Time Optimal Self-Stabilizing Spanning Tree Algorithms

Sudhanshu Aggarwal[1] and Shay Kutten[2]

[1] MIT Laboratory for Computer Science and IBM T.J. Watson Research Center
[2] IBM T.J. Watson Research Center

Abstract. In this paper we present time-optimal self-stabilizing algorithms for asynchronous distributed spanning tree computation in networks. We present both a randomized algorithm for anonymous networks as well as a deterministic version for ID-based networks. Our protocols are the first to be time-optimal (i.e. stabilize in time *O(diameter)*) without any prior knowledge of the network size or diameter, assuming we are allowed messages of size *O(ID)*. Both results are achieved through a new technique of symmetry breaking that may be of independent interest.

1 Introduction

The task of *spanning tree construction* is a basic primitive in communication networks. Many crucial network tasks, such as network reset (and thus any input/output task), leader election, broadcast, topology update, and distributed database maintenance, can be efficiently carried out in the presence of a tree defined on the network nodes spanning the entire network. Improving the efficiency of the underlying spanning tree algorithm usually also correspondingly improves the efficiency of the particular task at hand. In practice, computation in asynchronous distributed networks is made much more difficult because of the possibility of numerous kinds of *faults*. The property of *self-stabilization*, first introduced by Dijkstra [Dij74], implies the ability of the system to recover from *any* transient fault that changes the state of the system. Self-stabilization is a very strong and highly desirable fault-tolerance property.

We would therefore like to have an *efficient self-stabilizing algorithm for spanning tree construction in asynchronous networks.* There are two principal measures of efficiency - *stabilization time*, which is the maximum time taken for the algorithm to converge to a "spanning tree" state, starting from an arbitrary state, and the *space required at each node* (i.e. size of local memory needed). Let d be the *diameter* of the network, and let n be the *network size* – the number of nodes in the network. Note that the optimal stabilization time must necessarily be $\Omega(d)$, and the optimal space requirement for an ID-based protocol must be $\Omega(\log n)$ (since there must exist IDs of size $\Omega(\log n)$).

Several factors influence the "difficulty" of the protocol. The protocol can be designed for networks that are either *ID-based* (each node has a unique "hard-wired" ID), or for networks that are *anonymous* (in which nodes lack unique IDs, so there is no *a priori* way of distinguishing them). The protocol may either "know" the network size n, or it may "know" some upper bound on n, or it may "know" nothing whatsoever. Of course, the more "knowledge" a protocol "is given" about the network, the easier it becomes to achieve its objectives.

Following the pioneering work of [Dij74], there has been considerable work in this area. [Ang 80] showed that no deterministic algorithm can construct a spanning tree in an anonymous network, so any protocol for the anonymous case must necessarily employ randomization. [AKY90] give an ID-based self-stabilizing spanning tree protocol with a stabilization time of $O(n^2)$ and space requirement $O(\log n)$; they subsequently also give a randomized protocol. They presented the technique of "local checking" and "local detection", used in many subsequent papers. [AG90] give an ID-based self-stabilizing spanning tree protocol with time complexity $O(N^2)$, where N is a pre-specified *bound* on the network size n. [APV91] give an ID-based self-stabilizing *reset* protocol that stabilizes in $O(n)$ time and requires $O(\log n)$ space.

[DIM91] give a self-stabilizing spanning tree algorithm for anonymous networks that runs in expected $O(d \log n)$ time and $O(\log n)$ space. [AV91] present a self-stabilizing *synchronizer*, which stabilizes in time $O(D)$, where D is a pre-specified *bound* on the network diameter. [AM89] give a Monte-Carlo spanning tree protocol for anonymous networks that works in $O(d)$ time and expected $O(\log \log n)$ space; however, their protocol is *not* self-stabilizing.

