Memory-Efficient Self Stabilizing Protocols for General Networks

Yehuda Afek
AT&T Bell Laboratories, and
Computer Science Department,
Tel-Aviv University,
afek@taurus

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

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

EXTENDED ABSTRACT

Abstract

A self stabilizing protocol for constructing a rooted spanning tree in an arbitrary asynchronous network of processors that communicate through shared memory is presented. The processors have unique identifiers but are otherwise identical. The network topology is assumed to be dynamic, that is, edges can join or leave the computation before it eventually stabilizes.

The algorithm is design uses a new paradigm in self stabilization. The idea is to ensure that if the system is not in a legal state (this is a global condition) then a local condition of some node will be violated. Thus the new could restart the algorithm.

The algorithm provides an underlying self-stabilization mechanism and can serve as a basic building block in the construction of self stabilizing protocols for several other applications such as: mutual-exclusion, snapshot, and reset.

The algorithm is memory efficient in that it requires only a linear size memory of words of size $\log n$ (the size of an identity) over the entire network. Each processor needs a constant number of words per incident link, thus the storage requirement is in the same order of magnitude as the size of the traditionally assumed message buffers size. The adversary may be permitted to initiate the values of the variables to any size. Still, in this case the additional memory used by the algorithm is the amount stated above.

Extensions of our algorithm to other models are also discussed.

1 Introduction

In 1974 Dijkstra has introduced a new approach to fault tolerant distributed computing, called *self stabilization* [Dij74]. In this approach protocols are designed with a built-in automatic mechanism that whenever the distributed system is placed in an illegal global state the protocol guarantees that it will converge back to a legal state and will stay legal thereafter.

The first problem for which self-stabilized protocols were designed was the mutual exclusion problem ([Dij74, BGW89, BP89] and others); recently, other tasks have been shown to have this property (e.g., [DIM90, AB89, KP90]). In general one can formulate the question with respect to any distributed computing task and think about a general underlying mechanism which can support several tasks and provide them with a self-stabilization augmentation.

In this work we present a self stabilizing protocol to construct and maintain a rooted spanning tree in a dynamic network. This protocol serves as an underlying primitive for a self-stabilizing compiler; that is, a general mechanism to self stabilize arbitrary distributed algorithms, in the same way as the protocol of [AAG87] is used to run distributed algorithms in a fail-safe fashion (see [KP90] for discussion of such self-stabilizing compilers).

We assume the model of an asynchronous network of processors with unique IDs communicating via shared memory. The IDs are unchangeable throughout the protocol (each is embedded in the machine's hardware).

In [SG89], Spinelli and Gallager present a topology maintenance algorithm for arbitrary dynamic networks. It turns out that their algorithm is easily modified into a self-stabilizing spanning tree algorithm, however it is an expensive algorithm both in its communication complexity (nodes have to periodically read O(nE) memory words, where n and E are the total number of nodes and links in the network, and the O(nE) words are the whole description of the network as viewed by the neighbor), and in its memory requirements (each node has to maintain one copy of the network topology for each of its incident links).

It is well known [Ang80] that no deterministic algorithm can construct a spanning tree in an anonymous (i.e., uniform) symmetric network. Dijkstra has shown that deterministic self-stabilizing protocols for mutual exclusion cannot be constructed in an anonymous ring whose size is a composite number (Burns and Pachl [BP89] gave a uniform protocol for the self-stabilizing mutual exclusion problem on rings whose size is a prime number). Independent of our work, S. Dolev, Israeli and Moran [DIM90] have recently designed a self stabilizing spanning tree protocol. To break the symmetry their protocol assumes the existence of a unique "special" processor in the network (a pre-distinguished leader) and the constructed tree is rooted at that node. Naturally they assume that the adversary cannot change the status of the unique node such that it stops being in the "special" status, neither can it cause any other node to become "special". Note that the network cannot be allowed to be partitioned such that no special node will be included in one of the partitions. In another independent work, Arora and Gouda [AG90] have designed a spanning tree construction self-stabilizing protocol that rather than assuming a predistinguished node assumes that each node has a unique ID (as we do). However they assume a known bound on the network size in order for the algorithm to self stabilize. Moreover, the quiescence time complexity of their algorithm depends on that bound, regardless of the actual size. Note that faults or partitions may decrease the actual size significantly. Our quiescence time complexity is $O(n^2)$ where n is the unknown size of the network (component).

