Self-Stabilizing Unidirectional Network Algorithms by Power-Supply (Extended Abstract)

Yehuda Afek * Anat Bremler *

Abstract

Power-supply, a surprisingly simple and new general paradigm for the development of self-stabilizing algorithms in different models, is introduced. The paradigm is exemplified by developing simple and efficient self-stabilizing algorithms for leader election and either BFS or DFS spanning tree constructions, in strongly-connected unidirectional and bi-directional dynamic networks (synchronous and asynchronous). The different algorithms stabilize in $O(n)$ time in both synchronous and asynchronous networks without assuming any knowledge about the network topology or size, where n is the total number of nodes. Following the leader election algorithms we present a generic self-stabilizing spanning tree and/or leader election algorithm that produces a whole spectrum of new and efficient algorithms for these problems. Two variations that produce either a rooted Depth First Search tree or a rooted Breadth First Search tree are presented.

1 Introduction

A distributed system is self-stabilizing if never mind what local state each of its processors is placed in, how badly the RAM of each is corrupted and which messages are placed on its communication channels, the system automatically enters a global legal state a bounded time after the last fault or corruption has occurred. Once in a legal and correct state, the system remains in legal states unless another fault or topological change has happened. The notion of self-stabilization was introduced by Dijkstra in 1973 [16] and since then many algorithms for different problems and configurations were developed. Self-stabilizing algorithms for message passing systems were developed for either unidirectional or bidirectional rings [2, 5, 6, 12, 13, 14, 16, 17, 25, 26, 27, 29, 30, 31] and for bidirectional arbitrary topology networks [4, 7, 8, 9, 10, 12, 20]. In this paper we develop simple and efficient self-stabilizing algorithms for unidirectional arbitrary topology dynamic networks. The techniques developed in this paper produce new simple and efficient algorithms for the bidirectional case as well. In either case our algorithms do not make any assumptions on the size of the network or of messages and variables used in the algorithms.

The major obstacle in designing unidirectional self-stabilizing algorithms is the lack of acknowledgments. Bidirectional communication is heavily used by the nodes in the bidirectional self-stabilizing system to compare the states of neighboring nodes and to check their consistency, as for example in the local checking algorithms [4, 10, 11].

In this paper we overcome the lack of bidirectional communication by a new and surprisingly simple technique called *power-supply*. Using this technique we present leader election algorithms for synchronous (Section 3) and asynchronous (Section 4) networks. Subsequently we generalize these algorithms (Section 5) with a new observation of Collin and Dolev [15] to a family of algorithms either for electing a leader, or for constructing a spanning tree. Two versions of the general algorithm produce one a BFS tree and the other a Depth First Search (DFS) tree with pre-distinguished leader or without one, while other versions are possible. While there could be as much as $O(E)$ corrupted messages in a global state of the system the time complexity (stabilization time) of our algorithms is $O(n)$ without making any assumption on the size of the network.

1.1 The paper in a nutshell and Related works

The design of self-stabilizing algorithms for unidirectional networks has started to receive attention only in recent years with the works of Mayer Ostrovsky and Yung [30] and with [5]. These works have designed algorithms mostly for unidirectional rings and left the arbitrary topology open. Our work is motivated by these works and by the requirements posed by SDH/SONET unidirectional networks [5].

Only a few non-fault tolerant distributed algorithms for unidirectional networks have been developed in the past, e.g., [3, 18, 21, 22, 23, 24, 28, 32, 33].

The paper goes from simple (Section 3) to more difficult (Sections 4, and 5). Let us go over by enumerating the key ideas:

*Computer Science Department,Tel-Aviv University, Israel 69978

1. The basic algorithms developed are leader election algorithms that elect the smallest id as a leader. However, in self-stabilization simply choosing the smallest id is not safe since a fake id smaller than all the ids in the network may falsely be chosen by all the nodes.

2. A standard technique to overcome this problem is to assign each node with its distance from the node whose identity it has selected as a leader [20, 34]. To ensure self-stabilization every node periodically checks that its distance is one more than its parent distance (parent is the neighbor through which the node has discovered the leader with the distance parameter it believes in, *nil* if the node under consideration is the root). Still, this principle by itself is not enough, since for example, a false id might circulate around a cycle increasing its distance parameter without bound and never detecting the problem.

3. This problem was overcome in several different ways: in [20] a predefined tree subnetwork was assumed, in [4] a special mechanism was developed to overcome the problem, in [8, 10] a reset protocol was invoked each time an inconsistent situation was detected, and in [7], the knowledge of a bound b on the network diameter was assumed. In this paper a new technique is suggested, that has the advantage over the above in that it works also in unidirectional networks without making any assumptions ([7] also works in unidirectional networks but requires the knowledge of a bound on the network size).

