = color_r(c')$ (line 24). Hence, if P_l assigns $ot_l := true$ then during E' P_l reads from one of its neighbors a color that is

different from $color_r(c'_b)$ and from $color_r(c')$ (line 26). Contradiction, since during E' the color of a processor is either $color_r(c'_b)$ or $color_r(c'_b)$. Hence P_l does not assign ot_l: = *true* and hence the last configuration of E' is a *BFS* configuration. □

Note that by Lemmas 10, and 11, and Corollary 12 it holds that $O(\Delta\mathcal{D})$ rounds after the root extends its *tid* the flooding of the new *tid* is ended. Thus the time complexity of the propagation of information and feedback protocol is $O(\Delta\mathcal{D})$ rounds.

4 Ranking

In this section we present a self-stabilizing protocol which *ranks* the systems' processors. First we present a self-stabilizing ranking protocol that works on systems whose communication graph is an in-tree. This protocol assigns each processor a rank which equals its DFS number. The time complexity of this protocol is $O(\Delta D)$ rounds, and its space complexity is $O(\Delta \log n)$. Then we use the technique of fair protocol combination of [15], and achieve a uniform self-stabilizing ranking protocol for general graphs by composition of this protocol with the protocol presented in the previous section. The time and space complexities of the combined protocol are the sum of the complexities of the two protocols.

The ranking protocol, for a tree system, is also a composition of two other protocols that are composed by a fair protocol composition. In one of those protocols each processor P_j computes the number of processors, n_j, in the subtree rooted at P_j, this protocol is called *counting protocol*. The second protocol is called the *naming protocol*. The naming protocol provides a distinct name for every processor. Each of those protocols stabilizes within $O(\Delta D)$ rounds and the space complexity of a processor is $O(\Delta \log n)$.

4.1 Counting Protocol

Every processor P has a register in which P writes to its father in the tree. A leaf processor writes in every atomic step the value 1 to its father. A non leaf processor repeatedly reads its sons' registers, sums their values, adds one to this sum, and writes the result to its father. Define the *height* of a processor Q in a tree to be the maximal number of processors in a path that starts in a leaf and ends in Q. The correctness and complexity proof of the counting protocol is by induction on the height of the processors. The induction assumption is that following $O(\Delta h)$ rounds the register of a processor, Q, that is in height less than or equal to h, holds the number of processors in the subtree rooted at Q. The induction base is by the fact that every processor that is in height 0 is a leaf. The induction step is derived from the induction assumption for processors in height h and the way processors in height $h + 1$ calculate the number of processors in their subtree.

4.2 Naming Protocol

Every processor, P, orders its sons, $(P_1, P_2, \ldots, P_i)$ and uses a register to write to each of his sons. Let $n_1, n_2, \ldots, n_i$ be the number assigned by the counting protocol for $P_1, P_2, \ldots, P_i$, respectively, and read by P. The root chooses the identity 1 and repeatedly writes it to its sons. The root writes the value $2 + n_1 + n_2 + \ldots + n_{j-1}$ to its j son. A nonroot processor Q repeatedly reads the number in its father register and choose its identity to be that value, say the value x. Q repeatedly writes to each of its sons. Q writes the value $x + 1 + n_1 + n_2 + \ldots + n_{j-1}$ to its j son. Define the *depth* of a processor Q in an in-tree to be the number of processors in the directed path that starts in Q and ends in the root. The correctness and complexity proof of the naming protocol is by induction on the depth of the processors. The induction assumption is that following $O(\Delta d)$ rounds every processor, Q, that is in depth less than or equal to d, chooses an identity that is equal to its *DFS* index in the tree. The induction base is by the fact that the root processor chooses the identity 1. The induction step is derived from the induction assumption for processors in depth d and the way processors in height d and height $d + 1$ communicate.

