Time Optimal Asynchronous Self-stabilizing Spanning Tree

Janna Burman and Shay Kutten

Dept. of Industrial Engineering & Management
Technion, Haifa 32000, Israel
bjanna@tx.technion.ac.il, kutten@ie.technion.ac.il

Abstract. This paper presents an improved and time-optimal self-stabilizing algorithm for a major task in distributed computing- a rooted spanning tree construction. Our solution is decentralized ("truly distributed"), uses a bounded memory and is *not* based on the assumption that either **n** (the number of nodes), or **diam** (the actual diameter of the network), or an existence of cycles in the network are known. The algorithm assumes asynchronous and reliable FIFO message passing and unique identifiers, and works in dynamic networks and for any network topology.

One of the previous time-optimal algorithms for this task was designed for a model with coarse-grained atomic operations and can be shown not to work properly for the totally asynchronous model (with just "read" or "receive" atomicity, and "write" or "send" atomicity). We revised the algorithm and proved it for a more realistic model of totally asynchronous networks.

The state in the presented algorithm does not stabilize until long after the required output does. For such an algorithm, an increased asynchrony poses much increased hardness in the proof.

1 Introduction

A system that reaches a legal state starting from an arbitrary one is called *self-stabilizing* [15]. The *stabilization time* is the time from the moment of the faults till the system reaches a legal state.

The task of a directed spanning tree construction requires the marking, in each node, of some of the node's edges such that the collection of marked edges forms a tree. Moreover, we mark in each node the edge leading to its parent on the tree. Given a spanning tree, most of the major tasks for distributed network algorithms become much easier, including the tasks of reset, broadcast, topology learning and updating, mutual exclusion, voting, committing, querying, scheduling, leader election, and others.

In this extended abstract, we directly address only the task of constructing a spanning tree with **diam** height in $O(\mathbf{diam})$ time, but this, together with the method of [9], also yield an $O(\mathbf{diam})$ time reset algorithm. In the context of self stabilization, it was observed that a self stabilizing reset protocol can translate

A. Pelc (Ed.): DISC 2007, LNCS 4731, pp. 92–107, 2007.

a non-self stabilizing protocol into a self stabilizing one [4,9,6,7]. Another application of a reset protocol is to translate protocols that use unbounded event counters (e.g. sequence numbers of messages) to use bounded ones [8]. These applications of the reset simplify protocols' design.

An optimal self stabilizing algorithm for constructing a spanning tree was presented in [7]. As opposed to some previous protocols, it was not based on the assumption that either **n** (the number of nodes) or **diam** (the actual diameter of the network) or an existence of cycles in the network were known. It used a bounded memory. For that, it assumed that *some* bound on the diameter was known. This bound may have been very large, but it did not affect the time complexity which was $O(\mathbf{diam})$. The effect of the bound on the size of the memory was only polylogarithmic.

The algorithm of [7], however, was designed for a model with a coarse-grained atomicity. That is, it assumed that a node could read a value of a variable written by a neighbor, and also perform an operation based on that read value in one atomic step. Only after both actions, could the neighbor change the value of its own variable. We show a scenario where that algorithm does not yield a correct result in a totally asynchronous model, that is, when an atomic operation contains either a read, or a write, but not both.

In this paper, we modify the algorithm of [7] so that it functions correctly also in a fully asynchronous environment. We kept the main algorithmic ideas of [7], while adding a few tricks that may prove useful in translating also other such protocols. The main contribution of this paper is in the proof that the algorithm after the changes is correct for the more realistic *asynchronous send/receive atomicity* model (similar to "Atomic Read/ Atomic Write" model, but for a message passing model).

Note, that combining the algorithm in [7] with some existing efficient transformer (e.g. [10] or [26]) to refine the atomicity would not have yielded optimal stabilization times in our case. When such a transformer is combined with a coarse-grained atomicity algorithm, a resulted fine-grained atomicity algorithm suffers from a reduction in concurrency due to the mutual exclusion procedures used in the implementation of the transformer. This loss of concurrency results in a higher than a constant delay (up to $\Omega(n)$) between two successive atomic step executions by a particular process.

Numerous self stabilizing reset and spanning tree algorithms that were less efficient than [7] also appeared. We mention some of them below. Many (starting with [17]) championed the claim that a self stabilizing algorithm should use fine-grained atomic operations. We did not see how to use the methods used in [17] (for making their $O(\mathbf{n})$ time algorithm work under fine-grained atomicity) to make the algorithm of [7] also work for fine-grained atomicity. We note that moving from a coarse-grained atomicity to fine-grain atomicity of operations is even impossible for some tasks in some models, and for other tasks it is tricky. This is especially tricky for the algorithm of [7], since the latter keeps multiple trees that do not stabilize in the required time $O(\mathbf{diam})$, although the output in their algorithm does stabilize in $O(\mathbf{diam})$ time in the coarse-grained operations

model. The proof requires us to reason about these not yet stabilized trees (to show that they do not prevent also the output tree from stabilizing in $O(\mathbf{diam})$ time). Proving a property of a not yet stabilized tree is made more difficult by the asynchronicity. It would have been much easier to prove that some condition holds for a node in a tree if it was certain (as it is in a coarse-grain atomicity model) that this node has not lost its parent or children without yet knowing about the loss. The current paper makes this necessary step from coarse-grained atomicity to asynchronous networks for the major tasks it solves.

Other related work. Due to the lack of space, we give here a somewhat limited survey of the related work for the well-studied problem of a spanning tree construction. A thorough survey is deferred to the full paper [12].

In [3], a spanning tree construction algorithm with the stabilization time of $O(\mathbf{n}^2)$ (where $\mathbf{n}$ is unknown) is given. In [18], a randomized spanning tree protocol is given implicitly, with the expected stabilization time of $O(\mathbf{diam} \cdot \log \mathbf{n})$, where **diam** is unknown. (If $\mathbf{n}$ is known in advance then the stabilization time is $O(\mathbf{diam})$). In [5], the time complexity is $O(\mathbf{diam})$ (where **diam** is unknown), but the memory space (and the length of a message) is not bounded. In [2] and [19,13,14], generic self-stabilizing solutions solving also the task of a spanning tree construction are given. These papers present algorithms for weaker models (with unidirectional communication links and even with unreliable communications in [13,14]), but the time-optimal stabilization is not achieved in them. In [20], an algorithm for maintaining a spanning tree for a *completely connected topology* stabilizes in $O(log\mathbf{n}/loglog\mathbf{n})$ with high probability. In [24], a completely connected topology is assumed too, but the model is weaker than in [20]. In [25], a self-stabilizing algorithm for a minimum spanning tree construction is presented for an asynchronous message-passing reliable network. The time complexity in [25] depends on $\mathbf{n}$ (and hence, cannot be time-optimal). In addition, that algorithm of [25] requires a bound on the time that it takes to travel over a path of $\mathbf{n}$ nodes in the network. In [23] and [1], spanning tree algorithms for ad hoc networks with larger than $O(\mathbf{diam})$ stabilization times are given. For an additional survey one can refer to [21]. Finally, we note that the algorithm presented in this paper can be combined with some hierarchial structure (e.g. [22]) to improve the stabilization time in some favorable settings (see [22]), but not in the worst case.

