Self-Stabilizing Unidirectional Network Algorithms by Power-Supply*

Yehuda Afek † Anat Bremler ‡
Computer Science Department,
Tel-Aviv University, Israel 69978.

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 1974 [Dij74] and since then many

*This work was supported by the Broadband Telecommunications R&D Consortium administered by the Chief Scientist of the Israeli Ministry of Industry and Trade.

†E-mail:afek@math.tau.ac.il.

‡E-mail:natali@math.tau.ac.il.

algorithms for different problems and configurations were developed. Self-stabilizing algorithms for message passing or shared memory systems were developed for either unidirectional or bidirectional rings [Dij74, AB93, AL95, MOY96, MOOY92, Mul88, BP89, BGM90, Her90, IJ93, KP89, AEYH92, BGW89] and for bidirectional arbitrary topology networks [DIM94, AKY90, APSV91, AV91, BGW89, AG94b, AKM+93]. 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 [AKY90, APSV91, APSVD94].

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 [CD94] 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 Breath First Search (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. The space complexity (overhead) of our algorithms is $O(\log n)$ bits per node.

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 [MOY96] and with [AL95]. 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 [AL95].

Only a few non-fault tolerant distributed algorithms for unidirectional networks have been developed in the past, e.g., [GA84, AG94a, GKA83, GK84, Kut88, ELW90, Pet82, OW95, DKR82].

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

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 [Taj77, DIM94]. 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 [DIM94] a predefined tree subnetwork was assumed, in [AKY90] a special mechanism was developed to overcome the problem, in [APSV91, AV91] a reset protocol was invoked each time an inconsistent situation was detected, and in [AG94b], 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 ([AG94b] 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 [CD94]. 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 [AG94a]. 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 [DIM91] 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 [AAG87].

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 number of edges in 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 distinct distances associated with a fake leader is denoted as 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 the fake ids have disappeared, where D is the diameter of the network. The smallest id in the network is elected leader by all the nodes.

In theorem 1 we prove that fake leaders eventually disappear.

Theorem 1 *Eventually, all nodes have in the variable* `leader` *have* $id \in ID$, *where* $ID = \{id_v | v \in V\}$.

Proof: Let fid be a fake id in the network, i.e., $fid \notin V$.

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

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

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

Figure 1: Synchronous Algorithm

We claim that for any fake id fid, the size of $heights(fid)$ is decreasing with time. In Lemma 1 it is proved that the size of $heights(fid)$ may not increase, and in Lemma 2 it is proved that the size of $heights(fid)$ decreases every two rounds.

Let $\mathtt{X}_v^i$ denote variable X at node v at round i.

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

Proof: The lemma follows from the code since for each $d \in heights^{r}(fid)$, $d \geq 2$ 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: $[fid,\ d-1]$ (By Line 3). Hence, there must have been an incoming neighbor of u, v, such that in round $r-1$ $\mathtt{leader}_v$:=$[fid,\ d-1]$. □

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

Proof: By Lemma 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 $\mathtt{leader}_v^r$=$[fid,\ dm+2]$.

Clearly in round $r-2$ $\mathtt{leader}_v \neq [fid,\ dm+2]$.

Hence there are two possible cases. Either $\mathtt{leader}_v$=$[fid,dm+2]$ also in rounds r and $r-1$ or, only at round r. We show that in either case new_v^{r-1}=$[fid,dm+2]$, hence node v must have received a message $[fid,\ dm+1]$ in round $r-1$. Thus there has been a u, an incoming neighbor of v, such that $\texttt{leader}_u$=$[fid,\ dm+1]$ in round $r-2$, a contradiction.

In the first case, since $\texttt{leader}_v^{r-1}$=$[fid,dm+2]$, node v preformes Line 5 and $\texttt{leader}_v^{r-1}$=new^{r-1}=$[fid,dm+2]$.

In the second case, since $\texttt{leader}_v^{r}$=$[fid,dm+2]$, node v preformes Line 5 and $\texttt{leader}_v^{r}$=new^{r}=$prev^{r}$=$[fid,dm+2]$. Since $prev^r$=new^{r-1} (by Line 9), new^{r-1}=$[fid,dm+2]$. □

From the two lemmas it follows that the size of $heights(fid)$ decreases by at least one every two rounds. Hence, Theorem 1.

Corollary 3 *$O(n)$ rounds after the last fault or topological change all fake ids disappear.*

Theorem 2 *$O(D)$ rounds after all fake ids have been eliminated the minimum id in the network is elected leader by all the nodes, where D is the diameter of the network.*

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. □

Corollary 4 *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 appendix A.

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

```
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
   if parent=nil then current_leader:=[id_v, 0] ;                         {To be consistent}
   if [id_v, 0] ≤_lexic current_leader then current_leader:=[id_v, 0]; parent:=nil;
   if (p=parent) ∧ (msg = current_leader)
      then if (mtype = strong) then send_neighbors(strong) ;
   if (p= parent) ∧ (current_leader ≠ msg)             {inconsistent message from the parent}
      then current:=[id_v,0] ;
           parent:=nil ;
           send_neigbours(strong) ;
   if (msg <_lexic current_leader) then
      if (mtype= strong) ∧ (prev=msg) ∧ ( p ∈ prev_ports)
                                                 { The second lexicographically smallest message}
           then current_leader:=msg ;                                   {The node is captured }
                parent=p ;
                send_neigbours(strong) ;
           else current_leader:= [id_v,0];          { The first lexicographically smallest message }
                parent:=nil ;
                send_neigbours(strong) ;

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

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

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

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

Figure 2: The Asynchronous Power Supply Algorithm

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 Section 6.

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.

Similarly at the receiving end the process maintains a two messages buffer and keeps in it only the last two different messages that have 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 [DIM94, AKY90, APSV91, AG94b, AKM+93] 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 [CD94] 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 [Taj77] is one instance of the general scheme to construct a BFS spanning tree while the Collin-Dolev principle given in [CD94] 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 pre-distinguished 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 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. Each such scheme has three components: (1) the *next* function, used to compute the suggestions, (2) the selection rule, which each node applies

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` **of type** `info`**,** p **of type 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` **of type** `info`**,** *msg* **of type** `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

to choose its *selected* variable from the suggestions it receives, and (3) a set of *zero* values, which are all the *zero* values of nodes in the network. This set of *zero* values has 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 [CD94]. 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 given in Section 6.

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 $O(n)$ (proof in Section 6). 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.

