The Local Detection Paradigm and its Applications to Self Stabilization*

Yehuda Afek
Computer Science Department,
Tel-Aviv University, Israel 69978,
afek@math.tau.ac.il

Shay Kutten
IBM T.J. Watson Research Center,
P.O. Box 704, Yorktown Heights, NY 10598
kutten@watson.ibm.com

Moti Yung
IBM T.J. Watson Research Center,
P.O. Box 704, Yorktown Heights, NY 10598
moti@watson.ibm.com

*A preliminary version of this paper appeared in the Proc. of the 1990 International Workshop on Distributed Algorithms, Bari, Italy.

Abstract

A new paradigm for the design of self-stabilizing distributed algorithms, called *local detection*, is introduced. The essence of the paradigm is in defining a local condition based on the state of a processor and its immediate neighborhood, such that the system is in a globally legal state if and only if the local condition is satisfied at all the nodes. In this work we also extend the model of self-stabilizing networks traditionally assuming memory failure to include the model of dynamic networks (assuming edge failures and recoveries). We apply the paradigm to the extended model which we call "dynamic self-stabilizing networks." Without loss of generality, we present the results in the least restrictive shared memory model of read/write atomicity, to which end we construct basic information transfer primitives.

Using local detection, we develop deterministic and randomized self-stabilizing algorithms that maintain a *rooted spanning tree* in a general network whose topology changes dynamically. The deterministic algorithm assumes unique identities while the randomized assumes an anonymous network. The algorithms use a constant number of memory words per edge in each node; and both The size of memory words and of messages is the number of bits necessary to represent a node identity (typically $O(\log n)$ bits where n is the size of the network). These algorithms provide for the easy construction of self-stabilizing protocols for numerous tasks: reset, routing, topology-update and self-stabilization transformers that automatically self-stabilize existing protocols for which local detection conditions can be defined.

Key words: self-stabilizing distributed algorithms, spanning-tree algorithm, leader election, reset protocol, dynamic networks algorithms, locality in distributed computing, local detection/ local checking.

1 Introduction

In a distributed system it is usually impossible to detect an illegal global state by individually observing only private states of processors. For example, in a token ring, the fact that each node has at most one token does not imply that there is exactly one token in the ring, as required from any legal global state. However, this individual condition is true at any node in any globally legal state. In another example, the fact that each node in a network has at most one distinguished outgoing link does not imply that the nodes have distinguished a spanning tree, though the converse is true. In 1974 Dijkstra suggested the notion of *self-stabilizing systems*. The notion is particularly interesting because of the above phenomenon: a system can be placed in an illegal global state, while each process is individually in a legal state. The self-stabilizing property assures that such a system automatically moves into and stays in a globally legal state regardless of its initial condition.

The implementation of the self stabilization methodology places a set of procedures, one at each processor, that govern and dictate the necessary state transitions to guarantee the eventual entry into a globally legal state and its maintenance thereafter. For example, if a self-stabilizing token ring would be placed in a state in which three tokens are present in the ring then the self-stabilizing algorithm would automatically move the system into a state such that, in any subsequent state exactly one token is present in the ring, which circulates in a certain legal pattern.

Since the introduction of this paradigm by Dijkstra in 1974 many self-stabilizing algorithms were developed to stabilize different distributed systems [Dij74, BGW89, BP89, AB89, KP94, GM91, GH89, IJ90, DIM94, Kru79].

In this paper we first introduce a new paradigm and methodology for the development of self-stabilizing algorithms, called *local detection*. Secondly we employ the new methodology to design a self-stabilizing spanning tree algorithm for the standard model of asynchronous networks with a dynamically changing topology as in [AAG87]. Finally we combine our self-stabilizing algorithm with the techniques of [AM94] to construct a randomized self-stabilizing algorithm for anonymous networks.

There are several general implications to the spanning tree algorithm and the methodology presented here, the most important of which is the introduction of a self-stabilizing reset algorithm and its combination with the transformer methodology of Katz and Perry [KP94]. In that paper a general methodology to transform non-self-stabilizing algorithms into self-stabilizing counterparts is presented. Katz and Perry assume that a unique leader exists and use it to manage and coordinate the global detection of illegal states. The self-stabilizing spanning tree algorithm presented here may be combined with [KP94] to remove the unique leader assumption. Furthermore, our methodology combined with our spanning tree algorithm produce a new self-stabilizing transformer. In the new transformer the global detection of Katz and Perry is replaced by our local detection, and the spanning tree algorithm is used to reset the application if necessary. Thus we construct a self-stabilizing algorithm for any task suitable for local detection.

Another contribution of this work is the model of "dynamic self-stabilizing networks" that combines dynamically changing networks that undergo topological changes and the traditional self-stabilizing model in which memory fault may occur. In addition, the algorithms presented here are under a very strict notion of read/write atomicity of operations (unlike their original presentation in [AKY90]). A mechanism to assure the self-stabilizing exchange of messages under this condition is designed and presented.

Related work: First, we observe that a self-stabilizing spanning tree algorithm may be easily derived from the dynamic topology maintenance algorithm of Spinelli and Gallager [SG89]. However, being based on the maintenance and exchange of full topology information the modified Spinelli-Gallager algorithm would have very high communication complexity (each node reading $O(E)$ words from each of its neighbors in each round, where E is the total number of links in the network) and high space complexity ($O(E)$ words of memory for each incident link).

Two other self-stabilizing spanning tree algorithms were developed at the same time that we developed ours but for a different model. In [DIM94] Dolev, Israeli and Moran have designed a self stabilizing spanning tree algorithm with read/write atomicity, assuming the existence of a unique distinguished processor in the network. If the network is partitioned, then the parts that do not include the unique leader do not stabilize, thus their model does not generalize to dynamic networks, where nodes and edges may fail and recover. Arora and Gouda [AG90] have designed a self-stabilizing spanning tree protocol that assumes that each node has a unique identity; however they assume a known bound on the network size in order for the algorithm to self stabilize. The quiescence time complexity of their algorithm depends on that bound, regardless of the actual size. Note that dynamic faults may significantly decrease the actual size of a network component (e.g., to a logarithmic size in the bound). In our algorithms we either assume a network bound is unknown or we assume a bound on the network which is only used to allocate an effective register and message size but does not influence the actual stabilization-time. In [IJ90] Israeli and Jalfon present an interesting randomized algorithm for passing a token (i.e., random walk) in general networks, under the assumption that n the total number of nodes is known. This paper was then extended by Coppersmith, Tetali and Winkler in [CTW93]. Many other self-stabilizing algorithms were designed and the above is in by no means a complete reference list, however the above four works are closely related to ours in terms of models and assumptions.

The methodology of combining self-stabilizing building blocks to compose a more complex self-stabilizing algorithm is beyond the scope of our paper. Such methodologies were introduced and studied in [KP94, DIM94, IJ90, Sto93]. Such a composition is employed when we use our tree algorithm as a building block for designing self stabilizing transformers, self stabilizing reset, etc.

Subsequent to [AKY90] and independently of the current paper S. Dolev, Israeli and Moran [DIM91] have also presented a randomized election algorithm for anonymous networks.

Model: Unlike [DIM94, AG90, IJ90], in this paper we make no extra assumptions that may limit the standard asynchronous network model or their running times [GHS83, AAG87]. We introduce a model of a fault tolerant self-stabilizing network that generalizes the model of dynamic networks. Thus, we require that connected components of a possibly partitioned network continue executing the protocol and eventually stabilize in a time proportional to their size. Our algorithm support this requirement since connectivity to a distinguished node is not assumed and the algorithm is symmetric, i.e., works correctly over any set of connected processors.

In the basic model we consider here each node has a unique identity represented in $O(\log n)$ bits (we relax this model later on and use randomization). The spanning tree algorithm stabilizes in a finite amount of time after the end of any sequence of topological changes and nodes memory corruption (node IDs may not be corrupted). The communication model assumed is that of shared memory, i.e., each pair of neighboring nodes share a pair of atomic read/write registers. Each atomic operation is either a read, or a write, or an internal operation (in this respect we follow [DIM94]). The algorithms presented here can be adapted to the message passing model by using the techniques of [AB89, AGR92, AV91, Var92]. This transformation to the message passing model is possible under the realistic assumptions that (1) the capacity of each link is bounded, and (2) for each link one of its incident nodes is distinguished as the link master (an assumption that trivially holds if nodes have unique identities). This adaptation will not be treated here.

The space complexity of our spanning tree algorithm is $O(\log n)$ bits per node. This space complexity overhead is negligible since the standard message size is also $O(\log n)$ bits and one message buffer is kept in any event for communication purposes at each incident link. We remark that for bounded-degree networks or networks where special control signals (channels) are used (rather than transferring control over regular messages) it makes sense to further reduce the space requirements, as was recently pursued in a number of works [MOOY92, AO94, IL94]. Our stabilization-time complexity is $O(n^2)$.

When unique ID's are not available, we employ randomized algorithms that do not need predefined ID's. The space bound in this case is considered under two model variations: Either, as in practice, a $O(\log n)$ bit size register is given (and, this is the only usage of n), or (which is more theoretically appealing) the register size is unbounded. In the later case, the adversary is constrained to access (and corrupt) any finite prefix of the registers, while the algorithm is capable of accessing the prefix plus $O(n)$ additional bits of the register in one step.

Local detection and its extensions: The essence of *local detection* is that the system is in a globally legal state if and only if a certain local condition is satisfied in all the nodes of the network. The local condition in each process is a boolean expression over the variables of the process and the variables of its immediate neighbors in the network. Thus, it is enough for a node to exchange state information with its neighbors to either maintain correctness, or to restart the algorithm in case it enters an illegal state. Hence, in a constant amount of time after the global state is becoming illegal, a reset or some other correction is activated. This is analogous to a jigsaw puzzle where local matchings throughout imply a global legal state.

A straightforward example where the local detection method may be applied is in the topology update algorithm. In this algorithm each node maintains a description of the entire topology in its memory. To facilitate the local detection each node is also assumed to eventually know the correct state of each of its incident links, this is achieved by a self stabilizing link level protocol [AB89, APSV91]. Each node repeatedly checks the consistency of its information, i.e., it compares its topology description to that of its immediate neighbors, and it checks that its own topology description agrees with its knowledge about the state of its incident links. It is easy to verify that if the network is in a global state in which some of the nodes have an incorrect topology description then some of the nodes would detect an inconsistency. If a discrepancy is detected, then there is an inconsistency that the topology update algorithm must correct.

Note that in the simple example above, a node has to read the entire topology of each of its neighbors in each round of computation. Since such an amount of information can be neither stored in one register nor held in one message, (because each of these is typically restricted to $O(\log n)$ bits) then the exchange of local information in each round would take $O(\min\{E, n^2/\log n\})$ time. In the algorithms presented herein, the amount of locally exchanged information is $O(\log n)$ bits, thus each round takes $O(1)$ time.

Unlike the local detection suggested here, in [KP94] an illegal state is detected by collecting information about the global state of the network to one processor. This global detection is harder and much slower to implement since it requires knowing an upper bound on the delay for collecting the global information.