1.1 Notations and Model of Computation

System Model. The system topology is represented by an undirected graph $G = (V, E)$, where nodes represent processors and edges represent communication links. Each node has a unique identifier denoted by *ID*. The number of the nodes is denoted by $\mathbf{n} = |V|$. The actual diameter of the graph is denoted by **diam**. We assume that there is a known upper bound on the diameter of the network, denoted by D and called the *bound*. This upper bound serves only for the purpose of having finite space protocols, and does not appear in the time complexity. For $v \in V$, we define $N(v) = \{u \mid (v, u) \in E\}$, called the *neighbors* of v. We assume that the topology is dynamically changing- node/link addition

or removal are possible (and modeled as faults). We consider an *asynchronous message passing network* model. The message delivery time can be arbitrary, but for the purpose of time analysis only, we assume that each message is delivered in at most one time unit. On each link, messages are delivered in the same order as they have been sent. The number of messages that may be in transit on any link in each direction and at the same time is bounded by some parameter B (independent of the network size). It is necessary, as shown in [16], for self stabilization.

We adopt the usual definitions of the following: a *local state* of a node (an assignment of values to the local variables and the program counter); a *global state* of a system (the cross product of the local states of all the nodes, plus the contents of the links); the semantics of protocol actions (possible atomic (computation) steps and their associated state changes); an *execution* of a protocol P (a possibly infinite sequence of global states in which each element follows from its predecessor by the execution of a single atomic step of P).

Informally, a distributed system allows the processors to execute steps concurrently; however, when processors execute steps concurrently, we assume that there is no influence of these concurrent steps on each other. To catch this formally, we assume that at each given time, only a single processor executes a step. Each step consists of an internal computation and a single communication operation: a *send* or *receive*. Every state transition of a process is due to a communication step execution (including the following local computations). Let us call this the *send/receive atomicity* network model.

Fault Model. The *legality predicate* (defined on the set of global states) of our protocol becomes true when the collection of internal variables of the nodes defines a spanning tree with **diam** height, rooted at the minimal *ID* node. A protocol is called *self-stabilizing* if starting from a state with an arbitrary number of faults (or from an arbitrary state) such that no additional faults hit the system for "long enough", the protocol reaches a *legal* global state eventually and remains in legal global states thereafter. When this happens, we say that the *stabilization* has occurred. The maximum number of time units that takes to reach the stabilization is called the *stabilization time* of the protocol.

2 The Algorithm

Due to the fine-grained communication atomicity, we use the notion of *base* and *image* variables. Each node v has a set of internal variables it writes to- the base variables. Consider some base variable of v, $\texttt{var}_v$. Every neighbor u of v maintains an internal copy of $\texttt{var}_v$ in its corresponding image variable $\texttt{var}_u[v]$ at u. The copies get their values from `InfoMsg` messages sent from v to u repeatedly. Node u reads $\texttt{var}_u[v]$ for algorithm computations. By then, this copy can have a different value then $\texttt{var}_v$ at v, if v has changed it meanwhile. This is the main difficulty encountered by the current paper, as opposed to [7].

2.1 Ideas Adopted from [7]

The algorithm runs multiple versions of the Bellman-Ford's algorithm ([11], see Rule 1 bellow). When running a single version alone, a stabilized tree results (this was observed by [17] for their algorithm which is similar to Bellman-Ford's, and by [7] and others for Bellman-Ford's algorithm itself). The stabilized tree is rooted at the minimal *ID* node in the network and the stabilization occurs in $O(D)$ time when D is given. This is similar e.g. to [6,17]. Therefore, if the bound parameter D is close to the actual diameter **diam**, Rule 1 is close to optimal. Unfortunately, typically, a hardwired bound will be much larger than the actual diameter, to accommodate for extreme cases, and to have room for scaling up the network size. Nevertheless, coming up with *some* bound is pretty easy. Hence, $\log D + 1$ versions are run in parallel. Each version i, $0 \leq i \leq \log D$, executes the Bellman-Ford version with bound parameter 2^i. Versions with "large enough" bound parameters 2^i (the "higher versions") will stabilize to a desired spanning tree in $O(2^i)$ time. However, the other versions (the "lower versions") can stabilize only in $\Theta(\mathbf{n})$ as demonstrated in [7]. The trick there was that one does not need the smaller i versions to stabilize. One only needs them to detect and inform each node that the version has not yet constructed the required spanning tree which contains all the nodes. This is done by the standard technique of a broadcast over a tree. That is, each version i maintains at each node two bits: `up_cover` and `down_cover`. Using the `up_cover` bit, nodes report towards the root that the version tree has not yet spanned all the nodes; this information is propagated up the tree by having each node take a logical *and* of the `up_cover` bits of its children repeatedly. The purpose of the `down_cover` bit is to disseminate this fact down the tree, by having each node copy the `down_cover` bit of its parent repeatedly. See Rule 2.

Then, each node v selects its output by finding the minimum i such that $\texttt{down\_cover}_v = 1$ for version i. The tree edges of that version are the output of the combined protocol.

A Counter Example. In the sequel we mention some problems that prevent one from using the solution of [7] in the weaker model we address in this paper. This is demonstrated in [12], because of the lack of space here. We just mention here that the problems manifest in the trees that have not stabilized yet, when the output was supposed to have stabilized already. In an asynchronous network, such a not yet stabilized tree may "pretend" successfully to have stabilized, and to cover the network.

2.2 Revising the Algorithm

Now, we introduce three mechanisms we incorporate in the new algorithm to make the ideas above also work in the weaker model we use.