```
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
   if parent=nil then selected:=zero ;                                   {To be consistent}
   if (select(selected,zero)= zero) then selected:=zero; parent:=nil
   if (p=parent) ∧ (msg = selected)                                       {send a message}
      then if (mtype = strong) then send_neighbors(strong) ;
   if ((p= parent) ∧ (selected ≠ msg))                  {inconsistent message from the parent}
      then selected:=zero ;
           parent:=nil ;
           send_neighbors(strong) ;
   if (select(selected,msg)=msg) then
      if (mtype= strong) ∧ (prev=msg) ∧ ( p ∈ prev_ports)     {The second selected message}
           then selected:=msg ;                                     {The node is captured }
                parent=p ;
                send_neighbors(strong) ;
           else selected:=zero ;                                   {The first selected message }
                parent:=nil ;
                send_neighbors(strong) ;

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

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

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

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

Figure 4: The Generic Power Supply Algorithm

6 Correctness of the generic asynchronous algorithm

Main theorem: $O(nB)$ *units of time after the last topological change, or erroneous state change, a stable tree spans the network.*

In this section we prove the main theorem. There are three major steps in the proof:

1. First we show that in linear time after the last change or error in the network all erroneous and illegal strong messages disappear (Lemma 15) and are never generated again.
2. Secondly we show that $O(n)$ time after the first step erroneous or illegal weak messages do not exist and are never generated again (Lemma 22).
3. Thirdly we show that $O(n)$ time after step 2 a stable rooted tree spans the network (Lemma 30).

Let us start with some definitions:

Definition 2 *With each message m two functions are associated, mtype(m) the type of the message, strong or weak, and msg(m) the body of the message, which is of type* `info` *(see the code).*

Definition 3 *A* message sequence, *denoted by* $\vec{m} = \{m_1 m_2, \ldots, m_k\}$, *is a contiguous sequence of messages (strong and weak) on a link such that they all have the same msg value. Remark: A message sequence can be empty.*

Definition 4 *We denote by* $in\_port_v(w)$ *the port id through which the* (wv) *link enters* v. *We denote by* $out\_port_v(w)$ *the port id through which the* (vw) *link leaves* v.

If we observe the edges that are covered by the union of message sequences that appear to have been originated from the same candidate, then this structure may look like a forest. However, in the proof it is easier and much simpler to argue about union of such sequences along a path, rather then along the edges of a tree. Below we formally define such sequences as *block*. Clearly, if we prove statements like "there are no more blocks with property p in the network" then it is true that the same holds for the corresponding trees with the same property. Therefore, it is enough to consider blocks, as defined below, rather then the more complicated trees.

Definition 5 Block - *denoted by* $bl = \{\vec{m}_0 v_0 \vec{m}_1 v_1 \ldots \vec{m}_l v_l \vec{m}_{l+1}\}$ $l \geq 0$, *is a sequence of messages interleaved with zero or more nodes in one state of the system, such that (see Figure 5):*

1. $\{v_0,\ v_1, \ldots v_l\}$ *is a directed path.*
2. *For* $i = 0, \ldots, l+1$, $\vec{m}_i = \{m_{i,1} m_{i,2}, \ldots m_{i,k_0}\}$ *is a message sequence on the link from* v_i *to* v_{i+1}. *Except for* $\vec{m}_{l+1}$ *each* $\vec{m}_i$ *contains* all *the messages on the corresponding link. If* $\vec{m}_0$ *is not empty then* m_{0,k_0} *is the first message on the link entering* v_0 *and we denote by begin_port the port-id at node* v_0 *through which message* m_{0,k_0} *arrives. If* $\vec{m}_0$ *is empty then begin_port is a port-id of some incoming neighbor of* v_0. *If* $\vec{m}_{l+1}$ *is not empty then* $m_{l+1,0}$ *is the last message sent by* v_l *on the corresponding outgoing link and we denote by end_port the port id at node* v_l *through which message* $m_{l+1,0}$ *was sent.*
3. *For* $i = 0, \ldots, l, \forall m \in \vec{m}_i$, $msg(m) = selected_{v_i}$.
   For $i = l+1, \forall m \in \vec{m}_{l+1}$, $msg(m) = next(selected_{v_l}, end\_port)$.
4. *For* $i = 0, \ldots, l-1$, $next(selected_{v_i}, out\_port_{v_i}(v_{i+1})) = selected_{v_{i+1}}$.

5. *For* $i = 1, \ldots, l$, $parent_{v_i} = in\_port_{v_i}(v_{i-1})$.
   For $i = 0$ $parent_{v_0} = begin\_port$.

6. *There is no* v *in* G *such that* $\{v\vec{m}_0 v_0 \vec{m}_1 v_1 \ldots \vec{m}_l v_l \vec{m}_{l+1}\}$ $l \geq 0$, *satisfies conditions 1-5, and there is no message,* m *such that*
   $\{\{m\} \cup \vec{m}_0 v_0 \vec{m}_1 v_1 \ldots \vec{m}_l v_l \vec{m}_{l+1}\}$ $l \geq 0$, *satisfies conditions 1-5.*

Observation 5 *The maximum number of nodes in a block is* n.

Observation 6 *Every block induces a simple directed path.*

Observation 6 follows from point 4 of Definition 5, and from the particular properties of the spanning tree construction scheme. The no cycle property has to be proved separately for each spanning tree scheme used. It is easy to verify it for the three major schemes we use, the BFS, DFS and FP (see the discussion in the previous section).

Definition 6 *The potential(bl) of a block bl is the total number of strong messages in the block (see Figure 5).*

Corollary 7 *The potential of a block is at most* nB.