[DIM91] also mention a self-stabilizing spanning tree protocol for anonymous networks that requires $O(d)$ time (and is thus time-optimal), but requires prior knowledge of a bound N on the network size. Recently, [AKMPV93] have developed a time-optimal self-stabilizing spanning tree protocol for ID-based networks; they, too, require prior knowledge of a bound D on the diameter of the network. Their protocol requires $O(d)$ time and $O(\log n \log D)$ space. However, they need messages of size $O(\log n \log D)$. The usual convention only "allows" messages of size $O(\log n)$, so their messages are longer than permitted (which implies that if they were to follow the convention, their time complexity would actually be $O(d \log D)$).

We present the first time-optimal self-stabilizing spanning tree algorithms *that do not need any prior knowledge of the network size or diameter.* We present both a randomized Las-Vegas algorithm for anonymous networks and a deterministic version for ID-based networks. Both our protocols stabilize in $O(d)$ time, and require space only $O(\log n)$ bits larger than the space occupied at the "start" of the algorithm (i.e. set by the adversary in the "bad" initial state; not by the algorithm).

Thus, with respect to the $O(d \log n)$-time protocol of [DIM91], we decrease the time complexity to $O(d)$, and compared to their $O(d)$-time protocol, we do not need a bound N on the network size. Unlike [AKMPV93], we do not need a bound D on the diameter. Further, our deterministic algorithm is the first to be time-optimal under the conventional measure of time complexity which allows messages of only $O(\log n)$ bits to be sent in one unit of time (provided the adversary has not set more than $O(\log n)$ bits).

2 The Model

We assume that the network is represented by a graph $G = (V, E)$ of *processors* V and *links* E. Our protocols are developed for the popular *shared memory* model. In the shared memory model, each processor is associated with a set of *local registers*. Processors communicate by performing write operations on their local registers and read operations on the registers of their neighbors. All reads and writes are *atomic*. Each processor is a *state machine*; the local computation at each processor consists of a sequence of atomic *steps*. The *state* of a processor fully describes its internal state and the values written in all its registers. A *configuration* C of the network is a vector of states of all its processors. 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 step is performed. An *execution* of the system is an infinite sequence of configurations $\{C_1, C_2, ...\}$.

The protocol is presented as a "loop" of code performed repeatedly; each processor will execute the entire loop infinitely often. As in [DIM91], we partition fair executions into *rounds*. We define the *first round* of an execution E to be the minimal prefix E_1 of E in which each processor has executed steps comprising one loop iteration. Let E_1' be the suffix of E such that $E = E_1 \cdot E_1'$. We define the second round of E to be the first round of E_1', and so on. Rounds mark the passage of time—the stabilization time of the protocol is the maximum number of rounds required for the system to reach its desired state. For the purpose of self-stabilization, we assume that the network is *dynamic* – nodes or links may enter or leave the network at any time. Further, the *state* of a processor may change arbitrarily. We assume that the sequence of topological changes and non-algorithmic state changes is finite and that eventually such events cease.

3 Comparison of our work with previous approaches

Spanning tree algorithms usually utilize variants of a common overall scheme. We first describe the basic scheme which assumes the existence of unique node IDs. The network is logically partitioned into a spanning forest, which is defined by *parent* pointers maintained by the nodes. Initially (unless initialized by the adversary), this forest consists of the single-node trees defined by the network nodes themselves. Starting from this configuration, the trees gradually coalesce into larger trees. Each node keeps track of the ID of the root of its tree. The goal is to produce a spanning tree rooted at the node with the highest ID. When a node v notices a neighbor u with a higher root ID, it attaches itself to u's tree by making u its parent ($parent_v \leftarrow u$). Thus, trees with higher root IDs *overrun* trees with lower ones. Eventually, all nodes in the network form a single tree rooted at the node with the highest ID.

To adapt the ID-based scheme to an anonymous network (i.e. with no pre-assigned IDs), we need randomization to break symmetry between the processors. Each node in the network flips coins to arrive at a random ID, and participates in the tree construction process described above. Since IDs are chosen randomly, it is possible that the node with the highest priority in the network is not unique. To detect such "multiple highest priorities", [AKY90] and [DIM 91] proposed the method of *recoloring* trees. In their scheme, each tree is associated with a randomly chosen *color*. The root chooses a color at random from a small set of colors. This color is propagated through the entire tree rooted at that root. When the root receives confirmation that the entire tree has been colored with its color (through a simple acknowledgement mechanism), it chooses a new color. The process is repeated forever. This allows neighboring trees with the same priority p to detect the existence of each other.