• **Non-stabilization Detector:** The algorithm of [7] propagated up a tree the information that a node in that tree has a neighbor not in the tree. This information turned out to be unstable, causing the counter example mentioned above. Hence, we augment the above with a new notion of local "non-stabilization detector": if it looks as if any of the neighbors of a node v may still have to

change its state, then v observes that the current configuration is not yet stable, and propagates this observation upwards. Below, we give the formal definition of this idea- the $\texttt{consistent}_v$ predicate for each node v (see Def. 1). Now, in contrast with the previous solution, the definitions of the rules for the `up_cover` and `down_cover` bits use the `consistent` predicate. They are given in Rule 2.

• **Strict Communication Discipline:** We adopt this module as is from [4] (see Sec. 2.3 below). The main ***property of the discipline*** is that before a node v may change any of its base variables, all its neighbors "know" the value of v's base variables (see Lem. 3). The proof of Lem. 4 uses this property heavily.

• **Local Reset:** We use this mechanism to ensure the following. Whenever a node v changes the values of its tree variables (line 6, Fig. 1), no neighbor u considers v as its parent (that is, $\texttt{par}_u \neq v$). Note, that in our model this is not a condition that v can check directly, since by the time v reads u's variables, u may have changed them. We found this property very useful in several places in the proof. For example, whenever a node joins a tree on some specific path called a *legal branch* (see Def. 4), it joins initially with $\texttt{up\_cover} = 0$ (Lem. 7). This helps to prove that even though the lower versions trees do not stabilize (at least, not in $O(\mathbf{diam})$ time), their `up_cover` does stabilize to zero (Lem. 8). Lines 6-11 (Fig. 1) implement the local reset mechanism.

Intuitively, the main difficulty the revised algorithm overcomes is the following. Consider a branch of a tree whose nodes have a root value of v, however, they are disconnected from v. (A formal definition of this structure we call a "sprig" will follow (Def. 3).). Note, that v does exist. This phenomena is different than branches of a tree of a ghost root- that is, branches with nodes whose root variable contains an *ID* of a node that does not exist. The latter branches disappear by time $T_{valid} = O(2^i)$ for version i (see Def. 2 and Obs. 1), but new sprigs can be created by the lower versions of the algorithm up until time $\Omega(\mathbf{n})$. We needed the above revisions in the algorithm to show that the sprigs do not confuse themselves to have $\texttt{up\_cover} = 1$ (Lem. 10). Nor do they confuse the nodes of the legal tree of v (nodes that are indeed connected (via a chain of parent markings) to v where v is the value of their root variable). See Lem. 8.

2.3 Algorithm Details

Variables. Graph property variables are represented using a **boldface font**, as in $\mathbf{dist}(v, u)$, which denotes the true distance in the graph between nodes v and u. Protocol variables are represented using `teletype font`. The variables appearing and manipulated in the code of Fig. 1 are local protocol variables of process v. The subscript v of each variable in the figure emphasizes that fact.

Each node $u \in V$ sends a set of values of its internal variables (its *base variables*) periodically to all its neighbors by an $\texttt{InfoMsg}_u$ message. Each neighbor v of u copies these values to its local copies (to simplify the code, we omit this

simple operation from the algorithm code, but we assume it is performed for each message receive event). The copy of a variable $\mathtt{var}_u$ at v (an *image variable* of u at v) is denoted by $\mathtt{var}_v[u]$. Node u does not send its neighbors any of its image variables.

Note, that for each $v \in V$, in addition to $N(v)$, we use a variable $\mathtt{Nlist}_v$ which 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.

A detailed verbal explanation of all the variables appears in [12].

Communication Discipline. Every node sends the `InfoMsg` messages repeatedly, using the communication discipline mentioned above. We treat the discipline as being embodied by a lower layer process. Whenever any `InfoMsg` arrives at some node v, the discipline (layer) copies the content of the message into the appropriate image variables. Then, the communication discipline can decide either to pass the message to the higher level algorithm or to discard it. If it decides to pass the message, we say that `InfoMsg` is *accepted* (then, this message is processed according the code in Fig. 1). Otherwise, we say that `InfoMsg` is *received* (and no further processing takes place). Note, that in both cases, v copies the message content (*receives* `InfoMsg`). For the complexity analysis, we define a *time step*, which is the maximum time for a message to get accepted (it is equivalent to 3 time units). Lem. 3 states the communication discipline property formally.

Algorithm Formal Definition. The formal code for version i of the algorithm in node v appears in Fig. 1; it applies Rules 1 and 2 below. In each iteration, node v outputs its tree edges according to the lowest version in v for which $\mathtt{up\_cover} = 1$.

Rule 1. *(Bellman-Ford with* ID*s and Bound Parameter* D *(used in [11,17,6] for a single version)) Let v be a node.*

1. $\mathtt{r}_v \leftarrow \min\{\mathrm{ID}_v, \mathtt{r}_v[u]\}$ *where* $u \in \mathtt{Nlist}_v$ *and* $\mathtt{d}_v[u] < \mathrm{D}$.

2. $$\mathtt{d}_v \leftarrow \begin{cases} 1 + \min\{\mathtt{d}_v[u] \ : u \in \mathtt{Nlist}_v \text{ and } \mathtt{r}_v = \mathtt{r}_v[u]\}, & \text{if } \mathtt{r}_v \neq \mathrm{ID}_v \ , \\ 0, & \text{if } \mathtt{r}_v = \mathrm{ID}_v \ . \end{cases}$$

Definition 1. *(Used in Rule 2 below)*

- *The* parent *of a node v, denoted* $\mathtt{par}_v$*, is (supposed to be) the smallest* ID *neighbor u of v for which* $\mathtt{r}_v = \mathtt{r}_v[u]$ *and* $\mathtt{d}_v[u] = \mathtt{d}_v - 1$*, if* $\mathtt{r}_v \neq v$*. Otherwise, if* $\mathtt{r}_v = v$*, the parent of v is* `null`*.*
- *The* children *of v, denoted* $\mathtt{child}_v$*, are the set* $\{u \mid \mathtt{par}_v[u] = v\}$*.*
- *Given a node v, the predicate* $\mathtt{consistent}_v$ *is defined by*
  $\bigwedge_{u \in \mathtt{Nlist}_v} (\mathtt{r}_v[u] = \mathtt{r}_v) \wedge (|\mathtt{d}_v[u] - \mathtt{d}_v| \leq 1) \wedge$
  $(u = \mathtt{par}_v \iff u$ *is the minimal* ID *node such that* $\mathtt{d}_v - \mathtt{d}_v[u] = 1) \wedge$
  $(\mathtt{d}_v[u] - \mathtt{d}_v = 1 \implies \mathtt{par}_v[u] \leq v)$
- *Node v is a* consistent *node, if* $\mathtt{consistent}_v = \mathbf{true}$*.*