4. **Power supply:** "Power", the first basic idea in this paper has two parts; First, a legal leader becomes a source of power which is disseminated around, while fake identities may not produce power, (a legal leader is a node which is the leader of itself). Second, to be captured by a new leader, a node consumes a fixed amount of that leader power. Hence, fake ids that have no source of power, eventually disappear.

5. **Synchronous case:** The implementation of the "power" idea in a synchronous network is simple: To be captured by a new id_{min} a node has to receive a message with id_{min} in two consecutive rounds from the same neighbor and no smaller id from any other neighbor. However, if after being captured by id_x a node does not receive an id_x message in any one round, it immediately becomes the leader of itself.

6. **Asynchronous case:** The implementation of the above idea in an asynchronous network is problematic since, on the one hand, nodes in a self-stabilizing asynchronous network have to periodically transmit messages and, on the other hand, the transmission of such messages may "give" power to a fake id. This problem is solved in this paper by distinguishing between two kinds of messages, *weak* and *strong*. Weak messages have no power and are sent periodically between neighbors to ensure the consistency of the global state. Any inconsistency detected causes the detecting node to become the leader of itself. Strong messages, on the other hand, carry power. Only leader nodes periodically produce strong messages. Every other node relays a strong message to each of its neighbors only when it receives a strong message from its parent. To be captured by a new id, id_{min}, a node has to receive two strong messages with id_{min} from the same neighbor, and at the same time, this id is smaller than any other id it receives.

7. **A generic self-stabilizing algorithm:** In this part we introduce another idea which is orthogonal to the power supply idea. We replace the minimum distance parameter (Point 2 above) by a general metric that may accommodate different parameters and rules for their update, e.g., the beautiful and simple parameter introduced by Collin and Dolev [15]. This new metric generates in combination with any of our other techniques (e.g., power-supply), a DFS tree (instead of BFS with the distance parameter) rooted at the elected leader. An example of a third metric is given in Section 5.

2 Model

We consider a unidirectional strongly connected network with a set V of n nodes and a set E of unidirectional links [3]. A unidirectional link is a point to point (node to node) communication line over which information may flow only in one direction. We make the standard and realistic assumption that each node v has a unique identity called id_v.

An incoming link of a node is a link directed into the node and an outgoing link of a node is the link directed away from that node. In the asynchronous network, the number of messages that may be present on any link at the same time is bounded by a constant B (independent of the network size). This assumption is not only realistic but also a necessary assumption as it is shown in [19] that almost any non-trivial task cannot be solved in a self-stabilizing manner if link capacities are unbounded. However, when a bounded capacity links are used a deadlock may be formed

unless messages are handled with care. In this paper we maintain a buffer for each outgoing link (incoming link) where the last two different messages that could not have been sent (received) because the link is too slow (too fast), are stored. When the link (processor) becomes available again, these messages are the first to be sent (processed). However, for the sake of clarity in describing the algorithms we disregard this mechanism, i.e., we assume bounded capacity links and that deadlocks are not formed. It is easy to see that the two models are equivalent (see the remark at the end of Section 4 for more details).

Our algorithms recover also from corrupted messages and transmission errors. During stabilization, after failures and topological changes stop, each link is assumed to reliably transmit messages from the tail node to the head node of the link. Each message that arrives at a node is tagged by the port number over which it arrives and messages are processed in the order of arrival. Moreover, messages arrive at one end of a link in the order that they have been sent from the other end (FIFO), otherwise bounded time self-stabilization is prohibited.

Another assumption that we make and without which no self-stabilizing message passing algorithm may work in a dynamic network, is that each node knows which of its incoming links is up and operational and which is down, otherwise nodes may be stuck in the asynchronous case, waiting for messages over a link that is no longer operational [1].

In Section 3 we describe our power-supply algorithm in a synchronous network. All the processors in a synchronous network receive, at the same times, an infinite sequence of evenly spaced clock ticks. In each clock tick (pulse), the processor, based on its local state and on messages received from its neighbors in the beginning of the pulse, makes a transition into a new state and may send a message to any of its neighbors. Each message sent immediately after a pulse is received by its destination before the next pulse. The time interval between two consecutive clock pulses is called a *round*.

In an asynchronous network the processors operate at arbitrary rates which might vary over time, and the messages incur an unbounded and unpredictable, but finite, delay.

The diameter of a network G whose set of nodes is V, is defined as follows: $diameter(G) = \max_{u,v \in V}\{dist(u,v)\}$ where $dist(u,v)$ is the shortest directed path from u to v.