When a root learns of the existence of another tree rooted at the same ID, in the [AKY90] and [DIM91] schemes the root *extends* its ID by a randomly chosen bit and continues the protocol. Extending IDs is a way of breaking symmetry; eventually the roots in the network have appended enough random bits to their IDs so that there is a unique root with the highest ID, and consequently a unique tree spanning the entire network.

Our main technical contribution in this paper is to develop an intuitive framework for ID extension and generalize the concept. Our generalization enables us to reduce the time complexity of the randomized protocol to $O(d)$, without prior knowledge of the size or diameter of the network. Intuitively, the $\log n$ factor in the previous randomized result came from the need to initiate a new *competition* every time two trees "collided". Every time a tree T_v noticed another tree $T_{v'}$ with the same ID, T_v would randomly extend its ID to try to "win" over $T_{v'}$. Our new method usually needs just $O(1)$ ID extensions per node to converge to a spanning tree, as opposed to $O(\log n)$ extensions in the previous scheme.

4 ID representation

IDs are represented as tuples of *entries*. In the deterministic protocol, each entry is an *integer*. In the randomized version, each entry is itself a *pair* (t_i, s_i) chosen according to the scheme in [AM89]. [AM89] proposed a scheme by which if several (say k) pairs (t_i, s_i) are randomly computed, there is a unique highest pair (lexicographically) with probability at least $\bar{\epsilon}$, where $\bar{\epsilon}$ is a constant independent of k. The number t_i is randomly selected according to the probability distribution $P(t_i = x) = 1/2^x$ and the number s_i is randomly uniformly selected from the range $[1, 20\ln(4r)]$ where $r = 1/\epsilon$ ($\bar{\epsilon} = 1 - \epsilon$; ϵ is the probability of *error* we are prepared to tolerate for a given collection of random t_is). For our purposes, we choose $\bar{\epsilon} = \epsilon = 1/2$.

Thus, an ID is a tuple $(a_1, \ldots, a_j)$ consisting of an arbitrary number j of *entries*. We impose a lexicographic order $\prec$ on IDs; this order is a total order. If $X \prec Y$ and X is a proper prefix of Y, we define the precedence to hold in the *weak sense*, or $X \overset{w}{\prec} Y$. Otherwise, if $X \prec Y$ and X is not a prefix of Y, we define the precedence to hold in the *strong sense*, or $X \overset{s}{\prec} Y$. As mentioned earlier, nodes *compete* with one another for being the root of the eventual spanning tree. The competition is on the basis of IDs; a higher ID "beats" a lower one. If two trees with the same ID detect the existence of each other, they need to break symmetry so that only one of the two advances in the competition. A highly desirable model to impose on this competition is the *tournament* model, to pick a unique winner starting with n competitors. As the tournament progresses, we have a *shrinking pool* of "candidates" for the eventual winner; once a player leaves the pool, it is out of the running.

Our definition of IDs and the ordering defined on them captures the tournament model. A node can only change its ID by *appending* an *entry* to it. When two equal IDs are independently extended in this manner, one of the new IDs is ordered higher than the other (if they are different). Further, note that the first ID is now higher in the *strong* sense: if both the changed IDs undergo further (possibly none) extensions, the first ID will remain higher even after additional extensions. The second, lower, ID can never compete with the first after this extension. Hence our shrinking pool of "candidate" IDs. On the other hand, if an ID X is higher than ID Y in the *weak* sense, it is still possible for Y, through some sequence of extensions, to eventually be higher than X in the strong sense. Thus a weak-sense relationship between two IDs implies that the IDs are not yet "differentiated" in the competition; any of them might eventually "beat" the other.