Following the original version of our paper several other self-stabilizing algorithms that employ the *local detection* paradigm were designed. In [AV91] methods are suggested to apply the local detection (called there *local checking*) to other tasks, such as, shortest paths, topology update, leader election, and computing a maximum flow. Recently [APVD94] developed a methodology to combine the local detection principle with any self stabilizing reset protocol that has certain properties. Another contribution of [APVD94] is an exact and formal definition of local detection (checking) and a rather general characterization of the tasks that can be locally checked, and globally corrected. Together with [AV91] and [APSV91] the work of [APVD94] yields a comprehensive understanding of the usage of local detection for self stabilization. In [APSV91], the methodology of local detection and global correction developed herein is extended to local detection and local correction. In the local correction a reset procedure is locally activated. The methodology is applied in [APSV91] to develop self-stabilizing interactive distributed protocols, such as, end-to-end communication and network reset. In [AKM+93] and [A94], local detection is used to self stabilize a network synchronizer [Awe85]. Another body of work in [NS93] considers a related question where a class of Locally Checkable Labels is defined and shown as a useful tool in the definition of local checking and computing in a distributed setting. In particular, Naor and Stockmeyer [NS93] show other tasks that can be easily and locally checked, such as the maximal independent set; further results in this model are in [MNS94].

Organization: In Section 2 we present the model of dynamic self-stabilizing networks. In Section 3 we present our main algorithm and in Section 4 its correctness. Related discussions and extensions are in Section 5, and conclusions in Section 6.

2 The Model

The network consists of a set of n processors communicating by reading and by writing shared memory. Each processor has its own set of single-writer multi-reader registers. A pair of processors that communicate directly can read each other's register. The direct communication relations between the processors is represented by an undirected graph (V, E), where V is the set of processors, and $(p, q) \in E$ if and only if p and q can read each other's registers. To break symmetry we first assume that each processor has a unique Id (hardwired in its code). The total number of processors, n, is unknown to the processors and may change dynamically.

Processors communicate only by reading the memory of neighboring processors and by writing, each to its own local memory. Each processor is a state machine with a bounded number of states (which can be a function of n). The local computation at each processor is a sequence of transitions, each consisting of an operation that moves the processor from its given state to a new (possibly the same) state. Each processor operation is either a local computation step, or an atomic read of a neighbor's memory, or an atomic write of its own memory. Without loss of generality we assume that in one atomic step a processor can both read and write its own (non shared) memory. The fair scheduler (demon) of the global computation is an infinite sequence of processors such that each processor appears in the sequence infinitely often. Whenever a processor appears in the schedule its next transition is performed (every processor always has an operation (e.g. read one of the neighbors memory) that is enabled unless the processor experiences a fault). Such an Atomic Read/Write demon for self-stabilizing computations was first introduced in [DIM94]: they show how to convert self stabilizing protocols to work with Read/Write atomicity.

We use the notion of time only for the sake of analyzing the complexity (but not for the specification of the algorithm). We use the rather standard definition where each action of communication (a read of a neighbor's register, or write of an own register) takes at most one unit of time. In each time unit one register may be read from or written into. The time-complexity measure is affected by an increment of one when the current transition has causal relationship with (i.e., must rely upon and is affected by) a previous transition. In a unit of time, traditionally, an $O(\log n)$-bit size message can be read by a neighbor; we employ such messages.

A local state is the memory content of one node. The Cartesian product of local states of all processors defines the global states. In a self-stabilizing system, a subset of the set of global states is defined as legal global states. From a legal state the computation moves the system only to another (possibly the same) legal state and, starting from any state a fair scheduler eventually brings the system to a legal state. At the start of the computation the adversary may put each processor in an arbitrary local state.

A projection of the global state on a set VAR of variables is the Cartesian product of local states where all the variables except for those in VAR are omitted. A self stabilization problem P is defined by a set of legal global states using some set of variables VAR_P. A (self stabilizing) protocol (or system) solves P if the projection of the global states of the system on VAR_P induces a mapping from the legal states of the system onto the legal states of the problem.

We assume the model of a dynamic network, that is, links and processors can be removed and added an arbitrary number of times. We further assume that there is a local self-stabilizing mechanism that eventually updates at each processor the status of its incident links and neighboring processors. When a link is down the processors incident to that link cannot read each others memory. We assume that the sequence of topological changes is finite (i.e., eventually topological changes cease). Such assumptions represent a system which reaches a working state for large enough time and are common in the literature (see for example [AAG87]).

In a dynamic self-stabilizing network, a state is defined as legal if first, the topological changes have ceased, and second in this final topology it is in a legal state.

In the spanning tree problem each node has a Parent variable, that holds an Id of a neighboring processor. We say that a tree spans the network (component) when the collection of the Parent variables of the network (component) defines such a rooted tree in the graph-theoretic sense, i.e., a rooted tree data-structure of records labeled by the Id's. (For convenience in our algorithm the parent pointer of the root points at the root itself.) Our main algorithm is a self stabilizing procedure that computes such a tree. Using this algorithm as a building block it is easy to construct several other self stabilizing algorithms such as: mutual exclusion, snapshot, reset, and leader election (using the methodology mentioned above). Given a reset protocol many problems can be solved in a self stabilizing manner. Given the tree, a global checking such as that of [KP94] can be performed more efficiently along the tree.

3 A Self Stabilizing Spanning Tree Algorithm

Our methodology distinguishes between two parts of self-stabilizing algorithms, namely the *detection* (checking) part, and the *reformation* (correction) part. The main observation, as was defined above, is that it is possible to detect a globally illegal state by verifying only local conditions. If the system is in a globally illegal state then at least one of the nodes observes that its local condition is false. Nodes with a false local condition start the reformation part of the algorithm, which will bring the system back to a legal state. We remark that when the "detection via local checking" idea is translated to the message-passing model, we require an ongoing local exchange of state information between neighboring nodes. (This is needed in order to check the local condition, since this local condition depends on the values of variables at the neighbors.) It is know that self-stabilizing systems have to be indefinitely active when solving a non trivial (global) task. Otherwise, if there is a legal global state in which the system may be dormant, the adversary would place the system in such a global, but inconsistent state. The advantage of our methodology is that only the detection part, which is local, is indefinitely active and reacts to memory corruptions and topological changes in constant time by activating the correction part.

In a nutshell, the algorithm can be described as follows. Upon detecting a globally illegal state any node v starts a process of constructing a spanning tree labeled by its

Id and rooted at v. During that process "larger identity trees" take over "lower identity trees". That is, a node that belongs to a tree that seems to be rooted in a smaller identity node and has a neighbor belonging to a tree with a larger identity root, takes (careful) measures to join the tree rooted at the larger identity node. Eventually, the tree of the largest Id node overruns all other trees and spans the connected component.

A spanning tree is distributively maintained in the network by keeping at each node a pointer to the node's parent in the tree. The parent pointer of the root points to itself. One of the main difficulties in self-stabilizing spanning tree algorithms stems from the fact that in an erroneous state the parent links could be placed in a cycle. For example, if a spanning tree algorithm is based on the spanning tree that is labeled and rooted at the node with the largest Id, then a Hamiltonian cycle labeled with a label larger than all the Id's in the network would look to each node locally as a legitimate tree. One way to circumvent this problem is, for example, to assume that nodes know the value of the maximum identity, or equivalently to require that exactly one node can be the root [DIM94]. Another possibility is to assume that the nodes know an *a-priori* bound on the network size and use a distance parameter to detect cycles [AG90]. That is, illegal situation is detected in [AG90] when the value of the distance parameter is larger than the bound on the number of nodes. These two kinds of restrictions are undesirable in the context of dynamic networks, where availability of a unique node or a tight bound on the size of a network connected component cannot be assumed.

The algorithm first uses local detection to observe an illegal state. For the algorithm to progress, we then define *transient* states that although are illegal satisfy some other local conditions. The transient states are a sequence of steps in the algorithm that move the system from an illegal state, to the desired globally legal state.

When a node wishes to join another tree whose root Id is larger, that node has to propagate a *request* "message" along the new tree branches to the root and to receive a *grant* message back. Of course, these messages propagate through the shared memories of the nodes via a sequence of read and write operations. The paradigm of local detection is applied also to the mechanism that propagates these messages. That is, an illegal message (e.g., one that was not initiated by the node that is claimed to be its source) is detected and eliminated.

We show that once the network reaches a global legal state, that state will persist, while the transient states are non-recurring. That is, the system passes through those states but will not return to them unless the network topology changes again or new faults occur.

3.1 Detailed Description

To simplify the algorithm and its proof of correctness we assume that each processor maintains all its *shared* variables in one single-writer-multi-reader atomic register. (No generality is lost since this register can be implemented in the case that the variables are in fact given each as a separate register.)

Performing the algorithm (the code is given in Figure 1), each node loops forever and follows a three step iteration, comprising of the following commands: (1) read all the neighbors' memories, (2) update a local copy of the neighbors variables, and (3) check various conditions and decide whether to perform a state transition or not, if yes then perform the transition.

We distinguish between two kinds of shared variables in each node. The variables that intuitively capture the "actual" state of the node, and those that are used for communication and synchronization between neighboring nodes. A variable Var of the first type at node v is denoted $v.\mathsf{Var}$ and is called a state variable.

We impose a strict discipline of communication between the nodes in the algorithm. Before a node may change any of its state variables all its neighbors must know and record the value of its current state variables. To accomplish this synchronization we give each node a sequence number state variable (which may have one of three allowed values) called $v.\mathsf{Toggle}$ and synchronization variables, called *echo*, one for each of its neighbors.

Before any change in its state variables each node u reads its neighbors variables in two steps: In the first step, for each neighbor v the processor at node u reads the variables of v into an internal variable, local to u. In the second step, for each neighbor v, the processor at u writes the value of $v.\mathsf{Toggle}$ read in the first step, into the corresponding echo shared register that node u maintains for neighbor v. We denote the echo of $v.\mathsf{Toggle}$ at node u by $u(v.\mathsf{Toggle})$ and the internal copy at node u of any other state variable $v.\mathsf{Var}$ by $u(v.\mathsf{Var})$. These copy and echo variables are *not* state variables (neither of node u nor of node v).

Each time a node changes the value of its state variables it increments its toggle variable modulo 3. Thus, if $v.\mathsf{Toggle}$ equals to the last value v read in $u(v.\mathsf{Toggle})$, then node v "knows" that u "knows", and agrees with v on the value of its current state variables (see Lemma 4.2).

In the code of a node, each reference to a neighbor's variable (except when reading it) is to the internal copy of the neighbor's variable. For the sake of clarity, such a reference (to a state variable Var of node u) is written as $u.\mathsf{Var}$ in the code of the processor at node v (instead of writing "v"s internal copy of $u.\mathsf{Var}$, or of writing $v(u.\mathsf{Var})$).

Each node v in the network has 10 state variables which we classify in four groups.

1. $v.\mathsf{Toggle}$ is the three valued integer, toggle counter, as explained above.