Were we proving the leader election algorithm of section 4, we would argue that blocks with fake ids eventually disappear. However, in the generic algorithm we have generalized the notion of id and thus we define the notion of a *phony* block:

Definition 7 *A block* $\{\vec{m}_0 v_0 \vec{m}_1 v_1 \ldots \vec{m}_l v_l \vec{m}_{l+1}\}$ $l \geq 0$, *is a phony block, if this block is a result of a "maliciously" erroneous initial state. That is, a block that was created after state* s_0 *and was truncated may not be considered as a phony block. Formally, a block* $\{\vec{m}_0 v_0 \vec{m}_1 v_1 \ldots \vec{m}_l v_l \vec{m}_{l+1}\}$ $l \geq 0$, *is a phony block, if there is no way to assign values to* `selected` *fields of nodes* $v_{-h}, v_{-(h-1)}, \ldots, v_{-1}$ *such that* $\{v_{-h}, v_{-(h-1)}, \ldots, v_{-1} \vec{m}_0 v_0 \vec{m}_1 v_1 \ldots \vec{m}_l v_l \vec{m}_{l+1}\}$ *is a legal block and* $\texttt{selected}_{v_{-h}}$ *equals* $zero_{v_{-h}}$ *(see Figure 5).*

Definition 8 *Let* $\{\vec{m}_0 v_0 \vec{m}_1 v_1 \ldots \vec{m}_l v_l \vec{m}_{l+1}\}$ $l \geq 0$, *be a block. Then* $m_{0,0}$ *is the* tail *of the block or if* $\vec{m}_0$ *is empty then* v_0 *is the* tail *of the block. Similarly,* $m_{l+1,k_{l+1}}$ *is the* head *of the block or if* $\vec{m}_{l+1}$ *is empty then* v_l *is the* head *of the block (see Figure 5).*

Definition 9 *Sub block is a sub segment of a block that satisfies all the conditions of Definition 5 except condition 6 (see Figure 5).*

Observation 8 *Every block is also a sub-block.*

Formal proof

The proof argues about runs of the system that start in an arbitrary state s_0 and in which there are no failures or topological changes. Starting in state s_0 the system behavior is modeled by a run which is an infinite sequence $q_0 \pi_0 q_1 \pi_1 \ldots$ of alternating states and atomic operations, such that $q_0 = s_0$. Each atomic operation π_i is either receiving, and processing a message and sending any resulting message, or a time-out event that results in sending messages. Each state includes a complete description of all the variables and messages in all the processors of the system. State q_{i+1} is the state of the system after applying operation π_i to state q_i.

For the purpose of time complexity analysis, we assume that each message is delivered in at most one unit of time, i.e., one unit of time is the time it takes the slowest message to reach its destination.

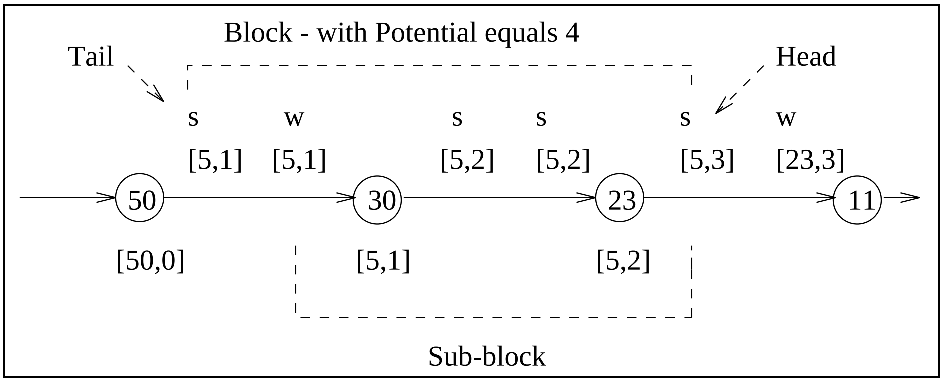

Figure 5: Example of a block, head of a block, tail of a block, potential, phony block and sub-block. The example is given in the framework of LE+BFS. Each node has id and under the node we see the node record: [leader, distance]. Similarly the record of the messages on the links. We symbolize the type of message by s for strong and w for weak. The block in the figure is $\{m_{0,1} = [s, [5,1]], m_{0,2} = [w, [5,1]], v_1 = \{id_v = 30, [5,1]\}, m_{1,1} = [s, [5,2]], m_{1,2} = [w, [5,2]], v_2 = \{id_v = 50, [5,2]\}, m_{2,1} = [5,3]\}$. The potential of the block is 4, the number of strong messages in the block. The message $m_{0,1} = [s, [5,1]]$ is the head of the block and the message $m_{2,1} = [5,3]$ is its tail. The sequence: $\{m_{0,1} = [w, [5,1]], v_1 = \{id_v = 30, [5,1]\}, m_{1,1} = [s, [5,2]], m_{1,2} = [w, w[5,1]], v_2 = \{id_v = 50, [5,2]\}, m_{2,1} = [5,3]\}$, is an example of a sub block, as any other suffix of the block.

We proceed by proving that following global state s_0 the system must progress through a sequence of global states that contains subsequence s_0, s_1, s_2 s_3 and s_4 such that in each of these states an additional global and *stable* property holds until in state s_4 the desired spanning tree property stably holds. The sequence of stable properties that hold in the run suffixes starting at states s_0, s_1, s_2, s_3, s_4 are correspondingly as follows:

1. Starting with global state s_0, there are no failures or topological changes by assumption.
2. Starting with global state s_1, all messages and all state variables (`parent`,`selected`,`prev`,`prev_ports`) hold values that result from reception of messages sent after s_0. It is easy to see that state s_1 is at most two time units after state s_0.
3. Starting from global state s_2, there are no erroneous and illegal strong messages in the network.
4. Starting from global state s_3, there are no erroneous or illegal weak messages in the network.
5. Starting from global state s_4 a stable rooted tree spans the network.

Proof intuition

In the first and major step of the proof we show that phony blocks eventually disappear, by arguing that the life time of a phony block is bounded by the potential of the block (i.e., the potential of a phony block monotonically decreases). The potential of a phony block (by Definition 6) equals to the maximum number of new nodes that the phony block may capture before disappearing (because the potential is the total number of strong messages and the capture of a new node consumes at least one strong message). Once all the phony blocks disappear, the system reaches a stable legal state in $O(n)$ time.