5 Definitions

In Section 2, we defined an *execution* of our protocol as an infinite sequence of *configurations* $C_1, C_2, C_3, \ldots$. Configurations are ordered chronologically in the sequence; we define the relation $\prec$ on configurations to be chronological order. Let the *round number* of a configuration be the number of the round it occurs in. We define the special configurations $R_1, R_2, R_3, \ldots$: configuration R_i is the *latest* configuration in the i'th round. Thus R_3 denotes the last configuration of the 3rd round, and so on.

We define $VAR(v, C)$ to be value of local variable VAR held by node v in configuration C. Thus *distance*(v,C) is the value of the *distance* variable in node v's memory in configuration C. $IDS(C)$ is the set of node IDs existing in configuration C. $ROOTS(C)$ is the set of root nodes in configuration C (i.e. for which *distance* $= 0$). The following additional terms are defined:

$CANDIDATE\text{-}IDS(C) : \{\, ID(V_i, C) \mid \neg(\, \exists\, I \in IDS(C) \mid I \overset{s}{\succ} ID(V_i, C))\,\}$, i.e the set of all IDS I_j existing in configuration C such that no other ID is greater than I_j in the strong sense. $CANDIDATE\text{-}ROOTS(C)$ is the restriction of this set to roots (*distance* $= 0$).

$MAXID(C)$: $(I \mid I \in IDS(C)$ and $I \succeq I' \forall I' \in IDS(C)$). This is the value of the maximum ID in configuration C. Note that this ID must be in $CANDIDATE\text{-}IDS(C)$. $MINID(C)$ is analogous to $MAXID(C)$.

6 The Randomized Algorithm for node v

Each node v maintains a set of *local variables* partitioned into eight *arrays* called *ID, distance, parent, color, direction, other-trees, recorded-color*$_{neighbor}$

and *recorded-color$_{self}$*. Each array is indexed from 0 through DEG$_v$, where DEG$_v$ is the *degree* of node v.

From node v's point of view, the most "important" variables are what we refer to as its *core* variables — all the eight variables indexed by 0 in its arrays. (Henceforth, we will drop the array index when mentioning values indexed by 0; thus we refer to *parent*[0] simply as *parent*). Values indexed by a nonzero index i store the most recently read values of the core variables of the neighbor numbered i at v.

Nodes maintain *ID*s; these IDs are *not* "hardwired". The *parent* variable at v is the number of a neighboring node (possibly itself); the set of *parent* variables at all nodes define a subset E_{parent} of the set of edges E. We attempt to make the *parent subgraph* $G_{parent} = (V, E_{parent})$ represent a *forest*; thus we attempt to make each node v belong to a *tree* T_v. The *distance* variable is an estimate of the distance from v to the root of its tree T_v (if such a tree exists).

```
DO PERIODICALLY:

    /* copy neighbor variables into local memory */
    LOOK-AT-NEIGHBORS,
    /* become child of neighbor with maximum priority, or become root */
    MAXIMIZE-PRIORITY,
    /* if local neighborhood "looks" stable, participate in recoloring etc.*/
    if $\forall j \in$ { 1,2, ...DEG$_v$ } , ID[j] = ID and | distance[j] - distance | $\leq$ 1,
    then DETECT-OTHER-TREES
    /* if root has detected other trees with same ID, extend ID */
    if (distance = 0 and direction = echo and other-trees = true)
    then EXTEND-ID
```

Fig. 1. Main Loop of Algorithm.

The protocol at each node is in the form of a *loop* executed infinitely often (Fig. 1). The procedure LOOK-AT-NEIGHBORS reads the values of each of its neighbors' core variables and copies it into the corresponding local "opinions". The procedure MAXIMIZE-PRIORITY (Fig. 2) makes v participate in the important task of tree overrunning; it sets the *ID, distance* and *parent* variables. The procedure DETECT-OTHER-TREES makes v participate in *recoloring* its tree to detect "competing" trees with the same ID. If v is a root node and the recoloring process has informed it of a "competitor" tree, v invokes the procedure EXTEND-ID to *extend* its ID randomly to break symmetry.