5 Conclusions

We presented a uniform self-stabilizing leader election protocol. In a uniform system processors do not have unique identifiers. The protocol uses randomization in order to break symmetry. Our protocol is the only protocol which solves the problem without any prior knowledge on the communication graph. Self-stabilizing local and global synchronization protocols are used as building blocks for the leader-election protocol. Those protocols can be used to implement many distributed tasks as synchronizers and reset protocols. Part of the self-stabilizing leader election is a new propagation of information with feedback protocol that terminates within $\Omega(\Delta\mathcal{D})$ rounds.

The self-stabilizing leader election protocol and ranking protocols can be combined, by the fair protocol combination method introduced in [15], with other self-stabilizing protocols that assume either a unique leader or unique identifiers. Thus, the vocabulary of self-stabilizing protocols for uniform distributed systems is enriched.

Throughout the presentation of the leader election protocol in Section 3 it is assumed that a processor can toss a single coin in a single atomic step. Under this restriction it is proven that a leader is elected in $O(\Delta\mathcal{D} \log n)$ rounds. It is easy to verify that when $O(\log n)$ coin tosses are executed in a single atomic step then the protocol stabilizes in $\Omega(\Delta\mathcal{D})$ rounds. Note that $\Omega(\Delta\mathcal{D})$ rounds are required to convey information from one side of the system to the other and thus to elect a leader.

Note that following the stabilization phase of our leader election protocol the protocol repeatedly colors a spanning tree of the system communication graph. Hence, the protocol can be used as a self-stabilizing synchronizer (β synchronizer in the terms of [9]) that converts self-stabilizing synchronous protocols to work in asynchronous system. One of the anonymous referees point out the relation to gossiping algorithms e.g. [13], [19], where processors communicate information among themselves. Our protocol can sport self-stabilizing gossiping easily by repeatedly collecting the information to the elected leader through the spanning tree and then broadcasting it along the tree. Another possibility is to use the virtual ring defined by a DFS

tour on the spanning tree to forward information to the destination and to use hop-counter to eliminate corrupted information from the system.

APPENDIX—NOTATIONS AND TERMS

1	Δ	the maximal degree of a node in the graph.
2	$\mathcal{D}$	the diameter of the graph.
3	n	the number of processors in the graph.
4	*loop iteration*	see definition 1 and 2.
5	*read iteration*	see definition in 2.
6	*FSG*(*c*)	the father-son relation graph of the configuration *c*. *FSG* consist of a directed edge from each non-root processor to its father.
7	*tid*	the tree identifier to which a processor belongs.
8	*dis*	the distance of a processor from its root.
9	*color*	the variable used to repeatedly color a tree.
10	*ack*	a variable used to acknowledge the termination of the coloring procedure in the subtree of the processor.
11	*ot*	Boolean indication on the existence of other trees.
12	*leader configuration*	a configuration with exactly one leader such that in any execution that starts with this configuration the leader is fixed.
13	*forest configuration*	see definition 3.
14	*R*(*c*)	the set of root processors in configuration *c*.
15	*MTID*	see definition 5.
16	*uniform tid configuration*	see definition 5.
17	*utff-execution*	uniform tid fixed forest execution,
18	*flooding subexecution*	see definition 7.

ACKNOWLEDGMENT

We would like to thank the anonymous referees for their constructive remarks that helped to improve this paper. S. Dolov's work was supported in part by TAMU Engineering Excellence funds and the National Science Foundation's Presidential Young Investigator Award CCR-9158478. Part of his work was done while Dr. Dolov was at Technion and Texas A&M University.

REFERENCES

[1] K. Abrahamson, A. Adler, L. Higham, and D. Kirkpatrick, "Probabilistic Solitude Verification on a Ring," *Proc. Fifth Ann. ACM Symp. Principles of Distributed Computing*, Quebec City, pp. 161-173, Aug. 1986.

[2] H. Attiya, M. Snir, and M. Warmuth, "Computing on an Anonymous Ring," *J. ACM*, vol. 35, no. 4, pp. 845-875, Oct. 1988.

[3] K. Abrahamson, "On Achieving Consensus Using a Shared Memory," *Proc. Seventh Ann. ACM Symp. Principles of Distributed Computing*, pp. 291-302, Toronto, Canada, Aug. 1988.