Intuitively, the proof that phony blocks disappear after $O(nB)$ time is based on the following three points:
(P1) By Observation 6 a block cannot cycle. (P2) A new node can be added to a block only if it is captured by two strong messages sent from the block (the power supply principle). Therefore, in capturing a node the phony block potential decreases by one.
(P3) The potential of a phony block cannot increase. It could have increased if one of the following conditions exist: a. It had the power-supply to generate new strong messages. This is obviously not true in a phony block. b. Two or more phony blocks were united, thus creating a larger potential. But blocks cannot unit, because whenever a node changes its state it first sends a message with its *zero* state to each of its neighbors. Upon receiving this message each of the neighbors that was at the same block as the node, also goes through a change. Thus, an element (message or node state) separating blocks never disappear (until the blocks it separates disappear).

Based on points P1 and P2, $O(nB)$ time after the initial state the potential of a phony block reduces to zero. To observe this, let us track any strong message m on a phony block until it disappears. In each time unit message m propagates at least one more node along the phony block. The length of the phony block thus traversed is at most $n + nB + 1$, n nodes in the original block, and $nB + 1$ that may be captured until its potential reachs zero. Therefore, after $O(nB)$ units of time there is no strong message in any phony block, and the block potential is zero. Based on Point P1, a phony block disappears $O(n)$ time after its potential is equal to zero.

In Lemma 9 we prove that the potential of a phony block does not increase after state s_1 (Point P3). I.e., distinct blocks cannot glue together into a larger block. More formally, if in state q_{i+1} there is a block with m nodes then in state q_i there is a corresponding block with at least $m-1$ nodes and with at least as many strong messages. In the lemma we go over all possible events showing that in non of them the potential increases.
Let $\mathtt{X}_v^i$ denote variable $\mathtt{X}$ at node v in state q_i.

Lemma 9 *Let q_{i+1} be a state, after state s_1. If in state q_{i+1} there exists a phony block $bl^{i+1} = \{\vec{m}_0 v_0 \vec{m}_1 v_1 \ldots \vec{m}_l v_l \vec{m}_{l+1}\}$ $l \geq 0$, then in state q_i there is a sub-phony-block sbl^i and exactly one of the following holds:*

1. **Node v_j receives a weak message:**
   $sbl^i = \{\vec{m'}_0 v_0 \vec{m'}_1 v_1 \ldots \vec{m'}_{l-1} v_{l-1} \vec{m'}_l v_l \vec{m'}_{l+1}\}$ *only one message sequence* $\vec{m'}_j$, $0 \leq j \leq l+1$, *in* sbl^i, *is not the same as in* bl^{i+1}. *Let* $\vec{m'}_j = \{m_{j,1} \ldots m_{j,k_j-1} m_{j,k_j}\}$ *in state* q_i, *then* $\vec{m}_j = \{m_{j,1} \ldots m_{j,k_j-1}\}$ *in state* q_{i+1} *where* m_{j,k_j} *is a weak message, and* $k_j \geq 1$.

2. **Node v_j sends a weak message:**
   $sbl^i = \{\vec{m'}_0 v_0 \vec{m'}_1 v_1 \ldots \vec{m'}_{l-1} v_{l-1} \vec{m'}_l v_l \vec{m'}_{l+1}\}$ *only one message sequence* $\vec{m'}_j$, $1 \leq j \leq l+1$, *in* sbl^i, *is not the same as in* bl^{i+1}. *Let* $\vec{m'}_j = \{m_{j,1} \ldots m_{j,k_j}\}$ *in state* q_i, *then* $\vec{m}_j = \{m' m_{j,1} \ldots m_{j,k_j}\}$ *in state* q_{i+1} *where* m' *is a weak message.*

3. **Node v_j receives a strong message and sends a strong message:**
   $sbl^i = \{\vec{m'}_0 v_0 \vec{m'}_1 v_1 \ldots \vec{m'}_{l-1} v_{l-1} \vec{m'}_l v_l \vec{m'}_{l+1}\}$ *only two message sequences* $\vec{m'}_j$ *and* $\vec{m'}_{j+1}$, $0 \leq j \leq l$, *in* sbl^i *are not the same as in* bl^{i+1}. *Let* $\vec{m'}_j = \{m_{j,1} \ldots m_{j,k_j-1} m_{j,k_j}\}$ *and* $\vec{m'}_{j+1} = \{m_{j+1,1} \ldots m_{j+1,k_j}\}$ *in state* q_i, *then* $\vec{m}_j = \{m_{j,1} \ldots m_{j,k_j-1}\}$ *and* $\vec{m}_{j+1} = \{m' m_{j+1,1} \ldots m_{j+1,k_j}\}$ *where* m_{j,k_j} *and* m' *are strong messages.*

4. **A node that is not in the block receives m_{l+1,k_l}, a strong message from the block:** $sbl^i = \{\vec{m'}_0 v_0 \vec{m'}_1 v_1 \dots \vec{m'}_{l-1} v_{l-1} \vec{m'}_l v_l \vec{m'}_{l+1}\}$ *Only* $\vec{m'}_{l+1}$ *in* sbl^i *is not the same as in* bl^{i+1}. *Let* $\vec{m'}_{l+1} = \{m_{l+1,1} \dots m_{l+1,k_l-1} m_{l+1,k_l}\}$ *in state* q_i, *then* $\vec{m}_{l+1} = \{m_{l+1,1} \dots m_{l+1,k_j-1}\}$ *in state* q_{i+1} *where* m_{l+1,k_l} *is a strong message.*

5. **A new node is added to the block after receiving a strong message from the block:** $sbl^i = \{\vec{m'}_0 v_0 \vec{m'}_1 v_1 \dots \vec{m'}_{l-1} v_{l-1} \vec{m'}_l\}$. $\vec{m'}_l$ *in* sbl^i *is not the same as in* bl^{i+1} *and both* v_l *and* $\vec{m}_{l+1}$ *in* bl^{i+1}, *but not in* sbl^i. *Let* $\vec{m'}_l = \{m_{l,1} \dots m_{l,k_l-1} m_{l,k_l}\}$ *in state* q_i, *then* $\vec{m}_l = \{m_{l,1} \dots m_{l,k_j-1}\}$ *in state* q_{i+1}. *In operation* π_i, v_l *changes its state and is captured to* sbl^i, *after receiving the strong message* m_{l,k_l} *and sends the strong message* m', *where* $\vec{m}_{l+1} = \{m'\}$.