Rule 2

Calculate $\texttt{par}_v$, $\texttt{child}_v$, *and* $\texttt{consistent}_v$ *such that they conform to Def. 1.*

$$\texttt{up\_cover}_v \leftarrow \begin{cases} \texttt{consistent}_v, & \textit{if } \texttt{child}_v = \emptyset \\ \bigwedge_{u \in \texttt{child}_v} \texttt{up\_cover}_v[u] \ , & \textit{if } \texttt{child}_v \neq \emptyset \wedge \texttt{consistent}_v \\ 0, & \textit{otherwise} \end{cases}$$

$$\texttt{down\_cover}_v \leftarrow \begin{cases} \texttt{down\_cover}_v[\texttt{par}_v] \wedge \texttt{consistent}_v, & \textit{if } \texttt{r}_v \neq v \\ \texttt{up\_cover}_v \wedge \texttt{consistent}_v, & \textit{if } \texttt{r}_v = v \end{cases}$$

Procedure Send() (* sending $\texttt{InfoMsg}_v$ *)

1 Send $\texttt{InfoMsg}_v \equiv [\ \texttt{r}_v, \texttt{d}_v, \texttt{par}_v, \texttt{local\_reset}_v, \texttt{up\_cover}_v, \texttt{down\_cover}_v\ ]$ to $\texttt{Nlist}_v$

Procedure LocalReset() (* performing a local reset *)

2 $\texttt{r}_v \leftarrow v$, $\texttt{d}_v \leftarrow 0$, $\texttt{down\_cover}_v \leftarrow \texttt{up\_cover}_v \leftarrow 0$, $\texttt{local\_reset}_v \leftarrow$ **true**

3 *Send*()

Do forever:

4 *Send*() (* this line is executed atomically *)

Upon accepting $\texttt{InfoMsg}_v[u]$ *message from neighbor* $u \in \texttt{Nlist}_v$

(* the following is executed atomically (not including reception) *)

5 Use Rules 1, 2 to calculate temporary variables as follows:
a temporary variable $\texttt{t\_var}_\texttt{v}$ for each $\texttt{var}_\texttt{v}$ computed in the rules.

(* tree changes generate a local reset *)

6 **if** $[\ \neg\texttt{local\_reset}_v \wedge (\texttt{par}_v \neq \texttt{t\_par}_v \vee \texttt{r}_v \neq \texttt{t\_r}_v \vee \texttt{d}_v \neq \texttt{t\_d}_v)\ ] \bigvee$

(* reset propagates down the tree *)

7 $[\ \exists u \in \texttt{Nlist}_v : \texttt{local\_reset}_v[u] \wedge u = \texttt{par}_v\ ] \bigvee$

(* local reset not yet completed *)

8 $[\ \texttt{local\_reset}_v \wedge \exists u \in \texttt{Nlist}_v : \texttt{par}_v[u] = v\ ] \bigvee$

(* candidate parent reset not yet completed *)

9 $[\ \texttt{local\_reset}_v \wedge \texttt{t\_par}_v \neq \texttt{null} \wedge \texttt{local\_reset}_v[\texttt{t\_par}_v]\ ]$

then

10 *LocalReset*()

11 *return*

end if

(* a local reset exit (if $\texttt{local\_reset}_v$ switches from **true** to **false**) *)

12 $\texttt{local\_reset}_v \leftarrow$ **false**

13 Update each variable $\texttt{var}_\texttt{v}$ by the value of the temporary variable $\texttt{t\_var}_\texttt{v}$.

14 *Send*()

Fig. 1. Algorithm for version i at node v

3 Preliminary Analysis

In the following analysis, we prove stabilization assuming that there are no faults or topological changes after some time $t_0 = 0$ (at least till the time when algorithm reaches a global legal state).

From now on, let us consider the execution of the algorithm after time $t = 2$ (2 *time steps* after time t_0). It is easy to see that after 2 time steps, no damaged (by faults) messages exist in the network and at least one authentic `InfoMsg` message has been *accepted* at each node from each neighbor.

Definition 2. *Consider a node* $v \in \mathrm{V}$ *and some time* T.
- *Let* T_{valid} *be time* $t_0 + 2 + 2^i$.
- *Node* v *is a* root node, *if* $\mathtt{r}_v = \mathrm{ID}_v$ *(*$\equiv v$*).*
- *If* $\mathtt{r}_v \notin \{\mathrm{ID}_u | u \in \mathrm{V}\}$, *then* $\mathtt{r}_v$ *is a* ghost root.
- *Let* $u \in \mathrm{N}(v)$. *If at time* T, $\mathtt{var}_u[v] = \mathtt{var}_v$, *we say that node* u knows *the value of* $\mathtt{var}_v$ *at* T.
- *The* depth *of* v *is* $\mathbf{depth}(v) \stackrel{\text{def}}{=} \max\{\mathbf{dist}(v,u) \mid u \in V\}$.
- *Let us denote by* v_{min} *the node with the minimum* ID *that exists in the network.*
- *Let us denote by a* local reset operation *of node* v *an invocation of the* $LocalReset()$ *procedure at line 10 of the algorithm code.*
- *Assume that node* v *performs a local reset operation at time* T. *We denote by a* local reset exit *the first time after time* T *when* v *executes line 12 of the algorithm code.*
- *Let us denote by a* local reset mode *the state of node* v *between a local reset operation and the subsequent local reset exit at* v.

Lem. 1 is a rather known property of Bellman-Ford's algorithm. Its proof, as well as some of the following proofs, due to the lack of space, appear in [12].

Lemma 1. *Starting from any initial assignments of the* `d` *and* `r` *variables, for any node* v, *after* t *time steps,* $\mathtt{d}_v \geq min(t, \mathbf{dist}(\mathtt{r}_v, v))$.

The following observation follows from Lem. 1 and Rule 1-1. (Note, that for every ghost root v, $\mathbf{dist}(\mathtt{r}_v, v) = \infty$.)

Observation 1. *After time* T_{valid}, *for every node* v, $\mathtt{r}_v$ *is not a ghost root and* $\mathtt{d}_v \geq \mathbf{dist}(v, \mathtt{r}_v)$.