In this paper we use unique IDs at the nodes to break the symmetry but make no farther assumption such as knowledge of the network size, nor that the network is not partitioned. Our approach is that in a fault tolerant network it is more natural not to rely on connectivity to a specific processor (leader), thus we require that connected components of possibly partitioned network will continue the protocol and will independently stabilize as well. This is a crucial difference between a symmetric distributed protocol and an asymmetric protocol that assumes a leader. The price is the need to break symmetry using ID's.

The algorithm design uses a new paradigm in self stabilization. Previous algorithms did not rely on the detection of an illegal state. In the independent work of [KP90] such an illegal state was detected by collecting information about the global state. We detect an illegal global state by ensuring that in such a state a local condition of some node will be violated. Thus the node could restart the algorithm.

The size of the space consumed in a processor by our protocol is exactly what is traditionally the size of a message (and its buffers), i.e., log(Max-ID) times the degree of the processor. Thus, we can view this as an optimal storage requirement, since any link needs a message buffer in any global distributed computing (and if we implement the reading by message passing we assume buffers of this size are available). (The adversary may be permitted to initiate the values of the variables to any size. Still, in this case the additional memory used by the algorithm is the amount stated above.)

Following our deterministic self stabilizing election, a randomized election algorithm for anonymous networks was developed lately by S. Dolev, Israeli and Moran [DIM90a]. Our deterministic protocol has a natural extension (using [AM89]) to run on anonymous networks. In this modular extension we use an additional procedure which carefully draws and continually checks random ID's to verify that they are different. The presentation of the randomized algorithm is postponed to the full paper.

It is also possible to run our algorithm in the message passing model, simulating shared memory with the help of the datalink protocol of [AB89].

2 The Model

We present here the model of a dynamic network of processors communicating via shared memory. The network is represented by a graph (V, E), where V is the set of processors, and E is a set of links. If u and v are processors and (u, v) is in E then we say that there is a link between u and v, and that they are neighbors. Each processor has a unique ID (say, hardwired in its memory), this is the only assumption that makes our model non-uniform. 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 its own local memory. Reads of neighbors memory as well as read and write on local memory are atomic operations. Each processor is a state machine with a bounded number of states (which could be a function of n; the dependency on n can be in the size of its registers). At the beginning of the computation the adversary can put each processor in an arbitrary state. The local computation at each processor consists of a sequence of transitions where each transition consists of an operation that moves the processor from its state before the transition to another (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. The fair scheduler (demon) of the global computation consists of 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 is down). Such an Atomic Read/Atomic Write demon was first suggested by [DIM90].

At any system state, the set of local states of the processors defines a global state. In any self-stabilization problem, a subset of the set of global states is defined as legal global states. In a self-stabilizing protocol the computation moves the system from a legal state only to another (possibly the same) legal state and starting from any state a fair scheduler will eventually bring the system to a legal state.

The network topology is dynamic, that is links and processors can go down and come up an arbitrary number of times. We assume that there is a local self-stabilazing 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 other's memory. We further assume that the sequence of topological changes is finite and that eventually topological changes cease [AAG87].

In the spanning tree construction algorithm each node has a Parent variable, that holds an *id* of a processor. We say that a tree spans the network when the collection of the Parent variables defines such a rooted tree in the natural way. Our main algorithm is a self stabilizing algorithm 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.

3 A Self Stabilizing Spanning Tree Algorithm

In a nutshell the algorithm can be described as follows: Every node tries to construct a tree in the network from itself. The process of larger identity nodes overruns the processes of lower identity nodes. Eventually, the tree of the largest Id node overruns all the other trees. A node leaves its tree when it detects a local inconsistency. However, to join another tree whose root Id is larger a node has to propagate a *request* message along the new tree branches to the root and to receive a *grant* message back. This mechanism prevents the necessity to know additional information such as a bound on the network size. Of course these messages propagate through the shared memories of the nodes via a sequence of read and write operations. To maintain self-stabilization we define certain local conditions that if violated in the neighborhood of a node then that node detects it and restarts the algorithm by considering itself the root of a single node tree. (It is guaranteed that if the network is in an illegal state, then some node will notice the violation of these conditions.)

Let us now describe the algorithm in detail. There are three groups of variables at each node:

1. The usual local variables { Id, Edge_list }:
   We denote by v.Id the variable Id in node v. The same convention is used for the other variables.
   v.Id is the read-only identity of node v and
   v.Edge_list is a list of links incident to node v that are operational and such that the processor on the other side is up. This list is maintained by a lower level self-stabilizing protocol which is beyond the scope of this paper. Each change in a link or node status is eventually recorded in Edge_list. If a neighbor u is removed from Edge_list while v is performing a read of the memory of u, then that read may return any value.