6. **The block did not change:** $bl^{i+1} = sbl^i$.

Before we prove the lemma, let us state and prove few claims that are used in the proof of the lemma.

Claim 10 *If* $parent^i_w \neq nil$ *in state* q_i *which is after* s_1 *then* $\texttt{selected}^{\texttt{i}}_{\texttt{w}}$ *equals to the body part of the last message received through port* $parent^i_w$.

Proof: Let π_j be the last operation before state q_i in which a message m is received over port $parent^i_w$. By the *timeout* procedure and by the definition of state s_1 such a π_j exists. The value of $parent_w$ does not change in all the states from q_{j+1} to q_i because if it changes in π_k then it changes either to *nil* or to the port over which a message was received in π_k (Lines 11-13).

Since by the code whenever the value of selected changes the value of parent changes too, neither $\texttt{selected}_{\texttt{w}}$ has changed in any state from state q_{j+1} to q_i. Thus $\texttt{selected}^{\texttt{j+1}}_{\texttt{w}} = \texttt{selected}^{\texttt{i}}_{\texttt{w}}$. To complete the proof we need to show that $\texttt{selected}^{\texttt{j+1}}_{\texttt{w}} = msg(m)$. We consider two cases: either $\texttt{selected}_{\texttt{w}}$ changes in π_j or it does not. In the former, $\texttt{selected}^{\texttt{j+1}}_{\texttt{w}} = msg(m)$ by Lines 11-12, and in the latter, $\texttt{selected}^{\texttt{j+1}}_{\texttt{w}} = msg(m)$ by Line 5. □

Claim 11 *If* $prev^i_v \neq zero_v$ *in state* q_i *which is after* s_1 *then* $prev\_ports \neq \emptyset$ *and for every* p *such that* $p \in prev\_ports^i_v$ $prev^i_v$ *equals to the body part of the last message received through port* p *at operation* π_j, *and* $p \in prev\_ports^l_v$ *and* $prev^i_v = prev^l_v$ *for* $j+1 \leq l \leq i$.

Proof: Since $prev^i_v \neq zero_v$ then $prev\_ports^i_v \neq \emptyset$ (by Line 24).

Let π_j be the last operation before state q_i in which a message m is received over port p. Since port p is alive (Lines 22-23) as argued in the previous proof such a π_j exists.

Port $p \in prev\_ports_v$ in all the states from q_{j+1} to q_i because otherwise $prev\_ports^i_v$ does not include p (Lines 17-21).

Since by the code whenever the value of *prev* changes the group of *prev_ports* changes too (Line 18), neither $prev_v$ has changed in any state from state q_{j+1} to q_i. Thus $prev^i_v = prev^{j+1}_v$. To complete the proof we need to show that $prev^{j+1}_v = msg(m)$. We consider two cases: either $prev_v$ changes in π_j or it does not. In the former, $prev^{j+1}_v = msg(m)$ by Line 18, and in the latter, $\texttt{selected}^{\texttt{j+1}}_{\texttt{w}} = msg(m)$ by Line 20. □

Claim 12 *If in operation* π_i, *after state* s_1, *node* v *is captured by another node (i.e.,* v *performs Lines 11 - 12 and* $\texttt{selected}_{\texttt{v}}$ *is changed to a value different than* $zero_v$*) then: 1.* $\texttt{selected}^{\texttt{i}}_{\texttt{v}} = zero_v$ *2. the msg part of the last message that* v *sends to each of its outgoing neighbors before operation* π_i *equals to* $next(zero_v, p)$, *where* p *is the port-id through which the message is sent.*

Proof: Since node v is captured at operation π_i then in state, q_i, by Lines 9-10,

$$select(prev_v^i, selected_v^i) = prev_v^i \tag{1}$$

If $prev_v^i = zero_v$ then $selected_v^i = zero_v$ (by (1) and Line 2) and we are done. Otherwise $prev_v^i \neq zero_v$ and hence $prev\_ports_v^i \neq \emptyset$ (by Line 24). ¿From Claim 11 there exists an operation $\pi_k, k < i$ in which a message m is received over port p, $p \in prev\_ports_v^i$. Let π_j be the last such operation before q_i. ¿From Claim 11 $p \in prev\_ports_v^l$ and $msg(m) = prev_v^i = prev_v^l$ for $j+1 \leq l \leq i$.

To complete the proof of the first part of the claim we prove that, if the value of $\mathtt{selected_v}$ has changed in π_l, $j+1 \leq l \leq i$, then in the last such change $\mathtt{selected_v}$ was set to $zero_v$, otherwise it has been set to $zero_v$ in π_j and thus $\mathtt{selected_v^i} = \mathtt{selected_v^j} = zero_v$. To prove this consider an operation π_r, $j \leq r < i$ such that π_r is the last time before π_i that the value of $\mathtt{selected_v}$ is changed. Notice that such an operation exists.

Assume to the contrary, that there is no such operation π_r then in π_j v receives a message m such that $msg(m) = prev_v^i$ and select($prev_v^i$, $\mathtt{selected_v^i}$) = select($prev_v^i$,$selected_v^j$)= $prev_v^i$ and hence $\mathtt{selected_v}$ must have changed its value in π_j by Lines 9-16, hence contradiction to the fact that there is no change in the value of $\mathtt{selected_v}$ in any operation $j \leq r < i$.

By definition of π_r $\mathtt{selected_v^{r+1}} = \mathtt{selected_v^i}$. The value of $\mathtt{selected_v}$ changes in π_r to either $prev_v^r$ or $zero_v$ (either in Lines 11 or Lines 6-14).

In the latter case $\mathtt{selected_v^{r+1}} = \mathtt{selected_v^i} = zero_v$ and the claim follows. We claim the former is impossible, i.e., $\mathtt{selected_v}$ may not change to $prev_v$ in π_r. Otherwise, $prev_v^{r+1} = selected_v^{r+1}$, and since $prev_v^{r+1} = prev_v^i$ then $prev_v^i = selected_v^i$ contradiction to (1).