Definition 3. *Let* $0 \leq k \leq \mathbf{n} - 1$. *Let* $v, x_j \in \mathrm{V}$ *for each* $0 \leq j \leq k$.
- *A* parent path of v to x_0 *is a path of nodes* $(x_0, x_1, x_2, \ldots, x_k = v)$ *such that for each* j, $\mathtt{r}_{x_j} = \mathtt{r}_v$ *and for each* $1 \leq j \leq k$, $\mathtt{par}_{x_j} = x_{j-1}$.
- *We say that* v *is a* descendant *of* x_0 *and* x_0 *is an* ancestor *of* v.
- *A* connection path of v *is a parent path of* v *to* $\mathtt{r}_v = x_0$ *such that* x_0 *is a root node. Let us denote a connection path of* v *(to* x_0*) by* $\mathbf{C}_v(x_0)$.
- *Node* v *is* connected (to node x_0) *if there is a connection path of* v *(to* x_0*).*
- *Let* $\mathtt{r}_v = x_0$. *If node* v *has no connection path, node* v *is* disconnected (from node x_0).
- *Let a* sprig *of* v *be a maximal set of nodes* $X \subset \mathrm{V}$ *that satisfies the following*

conditions: (1) $v \notin X$; (2) $\forall x \in X$, $\mathtt{r}_x = v$ and x is disconnected; (3) Every ancestor or descendant x of any $x_0 \in X$ is in X.
• *Let us denote some sprig A of v at time t by $A_v(t)$.*

The following is one of our main lemmas. Informally, we found it harder to ensure properties for sprigs than for legal trees. We use the following lemma to show a property of sprigs that is somewhat similar to the property of the d values in a legal tree (shown in Lem. 1). Informally, the lemma shows that a node's d is high if it remains in the same sprig for a long time (an "old" sprig). This may not hold for a node who leaves one sprig (having a high d) and joins another (with a lower d). Such a leave and join may happen even very late in the execution since new sprigs may be created very late in the execution. However, we handle such "new" sprigs later.

Lemma 2. *Consider nodes $v, u \in$ V. Let $t \leq 2^i$ and $t_1 > 2$. Let α be the following set of assumptions on u: (1) node u is disconnected from $\mathtt{r}_u = v$; (2) node u does not change its $\mathtt{r}_u$ value. If α holds for u during the whole time interval $[t_1, t_1 + 2t]$, then at time $t_1 + 2t$, $\mathtt{d}_u \geq t$.*

Proof: By induction on the time t. For $t = 0$, the lemma holds since $\mathtt{d}_u \geq 0$ always. Assume, that the lemma holds for $t = k$ for every node. For $t = k + 1$, assume that α holds for some node u for the time interval $[t_1, t_1+2(k+1)]$. By Def. 3, node u is disconnected and is not a root. Hence, $\mathtt{par}_u \neq \mathtt{null}$. When α holds, node u cannot change its $\mathtt{par}_u$. Otherwise, the condition at line 6 must hold beforehand and then, u must perform a local reset and assign $\mathtt{r}_u \leftarrow u$ (becoming a root) in line 2- a contradiction to α. Hence, for some node $w \in \mathtt{Nlist}(u)$, $w = \mathtt{par}_u$ throughout $[t_1, t_1 + 2k + 2]$.

Let δ be an InfoMsg message sent by w at time t_{sent} and accepted at u at time t_{rcv} such that δ is the last such message accepted at u by time $t_1 + 2k + 2$. By our model assumptions on communication links , $t_{rcv} \geq t_1 + 2k + 1$. Hence, δ is sent by w at $t_{sent} \geq t_1 + 2k$. Clearly, α holds at w during $[t_1, t_{sent}]$, otherwise it would not have held in u during $[t_1, t_{rcv}]$ since in that case, either u would have been connected (if w would have been connected) or some $\mathtt{InfoMsg}_w$ and line 6 of the algorithm would have caused u to perform a local reset and become a root, violating α. Hence ,by the induction hypothesis, $\mathtt{d}_w \geq k$ at time $t_1 + 2k$.

Moreover, if $\mathtt{d}_w$ is changed at w in $[t_1, t_{sent}]$, then $\mathtt{InfoMsg}_w$ stating this fact and line 6 of the algorithm would have caused u to perform a local reset and become a root, violating α. Thus, $\mathtt{d}_w$ stays unchanged during $[t_1, t_{sent}]$. Hence, $\mathtt{d}_w \geq k$ during the whole interval $[t_1, t_{sent}]$ and $\mathtt{d}_u[w] \geq k$ at u during the whole of $[t_1, t_1 + 2k + 2]$. Now, since $\mathtt{par}_u = w$ during whole $[t_1, t_1 + 2k + 2]$, by Rule 1, $\mathtt{d}_u = \mathtt{d}_u[w] + 1$ during whole this interval. Thus, at time $t_1 + 2k + 2$, $\mathtt{d}_u = \mathtt{d}_u[w] + 1 \geq k + 1$. Hence, the lemma holds for $t = k + 1$ too. ∎

The following lemma is ensured by the communication discipline, which we assume for the algorithm (Sec. 2.3). The lemma is proved in [7]- Lem. 4.2 (we reworded it somewhat).

Lemma 3. [7] *If at some time T, after time $t_0 + 2$, some node v changes any of its base variables, then, at time T, $\forall u \in \mathtt{Nlist}(v)$, for every base variable $\mathtt{var}_v$*

of v, $\texttt{var}_u[v] = \texttt{var}_v$ *(*u*'s image variable of* $\texttt{var}_v$ *equals* $\texttt{var}_v$ *at time* T*; or in other words, node* u *knows the value of* $\texttt{var}_v$ *at time* T*).*

The following lemma can be proven for our algorithm, but not for the previous one [7]. This proved to be a major reason why our algorithm can function in the atomic send / atomic receive model.

Lemma 4. *Assume that at some time* T*, some node* u *with* $\texttt{par}_u \neq w$ *assigns* $\texttt{par}_u \leftarrow w$*. Then, at time* T*,* $\forall v \in \texttt{Nlist}(u)$*,* $\texttt{par}_v \neq u$*.*

Proof: (see Fig. 2) To prove the lemma, we assume, by way of contradiction, that some neighbor v of u assigns[1] $\texttt{par}_v \leftarrow u$ before or at time T and $\texttt{par}_v = u$ holds till time T inclusively.