2. The variables related to the tree structure { Root, Parent, and Distance} where,
   v.Root is supposed to hold the identity of the root of the tree to which node v belongs;
   (We omit the word "supposed" in the sequel.)
   v.Distance is the distance from node v to its root; and
   v.Parent is the identity of a neighbor of v which is the parent of v in the tree.

3. The variables related to passing the *request* and *grant* messages { Request, To, Direction, and From} where,
   v.Request is either an Id of a node that is currently requesting to join the tree to which v belongs, or equal to v.Id if v itself is trying to join another tree;
   v.From 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 the name of a neighbor of v through which it is trying to propagate the *request* message;
   v.Direction is either *Ask*, to signify that the node whose Id is in v.Request wishes to join the tree, or *Grant* to signify that this request has been granted.

Definition 3.1 *We say that v is a* child *of w and that w is the* parent *of v if:*
$(w.\mathsf{Id} \in v.\mathsf{Edge\_list}) \wedge (v.\mathsf{Parent} = w.\mathsf{Id}) \wedge (v.\mathsf{Root} = w.\mathsf{Root}) \wedge (v.\mathsf{Distance} = w.\mathsf{Distance} + 1)$.

To achieve self stabilization of the spanning tree construction we define a condition, called $C1$, on the variables at each node such that if (and only if) the condition holds at all the nodes then the network (or component) is in a globally legal state in which a correct spanning tree exists. This condition is checked periodically at all the nodes. The aim of the algorithm is to detect violations of the condition (either because of incorrect initial values of the nodes' variables, or because of topological changes), and to ensure that eventually the network will enter a globally legal state.

Condition C1:
$\{[(\mathsf{Root} = \mathsf{Id}) \wedge (\mathsf{Parent} = \mathsf{Id}) \wedge (\mathsf{Distance} = 0)] \vee$
$[(\mathsf{Root} > \mathsf{Id}) \wedge (\mathsf{Parent} \in \mathsf{Edge\_list}) \wedge (\mathsf{Root} = \mathsf{Parent.Root}) \wedge (\mathsf{Distance} = \mathsf{Parent.Distance} + 1 > 0)]\}$
$\wedge (\mathsf{Root} \geq \max_{x \in \mathsf{Edge\_list}} x.\mathsf{Root})$

A formal description of the algorithm is given in Figure 1. The program consists of a set of actions. Each action 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. An action can only be executed from a state in which its guard is checked and found true. However, since we assume atomicity of a single operation only, some variables already checked can be changed while others are still being read. Thus it it may be the case that the guard is actually false, and still was found true. The execution at each node proceeds by an infinite sequence of operations where in each operation the memory of one neighbor is read atomically, then the guards of the actions are checked. If any guard is true, then the corresponding commands are performed atomically. (It can be shown that the atomicity of the read of one register, is an atomic operation, as well as the write of one register.) Each neighbor in Edge_list after the last topological change, is read infinitely often by the sequence of operations (e.g., the nodes are read in a linear cyclic order).

A node that notices that Condition C1 does not hold, must become a root (Action 1).

Definition 3.2 *Node v is a* **root** *if* $(v.\mathsf{Root} = v.\mathsf{Id}) \wedge (v.\mathsf{Parent} = \mathsf{Id}) \wedge (v.\mathsf{Distance} = 0)$.

By becoming a root condition $C1$ does not necessarily become true, however we define condition $C1'$ which does become true. Moreover, if condition $C1'$ is satisfied at all the nodes then (by transitivity) the Parent variables in the nodes define a spanning forest in the network. If condition $C1'$ holds at a node but $C1$ does not then eventually that node moves on to the process of joining a tree with a larger Root (Action 2).

Condition $C1'$:
$[(\text{Root} = \text{Id}) \wedge (\text{Parent} = \text{Id}) \wedge (\text{Distance} = 0)] \vee$
$[(\text{Root} > \text{Id}) \wedge (\text{Parent} \in \text{Edge\_list}) \wedge (\text{Root} = \text{Parent.Root}) \wedge (\text{Distance} = \text{Parent.Distance} + 1 > 0)]$

If node v's Id (which is now also its Root) is not larger than the Roots of its neighbors, it attempts to join another tree. It chooses the neighbor 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 that it sets its Request and From to its own Id, its Direction to *Ask* and its To to u. (Note that some of the operations in the guards are not (and do not need to be) atomic.)