In action [A] of MAXIMIZE-PRIORITY, we make v determine the number l of its neighbor with the highest *priority* (defined in Section 3). Of course, many neighbors may all have the same highest priority; we break ties by choosing the highest-numbered neighbor. Action [B] determines whether node v can *increase*

```
MAXIMIZE-PRIORITY
    /* let $l$ be the largest index of all neighbors that have max priority */          [A]
    Let $l \leftarrow$ MAX { b | (ID[b], distance[b]) = $\mathrm{MAX}'_{j\in\{1,2,\ldots DEG_v\}}$ (ID[j], distance[j]) }
    (where MAX' is maximum over the relation $\prec$, defined in Section 2)

    /* if v can improve its priority, by becoming child of another */
    /* neighbor, do so, otherwise become root */
    if (ID[$l$] $\succ$ ID) or (ID[$l$] = ID and distance[l] < distance) /* see def. of $\succ$ */
    then                                                                              [B]
        ID          $\leftarrow$ ID[$l$]
        distance    $\leftarrow$ distance[$l$] + 1
        parent      $\leftarrow$ $l$
    else /* no neighbor has a larger priority; become root */                         [C]
        distance    $\leftarrow$ 0
        parent      $\leftarrow$ self        /* self = 0 */
```

Fig. 2. Procedure MAXIMIZE-PRIORITY

its priority by attaching to the "highest" neighbor l determined by action [A]. If the priority cannot decrease, it then makes the neighbor numbered l its *parent*, assumes its ID, and assumes its distance incremented by one. However, if node v can only decrease its priority by attaching to the neighbor numbered l, action [C] makes it become a root, keeping its ID unchanged and resetting its *distance* to zero. This is the mechanism of handling the *ghost root* problem—if node v notices that it was a nonroot node with a ID I_g that is *not* possessed by any of its neighbors and is higher than all its neighbors IDs, it was erroneously "believing" in the existence of a root node with ID I_g. In this situation, node v simply becomes a root with ID I_g by setting its *distance* to zero, thus obviating the need to "correct" erroneous belief in that root elsewhere in the network. Hence action [C] plays an important role in self-stabilization.

The details of procedure DETECT-OTHER-TREES are left to the full paper.

7 Correctness and Complexity Proofs for the Randomized Algorithm

We prove that starting from an arbitrary initial state of all local variables and program counters, the graph of *parent* pointers forms a spanning tree within expected O(d) time, and remains "fixed" ever after. The assumption of a *fair scheduler* allows us to partition the execution into an infinite number of *rounds* (defined in Section 2). We prove that the protocol converges to a spanning tree within $O(d)$ rounds; hence the time complexity is $O(d)$.

The main outline of the proof is as follows: We first prove that starting from an arbitrary initial state, the *parent* graph must always be a spanning forest after

2 rounds. We show that the set of root nodes in the forest can only diminish: roots can become non-roots, but not vice-versa. A key property of our algorithm (Ref. Section 4) is that the ID of the root node of the final spanning tree must belong to the set of IDs called CANDIDATE-IDS (defined in Section 5); once a root node's ID ceases to be a member of this set, it can never be the final rootid. We show that if in some configuration all the nodes in the forest (if there are ≥ 2 trees present) have the same ID, within (constant $\times$ diameter) rounds afterwards, *some* node will have extended its ID. Using this fact, we show that during each *epoch* of duration (constant $\times$ diameter) rounds, there is a constant probability of there being a unique root at the end of the epoch. Thus there exists a unique root node within the network at the end of expected O(diameter) rounds, and hence O(diameter) time.

The details of the proofs are left to the full paper; here we just state the main lemmas.

Lemma 1. *For any node v, the value of its (ID, distance) cannot decrease with time, i.e. (ID,distance) $(v,C_i) \preceq$ (ID,distance) (v,C_j) $\forall$ $C_j \succeq C_i$, $v \in V$.*

The following lemma and corollary show that after 2 rounds starting from the initial arbitrary state, the graph of the *parent* pointers is "consistent" in that a child always has a lower priority than its parent; thus the entire *parent* graph has no cycles, and must be a spanning forest.