In π_r node v also sends a message with msg part equal to $next(zero_v, p)$ (Lines 8 or 16). In all the operations $r < k < i$ $\mathtt{selected_v^k} = zero_v$, and v can send a message only by performing procedure timeout (Lines 28-30), such that the msg part equals to $next(\mathtt{selected_v^k}, p) = next(zero_v, p)$. Hence the second part of the claim. □

Claim 13 *Let π_l be an operation after s_1 but before state q_i. If in π_l v sends to its outgoing neighbor w message m_l, such that, $msg(m_l) = next(zero_v, out\_port_v(w))$, and if either*
(1) message m_l is the last message that arrives at w before q_i, or
(2) message m_l has not yet arrived at w in state q_i,
then in state q_i there is no phony block bl^i such that both v and w are $\in bl^i$.

Proof: Case (1): Let π_j be the operation in which message m_l is received by node w.

If in q_i there is a block bl^i such that $v, w \in bl^i$ then $parent_w^i = in\_port_w(v)$ and from claim 10 $\mathtt{selected_w^i}$ equals to the msg part of m_l, the last message that was received from the parent port. Hence $\mathtt{selected_v^i} = next(zero_v, out\_port_v(w))$, a contradiction to the fact that bl^i is a phony block.

Case(2): Message m^l has not yet been received by w. Assume the opposite that $v, w \in bl^i$. Thus $\{v\vec{m}w\} \subseteq bl^i$, $m^l \in \vec{m}$. By the definition of a block, selected$_w^i = msg(m^l)$ =$next(zero_v, out\_port_v(w))$, and again this is a contradiction to the fact that the block is a phony block. □

Proof of Lemma 9: Let us prove that sub block sbl^i must exist in state q_i. To prove this assume such a sbl^i does not exist.

Then if the tail(bl^{i+1}) element exists in q_i then let e be the first element (message or node) along the path induced by bl^{i+1} that may not be a member of sbl^i because either it has a different selected (when e is a node) or an inconsistent parameter (when e is a message). There are four cases to consider: (1) Tail(bl^{i+1}) does not exist in q_i, (2) e is a node, (3) e is a message, (4) e does not exist. A general remark: By the code an operation π_i is exactly one of the following: 1. sending one new message (by procedure timeout), 2. receiving one message, 3. receiving one message and sending one new message without changing the node state, 4. receiving one message, changing the state of node v and sending one new message from node v.

Case (1) - Tail(bl^{i+1}) does not exist in q_i: If tail(bl^{i+1}) is a node then this node has changed its selected in π_i. If tail(bl^{i+1}) is a message it must have been generated in π_i. Let us consider each of the two sub cases (tail(bl^{i+1}) is a node, or a message).

In the former sub case let v be the node captured in π_i. By Claim 12, if w is an outgoing neighbor of v then the last message that v sent before state q_i was message, m, such that $msg(m) =$next$(zero_v, out_port_v(w))$

By Claim 13 v and w may not be in the same phony block in state q_{i+1}. Since v is the tail of Block bl^{i+1}, bl^{i+1} contains only one node, v. In operation π_i node v was captured after receiving a message m' and sent a strong message, $\widehat{m}$, with $msg(\widehat{m}) = next(msg(m'), out\_port_v(u))$ where u is an outgoing neighbor of v (by Line 11-13). Thus we are at case 5 of the lemma.

In the later case, in operation π_i some node v generates message m, which is the tail of bl^{i+1}. We claim that this case is impossible. Message m is sent by v to outgoing neighbor w using the procedure send_neighbors. By performing this procedure node v sent the message $next(\mathtt{selected}_\mathtt{v}^{\mathtt{i+1}}, out\_port_v(w))$. Hence from the definition of block and the fact that $m \in bl^{i+1}$, $v \cup bl^{i+1}$ is also a block, a contradiction to the fact that bl^{i+1} is a block (violating Condition 6 of the block definition (Definition 5)).

Case (2) - e is a node: Thus $selected_e^i \neq selected_e^{i+1}$ and node e was captured in operation π_i (Lines 11 - 12). Thus by Claim 12, $selected_e^i = zero_e$ and before state q_i the last message e sent to its outgoing neighbor w, was a $message$, such that $msg(e) = next(zero_e, out\_port_e(w))$. From Claim 13 e and w cannot be at the same phony block at state q_{i+1}. Hence we are again in case 5 of the lemma.

Case (3) - e is a message: If e is a weak message, then the message was generated in π_i. Weak message can be generated only by time-out (Lines 29-31). Thus we are in case 2 of the lemma.

If e is a strong message, then the message was created in π_i by some node v. We claim that $v \in sbl^i$ and $v \in bl^{i+1}$. If $v \in bl^{i+1}$ and $v \notin sbl^i$ then e is that node v and not the message, a contradiction. If $v \notin bl^{i+1}$, then in state q_{i+1} the message $msg(e) = next(\mathtt{selected}_\mathtt{v}^{\mathtt{i+1}}, l)$ where l is the port id of the link on which e is sent (procedure send-neighbors). By the definition of block $v \cup bl^{i+1}$ is a block, contradiction to condition 6 in Definition 5 of a block. Therefore $v \in sbl^i$ and $v \in bl^{i+1}$. Thus $\mathtt{selected}_\mathtt{v}^\mathtt{i} = selected_v^{i+1}$. A Strong message may be sent without changing the status of a node by performing:

1) Line 29 (time-out) but then $parent_v = nil$ and $\mathtt{selected}_\mathtt{v}^{\mathtt{i+1}} = zero_v$ (from Line 1) and this is a contradiction to the fact that the block is a phony block.

2) Line 3-4 and this is case 3 of the lemma.

Case (4) - e does not exist: In this case we should check if there is an element induced by sbl^i, that is not a member of bl^{i+1}. Let d be the first element (message or node) along the path induced by sbl^i which is not a member of bl^{i+1}. We consider two cases (1) d is a node, (2) d is a message. If d does not exist then we are in case 6 of the lemma, $sbl^i = bl^{i+1}$.

Sub-Case 1 - d is a message: then in operation π_i some node, v, received d. If $v \in sbl^i$ then if d is a strong message we are at case 3 of the lemma (Lines 3 - 4) and if d is a weak message then we are in case 1 of the lemma (weak message can only change selected of a node to zero).

If $v \notin sbl^i$ then d is the head of sbl^i. If $v \in bl^{i+1}$ we are at the case that a new node is captured, case 5 of the lemma. If $v \notin bl^{i+1}$ then only this message disappeared thus we are in case 1 of the lemma if it is a weak message or case 4 if it is a strong message.