When condition $C1$ holds at node v that node 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 to join the tree to the root of the tree and grants from the root back to the requesting node (Actions $4 - 7$). As with the tree related variables, we define Condition $C2$ that is true if the variables related to that process are in a legal state. If $C2$ does not hold then the node must reset those variables (Action 3).

Definition 3.3 *We say that node v is* forwarding a request from node *w if the following condition holds:*

Condition $C2'$:
$((w.\text{Id} \in v.\text{Edge\_list}) \wedge$
$(w.\text{Root} = w.\text{Id} = w.\text{Request} = w.\text{From} = v.\text{Request} = v.\text{From} \wedge$
$(\text{w.To} = \text{v.Id}) \wedge (\text{w.Direction} = Ask))$
$\vee$
$((w.\text{Id} \in v.\text{Edge\_list}) \wedge (v\ parent\ of\ w) \wedge (w.\text{Request} = v.\text{Request} \neq w.\text{Id}) \wedge$
$(v.\text{From} = w.\text{Id}) \wedge (v.\text{To} = v.\text{Parent}) \wedge (w.\text{To} = v.\text{Id}))$

Condition $C2$:

$C2' \vee$
(v.Request, v.To, v.From, and v.Direction are undefined)

If node v is forwarding a request (i.e., $C2'$ holds at v) and it is a root, it can *grant* the request (Action 6). That is, it changes its Direction to *Grant* (unless its Direction is already *Grant*). A non-root node who is forwarding a request, can forward a grant, provided that its parent is (Action 7):

```
**The program at node $v$:**
**do forever: read next neighbor information and**

1  $\neg C1 \wedge \neg C1'$
       $\rightarrow$ $v$.Root :=$v$.Id; $v$.Parent := Id;
          $v$.Distance := 0;

2  $C1' \wedge$
   ($u$.Root = $\max_{x \in \text{Edge\_list}}$ x.Root) >$v$.Root
       $\rightarrow$ $v$.Request :=$v$.From :=$v$.Id;
          $v$.To :=$u$.Id; $v$.Direction := $Ask$;

3  $C1 \wedge \neg C2$
       $\rightarrow$ set $v$.Request, $v$.From, $v$.To,
          $and$ $v$.Direction to undefined

4  $C1 \wedge C2 \wedge \neg C2' \wedge$ ($\exists w \in v$.Edge_list|
   ($w$.Direction = $Ask$) $\wedge$ ($w$.To =$v$.Id)$\wedge$
   ($w$.Request =$w$.Id =$w$.Root =$w$.From))
       $\rightarrow$ $v$.Request :=$v$.From :=$w$.Id;
          $v$.To :=$v$.Parent; $v$.Direction := $Ask$

5  $C1 \wedge C2 \wedge \neg C2' \wedge$ ($\exists w \in v$.Edge_list|
   ($w$ $child\ of$ $v$) $\wedge$ ($w$.To = v.Id)$\wedge$
   ($w$.Direction = $Ask$))
       $\rightarrow$ $v$.Request :=$w$.Request; $v$.From :=$w$.Id;
          $v$.To :=$v$.Parent; $v$.Direction := $Ask$

6  $C1 \wedge C2' \wedge$ ($v$ $is\ a\ root$)$\wedge$ ($v$.Direction = $Ask$)
       $\rightarrow$ $v$.Direction := $Grant$

7  $C1 \wedge C2' \wedge$ ($v$.To =$v$.Parent =$u$.Id)$\wedge$
   ($u$.Direction = $Grant$) $\wedge$ ($v$.Direction = $Ask$)$\wedge$
   ($u$.Request =$v$.Request) $\wedge$ ($u$.From =$v$.Id)
       $\rightarrow$ $v$.Direction := $Grant$

8  $C1' \wedge \neg C1 \wedge$ ($v$.Direction = $Ask$)$\wedge$
   ($v$.Request =$u$.Request =$v$.From =$v$.Root =$v$.Id)$\wedge$
   ($u$.From =$v$.Id) $\wedge$ ($u$.Direction = $Grant$) $\wedge$ ($v$.To =$u$.Id)
       $\rightarrow$ $v$.Parent :=$u$.Id;
          $v$.Distance := u.Distance + 1;
          $v$.Root :=$u$.Root; reset request variables
```

Figure 1: The program at node v

1. Forwarding the same request (i.e. the parent and v have equal values in their Request variable); and
2. 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*).