Lemma 2. *Let the system be initialized in an arbitrary state, i.e. let the system be started with arbitrary values for the local variables. $\forall$ $C \succ R_2$, each node obeys the* distance invariant, *i.e.* $\forall u, v \in V$,
parent(u,C) = v $\Rightarrow$ (ID,distance)(u,C) $\prec$ (ID,distance)(v,C)

Corollary 3. *For any $C \succ R_2$, the graph of all parent(v,C) pointers defines a spanning forest.*

After the second round, the set of root nodes can only get smaller with time—a root may become a nonroot, but not vice-versa.

Lemma 4. *$ROOTS(C_j) \subseteq ROOTS(C_i)$ $\forall$ $C_j \succeq C_i \succ R_2$*

We now show that nodes must "learn" about "high" IDs existing in the network within *diameter* rounds—the highest ID in the network in some configuration R_i is no larger than the lowest ID in configuration $R_{i+diameter}$. In this sense, high IDs "overrun" lower IDs.

Lemma 5. *$\forall$ $v \in V$, $ID(v, R_{i+d}) \succeq MAXID(R_i)$.*

The following is a crucial property of our algorithm. To ensure fast progress, we want that if a root r_1 has an ID that is smaller than that of another root r_2, then the relationship will stay that way, even if the two roots never communicate directly. We can ensure this only if r_2's ID is higher *in the strong sense* — Corollary 6 below. However, later, in Lemma 12, we show (based on Lemmas 9-11) that even if the IDs are not related in the strong sense, then they become so in $O(d)$ time.

Corollary 6. *$\forall\ C_i \succ R_2,\ C_j \succ C_i$, if both r_1 and r_2 are roots in the interval $[C_i,\ C_j]$, then $ID(r_1, C_i) \overset{s}{\prec} ID(r_2, C_i) \Longrightarrow ID(r_1,\ C_j) \overset{s}{\prec} ID(r_2,\ C_j)$.*

We will show that the set CANDIDATE-IDS is the set of IDs that have a chance of "surviving" - a root not having an ID in this set will definitely be overrun by some other tree. We now have a "competition" between roots in the forest. The winner of the competition will be the root of the eventual spanning tree. The set CANDIDATE-ROOTS is the set of roots still in the fray; all other roots have "lost" and will be overrun. All roots change their IDs only by extension (unless they cease to be a root), and by changing their ID they may lose their membership in CANDIDATE-ROOTS.

Lemma 7. *$\forall\ C_i \succ R_2,\ C_j \succ C_i$, CANDIDATE-ROOTS($C_j$) $\subseteq$ CANDIDATE-ROOTS(C_i).*

Corollary 8. *If $r \in$ MAXROOTS(C_j), $r \in$ CANDIDATE-ROOTS(C_i) $\forall\ R_2 \prec C_i \prec C_j$.*

The set CANDIDATE-ROOTS of "candidates" for the root of the eventual spanning tree has been shown to *shrink* with time. However, "shrinking" of this set alone is not enough to get a good time complexity; the set must shrink *fast.* The following lemmas show that if we consider any *epoch* of k_1 * *diameter* rounds (where k_1 is some small constant), with at least a constant probability there will be just a single root node left in the set CANDIDATE-ROOTS at the end of the epoch.

Let the highest ID in some configuration C be I′. Then by Lemma 5, in some configuration C′ occuring within $O(diameter)$ rounds, *all* IDs in the network will be at least as large as I′. Thus we have two scenarios for C′: either some ID is greater than I′ in the strong sense, or all IDs are comparable to I′ only in the weak sense. Lemma 9 shows that if the first scenario holds, there will be a unique member in CANDIDATE-IDS with at least a constant probability.