2. The standard variables { Id, Edge_list }:

   $v.\mathsf{Id}$ is the read-only (hardwired) identity of node v and

   $v.\mathsf{Edge\_list}$ is a local list of node identities such that the incident link from v to each node u in the list is believed to be operational and the processor at each such node u is also believed to be up. This list is maintained by a lower level self-stabilizing protocol which is beyond the scope of this paper. The lower level protocol guarantees that each change in a link or node status is eventually recorded in Edge_list. If v is reading the memory of its neighbor u while u's status in Edge_list is incorrect, then that read may return any value.

3. The variables related to the tree structure { Root, Parent, and Distance} where:

v.Root is a node identity which in legal states is supposed to be the identity of the root of the tree to which node v belongs; (we omit the word "supposed" in the sequel.)

v.Parent is the identity of the parent of v in the tree.

v.Distance is a non-negative integer which is the distance, in the tree, from node v to its root.

4. The variables related to passing the *request* and *grant* messages {Request, From, To, and Direction} (all of which may also be assigned the special value $\perp$) where:

v.Request is a node identity which is either an Id of a node that is currently requesting to join the tree to which v belongs, or is equal to v.Id if v itself is trying to join another tree;

v.From is a node identity which is either the Id of the neighbor from which v copied the value of v.Request, or v.Id if v has initiated a request in an attempt to join a new tree;

v.To is a node identity that is the name of a neighbor of v through which v is trying to propagate the Request message;

v.Direction is either Ask, to indicate that the node whose Id is in v.Request wishes to join the tree, or Grant to indicate that this request has been granted.

A formal description of the algorithm is given in Figure 1. The program consists of a set of actions, each is specified as:

$$\langle guard \rangle \rightarrow \langle command \rangle$$

where the *guard* is a Boolean expression and the *command* is a sequence of assignments. A process executes an action if the action's corresponding guard was found true. The execution of the algorithm at each node proceeds by repeating forever an infinite loop, that includes a read of all the neighbor's shared-memory into internal copies. Then, the guards of all the actions are evaluated one after the other (using the internal copies). If any is true, then the corresponding action of one such true guard is performed atomically. Since the evaluation of the guard uses only internal (non shared) variables we assume (without loss of generality) that the evaluation of the guard and the writing of the shared memory together is an atomic operation. Starting at any point of time, any node reads the shared-memory of each neighbor in Edge_list infinitely often by this sequence of steps.

In the following we define a local condition, called st, on the variables at each node such that the condition holds at all the nodes if and only if the network is in a globally legal state in which a correct unique tree spans the network. The condition is periodically checked by each of the processors. If a violation is detected then the algorithm takes the network through a finite sequence of state transitions to a globally legal state in which the condition holds at all the nodes.

Condition $st(v)$:
$\{[(v.\mathsf{Root} = v.\mathsf{Id}) \wedge (v.\mathsf{Parent} = v.\mathsf{Id}) \wedge (v.\mathsf{Distance} = 0)] \vee$
$[(v.\mathsf{Root} > v.\mathsf{Id}) \wedge (v.\mathsf{Parent} \in v.\mathsf{Edge\_list}) \wedge (v.\mathsf{Root} = v.\mathsf{Parent.Root}) \wedge (v.\mathsf{Distance} =$

$v.\mathsf{Parent.Distance}+1)]\}$
$\wedge\,(v.\mathsf{Root} \geq \max_{x\in \mathsf{Edge\_list}} x.\mathsf{Root})$

which reads as follows:

$\{[v \textit{ is a tree root}] \vee [v \textit{ is on a tree branch}]\}$
$\wedge\,(v\textit{'s root identity is not smaller than its neighbors root identities})$.

Recall that when node v checks and finds that Condition st holds at node v, it holds for the internal copies at node v. Thus, for example, the term $(v.\mathsf{Root} = v.\mathsf{Parent.Root})$ is a shorthand for writing $(v.\mathsf{Root}= v(v.\mathsf{Parent.Root}))$. We use this shorthand whenever no ambiguity arises. Also note that for the global state to be legal it is required both that (1) Condition st holds; and that (2) the value of $v(v.\mathsf{Parent.Var}) = v.\mathsf{Parent.Var}$ for every variable $v.\mathsf{Parent.Var}$ that appears in the definition of Condition st. Recall, however, that nodes repeatedly read the variables of their neighbors. Thus, if (1) above starts holding indefinitely, then (2) above eventually (in a constant time) becomes true as well.

If Condition st is not satisfied at any of the nodes then the algorithm takes the network from the illegal global state to a legal state through a sequence of transient *semi-legal* states. Another local condition, called $frst$, is defined on the variables at each node such that Condition $frst$ holds at all the nodes if and only if the network is in a globally semi-legal state (and will enter a legal state in a finite number of transitions). If Condition $frst$ is satisfied at all the nodes, then the graph induced by the parent pointers in the network has no cycles, hence it is a directed forest.

Condition $frst(v)$:
$[(v.\mathsf{Root} = v.\mathsf{Id}) \wedge (v.\mathsf{Parent} = v.\mathsf{Id}) \wedge (v.\mathsf{Distance} = 0)] \vee$
$[(v.\mathsf{Root} > v.\mathsf{Id}) \wedge (v.\mathsf{Parent} \in v.\mathsf{Edge\_list}) \wedge (v.\mathsf{Root} = v.\mathsf{Parent.Root}) \wedge (v.\mathsf{Distance} = v.\mathsf{Parent.Distance}+1) \wedge (v.\mathsf{Root} \geq \max_{x\in \mathsf{Edge\_list}} x.\mathsf{Root})]$

Note that $frst$ differs from st by the omission of $\wedge\,(v.\mathsf{Root} \geq \max_{x\in \mathsf{Edge\_list}} x.\mathsf{Root})$ from the first term (the first pair of square brackets, namely when v is its own root). Clearly $st(v)$ implies $frst(v)$ but not vice-versa.

If the process at node v detects that Condition $frst(v)$ (and thus also Condition $st(v)$) does not hold, it becomes a root by performing Action 1. When node v is a *root* the following condition holds at v:
$(v.\mathsf{Root} = v.\mathsf{Id}) \wedge (v.\mathsf{Parent} = v.\mathsf{Id}) \wedge (v.\mathsf{Distance} = 0)$.
By becoming a root Condition $st(v)$ does not necessarily become true, however Condition $frst(v)$ does become true. If Condition $frst$ holds at all the nodes then the graph induced by the parent pointers is a forest. Each tree in the forest is labeled by the identity of its root. When Condition $frst$ holds in the network then Condition st holds at nodes that belong to the tree that is labeled with the maximum identity in the network. If Condition $frst(v)$ is true but $st(v)$ is false, then node v eventually joins the tree-construction process of a neighboring tree, with a larger Root (Action 2).

If node v is a root and its Id is not larger than all its neighbors' Roots, it attempts to join another tree. It chooses the neighbor u whose Root is the largest among the Roots of its neighbors, and makes a request to join as a child of this neighbor u (Action 2). For

While (true) **do**

read the shared memory of each neighbor in v.Edge_list into an internal copy;

write the value read from each neighbors Toggle variable into its corresponding *echo*;

if for every neighbor u: $u(v.\mathsf{Toggle})=v.\mathsf{Toggle}$

then select one of the following commands whose guard is true and perform it:

1. $\neg frst$ $\rightarrow$ $v.\mathsf{Root} := v.\mathsf{Id}$; $v.\mathsf{Parent} := v.\mathsf{Id}$ $v.\mathsf{Distance} := 0$

/*make $frst$ true*/

2. $frst \wedge (\exists u \in v.\mathsf{Edge\_list} | (u.\mathsf{Root} = \max_{x \in \mathsf{Edge\_list}} x.\mathsf{Root}) > v.\mathsf{Root}=v.\mathsf{Id}) \wedge \neg \mathsf{AlreadyAsking}$ $\rightarrow$ $v.\mathsf{Request} := v.\mathsf{From} := v.\mathsf{Id}$ $v.\mathsf{To} := u.\mathsf{Id}$ $v.\mathsf{Direction} := \mathsf{Ask}$

/*make a request*/

3. $st \wedge \neg rqst'$ $\rightarrow$ $v.\mathsf{Request} := v.\mathsf{From} := v.\mathsf{To} := v.\mathsf{Direction} := \perp$

/*make $rqst'$ true*/

4. $st \wedge rqst' \wedge \neg rqst \wedge (\exists w \in v.\mathsf{Edge\_list} | (w.\mathsf{Direction} = \mathsf{Ask}) \wedge (w.\mathsf{To} = v.\mathsf{Id}) \wedge (w.\mathsf{Request} = w.\mathsf{Id} = w.\mathsf{Root} = w.\mathsf{From}) \wedge (v.\mathsf{Parent.From} \neq v.id))$ $\rightarrow$ $v.\mathsf{Request} := v.\mathsf{From} := w.\mathsf{Id}$ $v.\mathsf{To} := v.\mathsf{Parent}$ $v.\mathsf{Direction} := \mathsf{Ask}$

/*Forward a request from a neighbor */

5. $st \wedge rqst' \wedge \neg rqst \wedge (\exists w \in v.\mathsf{Edge\_list} | (w.\mathsf{Parent} = v.\mathsf{Id}) \wedge (w.\mathsf{To} = v.\mathsf{Id}) \wedge (w.\mathsf{Direction} = \mathsf{Ask}) \wedge (w.\mathsf{Request} \neq \perp) \wedge (w.\mathsf{Request} \neq w.\mathsf{Id}) \wedge (v.\mathsf{Parent.From} \neq v.\mathsf{Id}))$ $\rightarrow$ $v.\mathsf{Request} := w.\mathsf{Request}$ $v.\mathsf{From} := w.\mathsf{Id}$ $v.\mathsf{To} := v.\mathsf{Parent}$ $v.\mathsf{Direction} := \mathsf{Ask}$

/*Forward a request from a child */

6. $st \wedge rqst \wedge (v\ is\ a\ root) \wedge (v.\mathsf{Direction} = \mathsf{Ask})$ $\rightarrow$ $v.\mathsf{Direction} := \mathsf{Grant}$

/*grant a request */

7. $st \wedge rqst \wedge (v.\mathsf{To} = v.\mathsf{Parent} = u.\mathsf{Id}) \wedge (u.\mathsf{Direction} = \mathsf{Grant}) \wedge (v.\mathsf{Direction} = \mathsf{Ask}) \wedge (u.\mathsf{Request} = v.\mathsf{Request}) \wedge (u.\mathsf{From} = v.\mathsf{Id})$ $\rightarrow$ $v.\mathsf{Direction} := \mathsf{Grant}$

/*forward a grant*/

8. $frst \wedge \neg st \wedge (v.\mathsf{Direction} = \mathsf{Ask}) \wedge (u.\mathsf{Id} \in v.\mathsf{Edge\_list}) \wedge (v.\mathsf{Request} = u.\mathsf{Request} = v.\mathsf{From} = v.\mathsf{Id} = v.\mathsf{Root}) \wedge (u.\mathsf{From} = v.\mathsf{Id}) \wedge (u.\mathsf{Direction} = \mathsf{Grant}) \wedge (v.\mathsf{To} = u.\mathsf{Id}) \wedge (u.\mathsf{Root} > v.\mathsf{Root})$ $\rightarrow$ $v.\mathsf{Parent} := u.\mathsf{Id}$ $v.\mathsf{Distance} := u.\mathsf{Distance} + 1$ $v.\mathsf{Root} := u.\mathsf{Root}$ $v.\mathsf{Request} := v.\mathsf{From} := v.\mathsf{To} := v.\mathsf{Direction} := \perp$

/*join*/

$v.\mathsf{Toggle} := v.\mathsf{Toggle} + 1 \bmod 3$;

end {of the while (true)}

Figure 1: The code of the spanning tree algorithm at node v.

that it sets its v.Request and v.From to its own Id, v.Direction to Ask and v.To to u.