When a grant has been received by the descendant of a node (i.e., the descendant was requesting from the node, and has now set its direction to grant) that node may reset its requesting related variables (action 3) and service other requests. 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 larget by one than that of its parent. In addition its resets its request variables to "undefined".

4 Correctness

In this section we prove the self stabilizing property of the algorithm. Let us first solve one difficulty that will repeatedly appears in the proofs. Intuitively this is the fear that a node "imagines" it read some value, but in fact did not. Note that since a high resulusion atomicity is assumed. Thus this involves also variables not explicityly mentioned in the the code, but also possible temporary variables used by the processor to store intermediate results, e.g. that some term of a computed condition has the value "true".

The formal definition of "imagines" is deferred to the full paper. However, we argue that there exists a time when such "imagination" is no longer possible. This follows from the fact that every node repeatedly reads (and the scheduler is fair). Thus, there exists a time t after the time when the faults cease, that from t on every value that is used in the computation was indeed read, or computed from values that where indeed read. In the following proof we speak only to the time after t. Thus it is not necessary to consider "imagination".

Next let us prove that eventually the node with the highest identity becomes a root, and no higher Root variables ever exist later.

Definition 4.1 *When Condition $C1$ holds for a node then it and its parent constitute together a* **branch**. *Each connected component of the transitive closure of the branch relation is also called a* **branch**.

A **request interval** *is a maximal path that is included in a branch, such that only the (possibly empty) suffix of the interval (the* Parents*' side) contains a* Grant, *the (possibly empty) prefix contains an* Ask, *and all the* Request *registers in the interval's nodes have the same value.*

Definition 4.2 *Let a* **false root** *be an identity that is a value of a* Root *variable of some node (in a connected component), but is not the identity of any node in the (connected component of the) network.*

Observation 4.3 *No identity that does not exist in the (connected component of the) network at any given time after the changes cease (either as an* Id *or as a* Root*) can become a false root later on.*

Proof Sketch: Notice that a node sets its Root either to its Id or to the Root of its neighbor. ▮

Lemma 4.4 *Consider the time after the changes cease. The number of times a node joins a tree whose* Root *is a false root is bounded.*

Proof Sketch: Let us first discuss only the highest false root (break ties arbitrarily). (If there are false roots then there exists such a Root by Observation 4.3.) The number of request intervals that will ever exist in branches with such a Root is bounded Also, if a request is granted by a grant that existed by mistake than the number of intervals decreases by one.

Consider now the second highest false root. The argument above bounds the number of times a node can join that tree, between times that it joins the tree of the highest false root. This argument can now be applied recursively to all the false roots. ▮

Lemma 4.5 *Eventually in every node* Root *equals some $w \in V$.*

Proof Sketch: By Lemma 4.4 and by Observation 4.3 it suffices to prove that no Root variables can contain a false root forever.

Assume the contrary and consider any time after the changes cease. Let $u \notin V$ be the highest identity such that from some time on there always exists a node $v_0 \in V$ whose Root equals some $u \notin V$.

Consider v_0 at a time after the topological changes. Note that v_0 must have a parent v_1 (i.e. its Parent does not equal its own Id.) whose Root equals u, otherwise, since $v_0 \neq u$, node v_0 would have set its Root to v.

Lev $v_0, v_1, ..., v_q$ be the maximal branch in which Root = u forever. (This is the case at least in v_0. Also, q is finite since v_{i+1}.Distance is smaller than v_i.Distance, and every Distance is not smaller than zero.) However, when v_q will not have a parent whose Root is u, node v_q will have to reset its Root too, contradicting the definition of v_q. This, together with Lemma 4.4 implies the lemma. ▮

Corollary 4.6 *Let u be the node with the highest* Id *in (its connected component of) the network. From some time on u is a root.*

Lemma 4.7 *Eventually every maximal branch contains the node whose* Id *is the value of* Root *in every node on the brach.*

Proof Sketch: Similar to Lemma 4.5. ▮

Let us now proof that a tree rooted at the node with the highest Id eventually spans the network. First let us show that from some time on there are no cycles.