3 Synchronous Unidirectional Power-Supply Algorithm

Here we present (see Figure 1) a simple algorithm for synchronous unidirectional networks. It stabilizes in $O(n)$ rounds and does not assume any bound on the diameter or on any other parameter of the network. (We remark that by a slight change in the algorithm, its stabilizing time may be reduced to be proportional to the length of the longest simple path in the network, which is $O(n)$ for general networks. However, this change leads to much more complex proofs, so we would not present it here).

In this algorithm again the smallest node identity is elected as a leader. Fake ids are eliminated by using the distance parameter and the following two rules: (1) to remain under the leadership of a leader L at distance d, $d > 0$, the minimum leader message the node receives at each round should be L with distance $d-1$, and (2) in order to be captured for the first time by a leader L at distance d the minimum leader message the node receives in *two* rounds in a row should be L with distance $d-1$. The first rule ensures that nodes owned by a fake leader and with the smallest distance parameter overall (which is necessarily larger than zero) at a round, would abandon that leader in the next round. Thus ensuring that the minimum distance parameter associated with a fake leader is increasing by one each round. The second rule ensures that the maximum distance parameter associated with a fake leader cannot grow by one every round, but only every two rounds and thus consuming from the power of the leader, because fake leaders do not have a power-supply (a source for leader messages). Thus, if in the initial faulty state the number of different distances associated with a fake leader is Δd then within at most $2\Delta d$ rounds that fake leader id vanishes (all its power is consumed).

Once all fake ids have been eliminated, the smallest id in the network would capture all the nodes, each with the correct shortest distance to the elected leader.

3.1 The correctness of the algorithm: For the proof of correctness we consider the execution of the algorithm in the network following the last changes and faults. That is, we assume that the execution starts in an arbitrary state and no faults or topological changes occur during the execution.

Clearly, two rounds after the initial state the variables (`new`, `prev`, and `leader`) at all the nodes hold values that were actually sent by their neighbors. In the first theorem we prove that $O(n)$ time after this state all fake ids disappear. In the second theorem we prove that $O(D)$ time after all fake ids disappear the smallest id in the network is elected leader by all the nodes.

THEOREM 3.1. *Fake leaders eventually disappear.*

Procedure for node v
Type
`leader_info` = record : [`id`, `dist`]
Var
id_v {The unique id of node v, fixed by the hardware}
`leader, new, prev` : of type `leader_info`
`m` : message of type `leader_info`
`M` : set of messages of type `leader_info` that have been received in the current round;

Each round do:
`M` := `M`$\cup\{[id_v, 0]\}$;
`new.id`:= $\min_{m \in M} m.id$;
`new.dist`:= $\min_{m \in M}\{$`m.dist` $+ 1 |$ `m.id` = `new.id`$\}$;
if `leader` $\neq$ `new` **then**
  if `prev` = `new` **then** {Second time the node receives the new information }
    `leader`:=`new`;
  else { First time the node receives the new information }
    `leader`:=[id_v , 0]; { Node v becomes a self-leader}
`prev`:=`new`; { Saving the information of the last round }
send(`leader` record) on all outgoing links.

Figure 1: Synchronous Algorithm

Proof. Let fid be a fake id in the network.

DEFINITION 1. *The* heights group *of fake id fid in a state of the system is:*

$$heights(fid) = \{\texttt{leader}_v.dist | \exists v \in G, \texttt{leader}_v.id = fid\}$$

We claim that for any fake id fid, the size of $heights(fid)$ is decreasing with time. In Lemma 3.1 it is proved that the size of $heights(fid)$ may not increase, and in Lemma 3.2 it is proved that the size of $heights(fid)$ decreases every two rounds.

We denote by $heights^r(fid)$, the set $heights(fid)$ in round r.

LEMMA 3.1. $|heights^{r-1}(fid)| \geq |heights^r(fid)|$.

Proof. The lemma follows from the code since for each $d \in heights^r(fid)$ there must have been a $d-1 \in heights^{r-1}(fid)$. Otherwise, no node would have distance d in the current round. Specifically, a node u whose leader is fid with distance d must have received in the beginning of round r the message: $[fid, d-1]$. Hence, there must have been an incoming neighbor of u, v, such that in round $r-1$ $\texttt{leader}_v$:=$[fid, d-1]$. ∎

LEMMA 3.2. $|heights^{r-2}(fid)| > |heights^r(fid)|$.