Of course, node v issues such a request only if it is not currently waiting for the answer of a similar request. That is, assume that v has a neighbor w whose Root, like that of u, is the largest among v's neighbors (i.e., is the same as u's Root). We would not like v to issue a request to u, then change it to w, then change it back to u, and so on and so forth. Thus Operation 2 is also conditioned on the negation of the following predicate:

Predicate AlreadyAsking:
$(\exists w \in v.\mathsf{Edge\_list} | \ (w.\mathsf{Root} = \max_{x \in \mathsf{Edge\_list}} x.\mathsf{Root}) > v.\mathsf{Root}) \ \wedge (v.\mathsf{Request} = v.\mathsf{From} = v.\mathsf{Id}) \wedge$
$(v.\mathsf{To} = w.\mathsf{Id}) \wedge (v.\mathsf{Direction} = \mathsf{Ask})$

Condition st does not hold for a node that makes a request to join another tree. When Condition $st(v)$ does hold, node v participates in the process of forwarding requests and grants in its tree in order to enable the addition of new nodes to the tree. Its task is to help forwarding requests (asking to join the tree) to the root of the tree and grants (allowing the joining) from the root back to the requesting node (Actions $4-7$).

Similar to the tree related variables, we define Condition $rqst$ for local checking of the legality of the issuing and forwarding of the requests. Condition $rqst$ is true if and only if the variables (at that node and its neighbors) related to the task of forwarding requests and grants are in a legal state. If $rqst(v)$ does not hold, but Condition $st(v)$ is true, then the process at node v resets the variables related to forwarding a request (Action 3). As with the tree related variables, we define a local condition $rqst'$ (that starts to hold when Action 3 is performed) which captures intermediate local states of nodes that have finished the participation in handling one request and can now start participation in handling another request. Such a condition is necessary to move a process from handling one request to handling another, in an orderly manner.

Condition $rqst$ has two terms capturing the states in which node v handles a request from either a child in its tree (started by Operation 5) or from a neighbor that is requesting to join the tree (started by Operation 4).

Condition $rqst(v)$**:**
$[(w.\mathsf{Id} \in v.\mathsf{Edge\_list}) \wedge (w.\mathsf{Parent} = w.\mathsf{Id} \neq v.\mathsf{Id}) \wedge (w.\mathsf{Request} = w.\mathsf{From} = v.\mathsf{Request} =$
$v.\mathsf{From} = w.\mathsf{Id} = w.\mathsf{Root}) \wedge (w.\mathsf{To} = v.\mathsf{Id}) \wedge (w.\mathsf{Direction} = \mathsf{Ask}) \wedge (v.\mathsf{To} = v.\mathsf{Parent})]$
$\bigvee$
$[(w.\mathsf{Id} \in v.\mathsf{Edge\_list}) \wedge (w.\mathsf{Parent} = v.\mathsf{Id}) \wedge (\perp \neq w.\mathsf{Request} = v.\mathsf{Request} \neq w.\mathsf{Id}) \wedge$
$(v.\mathsf{From} = w.\mathsf{Id}) \wedge (v.\mathsf{To} = v.\mathsf{Parent}) \wedge (w.\mathsf{To} = v.\mathsf{Id}) \wedge (w.\mathsf{Direction} = \mathsf{Ask})]$

Condition $rqst'$ holds if either Condition $rqst$ holds or if the related variables are in a predefined reset state ($= \perp$).

Condition $rqst'(v)$**:**
$rqst(v) \bigvee (v.\mathsf{Request} = v.\mathsf{To} = v.\mathsf{From} = v.\mathsf{Direction} = \perp)$

When $st(v)$ is true and $rqst(v)$ is false, the process at node v resets the corresponding variables to $\perp$, thus satisfying Condition $rqst'(v)$. As is the case with Conditions st and $frst$, note that Condition $rqst'$ implies Condition $rqst$, but the converse is not true.

Let us make a further comment on Operations 4 and 5: a pre-condition for each of these operations is that the parent is not currently forwarding any request from v (See the last term in the guards of Operations 4 and 5.) This is a crucial point for the proof of correctness.

If node v handles a request (i.e., $rqst$ holds at v) and it is a root, it can *grant* the request (Action 6). That is, it sets its Direction to Grant. A non-root node which is forwarding a request, can forward a grant, provided that its parent satisfies the following conditions (Action 7):

1. The node is forwarding the same request (i.e., the parent and v have equal values in their Request variable); and
2. it has received the request from v (i.e., the parent's From variable is v's Id); and
3. node v has sent the request to its parent (i.e., v's To is the Id of its Parent); and
4. the parent is forwarding the grant (i.e., the Direction in the Parent is Grant).

Note that Condition $rqst$ holds at node v as long as it handles the request of its neighbor u and u.Direction $= Ask$. As soon as node u changes its direction to Grant, Condition $rqst(v)$ is falsified (until it starts handling another request). In this case node v first resets its request related variables thus satisfying Condition $rqst'$ (Action 3). A node whose request has been granted (Action 8) joins the tree by setting its Root to the tree root, its Parent to its neighbor from which it read the grant and its Distance to be one more than that of its parent. In addition it resets its request variables to $\perp$.

4 Correctness

In this section we prove that the code of Figure 1 is a self stabilizing implementation of a spanning tree algorithm in a dynamic network environment. That is, if at some time t_0 faults and topological changes cease, then eventually the parent pointers define a spanning tree at each connected component of the network. Henceforth, we assume without loss of generality that after time t_0 the network of interest consists of one connected component. All the claims in the sequel can be proved for each connected component separately.

Before dwelling on the details let us describe the structure of the proof. The proof argues about runs of the system in which there are no failures or topological changes. Starting in state s_0 the system behavior is modeled by a run which is an infinite sequence $q_0\pi_0q_1\pi_1\ldots$ of alternating states and atomic operations, such that $q_0 = s_0$. Each atomic operation π_i is either a local step, or a read of shared-memory, or a write of a shared-memory, of one processor in the system. Each state includes a complete description of all the variables in all the processors of the system. State q_{i+1} is the state of the system after applying operation π_i to state q_i. The sequence keeps the causal order of an actual execution which has concurrent actions of processors which are far away from each other (whereas neighboring processors take atomic actions regarding shared variables one at a time).

We assume a *fairness* assumption that every non-faulty processor is scheduled to take a step infinitely often in the system.

Assume that following the last failure or topological change the system is placed in an arbitrary global state, s_0. We proceed by proving that following global state s_0 the system must progress through a sequence of global states that contains a subsequence $s_1, s_2, ... s_7$ such that in each of these states an additional global and *stable* property holds until in state, s_7, the desired spanning tree property stably holds. The set of stable properties starting at the points $s_0, s_1, s_2, s_3, \ldots s_7$ and holding thereafter in the suffix subsequence starting at these points, are correspondingly as follows:

1. By assumption, there are no faults or topological changes in any suffix of a run starting with s_0.

2. Starting with global state s_1: the internal copies and the echo copies that each node has of its neighbors shared variables, hold an actual value read from these shared memory variables in some step which has occured after s_0.

3. Starting from global state s_2: each node has read, at least once after s_1, its echo variable from its neighbors and has found them equal to its own Toggle. That is, each neighbor u of every node v read v's variables into u's internal variables, updated $u(v.\mathsf{Toggle})$, and v has later read the echo variable of u and found that it is equal to its own Toggle variable.

4. Starting from global state s_3: for any node v in any state that immediately precedes the change of state by node v, all the internal copies of v's variables at its neighbors have the same value as in v. (see Lemma 4.2). That is, at the time that v changes the value of any of its state variables from *old* to *new*, the value of the internal copy of this variable in each of its neighbors is *old*.

5. Starting with global state s_4: all the Id type variables (e.g. Id, Root, Parent, From, and To) in the system hold identities of actual nodes in the network, i.e., there are no false Ids in the network.

6. Starting with global state s_5: the largest identity in the network, r, is the root of at least node r.

7. Starting with global state s_6: there is no cycle of parent pointers among nodes whose Root variable equals r.

8. Starting with global state s_7: the parent pointers of nodes whose root is r constitute a spanning tree of the network. That is, all the nodes are included in the tree of r.

In global state s_0 the adversary may place arbitrary values in the memory of some processors, and the topology has stabilized (the edge lists reflect the actual topology). The second and third stable properties above capture the fact that starting from any global state s_0 a state s_2 is eventually reached in which the copies (both internal copies, and echo variables) of all the values used in the guards evaluation or in commands execution were actually read from registers in the shared-memory. This means that no new "fake" (made up by the adversary) values can be introduced by the algorithm, however old fake values may be still propagating around.

The fourth stable property above is used to prove the crucial property, that the number of such fake values decreases.

Let us start arguing the correctness:

Lemma 4.1 *In any run α of the system that starts in state s_0 (as defined above) there are states s_1, s_2 such that $\alpha = s_0\beta s_1\gamma s_2\delta$ and*

1. *The following stable property holds in any state q in $s_1\gamma s_2\delta$:* for every node, the values in the internal and in the echo variables of the node's shared-memory were read from the neighbors memories in a read operation in α.
2. *The following stable property holds in any state q in $s_2\delta$:* for every node v, the value v last read from each neighbor u's echo variable, i.e., $u(v.\mathsf{Toggle})$, is a value that was read by u after s_1. *(Notice the handshake implied in the second part of the lemma: operations of both v and u are referred to.)*

Proof: The lemma follows directly from the fairness assumption (each process being scheduled infinitely often) and from the fact that every processor infinitely often reads all the shared variables of its neighbors and updates the corresponding echo variables. ∎

Although all the values used in any computation in the guards and in the actions have been actually read from some shared-memory variable, there could be some old contaminated value moving around. The next part of the proof, established by the following lemmas, shows that these values are also doomed to disappear.

Lemma 4.2 *Let $s_2\alpha$ be a run of the system starting in state s_2 that satisfies the conditions of Lemma 4.1. Then, there is a state s_3 and a bounded length run fragment β such that $\alpha = \beta s_3\gamma$, and the following holds:*
If $q_j\pi_j q_{j+1}$ is in γ such that π_j is a change of state *variables of some process v then, $\forall$ u neighbor of v the following condition holds in state q_j:*
Neighbor Agreement:
$u(v.\mathsf{Toggle}) = v.\mathsf{Toggle}$ *and for every* state *variable* $v.\mathsf{Var}$ *of v the internal copy* $u(v.\mathsf{Var})$ *equals* $v.\mathsf{Var}$.