Lemma 4.8 *There exists a time from which on there are no cycles in the parent relation.*

Proof Sketch: Similarly to the proof of Lemma 4.5 every cycle must be broken. New cycles cannot be created (after time t) by the definition of request intervals (that cannot be cyclic because of the Distance register, and the observation that a node joins only when a request interval exists, and it is a root. ▮

Definition 4.9 *Consider the time from which on all* Root *values belong to nodes in the (connected component of the) network. A* correct tree *is a tree defined by the* Parent *relation, such that its root is the node with the highest identity, and Condition $C1$ holds for all the nodes in the tree.*

Observation 4.10 *There is a time from which on a correct tree exists.*

Proof: Follows from Corollary 4.6 and Lemma 4.5. ▮

Lemma 4.11 *Consider the time from which on a correct tree exists. A node that is not in this correct tree, but is a neighbor of a node in that tree, will eventually try to join the correct tree.*

Proof Sketch: Consider such a node v after the times mentioned in Lemmas reflem:false small root disappear, 4.5 and 4.8. If it is not a root then either its Root is that of the correct tree, or its Root is smaller than the highest. In the second case clearly Condition $C1$ does not hold for it. In the first case if Condition $C1$ did hold for it then it would have belonged to the correct tree. Thus condition $C1$ does not hold for it in both cases, and it must become a root. At that time, the algorithm dictates that it makes a request to join the neighboring tree with the highest Root value, which is the correct tree. ▮

Observation 4.12 *A node that joins the correct tree never leaves it.*

Proof Sketch: The reason is that the correct tree is defined only after there are no false roots, and since a node leaves a tree only when Condition $C1$ does not hold. ▮

Definition 4.13 *An* Id *v 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 in 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.

Lemma 4.14 *If there are nodes in the correct tree that contain a request that is not a correct request, then this request will eventually disappear from the nodes of the correct tree.*

Proof Sketch: By Observation 4.12 the request interval of this request can change only by growing, and it can grow only to a bounded size. Consider now the prefix of this interval, clearly the node at the beginning (of the prefix) notices that this is an incorrect request, and resets all the variables that are related to the request. 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. ▮

Lemma 4.15 *Consider a time in which the correct tree does not span the entire (connected component of the) network. Eventually another node will join it.*

Proof Sketch: By Lemma 4.11 every neighbor of the correct tree will eventually try to join the correct tree. By Lemma 4.14 incorrect requests will disappear. Note also that the requests are forwarded over a tree. Thus, eventually there must be a branch from some node v not on the correct tree to the root of the correct tree that includes only one request interval – that of the request of v. Thus this request is granted, and v joins the correct tree. ▮

Lemma 4.16 *Eventually a correct tree spans the network.*

Proof Sketch: Follows from Lemma 4.15 and from Observation 4.12. ▮

Theorem 4.17 *Eventually the following holds:*

(convergence) *the (connected component of the) network is spanned by a correct tree, and*

(termination) *no node performs any operation in the tree construction algorithm.*

Proof Sketch: Follows from Lemmas 4.16 and 4.14 and from observation 4.12 ▮

5 Conclusions and Discussion

A motivation for the construction of this algorithm was the possibility to use it as a modular component in other algorithms. One example is 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 [DIM90].) Another example is the randomized self stabilizing symmetry braking in anonymous networks. We can further make the reset procedure presented in [AAG87] self stable. A generalization of the self-stabilized snapshot of Katz and Perry [KP90] can be achieved, which implies in turn that general protocols can be self-stabilized.

As mentioned above our memory is optimal when message buffers are used. It is interesting to find out whether one can show a more memory-efficient protocol. In particular it is interesting to compare memory-efficiency of protocols in the unique ID model to other self-stabilized protocols in other models.

References

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

[AB89] Y. Afek and G. M. Brown. Self-stabilization of the alternating-bit protocol. In *Proceedings of the 8th IEEE Symposium on Reliable Distributed Systems*, pages 10–12, October 1989.

[AG90] A. Arora and M. Gouda. Distributed reset. Extended Abstract, 1990.

[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.

[AM89] Y. Afek, and Y. Matias Simple and Efficient Election Algorithms for Anonymous Networks, *3rd International Workshop on Distributed Algorithms*, Nice, France, September 1989.

[BGW89] G. Brown, M. Gouda, and C.L. Wu. Token systems that self stabilize. *IEEE Transactions on Computers*, 38(6):845–852, 1989.

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

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

[DIM90] S. Dolev, A. Israeli, and S. Moran. Self stabilization of dynamic systems assuming read/write atomicity. In *Proc. of the ACM Symp. on Principles of Distributed Computing*, August 1990.

[DIM90a] S. Dolev, A. Israeli, and S. Moran. Private communication

[KP90] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions. In *Proc. of the ACM Symp. on Principles of Distributed Computing*, August 1990.

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