By Lemma 3.1 for every $d \in heights^r(fid)$ there is a $d-2 \in heights^{r-2}(fid)$. To prove the current lemma we show that there is at least one value dm in $heights^{r-2}(fid)$ for which there is no $dm+2$ in $heights^r(fid)$. Let $dm = \max\{heights^{r-2}(fid)\}$. Assume by contradiction that $dm+2 \in heights^r(fid)$ and that v is a node with $\texttt{leader}_v$=$[fid, dm+2]$. Clearly in round $r-2$ $\texttt{leader}_v \neq [fid, dm+2]$. Thus node v must have received a message $[fid, dm+1]$ in rounds $r-1$ and r. Thus there has been an incoming neighbor of v, u, such that $\texttt{leader}_u$=$[fid, dm+1]$ in round $r-2$, a contradiction. ∎

From the two lemmas it follows that the size of $heights(fid)$ decreases by at least one every two rounds. Hence, Theorem 3.1.

COROLLARY 3.1. *$O(n)$ rounds after the last fault or topological change all fake ids disappear.*

THEOREM 3.2. *$O(D)$ rounds after all fake ids have been eliminated the minimum id in the network is elected leader by all the nodes.*

Proof. Let ID_v be the smallest id in the network. The theorem follows by a simple induction on the rounds since the elimination of all fake ids in round r_0. Clearly, $\texttt{leader}_v.id = ID_v$ in round r_0. In round r_0+2 every node u whose distance from v is one has v as it's leader at distance one. In round r_0+2D the leader of all the nodes is ID_v, where D is the diameter of the network. ∎

COROLLARY 3.2. *The time complexity of this part is $O(D)$.*

Hence the time complexity of the algorithm is $O(n)$. $\Omega(n)$ is also the lower bound on the time complexity of our algorithm as is shown in the full paper.

4 The asynchronous power supply algorithm

A fundamental characteristic of asynchronous self-stabilizing algorithms is that nodes have to periodically exchange messages with their neighbors (using timeouts). Otherwise the system could be placed in a global state in which each node is waiting for a message from another one. This fundamental characteristic breaks our power-supply algorithm since every node in an asynchronous environment spontaneously generates an unbounded number of messages, regardless of the messages it receives.

Therefore, we introduce a new idea in order to implement the power-supply principle in an asynchronous network. We distinguish between two types of messages, *weak* and *strong*. *Weak* messages are periodically sent by each node to its neighbors ensuring that neighboring nodes are in a consistent state and no node is stuck indefinitely waiting for a message from the other one. *Strong* messages on the other hand, play the role of the power messages from the synchronous algorithm. That is, only leader nodes generate strong messages spontaneously and each other node sends one strong message to each of its neighbors for each correct and consistent strong message received over its parent port-id. Parent port-id is the port through which a leader has captured a node, by two consecutive strong messages (details below).

Specifically, (the code is given in Figure 2) each node has a `Current_leader` record with an id field and a distance field as in the synchronous algorithm, plus a `parent` pointer which is either *nil* if the node is itself a leader, or is pointing to one of its ports. A node that is owned by another *id* becomes a leader if its state is inconsistent with the neighbors message, which happens in either of the following two cases: (1) it receives a message (weak or strong) through its `parent` port-id different than its `current_leader` (i.e., a message with an id different than the node's `Current_leader`.id or with a distance different than `Leader`.dist), (2) it receives a message, *msg*, weak or strong, through any port-id that is lexicographically smaller than `Current_leader` (i.e., either *msg*.id is smaller than its `Current_leader`.id or *msg*.id equals `Current_leader`.id and *msg*.dist is smaller than `Leader`.dist).

A node that has been captured by a certain leader will be captured by a new leader only if either the new leader identity is smaller, or the new leader identity is the same as the old one but it comes with a smaller distance parameter (that is, the new leader information is lexicographically smaller than the old leader information).

Principle of power supply To be captured *two* consecutive strong messages with the new lexicographically smaller information have to be received through the same port-id, (i.e., only consistent weak messages received through the port-id in between them) and at the same time no lexicographically smaller message arrives through any other port-id. The first lexicographically smaller message to arrive immediately changes the `current_leader` of the node to itself at distance zero and only the second one changes the `current_leader` to the new information.

This principle ensures that strong fake id messages eventually disappear from the network since strong messages cannot flow in a cycle and the number of strong fake-id messages is reduced for each node being captured by the fake-id. On every path, the number of strong messages can not increase since a node sends a strong fake-id message only in response to receiving one. An important point for the proof of correctness is that whenever a node changes its `current_leader` the node sends a strong message with its new `current_leader`. Thus all the neighbors of this node would notice that it went through a state change. In particular, whenever node v that is owned by *old* is being captured by a new leader, *new*, it assigns id_v to its `current_leader` in between these changes and sends strong messages containing id_v before sending the new strong messages.