Sub-Case 2 - d is a node: We show that in this case, $d \in sbl^i$ but $d \notin bl^{i+1}$, which is impossible - thus this whole case is impossible. If d is the tail of sbl^i then without loss of generality sbl^i is defined as the sub-block of $sbl^i \setminus \{v\}$. If the head of sbl^i is node x but $x \notin bl^{i+1}$ then with out loss of generality sbl^i is define as the sub-block of $sbl^i \setminus x$. Hence $head(sbl^i) \in bl^{i+1}$ and $tail(sbl^i) \in bl^{i+1}$.

Thus without loss of generality, d is not the head or the tail of sbl^i and there exists a node or a message $r \in sbl^i$ which is the following node or message along the block sbl^i and also $r \in bl^{i+1}$. Similarly there exists a node or a message $y \in sbl^i$ that is a previous node or message along the block sbl^i and also $y \in bl^{i+1}$. In operation π_i, d changed it's $selected_d$ or $parent_d$ so that $d \in sbl^i$ but $d \notin bl^{i+1}$ but the state of r and y did not change, thus it easy to see that according to the definition of a block it cannot be that both r and y are in the same block bl^{i+1}. □

Corollary 14 *Let π_i be an operation after state s_1, then $potential(sbl^i) \geq potential(bl^{i+1})$.*

In lemma 15 we prove that after $O(nB)$ time the potential of any phony block reduces to zero. More specifically we show that all the strong messages in a phony block disappear after at most that much time. To do that we pick an arbitrary message on a phony block, track its traversal of the block, and argue that such a traversal cannot last more than $O(nB)$ units of time. For the purpose of the proof a strong message that is relayed by a node from its in-port to its out-port is considered the same message.

In the formal proof we follow the distance between the tracked message and the head of the block (defined as *remainder* in Definition 10). However, this process and definition is problematic because continuing from the location of a specific message a block may branch into several blocks, since the overall structure is that of a tree. Thus, the specific block traversed by the message is not well defined and neither is the remainder of the message. In order to overcome this difficulty we break the proof into three steps. In the first two steps of the proof we run the algorithm forward and then observe the block in a backward execution of the same run. This solves our problem, because when the algorithm is run backwards, every block has a unique source sub-block (based on Lemma 9). Then in the third step we make the arguments necessary for the proof on the sequence of steps and blocks defined in the second step.

The three steps of the proof are thus:
(1) Run the algorithm forward $nB + n + 3$ time units and assume to the contrary that there exists a block with a non-zero potential, and hence there is a strong message in it.
(2) By a backward simulation of the run on this block and this message in previous states, we build the progression of the block and the traversal of that message, in different states.
(3) We prove on the forward sequence of blocks that the remainder of the message in that block, $nB + n + 3$ time units after the initial step, is zero. This last step is based on three points:
(1) The remainder of the message in the initial state is at most $n + 1$.
(2) At least $nB + n + 3$ times, the remainder of the message decreases by one, since at each time unit the message advances at least one more hop in the block in each time unit.
(3) The number of times the remainder of the message increases by one is at most $nB + 1$, the initial potential of the block. The correctness of the third statement is based on the following points: (a) The remainder of the message increases if a new node is captured by the block. (b) The block loses one strong message for every node that is captured by the block (except maybe for the first node that is captured to the block, see Claim 20). (c) The number of strong messages in the block is at most $nB + 1$. (d) By Lemma 9 the number of strong messages in a block cannot increase.

Lemma 15 *$O(nB)$ time units after state s_1 the potential of every phony block is zero.*

Proof: Before we prove the lemma let us begin with some definitions.

Definition 10 *Let $bl = \{\vec{m}_0 v_0 \vec{m}_1 v_1 \ldots \vec{m}_{l-1} v_{l-1} \vec{m}_l v_l \vec{m}_{l+1}\}$ be a sub-block. Define for each message $m \in \vec{m}_j$, $remainder(m, bl) = l + 1 - j$.*

Observation 16 *In any state, for every message m, and sub-block bl, $0 \leq remainder(m, bl) \leq n+1$.*

Let π_i be an operation after s_1. Let bl^{i+1} be a phony block in state q_{i+1}.

Definition 11 *The sub-source of block bl^{i+1} is the sub-block sbl^i in state q_i as defined in Lemma 9 for block bl^{i+1}.*

Definition 12 *The origin of message m^{i+1} is the strong message m^i in sbl^i where:*

1. *If in π_i v sends m^{i+1}, then m^i is the message that in response to its reception v sent m^{i+1}.*
2. *Otherwise, $m^i = m^{i+1}$.*

Definition 13 *The source block of block bl^{i+1} is block bl^i, whose suffix is sub-source of block sbl^i.*

Observation 17 $remainder(m^i, sbl^i) = remainder(m^i, bl^i)$.

The observation follows from the fact that sub-source block sbl^i is a suffix of source block bl^i.

Observation 18 $potential(bl^{i+1}) \leq potential(bl^i)$.

The observation derives from the following facts: (1) $potential(sbl^i) \leq potential(bl^i)$ since sbl^i is the suffix of bl^i, and (2) $potential(bl^{i+1}) \leq potential(sbl^i)$ (by Corollary 14).

Corollary 19 *Let m^i be the origin of m^{i+1} then there are three possible values for $remainder(m^{i+1}, bl^{i+1})$ as a function of $remainder(m^i, bl^i) - 1$:*

1. *$remainder(m^{i+1}, bl^{i+1}) = remainder(m^i, bl^i) - 1$, if in π_i message m^i was relayed by a node and sent on the next link on the block.*
2. *$remainder(m^{i+1}, bl^{i+1}) = remainder(m^i, bl^i) + 1$, if in π_i block bl^i in which message m^i resides, captures a node.*
3. *$remainder(m^{i+1}, bl^{i+1}) = remainder(m^i, bl^i)$, this happens in either of the following two cases:*
   (a) *The head of bl^i is message m^i and in π_i message m^i captures a new node and m^{i+1} is relayed by the captured node. In this case the remainder of the head message remains zero. Hence $remainder(m^{i+1}, bl^{i+1}) = remainder(m^i, bl^i) = 0$. This case is considered as the simultaneous occurrence of the first two possibilities.*
   (b) *$remainder(m^{i+1}, bl^{i+1}) = remainder(m^i, bl^i)$, if neither of the first two cases occurs, then in π_i no new node is captured by bl^i, and π_i does not effect m^i.*

The Corollary is based on Lemma 9 and on Observation 17.