[4] E. Angnostou and R. El-Yaniv, "More on the Power of Random Walks: Uniform Self-Stabilizing Algorithms," *Lecture Notes in Computer Science 579*: *Distributed Algorithms, Proc. Fifth Int'l Workshop Distributed Algorithms*, S. Toueg, P.G. Spirakis, and L. Kirousis, eds, Delphi, Greece, pp. 31-51, Oct. 1991. Springer-Verlag, 1992.

[5] A. Arora and M. Gouda, "Distributed Reset," *Lecture Notes Computer Science 472*: *Foundations of Software Technology and Theoretical Computer Science, Proc. 10th Conf. Foundations of Software Technology and Theoretical Computer Science*, K.V. Nori and C.E. Veni Madhavan, eds., pp. 316-331, Dec. 1990. Springer-Verlag, 1990.

[6] Y. Afek, S. Kutten, and M. Yung, "Memory-Efficient Self-Stabilization on General Networks," *Lecture Notes in Computer Science 486*: *Distributed Algorithms, Proc. Fourth Int'l Workshop Distributed Algorithms*, J. Van Leeuwen and N. Santoro, eds, pp. 15-28. Springer-Verlag, 1991.

[7] Y. Afek, S. Kutten and M. Yung, "Local Detection for Global Self-Stabilization," preprint.

[8] B. Awerbuch and G. Varghese, "Distributed Program Checking a Paradigm for Building Self-Stabilizing Distributed Protocols," *Proc. 31st Ann. IEEE Symp. Foundations of Computer Science*, pp. 268-277, 1991.

[9] B. Awerbuch, "Complexity of Network Synchronization," *J. ACM*, vol. 32, no. 4, pp. 804-823, 1985.

[10] G.M. Brown, M.G. Gouda, and C.L. Wu, "A Self-Stabilizing Token System," *Proc. 20th Ann. Hawaii Int'l Conf. System Sciences*, pp. 218-223, 1987.

[11] J.E. Burns and J. Pachl, "Uniform Self-Stabilizing Rings," *ACM Trans. Programming Languages and Systems*, vol. 11, no. 2, pp. 330-344, Apr. 1989.

[12] J.E. Burns, "Self-Stabilizing Rings Without Demons," Technical Report GIT-ICS-8736, Georgia Inst. of Technology.

[13] B.S. Chlebus, K. Diks, and A. Pelc, "Fast Gossiping With Short Unreliable Messages," *Discrete Applied Math.*, special issue on gossiping, vol. 53, pp. 15-24, 1994.

[14] E.W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Comm. ACM* vol. 17, no. 11, pp. 643-644, 1974.

[15] S. Dolev, A. Israeli, and S. Moran, "Self Stabilization of Dynamic Systems," *Proc. Ninth Ann. ACM Symp. Principles of Distributed Computing*, Quebec City, pp. 103-118, Aug. 1990. *Distributed Computting* vol. 7, pp. 3-16, 1993.

[16] S. Dolev, A. Israeli, and S. Moran, "Uniform Dynamic Self-Stabilizing Leader Election," in *Lecture Notes in Computer Science 579*: *Distributed Algorithms, Proc. Fifth Int'l Workshop on Distributed Algorithms*, Delphi, Greece, S. Toueg, P.G. Spirakis and L. Kirousis, eds., pp. 163-180, Oct. 1991. Springer-Verlag, 1992.

[17] S. Dolev, A. Israeli, and S. Moran, "Analyzing Expected Time by Scheduler-Luck Games," *IEEE Trans. Software Eng.*, vol. 21, no. 5, pp. 429-439, May 1995.

[18] R. G. Gallager, "Finding a Leader in Networks With $O(E) + O(N \log N)$ Messages," Internal Memo., M.I.T., Cambridge, Mass., 1978.

[19] S.M. Hedetniemi, S.T. Hedetniemi, and A.L. Liestman, "A Survey of Gossiping and Broadcasting in Comm. Networks," *Networks*, vol. 18, pp. 319-349, 1988.