The implementation of the above in the code (Figure 2) uses a `prev` variable to store the smallest message body received in recent messages exchange, and `prev_ports` which is the set of port-ids through which this new information has arrived.

For the algorithm to operate correctly and in a self-stabilizing manner in a dynamic network several local conditions have to be repeatedly checked and if found inconsistent then they should be corrected, those are:

1. The link connected to the `Parent` port-id should be up. If the `parent` link is found to be down then the node should become a leader of itself (Line 25).
2. If any port-id in `prev_ports` is found to be a port to a link which is down, it is removed from the set. If the set `prev_ports` becomes empty, then `prev` is reset (Lines 22 - 25).
3. If the `parent` of node v is *nil* then `current_leader` has to be [id_v, 0] and vice versa (Lines 1 - 2).
4. If `current_leader`.id is larger than the node's id, then again `current_leader` is reset to [id_v, 0] (Lines 1 - 2).
5. Each message has to conform to the expected syntax, and negative numbers are not allowed. A node that receives an illegal message becomes a leader of itself.

Since the asynchronous algorithm is an instance of the generic algorithm, its time complexity is $O(n)$ as we show for the generic algorithm. Similarly, the correctness of the algorithm follows from the proof of correctness of the generic algorithm given in the full paper.

Remark about the model: As stated in the model section, the number of messages on a link in a certain state is bounded by B. Thus if either a tail node tries to transmit faster than the rate of the link, or if a receiving head node is too slow to receive the messages in the rate they arrive over the link, messages might be lost. For our algorithms this does not pose a problem. We assume that at both end ports of a link there is a process that works as follows: At each end it keeps a buffer with room for two messages. At the outgoing end (tail) whenever the algorithm produces messages at a rate higher than the link rate, the process keeps the last two different messages that were not sent. These messages will be sent out, as if the two messages buffer is part of the link.

```
Procedure at node v:
Type
   leader_info = record : [id, dist]
Var
   id_v ;                                          {The unique id of node v, fixed by the hardware}
   current_leader,prev,msg : of type leader_info ;
   parent : port-id ;
   set prev_ports of {port-ids} ;

Upon receiving message (msg,mtype) arriving at incoming port-id p
1  if parent=nil then current_leader:=[id_v, 0] ;                                  {To be consistent}
2  if [id_v, 0] ≤ lexic current_leader then current_leader:=[id_v, 0]; parent:=nil;
3  if (p=parent) ∧ (msg = current_leader)
4     then if (mtype = strong) then send_neighbors(strong) ;
5  if (p= parent) ∧ (current_leader ≠ msg)                  {inconsistent message from the parent}
6     then current:=[id_v,0] ;
7          parent:=nil ;
8          send_neigbours(strong) ;
9  if (msg <lexic current_leader) then
10    if (mtype= strong) ∧ (prev=msg) ∧ ( p ∈ prev_ports)
                                          { The second lexicographically smallest message}
11         then current_leader:=msg ;                                  {The node is captured }
12              parent=p ;
13              send_neigbours(strong) ;
14         else current_leader:= [id_v,0];          { The first lexicographically smallest message }
15             parent:=nil ;
16             send_neigbours(strong) ;

17 if (msg <lexic prev) then case(mtype)                          {Updating prev and prev_ports }
18                     Strong:prev=msg; prev_ports:= {p} ;
19                     Weak:  prev=[id_v,0];prev_ports:= ∅ ;
20 if (msg = prev) ∧ (mtype=strong) ∧ (p ∉ prev_ports) then prev_ports:=prev_ports ∪ {p} ;
21 if (msg >lexic prev ) ∧ (p ∈ prev_ports) then prev_ports:=prev_ports \ {p} ;

22 for every p ∈ prev_ports                  { To make the algorithm work in Dynamic Network}
23    if p is not alive then prev_ports:= prev_ports \ {p} ;
24 if (prev_ports = ∅) then prev:= [id_v, 0] ;
25 if (parent is not alive) then current_leader:= [id_v , 0]; parent:=nil;

26 Send_neighbors(mtype)
27    Send ( [current_leader.id, leader.dist+1], mtype) to all neighbors ;

28 Upon timeout() at node v
29    if parent=nil then send_neighbors (strong) ;
30                  else  send_neighbors (weak) ;
```

Figure 2: The Asynchronous Power Supply Algorithm

Similarly at the receiving end the process maintains a two messages buffer and keeps in it only the last two different messages that has arrived and not yet processed by the algorithm. This ensures that if a node changes its state several times the last change is never lost and each of its neighbors would notice that it went through a state change.