To assign a new value to $\texttt{par}_u$, node u must execute line 13 at time T (also see line 5 and Rule 2). This requires that just before that, u was at a local reset mode and performed a local reset exit at line 12. Since u changes a base variable at time T, by Lem. 3, every neighbor v' of u (and v in particular) knows at time T that u is in a local reset mode ($\texttt{local\_reset}_{v'}[u] = \textbf{true}$). Let time T^* be the first time when v assigns $\texttt{local\_reset}_v[u] \leftarrow \textbf{true}$ such that $\texttt{local\_reset}_v[u]$ stays true until T inclusively. Recall, that we consider the execution of the algorithm after time t=2. Thus, due to line 9, it is guaranteed that v does not *change* its parent pointer to point at u (v cannot perform $\texttt{par}_v \leftarrow u$) throughout $[T^*, T]$. (A local reset operation and then its exit must precede any tree structure base variables change at any node; but, a local reset exit is impossible at v in $[T^*, T]$, since the condition at line 9 holds throughout this time interval.)

Let T^x be the time when v sends message $\texttt{InfoMsg}_v\ x$ that is the last one to be *received* by u from v before time T. This message must cause u to set $\texttt{par}_u[v] \neq u$, otherwise, by the condition at line 8, u cannot become a descendant of w at time T. Hence, at time T^x, $\texttt{par}_v \neq u$ holds at v. Thus, and due to the guarantee of the time interval $[T^*, T]$ that we have shown above, we must assume that v assigns $\texttt{par}_v \leftarrow u$ (at line 13) in the time interval (T^x, T^*). When this happens, v sends an $\texttt{InfoMsg}_v\ (= y)$ (line 14) with this new information ($\texttt{par}_v = u$) to u.

Recall, that by Lem. 3, node u learns at some time $T^* < T^{**} < T$ that its neighbor v knows that u is in a reset mode (at T^{**}, u learns that $\texttt{local\_reset}_v[u] = \textbf{true}$). Clearly, for u, to learn this, a message of the communication discipline should be sent from v at or after time T^* and should be received at u before time T. Let z be such a message. We proved above that message y is sent before T^*. Hence, and because of the FIFO assumption for each link, message y is *received* at u before message z, and thus, before time T^{**} and before T. Since InfoMsg message y is sent after InfoMsg message x, it should arrive at u after x, but we have assumed above that x is the last InfoMsg to be received from v just before time T. Thus, $y = x$ - a contradiction since these messages bear different information about the value of $\texttt{par}_v$. ∎

[1] The algorithm calculates the par pointers by invoking Rule 2 at line 5 and then assigns them at line 13 (Fig.1).

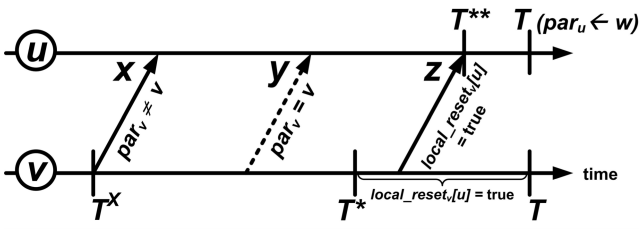

Fig. 2. Illustration for the proof of Lem. 4

4 Analysis of the Lower Version Case ($2^i < \mathbf{depth}(v_{min})$)

In this section we prove for every node in every lower version that `down_cover` $= 0$ holds in $O(2^i)$ time steps and remains thereafter. First, we prove this for legal trees and then for sprigs. To prove this for legal trees, we use the notion of a *legal branch*. There may be several shortest pathes between two nodes, however, only one of them is a *legal branch* as defined in the following definition. (This matches the way parent pointers are chosen by the algorithm. See Def. 1.)

Definition 4. *Let* $u, v \in \mathrm{V}$.
• *Node* u *is* foreign to node v *if after time* T_{valid}, $\mathtt{r}_u \neq v$ *always.*
• *Let* u *be foreign to* v. *A* legal branch $\mathbf{R}_v(u)$ of v via u *is a shortest path between* v *and* u $(x_1 = v, x_2, \ldots, x_k = u)$ *s.t.* $1 \leq k \leq \mathbf{dist}(v,u)+1$ *and for each* $2 \leq j \leq k$, x_{j-1} *is the smallest* ID *neighbor of* x_j *with* $\mathbf{dist}(v, x_{j-1}) = \mathbf{dist}(v, x_j)-1$. *Denote the length of* $\mathbf{R}_v(u)$ *by* $|\mathbf{R}_v(u)|$.

Lemma 5. *If* $2^i < \mathbf{depth}(v_{min})$, *then after time* T_{valid}, *for each node* v, *there exists a node* u *such that* u *is foreign to* v *and either (1)* $\mathbf{dist}(v,u) = 2^i + 1$ *or (2)* $\mathbf{dist}(v,u) < 2^i + 1$ *and* $v > u$.

Definition 5. *Let* $v, w \in \mathrm{V}$.
• *Let* $f_v = u$ *for some* u *as defined in Lem. 5.*
• *A* zero path *of node* v, *denoted by* $\mathbf{Z}_v$, *is a maximal parent path of nodes* $\{x | \exists \mathbf{C}_x(v) \wedge x \in \mathbf{R}_v(f_v)\}$. *A* fringe node of $\mathbf{Z}_v$ *is* $w \in \mathbf{Z}_v$ *which is the furthest node (in the number of hops) from* v. *Then, we also denote a zero path of* v *by* $\mathbf{Z}_v(w)$ *and its length by* $|\mathbf{Z}_v(w)|$ *(or* $|\mathbf{Z}_v|$*).*

Note, that every root node has a zero path (possibly containing only a root). A zero path may change dynamically in time. Nodes may join and leave a zero path. A zero path may "disappear" (if a root node stops to be a root) and "reappear" (if a node becomes a root again). The set of a zero paths stabilizes only in $\Omega(\mathbf{n})$ in some cases. Yet, by the following lemmas 6 to 8, we show that `up_cover` $= 0$ holds at every node of every zero path that exists after a certain time that is $O(2^i)$ time after the starting time t_0. The proofs of lemmas 6 - 8 establish an induction on the order of the nodes on a zero path, starting from a fringe node neighboring to a foreign node and proceeding to the root. This

induction shows that up_cover stabilizes fast to zero over each legal tree (one that has a root, as opposed to a sprig that is disconnected from its root). Let us comment that we needed to introduce the notion of a legal branch for these proofs. Moreover, we needed to add the check (in the consistent predicate, in Rule 2) that a branch is indeed legal when up_cover is updated. The proofs of lemmas 6 - 8 appear in [12].