Proof: If processors stop changing their state variables a bounded number of steps after s_2, then the lemma trivially holds. Otherwise, let β be any run fragment in which each processor either stopped changing its state variables and all its neighbors agree with it on their value, or has changed its state variables at least three times.

The lemma follows from the correctness of an alternating bit protocol that uses a three values toggle bit over a bidirectional link whose total capacity (in both directions) is 2 ([AB89]). In the analogy, the echo variable is the acknowledgment and together with the internal copy at node u is the capacity of the link. This is the crux of the proof, but nevertheless, herein we give a complete proof.

Assume to the contrary that in state q_j there is a processor u neighbor of v such that $u(v.\mathsf{Toggle}) \neq v.\mathsf{Toggle}$. By the definition of β, v has changed its state at least three times in β. By the code, the following subsequence of steps, taken by v, is repeated three times before π_j: (1) v reads the shared memory of all its neighbors, including u, (2) v assigns the values of the Toggles read to the corresponding echo variables, (3) v verifies that the echo of v's toggle, which it read in (1) from each of its neighbors (e.g. $u(v.\mathsf{Toggle})$), equals the actual value of $v.\mathsf{Toggle}$.

Assume that in step π_j node v changes its toggle from i to $(i-2) \bmod 3$, then it must have found the value of $u(v.\mathsf{Toggle})$ that it last read in $\pi_{v_i}^{read}$ (resulting in state $s_{v_i}^{read+}$) equal to $v.\mathsf{Toggle} = i$. Since the value of $v.\mathsf{Toggle}$ does not change between $s_{v_i}^{read+}$ and q_j, the following holds: $u(v.\mathsf{Toggle}) = v.\mathsf{Toggle}$ in state $s_{v_i}^{read+}$. (Otherwise node v would not have changed the contents of its state variables.) Thus the only way in which the lemma is falsified for $u(v.\mathsf{Toggle})$ in state q_j is if in state $s_{v_i}^{read+}$ node u holds a different value of $v.\mathsf{Toggle}$ in an internal variable. This internal value has not yet been posted in $u(v.\mathsf{Toggle})$ in $s_{v_i}^{read+}$ but is posted in q_j. Let this different value be z. Thus the following chronological subsequence of steps must have taken place:

Operation	value in $v.\mathsf{Toggle}$	value in u's internal copy of $v.\mathsf{Toggle}$	value in u's echo $u(v.\mathsf{Toggle})$
$\pi_{v_i}^{read}$: v reads $u(v.\mathsf{Toggle}) = i$	i	z	i
π_u: $u(v.\mathsf{Toggle}) := z$	i	z	$i \rightarrow z$
π_j: $v.\mathsf{Toggle}{:=}(i-2)$	$i \rightarrow (i-2)$		z

By the definition of β, v has changed its state variables at least three times after s_2 and before q_j. Let the last two changes of state variables in v that precede π_j be $\pi_{v_{i-2}}$, and $\pi_{v_{i-1}}$. Before each such state variables change v reads $u(v.\mathsf{Toggle})$ at least once. Let the last such read before each state variables change be $\pi_{v_{i-2}}^{read}$, and $\pi_{v_{i-1}}^{read}$ respectively. That is, we consider the following subsequence of events:

Operation	value in $v.\mathsf{Toggle}$	value in u's internal copy of $v.\mathsf{Toggle}$	value in u's echo $u(v.\mathsf{Toggle})$
$\pi_{v_{i-2}}^{read}$: v reads $u(v.\mathsf{Toggle}) = (i-2)$	$(i-2)$		$(i-2)$
$\pi_{v_{i-2}}$: $v.\mathsf{Toggle}{:=}(i-1)$	$(i-2) \rightarrow (i-1)$		
$\pi_{v_{i-1}}^{read}$: v reads $u(v.\mathsf{Toggle}) = (i-1)$	$(i-1)$		$(i-1)$
$\pi_{v_{i-1}}$: $v.\mathsf{Toggle}{:=}i$	$(i-1) \rightarrow i$		
$\pi_{v_i}^{read}$: v reads $u(v.\mathsf{Toggle}) = i$	i	z	i
π_u: $u(v.\mathsf{Toggle}) := z$	i	z	$i \rightarrow z$
π_j: $v.\mathsf{Toggle}{:=}(i-2)$	$i \rightarrow (i-2)$		z

Note that there are no other changes of state variables in node v in between $\pi_{v_{i-2}}$ and π_j.

Consider the last two times that u has read $v.\mathsf{Toggle}$ before $\pi_{v_i}^{read}$, denoted by $\pi_{u_i}^{read}$ and $\pi_{u_z}^{read}$. In the earlier one, $\pi_{u_i}^{read}$, u must have read $v.\mathsf{Toggle} = i$ and in the later one, $\pi_{u_z}^{read}$, u must have read $v.\mathsf{Toggle} = z$. Since $z \neq i$ we have the following fact:

F1: $\pi_{u_z}^{read}$ is before $\pi_{v_{i-1}}$ (because after that the value of $v.\mathsf{Toggle}$ is i).

Note that $\pi_{u_i}^{read}$ is before $\pi_{u_z}^{read}$. Thus

F2: $\pi^{read}_{u_i}$ must be before $\pi_{v_{i-2}}$ (because only before $\pi_{v_{i-2}}$ does $v.\mathsf{Toggle} = i$ and after it $v.\mathsf{Toggle} \neq i$ (until $\pi_{v_{i-1}}$); indeed $v.\mathsf{Toggle} = i$ also after $\pi_{v_{i-1}}$, but this is already after $\pi^{read}_{u_z}$, according to Fact F1, while $\pi^{read}_{u_i}$ is before $\pi^{read}_{u_z}$).

In the following table $\pi^{read}_{u_z}$ is placed in the latest place permitted according to Fact F1. Note that it can be placed in an earlier time.

Operation	value in $v.\mathsf{Toggle}$	value in u's internal copy of $v.\mathsf{Toggle}$	value in u's echo $u(v.\mathsf{Toggle})$
$\pi^{read}_{u_i}$: u reads $v.\mathsf{Toggle} = i$	i	i	
$\pi^{read}_{v_{i-2}}$: v reads $u(v.\mathsf{Toggle}) = (i-2)$	$(i-2)$		$(i-2)$
$\pi_{v_{i-2}}$: $v.\mathsf{Toggle}{:=}(i-1)$	$(i-2) \rightarrow (i-1)$		
$\pi^{read}_{v_{i-1}}$: v reads $u(v.\mathsf{Toggle}) = (i-1)$	$(i-1)$		$(i-1)$
$\pi^{read}_{u_z}$: u reads $v.\mathsf{Toggle} = z$	z	z	
$\pi_{v_{i-1}}$: $v.\mathsf{Toggle}{:=}i$	$(i-1) \rightarrow i$		
$\pi^{read}_{v_i}$: v reads $u(v.\mathsf{Toggle}) = i$	i	z	i
π_u: $u(v.\mathsf{Toggle}) := z$	i	z	$i \rightarrow z$
π_j: $v.\mathsf{Toggle}{:=}(i-2)$	$i \rightarrow (i-2)$	z	z

By Fact F1, in $\pi_{v_{i-1}}$, node u must be holding the values z (in an internal variable). By Fact F2, in $\pi_{v_{i-1}}$ and in $\pi^{read}_{v_{i-1}}$, node u must also hold (either in an internal variable or in the echo variable) the value i. However, at $\pi^{read}_{v_{i-1}}$ the echo variable at u has been equal to $(i-1)$. Since the echo variable is not copied to the internal one (while the internal variable may be copied to the echo variable) the value i must have been held in the internal variable at $\pi_{v_{i-1}}$. Thus, during $\pi_{v_{i-1}}$ the same internal variable at u holds two different values, which is impossible.

Thus in state q_j, $u(v.\mathsf{Toggle}) = v.\mathsf{Toggle}$ and this value has been read by u from $v.\mathsf{Toggle}$ after $\pi_{v_{i-1}}$. Hence the internal variables $u(v.\mathsf{Var})$ (read at the same time as $u(v.\mathsf{Toggle})$) equal $v.\mathsf{Var}$ in every state that follows $\pi^{read}_{v_i}$ up to and including q_j.

∎

Corollary 4.3 *Let π^u_r be the last time v read the variables of its neighbor u before step π_j that is after State s_3. Then Neighbor Agreement (Lemma 4.2) holds just after π^u_r as well.*

Proof: The corollary follows from the observation that v may take Step π_j immediately after Step π^u_r. Thus, if a case where the corollary does not hold exists, one could show a schedule for which Lemma 4.2 does not hold. ∎

The next three lemmas prove that eventually the following stable property holds: r_m, the largest identity in the network, is the largest Root value in the network.

Definition 4.4 *We define r* **branch***, r* **grant interval***, and r* **request interval** *in the system in a state s as follows:*

r **branch:** *a maximal path of nodes* $\{v_1, v_2, \ldots v_h\}$ *such that there exists a node* v_{h+1} *(possibly the same as* v_h*) and for each* i*,* $1 \le i \le h$ *the following holds:* v_i.Root$= r$*,* v_i.Parent $= v_{i+1}$.Id*, and Condition st holds at* v_i*.*

r **request interval:** *a path of nodes* $\{v_j, v_{j+1}, \ldots v_{j+l}\}$ *that is contained in an* r *branch such that for each* i*,* $0 \le i \le l$ *the following holds:* v_{j+i}.Request$= p$ *for some identity* $p \neq \perp$*.*

r **grant interval:** *(A) a maximal* r *request interval* $\{v_j, v_{j+1}, \ldots v_{j+l}\}$ *such that there exists* $j \le k \le j+l$ *such that* either *(A.1) each node* v_i $i \ge k$ *has its* v_i.Direction $=$ Grant *and if* $k > j$ *then* v_k.From $= v_{k-1}$.Id, or *such that (A.2)* $v_{j+l}(v_{j+l}$.Parent.Direction$)=$Grant*, and* $v_{j+l}(v_{j+l}$.Parent.From$)=v_{j+l}$.Id*. (B) Any subpath that is included in a grant interval is also a grant interval.*

The child end of an interval or a branch: *a prefix of the interval or the branch.*

The parent end of an interval or a branch: *a suffix of the interval or the branch.*

Note that in one branch one can place several grant intervals. Note also that there are two ways for a part of a request interval to be a maximal grant interval. One is by having its last nodes (i.e., the parent side) forward a grant. However, since an atomic operation is a single read, it may happen that when a node v_i reads its parent's Direction register the value read is Grant, and then the parent has changed the contents of its Direction before the child has made its next move. Since the child may still act based on the old value, we still call this interval a grant interval. (The internal variable may also contain the value Grant because of an initial assignment by the adversary.)

We now consider a major obstacle with which a self stabilizing algorithm needs to cope. This is the notion of false values that are allowed in a self-stabilizing systems.