[20] P. Humblet, "Selecting a Leader in a Clique in $O(N \log n)$ Messages. Internal Memo., Laboratory for Information and Decision Systems, M.I.T., Cambridge, Mass., 1984.

[21] A. Israeli and M. Jalfon, "Token Management Schemes and Random Walks Yield Self Stabilizing Mutual Exclusion," *Proc. Ninth Ann. ACM Symp. Principles of Distributed Computation*, Quebec City, pp. 119-132, Aug. 1990.

[22] A. Israeli and M. Jalfon, "Self Stabilizing Ring Orientation," *Lecture Notes in Computer Science 486*: *Distributed Algorithms* (*Proc. Fourth Int'l Workshop on Distributed Algorithms*, Bari, Italy, J. Van Leeuwen and N. Santoro, eds, pp. 1-14, Sept. 1990. Springer-Verlag, 1991.

[23] A. Itai and M. Rodeh, "Probabilistic Methods for Breaking Symmetry in Distributed Networks," *Information and Computation*, vol. 88, no. 1, pp. 60-87, 1990.

[24] E. Korach, S. Kutten, and S. Moran, "A Modular Technique for the Design of Efficient Distributed Leader Finding Algorithms," *ACM Trans. Programming Languages and Systems*, vol. 12, no. 1, pp. 84-101, 1990.

[25] E. Korach, S. Moran, and S. Zaks, "Tight Lower and Upper Bounds for Some Distributed Algorithms for Complete Network of Processors," *Proc. Third Ann. ACM Symp. Principles of Distributed Computing*, pp. 199-207, 1984.

[26] S. Katz and K.J. Perry, "Self-Stabilizing Extensions for Message-Passing Systems," *Proc. Ninth Ann. ACM Symp. Principles of Distributed Computing*, Quebec City, pp. 91-101, Aug. 1990.

[27] L. Lamport, "The Mutual Exclusion Problem: Part II—Statement and Solutions," *J. ACM*, vol. 33, no. 2, pp. 327-348, 1986.

[28] Y. Matias and Y. Afek, "Simple and Efficient Election Algorithms for Anonymous Networks," manuscript, 1989.

[29] A. Segall, "Distributed Networks Protocols," *IEEE Trans. Comm.*, vol. 29, no. 1, pp. 23-35, Jan. 1983.

[30] B. Schieber and M. Snir, "Calling Names on Nameless Networks," *Proc. Eighth Ann. ACM Symp. Principles of Distributed Computing*, Edmonton, pp. 319-328, Aug. 1989.
[31] M. Tchuente, "Sur l'auto-stabilisation dans un réseau d'ordinateurs," RAIRO Inf. Theor., vol. 15, pp. 47-66, 1981.

Shlomi Dolev received his BSc in engineering and BA in computer science in 1984 and 1985, respectively, and his MSc and PhD in computer science in 1990 and 1992, respectively, from the Technion Israel Institute of Technology. From 1992 to 1995 he was at Texas A&M University as a visiting research specialist. He visited Carleton University in 1994. In 1995 he joined the Department of Mathematics and Computer Science at Ben-Gurion University. His current research interest include the theoretical aspects of distributed computing and communication networks.

Amos Israeli received his BSc in mathematics and physics from Hebrew University in 1976, and his MSc and DSc in computer science in 1980 and 1985, respectively. He was a postdoctoral fellow at the Aiken Computation Laboratory at Harvard. His research interests are in parallel and distributed computing and robotics. He has worked on the design and analysis of wait-free and self-stabilizing distributed protocols.

Shlomo Moran received his BSc and PhD degrees in Mathematics from the Technion in 1975 and 1979, respectively. In 1979-1981 he was at the University of Minnesota as a visiting research specialist. In 1981 he joined the Computer Science Department at the Technion, where he is now a full professor. In 1985-1986 he visited at IBM T.J. Watson Research Center. In 1992-1993 he visited at AT&T Bell Laboratories and in Centrum voor Wiskunde en Informatica, Amsterdam. His research interests include distributed computing, combinatorics and graph theory, and complexity theory.