Lemma 6. *Let $v, u \in \mathrm{V}$ and let u be a fringe node of $\mathbf{Z}_v$ (a zero path of v), at some time T after T_{valid}. Then, if u stays a fringe node of $\mathbf{Z}_v$, in at most 1 time step, at time $T \leq T1 \leq T+1$, $\mathtt{up\_cover}_u \leftarrow 0$.*

Lemma 7. *Let $v, u \in \mathrm{V}$ and assume that u joins a $\mathbf{Z}_v$ (a zero path of v) at some time T after T_{valid} (u was not in $\mathbf{Z}_v$ just before T). Then, at time T, $\mathtt{up\_cover}_u \leftarrow 0$.*

Lemma 8. *Let $v, u, x \in \mathrm{V}$ and $\mathbf{Z}_v(x)$ be a zero path of v. Let $u \in \mathbf{Z}_v(x)$ be such that $\mathbf{dist}(v,u) = \mathbf{dist}(v,f_v) - j$ (for some $1 \leq j \leq \mathbf{dist}(v,f_v) \leq 2^i + 1$). Then, after $T_{valid} + j$ time steps, $\mathtt{up\_cover}_u = 0$.*

Let T_{ccover} be the time that is $T_{valid} + 2 \cdot 2^i$ time steps after time t_0. By lemmas 8 and 4 and Rule 2, it is easy to show that down_cover also stabilizes fast to zero on legal trees. Thus, Lem. 9 also follows.

Lemma 9. *After T_{ccover} time for any connected node $u \in \mathrm{V}$ the following holds:* $\mathtt{down\_cover}_u = 0$.

To conclude the analysis for versions for which $2^i \leq \mathbf{depth}(v_{min})$, we need to show that for every node u that is *disconnected* from $\mathtt{r}_u$ (u is a sprig node), $\mathtt{down\_cover}_u = 0$ holds in $O(2^i)$ time steps and remains such thereafter too. First, let us formalize all the possible modifications that a sprig can encounter. Note, that there may exist several sprigs of v at the same time.

Definition 6. *Consider a sequence of events in the execution, and let t_j be the time of the j-th event in the sequence. Consider some sprig $A_v(t_j)$. At time t_{j+1}, we consider sprigs that are non-empty and that have non-empty intersection of nodes with the original sprig $A_v(t_j)$. That is, we consider all the possible modifications of A_v at time t_{j+1}:*
(1) Sprig $A_v(t_{j+1}) \neq \emptyset$ is one of the following:
– $A_v(t_j)$.
– $A_v(t_j) \cup \{x\}$ (a join *node) for some $x \notin A_v(t_j)$. Note, that by Lem. 4, x has no descendants at t_{j+1}. (Also, w.l.o.g., no two events happen at the same time.)*
– $A_v(t_j) \setminus \{y\}$ (a loss *node) for some $y \in A_v(t_j)$.*
(2) Non-empty sprigs $A1_v(t_{j+1}), A2_v(t_{j+1}), \cdots$ such that $A1_v(t_{j+1}) \cup A2_v(t_{j+1}) \cup \cdots \{z\} = A_v(t_j)$ (a split *sprig A_v) for some node z who left sprig A_v at t_j.*

Lemma 10. *Let $u \in \mathrm{V}$ be disconnected from $\mathtt{r}_u$ at some time after $T_{ccover} + 2 \cdot 2^i + 1$. Then,* $\mathtt{down\_cover}_u = 0$.

Proof: Any disconnected node u belongs to some sprig. Let $\mathtt{r}_u \equiv v$. Only two kinds of sprigs exist after time T_{ccover} by Obs. 1:

(1) The old sprigs- these that already exist at T_{ccover}. We consider them old even when they get modified later. Moreover, if $A_v(t_j)$ is an old sprig at some time $t_j \geq T_{ccover}$, then any resulting sprig $A_v(t_{j+1})$ (see Def. 6) at time $t_{j+1} > t_j$ is an old sprig too.
(2) The new sprigs- those that are newly created after T_{ccover} by the following event. Let $w \in V$ be a root node and X be a set of the nodes connected to w (X is a tree rooted at w). When some node $x \in X$ (not a leaf node) leaves X, a new sprig (or sprigs) of w is (or are) created. When a new sprig gets modified, the resulting sprig (or sprigs) is (or are) considered a new sprig (or sprigs).

First, let us consider a set of the old sprigs Φ after T_{ccover}. We show that after at most $2 \cdot 2^i + 1$ time steps $\Phi = \emptyset$. By Lem. 2, in at most $2 \cdot 2^i$ time steps, for any disconnected node $x \in B \in \Phi$, $\mathtt{d}_x \geq 2^i$. Hence, x leaves sprig B in at most additional 1 time step by Rule 1-1, if x has not left sprig B before that. Any node $y \notin B \in \Phi$ that joins sprig B, assigns $\mathtt{d}_y \leftarrow \mathtt{d}_y[z] + 1$ for $z \in B$ such that $\mathtt{par}_y \leftarrow z$. Thus, by this and Lem. 2, for any $x \in B \in \Phi$, at time $T_{ccover} + 2t$, $\mathtt{d}_x \geq t$. Hence, after at most $T_{ccover} + 2 \cdot 2^i + 1$, $\Phi = \emptyset$.

Finally, let us consider a new sprig A that is newly created at some time $T > T_{ccover}$. By Lem. 9, at time T for each $x \in A$, $\mathtt{down\_cover}_x = 0$. After time T sprig A can change in time: split or join/loose nodes. Note, that no sprig has a root node and thus, by Rule 2, the $\mathtt{down\_cover}_x$ bit is calculated by $\mathtt{down\_cover}_x \leftarrow (\mathtt{down\_cover}_x[\mathtt{par}_x] \wedge \mathtt{consistent}_x)$. Hence, a split or a loss of nodes cannot switch the down_cover bit (from 0 to 1) at the resulting sprig/s. Now, consider the case that some node $y \notin A$ joins A. Node y becomes a child of some node $x \in A$. At that time, by Lem. 4, a local reset exit occurs at y (line 12) and then, at line 13, node y assigns $\mathtt{par}_y \leftarrow x$ and by Rule 2, adopts the $\mathtt{down\_cover}_{\mathtt{par}_y}$ which is 0 as shown above. Thus, any node that joins A adopts 0 in its down_cover (recall that by Lem. 4, it joins alone).