Definition 4.5 *A value* r_f *is a* false root *in state* s *if in state* s *there does not exist a node* u *such that* u.Id $= r_f$ *but there exists a node* v*, such that either*

1. v.Root $= r_f$*; or*

2. *some other node* w *has an internal copy* $w(v$.Root$) = r_f$*.*

The next definition deals with a certain kind of a false tree, for which it is easier to prove that it eventually disappears.

Definition 4.6 *A value* f *is a* false-tree *in state* s *if in state* s *there exists a node* v *such that* v.Root $= f$ *and there does not exist an* f *branch that includes* v*, and that* f *is its last node (at the parent end).*

If node v *above belongs to an* f *branch (or an* f *request interval, or an* f *grant interval) then this* f *branch (or an* f *request interval, or an* f *grant interval) is said to be an* f branch of the false tree of f *(or an* r *request interval of the false tree of* f*, or an* r grant interval of the false tree of f*).*

Lemma 4.7 *Consider state s_1 satisfying lemma 4.1 and a value r_x, then if r_x is not a false root in s_1 than it cannot be a false root in any state in any run $s_1\alpha$.*

Proof: Follows from the fact that only values that are copied from other variables or internal variables are assigned to identity variables (e.g. Root). ∎

Our goal is to prove that eventually there are no false roots or false-trees, and that r_m, the largest identity in the network, becomes the root of itself and of all the other nodes. A major obstacle in proving this property is that there might be erroneous grant intervals in the network that give nodes the permission to join a false-tree. That is, if the r branch were not able to expand into new nodes then the proof would be rather simple since the parent end of these erroneous r branches would be continuously eroded (by Operation 1). This holds since Condition st does not hold at the node at the parent end. However, grant intervals have the property that they may enable the addition of new nodes to their r *branches* at the child end of the interval. Thus, we first have to prove that the number of grant intervals of false roots does not grow (even though the number of such r *branches* may grow!) and that eventually the false roots grant intervals disappear. The following lemmas establish this fact; that is, eventually there are no r grant intervals for any false root. First, Lemma 4.8 shows that no new grant interval is formed for a false root. This is the main lemma we use in the proof of correctness.

Lemma 4.8 *Let $s_0\alpha s_3\beta q_j\pi_j q_{j+1}\gamma$ be any run that starts from s_0, where π_j is an operation (and q_j, q_{j+1} are states), and state s_3 satisfies Lemma 4.2. Then any f grant interval of a false root f that exists in state q_{j+1} existed also in q_j.*

Proof:

We consider all possible operations π_j. Each π_j taken by some node v may be either one of the following:

1. a copy by v of a variable of a neighbor of v into an internal variable, or
2. the assignment of the internal copy of a Toggle variable into an echo variable, or
3. an atomic execution of one guarded command in the code of Figure 1 using the values in v's internal copies to evaluate the guards and compute the values to be assigned. This includes the check that v's Toggle equals $u(v.\mathsf{Toggle})$ for every neighbor u.

For each possible operation we show that if it results in a grant interval in state q_{j+1} then this grant interval must have existed also in state q_j.

π_j **is a read from a neighbor** u: The change of an internal variable at v may effect the existence of a grant interval by either causing Condition $st(v)$ to become true thus making v a part of a *branch* and a grant interval, or by causing Condition (A.2) in definition 4.4 to be true. In the later case, if v is a part of a grant interval by Condition (A.2) of Definition 4.4 after π_j, then v has been a part of this grant interval before π_j by Condition (A.1).

In the former sub-case, Condition $st(v)$ does not hold before the read, but holds after the read. Since π_j is after s_2, node v has no parent since v performed the operation π_j while $st(v)$ was not true. Thus, either $frst(v)$ held (and v did not have a parent different from itself, or $frst(v)$ did not hold either, and thus the only permitted operation is 1, after which v's parent is itself.

We have established that if $st(v)$ starts to hold after π_j then v is a root. By the definition of Condition $st(v)$ we know that $v.\mathsf{Root} = v.\mathsf{Id}$. Since v does exist, it does not belong to a grant interval of a false root of v.

π_j is a copy of an internal copy of a neighbor Toggle variable into an echo variable: This operation does not affect variables that are used in the definition of grant intervals (Definition 4.4).

π_j is the execution of Operation 1: This can cause grant intervals to cease to exist, but not vice versa.

π_j is the execution of Operation 2: Since v has no parent, and since it sets $v.\mathsf{Request}$ to Ask (and not to Grant) neither Condition (A.1) nor Condition (A.2) of Definition 4.4 can become true (and part (B) of that definition does not hold either).

π_j is the execution of Operation 3: This case is similar to that of Operation 1.

π_j is the execution of Operation 4: If v is not a part of a grant interval in state q_{j+1} then the lemma holds. Similarly, the lemma holds trivially for each grant interval that node v is a part of in both states q_j and in q_{j+1}. Thus, assume that there is a grant interval that includes v in state q_{j+1}, but did not include v in state q_j.

Note that in Operation 4 node v assigns the value Ask (and not Grant) to its Direction variable. Thus, for v to be a part of a grant interval in state q_{j+1} either (1) $v(v.\mathsf{Parent.From}) = v.\mathsf{Id}$ (a grant interval according to Condition (A.2) of definition 4.4), or (2) $v.\mathsf{Parent.From} = v.\mathsf{Id}$ and $v.\mathsf{Parent.Request}{=}v.\mathsf{Request}$ (a grant interval according to Condition (A.1) of Definition 4.4; see the definition of a request interval for the value of $v.\mathsf{Parent.Request}$ in this case).

If (1) holds then v does not perform Operation 4. Thus, it remains to show that neither does (2) hold by proving that in state q_{j+1} either $v.\mathsf{Parent.From} \neq v.\mathsf{Id}$ or $v.\mathsf{Parent.Request} \neq v.\mathsf{Request}$. To prove this, let us assume the converse.

Let w be the value of $v.\mathsf{Parent}$ in state q_j. By the definition of s_3 (as being after s_2), all the internal copies in v of the form $v(w.\mathsf{Var})$ in state q_j were actually read from node w. Let π_r (starting in state s_r and resulting in state s_{r+1}) be the last such read operation in $\alpha s_3 \beta$ where v copies the variables of $v.\mathsf{Parent}$ before state q_j. By the algorithm (and the definition of π_r) the current operation π_j is the first action (among operations 1-8) v takes after π_r. This means that no state variable of v (i.e., of the form $v.\mathsf{Var}$) changes its value between π_r and q_j. (internal and echo variables of neighbors other than $v.\mathsf{Parent}$ in v may have changed their values.) In particular, $v.\mathsf{Parent}$ has the same value w in state s_r and state q_{j+1}, as well as in all the states that are between these two in the run. Thus, we may speak of w and of the value of $v.\mathsf{Parent}$ interchangeably, without specifying the state.

Since π_r is the last time v read w before π_j, we know that $w.\mathsf{From} \neq v.\mathsf{Id}$ at π_r, otherwise π_j could not have been Operation 4. However, we assumed that at q_j the value of $w.\mathsf{From}$ is $v.\mathsf{Id}$. Thus there exists an operation π_w (that is later than π_r but earlier than π_j) where w assigned the value $v.\mathsf{Id}$ to the variable $w.\mathsf{From}$.

On the other hand, recall that by the definition of s_3 (and by Corollary 4.3) the values of w's internal copies (of v's state variables) at π_r are the values of v's state variables at that time. In particular, the value of $w(v.\mathsf{Request})$ at π_r is the value of $v.\mathsf{Request}$ at that time. Since v does not change that value until π_j, then, by the guard of Operation 4 the value of $v.\mathsf{Request}$ and thus of $w(v.\mathsf{Request})$ is $\perp$. (Otherwise v does not perform Operation 4: note that for $frst$ to hold while st does not, the following must hold: $v.\mathsf{Request} = \perp$.)

We claim that the value of $w(v.\mathsf{Request})$ is still $\perp$ at π_w. This is clearly true if w does not read v's variables again between π_r and π_w. Recall again that the values of v's state variables do not change between these two operations. Thus, even if w does read v's variables again in this interval, it still finds the value $\perp$ in $v.\mathsf{Request}$.

We established that w assigns the value $v.\mathsf{Id}$ to $w.\mathsf{From}$ at π_w although the value of $w(v.\mathsf{Request})$ is $\perp$ at that time. This contradicts the guards in the algorithm that control the assignment of values to $w.\mathsf{Request}$.

π_j **is the execution of Operation 5:** This case is similar to that of Operation 4.

π_j **is the execution of Operation 6:** If this operation takes place, then v does not belong to a grant interval of a false tree. (In q_{j+1} the following holds: $v.\mathsf{Root}$=$v.\mathsf{Id}$.)

π_j **is the execution of Operation 7:** This case is similar to the case of a read operation from a parent.

π_j **is the execution of Operation 8:** Note that in Operation 8 node v sets $v.\mathsf{Request}$ to $\perp$. Thus, v is not a part of a request interval.

∎

Lemma 4.9 *Let r_m be the largest node* Id *in the network in state s_1. Then in any sufficiently long run $s_1\alpha s_4$ the cardinality of the set of false roots $\{\tilde{r}|\tilde{r} > r_m\}$ is monotonically decreasing until a state s_4 is reached such that at any state in any run $s_4\beta$ there are no false roots greater than r_m.*

Proof: Let $r_1 > r_2 > \ldots > r_h$, be the list of false root Ids that are larger than r_m at state s_1, and let $s_1\alpha$ be any infinite run starting from s_1. We show that there is a sequence of states $s_4^1, s_4^2, \ldots, s_4^h = s_4$ that is a subsequence of α and in the suffix of α starting at state s_4^i, $1 \leq i \leq h$, no Root variable has the value r_i. By Lemma 4.7 if there are false roots that are larger than r_m then r_1 is well defined. The number of r_1 grant intervals in state s_1 is bounded, and by Lemma 4.8 any r_1 grant interval in a later state, existed already in s_1. Since r_1 is the largest value in the network, a node that changes its Root to r_1 because of an r_1 grant interval, would change its Root from r_1 to some other value only if the first node v (on the child side) on that grant interval has as well changed its Root. This means that all the grant intervals that start (on the child side) with v can be accounted (together) only once for an extension of a branch by one node. Consider an r_1 branch with l nodes in state s_3, it, therefore, can extend into at most l new nodes in α (actually, a more careful argument shows that it can extend into at most 1 new node).

On the other hand, at the parent of the r_1 branch there must be a node w such that w.Parent.Id $\neq r_1$, since node r_1 does not exist. Node w certainly resets its Root variable in a finite number of steps after s_1 (in at most 2 loops of node w through the algorithm), since once w reads the registers of its neighbors neither Condition st nor $frst$ will hold at w.