Lemma 9. *Let $|$CANDIDATE-IDS(C)$| > 1$ for some $C \succ R_2$. Let $I' =$ MAXID(C). Let MINID(C') $\succeq I'$ and suppose $\exists\ I_s \in$ IDS(C') $\mid I_s \overset{s}{\succ} I'$, for some $C' \succ C$. Then with probability $\geq 1/2$, $|$CANDIDATE-IDS(C')$| = 1$.*

We now show that the "recoloring" approach – the procedure DETECT-OTHER-TREES – accomplishes its objectives. Specifically, consider a configuration in which all root nodes in the network (> 2 roots) have the same ID. Then, we show that within $k_1 \times$ *diameter* rounds, with constant probability, *some* root node will have extended its ID.

Lemma 10. *Let MAXID(R_i) = MINID(R_i), i.e. let all IDS be equal in configuration R_i, and let $|$ROOTS(R_i)$| > 1$. Then with probability $\geq p_1$, $\exists\ I \in$ IDS($R_{i+k_1 d}$) such that $I \overset{w}{\succ}$ MAXID(R_i), where*

$$p_1 \geq 1 - (\frac{1}{2})^{\lfloor k_1/6-1/3 \rfloor}$$

Lemmas 11 and 12 show that in any epoch of duration of (constant × *diameter*) rounds, with probability $\geq 1/2$ there is a single member in CANDIDATE-IDS at the end of the epoch.

Lemma 11. *Let $I' \overset{w}{\succeq} MINID(C') \forall I' \in IDS(C')$, for some $C' \succ C$. If $MINID(C') \overset{w}{\succ} MAXID(C)$, then with probability $\geq 1/2$, $|CANDIDATE\text{-}IDS(C')| = 1$.*

Lemma 12. *Let $|CANDIDATE\text{-}IDS(R_i)| > 1$. Then with probability $\geq p_1/2$, $|CANDIDATE\text{-}IDS(R_{t+(k_1+2)d})| = 1$, where p_1 and k_1 are related by Lemma 12, and d is the diameter of the network.*

Finally, the main result:

Theorem 13. *Starting from an arbitrary initial state, the graph of* parent *pointers forms a spanning tree in expected $O(d)$ rounds.*

Corollary 14. *The stabilization time of the protocol is O(diameter).*

8 The deterministic version

For our deterministic algorithm, we assume that each node has access to a "hard-wired" unique ID. We refer to the unique ID as the node's UID to prevent confusion with the nodes "other" ID, which is a tuple of entries as in the randomized case. The deterministic protocol is very similar to the randomized version. The main simplification, compared to the randomized version, arises in the method for recoloring trees. We no longer need random coin flips to break symmetry: the unique UIDs are exploited for fully reliable symmetry breaking. Each node, as before, has a *color*. However, the main difference is that trees do *not* need to be repeatedly recolored. The root of a tree always attempts to propagate *its UID as the color of its tree*, so nodes repeatedly copy their *parent*'s color. If a leaf notices a neighbor with the same ID but a different color, it can safely conclude that its neighbor belongs to a different tree, and informs its root through the *other-trees* variable which is echoed to its root by its ancestors in the tree. When a root detects the presence of a competing tree, it *appends its own UID to its ID*; this change in its ID is automatically propagated to its leaves.

References

[AG90] Anish Arora and Mohamed G. Gouda. Distributed reset. In *Proc. 10th Conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 316–331. Spinger-Verlag (LNCS 472), 1990.

[AKMPV93] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time Optimal Self-Stabilizing Synchronization. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, May 1993.

[AKY90] Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self-stabilization on general networks. In *Proc. 4th Workshop on Distributed Algorithms*, Italy, September 1990.

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

[Ang80] Dana Angluin. Local and global properties in networks of processes. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, May 1980.

[APV91] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *32nd Annual Symposium on Foundations of Computer Science*, pages 268–277, October 1991.

[AV91] Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *32nd Annual Symposium on Foundations of Computer Science*, pages 258–267, October 1991.

[Dij74] Edsger W. Dijkstra. Self stabilization in spite of distributed control. *Comm. ACM*, 17:643–644, 1974.

[DIM91] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform self-stabilizing leader election. In *Proc. 5th Workshop on Distributed Algorithms*, pages 167–180, 1991.