Hence, after time $T_{ccover} + 2 \cdot 2^i + 1$, only new sprigs exist (beside legal trees) and every disconnected node u in such a sprig has $\mathtt{down\_cover}_u = 0$. ∎

5 Analysis of the Higher Version Case ($2^i \geq \mathbf{depth}(v_{min})$)

Lem. 11 is important to prove the stabilization of the higher versions (Lem. 12). Specifically, it helps to show that a local reset mode in each node has finite duration.

Lemma 11. *Assume a node v performing a local reset operation at some time T. (1) Then, in at most 2 time steps after time T, unless the condition of line 9 holds, v performs line 12 of the algorithm (a local reset exit occurs at v).*
(2) During the local reset mode at v (starting at time T and till the local reset exit at v), $\mathtt{up\_cover}_v = 0$ and $\mathtt{down\_cover}_v = 0$.

Lemma 12. *If $2^i \geq \mathbf{depth}(v_{min})$, then version i stabilizes in $O(2^i)$.*

Lemma 13. *If $2^i \geq \mathbf{depth}(v_{min})$, then in $O(2^i)$ time, at every node v, $\mathtt{down\_cover}_v = 1$ for version i.*

Lem. 12 above establishes that there exists some higher version that stabilizes at time $O(\mathbf{diam})$. Lem. 13 establishes that in this version `down_cover` $\leftarrow 1$. Hence, the algorithm can output the tree this version produces. Recall that Sec. 4 shows that in lower versions `down_cover` stabilizes to zero. All these establish the following theorem.

Theorem 1. *In* $O(\mathbf{diam})$ *time units, the algorithm produces a shortest paths tree rooted at the minimal* ID *node in the network.*

The proofs (see [12]) in this section are rather similar to the proofs for the higher versions in [7] except for one important point. The local reset we use here has the potential to destabilize these versions. We show that a local reset always ends. Moreover, since a local reset is transferred to children, not to parents, the reset does not destabilize the tree rooted in the minimal *ID* node.

References

1. Abbas, S., Mosbah, M., Zemmari, A.: Distributed Computation of a Spanning Tree in a Dynamic Graph by Mobile Agents. In: IEEEIS'06 (2006)
2. Afek, Y., Bremler-Barr, A.: Self-stabilizing Unidirectional Network Algorithms by Power-Supply. In: SODA'97 (1997)
3. Afek, Y., Kutten, S., Yung, M.: Memory-Efficient Self-Stabilizing Protocols for General Networks. In: van Leeuwen, J., Santoro, N. (eds.) Distributed Algorithms. LNCS, vol. 486. Springer, Heidelberg (1991)
4. Afek, Y., Kutten, S., Yung, M.: The Local Detection Paradigm and its Applications to Self-Stabilization. In: TCS' 97, vol. 186(1–2) (1997)
5. Aggarwal, S., Kutten, S.: Time Optimal Self-stabilizing Spanning Tree Algorithms. In: Shyamasundar, R.K. (ed.) Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 761. Springer, Heidelberg (1993)
6. Arora, A., Gouda, M.G.: Distributed Reset. In: Veni Madhavan, C.E., Nori, K.V. (eds.) Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 472. Springer, Heidelberg (1990)
7. Awerbuch, B., Kutten, S., Mansour, Y., Patt-Shamir, B., Varghese, G.: Time Optimal Self-stabilizing Syncronization. In: STOC'93 (1993)
8. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Bounding the Unbounded. In: INFOCOM'94 (1994)
9. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-stabilization by Local Checking and Correction. In: FOCS'91 (1991)
10. Beauquier, J., Datta, A.K., Gradinariu, M., Magniette, F.: Self-stabilization Local Mutual Exclution and Daemon Refinement. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914. Springer, Heidelberg (2000)
11. Bellman, R.: On routing problem. Qurterly of Applied Mathematics 16(1), 87–90 (1958)
12. Burman, J., Kutten, S.: Time Optimal Asynchronous Self-stabilizing Spanning Tree (extended version), `http://tx.technion.ac.il/~bjanna/`
13. Delaët, S., Ducourthial, B., Tixeuil, S.: Self-stabilization with r-operators in Unreliable Directed Networks. TR 1361, LRI (2003)
14. Delaët, S., Ducourthial, B., Tixeuil, S.: Self-stabilization with r-operators revised. In: JACIC (2006)

15. Dijkstra, E.W.: Self-stabilization in spite of Distributed Control. Comm. ACM 17, 643–644 (1974)
16. Dolev, S., Israeli, A., Moran, S.: Resource Bounds for Self-stabilizing Message Driven Protocols. In: PODC'91 (1991)
17. Dolev, S., Israeli, A., Moran, S.: Self-stabilization of Dynamic Systems Assuming Only Read/Write Atomicity. DC 7, 3–16 (1994)
18. Dolev, S., Israeli, A., Moran, S.: Uniform Dynamic Self-stabilizing Leader Election (extended abstract). In: Toueg, S., Kirousis, L.M., Spirakis, P.G. (eds.) Distributed Algorithms. LNCS, vol. 579. Springer, Heidelberg (1992)
19. Ducourthial, B., Tixeuil, S.: Self-stabilization with r-operators. DC 14(3), 147–162 (2001)
20. Garg, V.K., Agarwal, A.: Distributed Maintenance of A Spanning-Tree Using Labeled Tree Encoding. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648. Springer, Heidelberg (2005)
21. Gärtner, F.C.: A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms. TR, EPFL (October 2003)
22. Gärtner, F.C., Pagnia, H.: Time-Efficient Self-stabilizing Algorithms Through Hierarchical Structures. In: Huang, S.-T., Herman, T. (eds.) SSS 2003. LNCS, vol. 2704. Springer, Heidelberg (2003)
23. Gupta, S.K.S., Srimani, P.K.: Self-stabilizing Multicast Protocols for Ad Hoc Networks. JPDC 63(1) (2003)
24. Herault, T., Lemarinier, P., Peres, O., Pilard, L., Beauquier, J.: Self-stabilizing Spanning Tree Algorithm for Large Scale Systems. TR 1457, LRI (August 2006)
25. Higham, L., Liang, Z.: Self-stabilizing Minimum Spanning Tree Construction on Message-Passing Networks. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180. Springer, Heidelberg (2001)
26. Nesterenko, M., Arora, A.: Stabilization-Preserving Atomicity Refinement. J. Parallel Distrib. Comput. 62(5), 766–791 (2002)