All together, in a bounded number of steps taken after the network is in state s_1, all the nodes along any r_1 branch similarly reset their Root variable. Let the state reached at that point be s_4^1. At that state all r_2 grant intervals may have already disappeared too. Let r_{i_2} be the largest remaining false root. The above argument (used above to show that r_1 eventually disappears) also shows that eventually r_{i_2} disappears. This is inductively repeated until no more false roots remain. ∎

Corollary 4.10 *Let r_m be the node with the largest* Id *in the network. Then eventually a state s_5 is reached where r_m is a root and there is no larger* Root *variable in the network.*

Lemma 4.11 *Let r_m be the highest* Id *of a node in the network. Eventually there are no r branches for the false tree of r_m.*

Proof: Consider the suffix of the run where r is the highest identity. Nodes cannot leave the correct tree in this suffix. Thus, a part of the correct tree cannot be disconnected and become a false tree of r. The rest of the proof is similar to the proof of Lemma 4.9. ∎

Let us now prove that a tree rooted at r_m eventually spans the network.

Lemma 4.12 *For any node v_i in an r_m branch $\{v_1, v_2, ..., v_h\}$ with v_h.*Id $= r_m$ *and v_h.*Distance*=0, the* Root*,* Parent *and the* Distance *variables do no change in any run α starting from s_5*

Proof: Straightforward from the code and the definition of s_5. ∎

Lemma 4.13 *Eventually a state s_6 is reached such that there are no cycles in any r_m branch.*

Proof: By way of contradiction. Consider a run α that starts from state s_5. If there is a cycle then either it is a cycle in state s_5 or it is created in some state in α after s_5. If there is a cyclic branch in state s_5, then in a bounded number of steps at least one node, v, on the cycle, will observe that its v.Distance+1 $\neq v$.Parent.Distance. Thus, node v will reset its variables and will break the cycle. So the cycle must be created after the network is in state s_5.

By the definition of an r branch each node on it has at most one parent. Thus the only way to create a cycle is for the last node on it (the one with no parent) to adopt another node on the branch as a parent, while maintaining the identity r as the Root. This is not possible by the definition of s_5 and by the part of the code that selects a parent. ∎

Definition 4.14 *A* correct tree *is a tree defined by the* Parent *relation, such that its root r is the node with the largest identity, and for every node v in the tree Condition st holds and the value of the internal variables in v (that are used for computing Condition st) is the same as the value of their original* state *variables.*

Lemma 4.15 *A correct tree exists in the network in every state of any run that starts from some state s_6.*

Proof: Follows from Corollary 4.10 and Lemma 4.13. ∎

Lemma 4.16 *In any infinite run $s_6\alpha$ there exists a state s'_6 in which every node that is not in a correct tree, but is a neighbor of a node in a correct tree (if such exists), has its* Request *equal to r_m (the largest node* Id *in the network), its* Direction *to ask, and its* To *to its neighbor (which is in the correct tree).*

Proof: Consider such a node v after reaching s_6 and after the condition described in Lemma 4.11 starts to hold (i.e., no more false trees). If it is not a root then either its Root is that of the correct tree, or its Root is smaller than r_m. In the second case clearly Condition st does not hold at v. (At least it stops to be true when v reads the variables of its neighbors). In the first case, by Lemma 4.11 if Condition st does hold for it, then by the fairness assumption it will eventually join the correct tree by reading its neighbors. Thus Condition st eventually does not hold for it in both cases, and it must become a root. Then, the algorithm dictates that it makes a request to join the neighboring tree with the largest Root value, which by the assumption is the correct tree. ∎

Definition 4.17 *An* Id $v \neq \perp$ *in the* Request *variable of a node u in the correct tree is called a* correct request *if*

- *node u is in a request interval that starts at a node w (in the correct tree) that is a neighbor of node v, and*
- *v's request variable is equal to v, and*
- *v's* Direction *variable contains an* Ask*, and*
- *v is a root, and*
- v.To $= w$.

Lemma 4.18 *In any infinite run $s'_6\alpha$ there exists a state s''_6 such that $\alpha = \beta s''_6\gamma$ and the following holds for every state in γ: all the* Ids *in the* Request *registers of the nodes in the correct tree are correct requests.*

Proof: Consider an Id p in a Request register of a node v in the correct tree in state s'_6 that is not a correct request, and consider the request interval to which this node belongs.

First consider the case that in every two consecutive states s_j, s_{j+1} in α the interval in s_{j+1} does not contain any node not belonging to the interval in s_j. (More formally, this is the case where v belongs to request intervals of some incorrect request p both in s_j and in s_{j+1} and the interval in s_{j+1} does not include a node that does not belong to the interval in state s_j; Since there exists an obvious one to one mapping between these two intervals we treat them henceforth as one interval that changes in time.) Clearly, Condition *rqst* does not hold at the first node on the child end of this interval. Thus that node will reset all the variables that are related to the request (operation 3). Note that the request is copied only from a child to its parent, and not vice versa. Thus, a repeated procedure of reset will eventually cause this interval to disappear.

Now consider the case that the interval expands to new nodes. By Lemma 4.12 and Lemma 4.13 and the fact that the interval may expand only from a node to its parent the request interval can expand only into a finite number of nodes. Now consider a state after which that interval no longer expands. The argument for the previous case now shows that this request interval eventually disappears in this case too. ∎

Lemma 4.19 *If in the suffix of a run after s_6 the correct tree does not span the entire network then in bounded number of steps the tree size will grow.*

Proof: By Lemma 4.16 every neighbor v of the correct tree will eventually set its request variables (Action 4) requesting to join the correct tree. By Lemma 4.18 incorrect requests will disappear making way for correct requests. By Actions 4 and 5 in the code a request interval will expand from a requesting node v over a path of parent links in the direction of the root, and eventually (by Lemma 4.12) there must be a branch from some node v not on the correct tree to the root of the correct tree. Since this is a correct request, then by Actions 6 and 7 in the code this will eventually cause a grant interval to reach v, and v will join the correct tree. ∎

Lemma 4.20 *Eventually a correct tree spans the network.*

Proof: Follows from Lemma 4.19 and from Lemma 4.12. ∎

Theorem 4.21 *In a bounded amount of time after s_0 the following conditions hold:*

(Convergence) *the network is spanned by a correct tree, and*

(Termination) *Condition st is true, and remains so, at all the nodes.*

Proof: Follows from Lemmas 4.20, 4.18 and 4.12. ∎

5 Applications as Modular Extensions

One motivation for the construction of the above algorithm is the possibility to use it as a modular component in other algorithms. Exact composition of self-stabilizing tasks is described in [Sto93] (where composition based on combining protocols which use independent variables is put forth); this method can be employed to construct algorithms on top of the spanning tree procedure in a modular way, as will be shown below.

One example is solving the famous token passing problem: Once a spanning-tree protocol is constructed we can achieve mutual exclusion by token passing along a virtual ring embedded in a DFS traversal on the tree. (This idea was independently suggested in [DIM94].) The passing of the token on the virtual ring can use some known self stabilizing ring token passing algorithm, e.g., [Dij74, AB89].

We can further use our tree to have a self stabilizing reset procedure, i.e., a procedure that translates algorithms designed for static networks to run correctly over dynamic (changing topology) networks [AAG87, ACK90]. Given a self stabilizing spanning tree algorithm, the construction of a self stabilizing reset algorithm is simple.

Katz and Perry [KP94] suggested a novel self stabilizing general protocol extensions based on a snapshot collected at a leader. A generalization of their self-stabilizing snapshot can be achieved for the case that no leader is known in advance, which implies in turn that general protocols can be self-stabilized and even on a dynamic network. In addition, a spanning tree can also be used to improve the complexity of the procedure in [KP94].

Let us present here one extension of our algorithm in more details; this extension may be of interest by itself. This is the task of breaking symmetry and constructing a rooted spanning tree in an anonymous network. In fact, after this extension was suggested in the proceedings version of this paper, several algorithms to perform this task were developed, e.g. [AEYH92, A94]). Another modular construction was independently used in [DIM91]. Henceforth, we assume that nodes do not have unique identities and hence need to rely on randomization to break symmetry.

The main idea of the randomized extension is as follows: We would like to run the algorithm described in the previous section. In order to supply the anonymous nodes with IDs we give each $2\log n$ bits, which are randomly flipped to select a random ID as in e.g., [AM94, SS94], where n is now a bound on the number of nodes. Even when chosen at random (let alone by the adversary) it is still possible (with exponentially small probability) that the largest ID selected is not unique, i.e., is selected by more than one node. Therefore, we add a self-stabilizing procedure that runs on top of our self-stabilizing spanning tree algorithm and which guarantees to eventually (exponentially fast) detect the case in which there are two or more trees with the same highest ID value. If such is detected then the algorithm starts all over by each node selecting a new random ID, and performing Operation 1 in our code. In what follows we outline the procedure that detects the case in which there are two or more trees with the same highest ID. Note that node v selects a new ID if either condition $frst$ does not hold at v or if condition st holds at v and v is a root that detected the existence of another neighboring tree with the same identity.

The verification of uniqueness of the root identity is carried out by each root repeatedly "coloring" with a random color, and "uncoloring" its tree. If one of its descendants observes a neighboring node that was believed to belong to the same tree but is colored with a different color, then the alarm is set since a collision of ID'S is detected. On the other hand, if all the nodes of the tree always observe their neighbors colored with the same color as they are colored with, then with a probability that approaches 1 exponentially fast, this tree, and its root identity, are unique.

For the sake of completeness, let us elaborate more on the method by which a tree checks whether some other neighboring tree root has the same identity. Each tree root periodically initiates a "coloring" phase, in which every node in the tree is "colored" by a color taken at random from the set $\{0,1\}$. The coloring is then used by neighboring nodes, w and v, to detect whether they belong to the same tree, in which case they both have the same color in every coloring phase. If each of w and v belongs to a different tree, but with the same tree root identity, then with probability 1 (in a process that converges exponentially fast) w and v are colored with different colors at some point in time, since the roots repeat this random procedure to infinity.

The coloring is performed by a self-stabilizing broadcast and echo on the tree (see [Seg83, Cha79]). Every two phases of coloring are separated by a phase of broadcast and echo of a "reset" color, called NoColor. The "no-coloring" step ensures that nodes compare colors of the same phase. Otherwise, because of the asynchronous nature of the network, it could have happened that one node, v, in a tree is still colored by one color, c, while another node, w, in the same tree is already colored by a "new" color that may be different than c. Nodes w and v could have (in this case) considered themselves belonging to two different trees with the same root identity. We prevent that by using the NoColor phase, mentioned above.

The implementation of the coloring phases builds upon the same self-stabilizing techniques as we have developed in the spanning tree algorithm of Section 3. For that implementation each node v has the following additional variables whose usage is explained in the sequel: v.Color, v.Broadcast, v.collision, and v.Compare(u) for each neighbor u. A root r starts a coloring phase whenever Condition st becomes true at r, or when Condition st holds and its previous coloring terminated, that is the variable r.Broadcast=Echo. Furthermore, its previous coloring had to terminate successfully, i.e., with variable r.collision = False (no different tree with the same identity was found).

When starting a coloring phase root r first selects the new color as follows: If it is colored, (r.Color $\neq$ NoColor), then r needs to reset the tree color by assigning NoColor to r.Color. Otherwise, (r.Color = NoColor) node r assigns a random bit to r.Color. Then r assigns Wave to r.Broadcast to signal that the value of r.Color should be adopted by r's children. When any node adopts a new color it also resets r.Compare(x) to Undefined for every neighbor x, and it reset r.collision to False. This will enable the comparison of each node color and the colors of its neighbors.