5 A generic power-supply algorithm

The two self-stabilizing algorithms presented in Sections 3 and 4, and most of the other algorithms known [4, 7, 9, 10, 20] rely on the distance parameter, i.e., on the fact that each node selects the node closest to the leader and updates its distance to be one more. Yet, in [15] Collin and Dolev present a self-stabilizing algorithm that relies on another metric which in turn produces a DFS tree rather than a BFS tree. These results suggest that perhaps there is a basic principle unifying these metrics. In this section we develop a generic algorithm into which different metrics may be plugged, e.g., one of the above two, or new ones. An example of such a new one is given below.

The general algorithm produces a whole spectrum of self-stabilizing algorithms for both uni-directional and bi-directional networks, and is given in Figure 4. The algorithm is a combination of the power-supply principle from the previous sections with a general scheme to produce spanning trees. The BFS principle as in [34] is one instance of the general scheme to construct a BFS spanning tree while the Collin-Dolev principle given in [15] is another instance producing a DFS.

From any initial state the generalization guarantees to stabilize in $O(n)$ time units if the underlying principle that ensures a tree structure does not send huge amounts of information (i.e., as long as messages size is kept $O(\log n)$ bits, or in a model which allows sending large messages in one time unit). If messages are larger than it is allowed by the model, then the time complexity might be larger.

Let us first describe the underlying principles and properties of the family of tree producing schemes that fit our general algorithm. All these schemes work according to the following general mechanism: Each node which is a candidate for leadership has a unique value called the *zero* of that node. In the algorithms for constructing a tree rooted at a pre-distinguished node only that node is a candidate. The *zero* value of each candidate is fixed in hardware (i.e., in stable and reliable memory) and, it is usually based on the node unique identity. In the algorithm each candidate tries to "convince" all other nodes to choose its *zero* value as their selected value and thus to capture them. To do so, each candidate suggests each of its neighbors to be its selected parent by sending each of them a special value computed by a applying a function, called *next* on the *zero* value. Each node v selects, according to a particular selection rule, one of the suggestions it receives, assigns it to its *selected* variable, and selects the link over which it arrives as its parent. Node v transitively suggests its neighbors to join the same selected candidate by sending them a special message computed by again applying the function *next* on v's *selected* value. This

The type info:
  case ALGORITHM:
  LE+BFS, LE+DFS, LE+FP, FP:
    Type `info` = record : [`id`, `param`];
  BFS, DFS:
    Type `info` = record: [`param`];

The value *zero* **type** `info` **at node** v**:**
  case ALGORITHM:
  LE+BFS:
    zero:= [id_v, 0];
  LE+DFS, LE+FP:
    zero:= [id_v, $\perp$];
  BFS:
    if v is the root **then** *zero*:= [0];
                else *zero*:= [∞];
  DFS:
    if v is the root **then** *zero*:= [$\perp$];
                else *zero*:= [∞];
  FP:
    if v is the root **then** *zero*:= [0,$\perp$];
                else *zero*:= [∞,$\perp$];

If v **is a** *root* **then** `parent`:=*nil*;

Function next(`selected`**:** `info`**;** p**: port-id) :** `info`
  case ALGORITHM:
  FP, LE+FP, LE+DFS:
    return [`selected.id`, `selected.param` $\circ$ p];
  LE+BFS:
    return [`selected.id`, `selected.param`+1];
  BFS:
    return [`selected.param`+1];
  DFS:
    return [`selected.param` $\circ$ p] ;

Function select(`selected`,*msg***:** `info`**):** `info`
  case ALGORITHM:
  BFS+LE, DFS+LE, BFS, DFS:
    if *msg* $<_{lexic}$ `selected` **then return** *msg*;
                else return `selected`;
  FP, LE+FP:
    if ($id_v \notin$ *msg.param*) $\wedge$
    (($id_v \in$ `selected`.param) $\vee$ (*msg.id* $<$ `selected`.*id*))
                then return *msg*;
                else return `selected`;

Figure 3: Generic framework

process continues transitively until one candidate captures the entire network. The process thus described constructs a tree structure that traces the paths along which the *zero* value of the tree root has disseminated.