Consider a node v such that $st(v) = true$, and whose parent w is broadcasting color c. One legal state for v is to be broadcasting color c. Another is to be *echoing* color c. That is: v.Color = c = w.Color, v.Broadcast = Echo. However, for the latter state to be legal each child u of v (if such exists) must be echoing c as well, i.e., u.Color = c and

u.Broadcast = Echo. If v is neither broadcasting color c nor echoing this color when it notices that its parent is broadcasting c then v starts broadcasting c, resets v.Compare(x) to Undefined for every neighbor x, and resets v.collision to False.

Whenever v is broadcasting c it also checks the colors of each of its neighbors. That is, if u's color is not NoColor and v.Compare(u) = Undefined then v sets v.Compare(u) to u.Color.

For a node v to start echoing c it waits for the following conditions (*Condition Echo*) to hold:

- All v's children (if such exist) are echoing c.
- Node v's parent is broadcasting c. (Otherwise there is no more broadcast to converge, and v resets its coloring related variables.)
- If v's color is different from NoColor then it also waits that each of its neighbors x has noticed v's color. That is, x.Compare(v) =v.Color.
- Similarly, if v's color is different than NoColor then it waits until it has noticed the color of each of its neighbors x. That is, v waits until v.Compare(x) $\neq$ Undefined.

To avoid deadlocks, if v.Color = NoColor then v does not wait to have its color noticed by its neighbors. Similarly node v does not wait to notice its neighbors' colors.

When Condition *Echo* holds (together with Condition st) node v starts echoing c. In addition, if any child x of v detects a neighboring tree with the same identity but a different color (x.collision = True) or v itself detects a collision then v sets v.collision to True. Node v detects a collision if when it starts echoing there is a neighbor u such that v.Compare(u) is defined and v.Compare(u) $\neq$ v.Color $\neq$ NoColor.

Finally, if a root r is echoing a value c and another tree with the same identity was detected (r.collision = True) then r restarts the algorithm: it resets all its variables and randomly chooses another identity. This will cause condition st to fail at all the neighbors of r in the tree, and inductively at all the other nodes in its tree. This will restart the algorithm at these nodes. Any node that detects that condition st is violated, redraws a new id in the range $[1, \ldots, n^2]$.

The method thus described satisfies the following Claim whose proof is not included:

Claim 1 *The spanning tree algorithm for anonymous networks described above eventually satisfies the following properties:*

(Convergence:) *The network is spanned by a correct tree, and*

(Termination:) *Condition st holds forever at all the nodes.*

By standard methods, (Claim 10 in [AM94]) the algorithm takes expected $O(1)$ phases of drawing new identities, and the time of each phase might be $O(n^2)$ (the time necessary to construct a tree) and only expected $O(n)$ time to detect collisions.

We remark that if a bound on the network's size is not known but the ID register is (necessarily) of infinite size, a procedure like the above is still possible. In this case, each time a collision is detected between two highest value IDs, the different colors used to detect the collision are appended to the IDs thus forming new IDs that are guaranteed to be different. In this model the adversary is constrained to access (and corrupt) any finite prefix of the registers, while the algorithm is capable of accessing the prefix plus $O(n)$ additional bits of the register in one step. In the worst case the algorithm adds, on top of the bits set by the adversary, n additional bits (since n is the worst case bound on the potential number of node ID collisions). This scheme is expected to repeat the basic spanning-tree algorithm $O(\log n)$ times before it stabilizes (this holds with very high probability as well). Note that the advantage of this is that a bound on the network size is not necessary, but it comes for the price of unbounded size registers.

6 Conclusions

We have introduced and presented the local detection paradigm for self stabilization (later called by [AV91] "local checking"), as well as examples for its application to derive new spanning tree algorithms and applications thereof. The main example is the self stabilizing protocol for constructing a spanning tree (and derived tasks, e.g., reset) in a general topology network that does not have a pre-specified leader. Using this tree other tasks can be easily performed, such as, reset, topology update, and transformation of protocols to be self stabilizing.

We presented our algorithm with read/write atomicity and designed "local agreement" procedures for neighboring nodes to agree. Our network model combined dynamically changing topology and memory faults.

Following the initial proceedings version of this paper [AKY90] additional related works have has been carried out; some of which were motivated by this work. In [AEYH92, DIM91] algorithms with better time complexity are presented, in particular an optimal expected time randomized algorithm for the case that a polynomial bound on the diameter is known is presented in [DIM91]. It is not hard to check that our quiescence time complexity is $O(n^2)$, where n is the size of the network (not available to the nodes). [AKM+93] used the local detection paradigm to achieve a (O(diameter)) time and worst case $O(\log^2 n)$ message size self stabilizing reset and self stabilizing synchronizer, however, the reset algorithm requires an a-priori bound on the diameter (although the time complexity is not a function of this bound but rather the actual network parameters). In [A94] an O(diameter) time algorithms are presented that do not require such a bound. Novel recursive methods that in particular employ clever applications of our technique enable reduction of the space requirements for self-stabilizing protocols in the recent body of work done by Awerbuch, Itkis, Levin and Ostrovsky (see [AO94, IL94]).

Acknowledgments

We would like to thank Shlomi Dolev, Amos Israeli and Shlomo Moran for comments, and the anonymous referees for very thorough and helpful reviews.

References

[AAG87] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proc. of the 28th IEEE Ann. Symp. on Foundation of Computer Science*, pages 358–370, October 1987.

[AB89] Y. Afek and G. M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing Journal*, 7:27–34, 1993.

[AGR92] Y. Afek, E. Gafni, and A. Rosen. The slide mechanism with applications in dynamic networks. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, pages 35–46, August 1992.

[AKY90] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self stabilizing protocols for general networks. In *Proc. of the 4th Int. Workshop on Distributed Algorithms*, (WDAG), Springer-Verlag LNCS, September 1990.

[AM94] Y. Afek and Y. Matias. Elections in anonymous networks. *Information and Computation (formerly Information and Control)*, 113(2):312–330, September 1994.

[A94] S. Aggarwal. Time optimal self- stabilizing spanning tree algorithms. M.Sc Thesis, MIT, May 1994.

[AEYH92] E. Anagnostou, R. El-Yaniv, and V. Hadzilacos. Memory adaptive self-stabilizing protocols. In *Proc. of the 6th International Workshop on Distributed Algorithms: Springer-Verlag LNCS*, November 1992.

[Ang80] D. Angluin. Local and global properties in networks of processes. In *Proc. of the 12th Ann. ACM Symp. on Theory of Computing*, pages 82–93, May 1980.

[AG90] A. Arora and M. Gouda. Distributed reset. In *Proc. of the 10-th FSTTCS: Springer-Verlag LNCS 472*, pages 316–331, September 1990.

[ACK90] B. Awerbuch, I. Cidon, and S. Kutten. Communication-optimal maintenance of replicated information. In *Proc. of the 31st IEEE Ann. Symp. on Foundation of Computer Science*, pages 492–502, October 1990.

[AKM+93] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self stabilizing synchronization. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 652–661, May 1993.

[AO94] B. Awerbuch and R. Ostrovsky. Memory efficient and self stabilizing network reset. In *Proc. of the 14th ACM Symp. on Principles of Distributed Computing (PODC)*, August 1994.

[APSV91] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proc. of the 32nd IEEE Ann. Symp. on Foundation of Computer Science*, pages 268–277, October 1991.

[APVD94] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev. Self stabilization by local checking and global reset. in the Proc. of WDAG 94, Springer-Verlag LNCS, pages 226–239, October 1994.

[AV91] B. Awerbuch, , and G. Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *Proc. of the 32nd IEEE Ann. Symp. on Foundation of Computer Science*, pages 258–267, October 1991.

[Awe85] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[BGW89] G. M. Brown, M. G. Gouda, and C-L Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, c38(6):845–852, 1989.

[BP89] J. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, 1989.

[Cha79] E. J. Chang. *Decentralized Algorithms in Distributed Systems.* PhD thesis, University of Toronto, October 1979.

[CTW93] D. Coppersmith, P. Tetali and P. Winkler. Collisions among random walks on a graph. In *SIAM J. Disc. Math.*, 6,3:363–374, August 1993. Abstract In *Proc. of the 10th ACM Symp. on Principles of Distributed Computing*, pages 273–280, August 1991.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communication of the ACM*, 17:643–644, November 1974.

[DIM94] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing Journal*, 7, 1994. also In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, August 1990.

[DIM91] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. In P.G. Spirakis S. Toueg and L. Kirousis, editors, *Lecture Notes in Computer Science 579: Proc. of the fifth Int. Workshop on Distributed Algorithms*, pages 163–180. Springer Verlag, October 1991.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5:66–77, January 1983.

[GH89] M. G. Gouda and R. R. Howell. The instability of self-stabilization. Extended Abstract, January 1989.

[GM91] M. G. Gouda and N. Multari. *Stabilizing Communication Protocols.* In *IEEE Transactions on Computers*, 40(4):448–458 1991.

[IL94] G. Itkis, and L Levin. Fast and Lean Self-Stabilizing Asynchronous Protocols. In *Proc. of the 35th IEEE Ann. Symp. on Foundation of Computer Science*, pages 226-239, October 1994.

[IJ90] A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, pages 119–132, August 1990.

[KP94] S. Katz and K. J. Perry. Self-stabilizing extensions. *Distributed Computing Journal*, 7, 1994. also In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, August 1990.

[Kru79] H.S.M. Kruijer. Self-stabilization (in spite of distributed control) in tree-structured systems. *Information Processing Letters*, 8(2):91–95, 1979.

[MNS94] A. Mayer, M. Naor and L. Stockmeyer. Local Computations on Static and Dynamic Graphs. In *Proc. of the IEEE 3rd Israeli Symp. on Theory of Computing and Systems*, pages 268–278.

[MOOY92] A. Mayer, Y. Ofek, R. Ostrovsky, and M. Yung. Self-stabilizing symmetry breaking in constant space. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 667-678, May 1992.

[NS93] M. Naor and L. Stockmeyer. What can be computed locally. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 185–193. ACM, May 1993.

[SS94] Schieber and Snir. Calling names on nameless networks. *Information and Computation (formerly Information and Control)*, 113, 1994. also in: *Proc. of the ACM Symp. on Principles of Distributed Computing*, pages 319–328, August 1989.

[Seg83] A. Segall. Distributed network protocols. *IEEE Trans. on Information Theory*, IT-29(1):23–35, January 1983.

[SG89] J. Spinelli and R. G. Gallager. Broadcast topology information in computer networks. *IEEE Trans. on Comm.*, 1989.

[Sto93] F.A. Stomp. Structured design of self stabilizing programs. In *Proc. of the IEEE 2nd Israeli Symp. on Theory of Computing and Systems*, pages 167–176, June 1993.

[Var92] G. Varghese. *Dealing with Failure in Distributed Systems.* PhD thesis, MIT, Department of Electrical Engineering and Computer Science, 1992.