For such a scheme to generate a self-stabilizing algorithm when combined with the power-supply technique, it has to satisfy particular characteristics. The three components: (1) the *next* function, used to compute the suggestions, (2) the selection rule each node applies to choose from the suggestions it receives, and (3) the set of *zero* values, have to satisfy the following three properties:

1. No legal sequence of *selected* values along a path may cycle. That is, in any cycle of parent links and *selected* values at least one node locally detects (by observing its predecessor selection and its own selection) that its *selected* value is wrong.
2. If there are no faults or erroneous values then exactly one candidate node captures the whole network. This node does not select a parent link (its parent is *nil*).
3. If there are no faults or erroneous values then the process reaches a fixed point. That is, the network reaches a state after which no node changes its selection.

Any scheme that satisfies these properties reaches a stable state in which the parent links induce a rooted tree spanning the network. Different schemes (*next* function, selection rule and set of *zero* values) produce different trees. In this paper three basic schemes are used that produce either a BFS tree, or a DFS tree, or an arbitrary tree.

A scheme that satisfies the above guidelines to construct a DFS tree was given by Collin and Dolev [15]. The *zero* of a root node in that variation is the symbol $\perp$, the *selected* value of each node is a string of output port-ids along a simple path from the root (candidate for leadership) to that node. The selection rule selects the neighbor such that its $next(selected)$ value (sequence of link ports) is lexicographically smallest. Note that in this case the *next* function takes also the port-id leading to each neighbor as a parameter.

In another example of the generic algorithm, each node maintains the sequence of nodes on a path from the root to itself. In the generic implementation, each node v selects and extends the list of a neighbor whose list does not include v.

The different variations of the scheme are specified in Figure 3, for inclusion with the power-supply code in Figure 4. We present the parameterization of the *zero* values, *next* function and the selection rules to produce the following variations: (1) a leader election algorithm that produces also a rooted breadth first search (BFS) tree (as in the previous section, denoted as LE+BFS), (2) a leader election algorithm that produces also a rooted *depth* first search (DFS) tree, denoted as LE+DFS), (3) an algorithm that produces a BFS tree given a distinguished root (denoted as BFS), (4) an algorithm that produces a DFS tree given a distinguished root (denoted as DFS), (5) a leader election algorithm that produces an arbitrary rooted tree (denoted as LE+FP), and (6) an algorithm that produces an arbitrary tree given a distinguished root (denoted as FP).

The different schemes satisfy each property in different ways. In the BFS and DFS schemes the no cycle property is satisfied because:

1. The set of all possible suggestions and *zero* values, is a total ordered set.
2. For any value x, $next(x) > x$.

In the third scheme (FP) the no-cycle property is trivially satisfied since each suggestion is a string of ids, and the function *next* at node v simply appends id_v to v's selected string. A node does not select a suggestion that contains its own id.

The no-cycle property together with the power-supply technique ensure that any phony suggested value eventually disappear.

The formal proof of the algorithm properties is omitted from this extended abstract.

Note that in the cases with the pre-distinguished leader (BFS or DFS) it is not necessary that each node in the system has a unique *id*. The time complexity of the general algorithm is $O(n)$ (due to space limitations the proofs are omitted from the extended abstract). Note that in the case of BFS with pre-distinguished leader, the time complexity is lower, $O(d)$, which is also the optimal time complexity for this problem.

Acknowledgments: We would like to thank Shlomi Dolev for several helpful discussions.

References

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

[2] Y. Afek and G. M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing Journal*, 7:27–34, 1993. Abstract in *Proc. of the 8th IEEE Symp. on Reliable Distributed Systems*, 10-12 1989.

[3] Y. Afek and E. Gafni. Distributed algorithms for unidirectional networks. *Siam J. on Computing*, 23(6), December 1994.

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

[5] Y. Afek and T. Lev. Distributed synchronization protocols for SDH networks. Submitted for publication, November 1995.

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

[7] A Arora and MG Gouda. Distributed reset. 43:1026–1038, 1994.

```
Procedure at node v:
Type
   info = record : [id, param]
Var
   selected,prev,msg : of type info ;
   parent : port-id ;
   set prev_ports of {port-ids} ;

Upon receiving message (msg,mtype) arriving at incoming port-id p
1  if parent=nil then selected:=zero ;                                  {To be consistent}
2  if (select(selected,zero)= zero) then selected:=zero; parent:=nil
3  if (p=parent) ∧ (msg = selected)                                     {send a message}
4      then if (mtype = strong) then send_neighbors(strong) ;
5  if ((p= parent) ∧ (selected ≠ msg))          {inconsistent message from the parent}
6      then selected:=zero ;
7           parent:=nil ;
8           send_neighbors(strong) ;
9  if (select(selected,msg)=msg) then
10     if (mtype= strong) ∧ (prev=msg) ∧ ( p ∈ prev_ports)     {The second selected message}
11          then selected:=msg ;                                  {The node is captured }
12               parent=p ;
13               send_neighbors(strong) ;
14          else selected:=zero ;                                 {The first selected message }
15              parent:=nil ;
16              send_neighbors(strong) ;

17 if (select(prev,msg)=msg) then case(mtype)                    {Updating prev and prev_ports }
18                           Strong:prev=msg; prev_ports:= {p} ;
19                           Weak:  prev=zero;prev_ports:= ∅ ;
20 if (msg = prev) ∧ (mtype=strong) ∧ (p ∉ prev_ports) then prev_ports:=prev_ports ∪ {p} ;
21 if (select(prev,msg)≠ msg) ∧ (p ∈ prev_ports) then prev_ports:=prev_ports \ {p} ;

22 for every p ∈ prev_ports                  { To make the algorithm work in Dynamic Network}
23     if p is not alive then prev_ports:= prev_ports \ {p} ;
24 if prev_ports = ∅ then prev:=zero ;
25 if (parent is not alive) then selected:=zero;parent:=nil;

26 Send_neighbors(mtype)
27     for every p ∈ prev_ports
28              Send ( next(selected,p), mtype) to neighbor p ;

29 Upon timeout() at node v
30     if parent=nil then send_neighbors (strong) ;
31                   else  send_neighbors ( weak) ;
```

Figure 4: The Generic Power Supply Algorithm

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

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

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

[11] B Awerbuch, B Patt-Shamir, G Varghese, and S Dolev. Self-stabilizing by local checking and global reset. In *International Workshop on Distributed Algorithms, Springer-Verlag*, pages 326–339, 1994.

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

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

[14] J. E. Burns, M. G. Gouda, and R. E. Miller. Stabilization and pseudo stabilization. Technical Report TR-90-13, The University of Texas at Austin, May 1990.

[15] Z. Collin and S. Dolev. Self-stabilizing depth first search. *Information Processing Letters*, 49:297–301, 1994.

[16] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1973.

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

[18] D. Dolev, M. Klawe, and M. Rodeh. An O(n log n) unidirectional algorithm for extrema finding in a circle. *Journal of Algorithm*, 3:245–260, 1982.

[19] S. Dolev, A. Israeli, and S. Moran. Resource bounds for self stabilizing message driven protocols. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, pages 281–294. ACM, august 1991.

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

[21] S. Even, A. Litman, and P. Winkler. Computing with snakes in directed networks of automata. In *Proc. of the 31st IEEE Ann. Symp. on Foundation of Computer Science*, pages 740–745, October 1990.

[22] E. Gafni and Y. Afek. Election and traversal in unidirectional networks. In *Proc. of the 3rd Ann. ACM Symp. on Principles of Distributed Computing*, August 1984.

[23] E. Gafni and W. Korfhage. Distributed election in unidirectional eulerian networks. In *Proc. Twenty-Second Ann. Allerton Conference on Communication, Control, and Computing*, October 1984.

[24] M. Gerla, L. Kleinrock, and Y. Afek. A distributed routing algorithm for unidirectional networks. In *Proc. GLOBCOM 83*, 1983.

[25] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35:63–67, 1990.

[26] A Israeli and M Jalfon. Uniform self-stabilizing ring orientation. *Information and Computation*, 104:175–196, 1993.

[27] S. Katz and K. J. Perry. Self-stabilizing extensions for message-passing systems. In M. Evangelist and S. Katz, editors, *MCC Technical Report Number STP-379-89, Proc. of the MCC Workshop on Self-Stabilizing Systems*, 1989.

[28] S. Kutten. Stepwise construction of an efficient distributed traversing algorithm for general strongly connected directed networks. In J. Raviv, editor, *Proc. of the Ninth Int. Conference on Computer Communication*, pages 446–452, October 1988.

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

[30] A. Mayer, R. Ostrovsky, and M. Yung. General distributed self-stabilizing algorithms for unidirectional rings. In *Proc. of 8th Ann. ACM-SIAM Symp. on Discrete Algorithms*, January 1996.

[31] N. Multari. *Self-Stabilizing Protocols.* PhD thesis, Department of Computer Sciences, University of Texas, 1988. Ph.D. Thesis.

[32] Rafail Ostrovsky and Daniel Wilkerson. Faster computation on directed networks of automata. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 38–46. ACM, August 1995.

[33] G. L. Peterson. An O(n log n) unidirectional algorithm for the circular extrema problem. *ACM Trans. Program. Lang. Syst.*, 4(4):758–762, October 1982.

[34] W. P. Tajibnapis. A correctness proof of a topology information maintenance protocol for a distributed computer network. *CACM*, 20-7:477–485, 1977.