Definition 14 *The progression of bl^w from state q_k to state q_w, $k \leq w$ is the series of blocks $bl^k bl^{k+1} \ldots bl^w$ at states $q_k q_{k+1} \cdots q_w$, such that, for $k \leq i \leq w$, bl^i is the source block of bl^{i+1}.*

Remark: Note that the progression of a block is defined backwards, i.e., bl^w is the first step and the source relationship between bl^i and bl^{i+1} continues the definition from $i+1$ to i.

Definition 15 *Strong message m^w traversal of block bl^w, from state q_k to state q_w, $k \leq w$ is the series of strong messages $m^k, m^{k+1} \ldots m^w$ at states $q_k, q_{k+1} \ldots q_w$, such that, for $k \leq i \leq w$, m^i is the origin of m^{i+1}, $m^i \in bl^i$, and $m^{i+1} \in bl^{i+1}$, and bl^i is the source block of bl^{i+1}.*

Remark: Note that as in the definition of block progression, the message traversal definition is also inductively backwards, i.e., the basis of the induction is message m^w and block bl^w, and the step of the induction is the origin relationship between m^i and m^{i+1}.

Before we start the formal proof of Lemma 15 we state and prove a basic claim that corresponds to Point (b) in the end of the intuition paragraph (before Lemma 15). Let bl^w be a block in state q^w and let bl^0, $bl^1, \ldots, bl^w$ be the progression of the block from state q^0 to q^w. Let $\{\pi_{i_1}, \pi_{i_2}, \ldots \pi_{i_k}\}$ be a maximal series of operations such that for every j, a new node is captured by bl^{i_j-1} in π_{i_j}.

Claim 20 *For every j, $0 \leq j \leq k-1$, there exists operation π_l, $i_j < l < i_{j+1}$ such that $potential(b^{l+1}) \leq potential(bl^l) - 1$.*

Proof: Let v be the node that is captured by sub-source block bl^{i_j} in operation π_{i_j}. Let u be the node captured by source block $bl^{i_{j+1}}$ in operation $\pi_{i_{j+1}}$. Node u is the outgoing neighbor of v. This follows from the fact that the series of $\{\pi_{i_1}\pi_{i_2}\ldots\pi_{i_k}\}$ is maximal, and from the definition of the progression $bl^0, bl^1 \ldots bl^w$.

The last message that v sends to u before operation π_{i_j}, is $m1$, such that $msg(m1) = next(zero_v, out\_port_v(u))$ (by Claim 12). In operation π_{i_j} node v sends to u, a strong message, $m2$, such that $msg(m2) = next(\mathtt{selected_v}, out\_port_v(u))$ where $m2 \in bl^{i_j+1}$ (Case 5 of Lemma 9). Hence $msg(m2) \neq msg(m1)$ (since bl^{i_j} is a phony block).

Let π_l be the operation in which u receives message $m2$. To complete the proof we show that (1) $i_j < l < i_{j+1}$ and (2) $potential(b^{l+1}) \leq potential(bl^l) - 1$.

(1) By the definition of π_l clearly $i_j < l$. Since $u \notin bl^l$ it is clear that $l < j+1$. Assume to the contrary that $u \in bl^i$, then by Claim 10 $\mathtt{selected_u}$ equals to the body of the last message received from v. Since $m1$ is the last message u received from v, $\mathtt{selected_v} = next(zero_v, out\_port_v(u))$. This contradicts the fact that bl^l is a phony block.

(2) Assume to the contrary, that $potential(b^{l+1}) > potential(bl^l) - 1$. Hence in π_l after receiving $m2$ u sends another strong message that belongs to bl^{l+1} and u performed Lines 3-4 or 10-12. Let us consider the two cases:

(a) In case Line 4 is performed, then $in\_port_u(v) = parent_u^l$ and $msg(m2) = \mathtt{selected_u^1}$ (Line 3) but by Claim 10 $\mathtt{selected_u^1} = msg(m1)$ and this is a contradiction since $msg(m1) \neq msg(m2)$.

(b) In case Lines 8-9 are performed then $in\_port_u(v) \in prev\_ports^l$ and $prev_u^l = msg(m2)$ (Line 10) but by Claim 11 $prev_u^l = msg(m1)$ and this is a contradiction since $msg(m1) \neq msg(m2)$. Hence $potential(b^{l+1}) \leq potential(bl^l) - 1$. □

Proof of Lemma 15: Let q_s be a state after s_1 and let q_w be a state which is $nB + n + 3$ time units after q_s. We prove that at state q_w the potential of every phony block reaches zero. Assume to the contrary that in state q_w the potential of a particular block bl^w is not zero. Hence, there exist a strong message which we denote as m^w, where $m^w \in bl^w$. Let $bl^s bl^{s+1} \ldots bl^w$ be the progression of the block from state q_s to q_w, and let $m^s m^{s+1} \ldots m^w$ be the traversal of the message in the different states $q_s q_{s+1} \ldots q_w$.

Let us look at the group of operations $\{\pi_i\}_{s \leq i \leq w-1}$. The following two observations about the remainder of m^i and potential of bl^i are true:

1. At least $nB + n + 3$ times, there is an operation π_i such that a node in bl^i received m^i and sends m^{i+1} (Case 1 and 3(a) of Corollary 19). Hence at least $nB + n + 3$ times: $remainder(m^{i+1}, bl^{i+1}) = remainder(m^i, sb^i) - 1$.

2. At most $nB + 1$ times an operation π_i exists such that a node is captured by block bl^i (case 2 and 3(a) of Corollary 19).
   This follows from 3 main points:

(1) By Observation 18 $potential(bl^{i+1}) \leq potential(bl^{i})$.
(2) By Claim 20 between every two consecutive operations, in which a node is captured by the block, there is an operation π_l such that $potential(bl^{l+1}) \leq potential(bl^{l}) - 1$.
(3) By Corollary 7 $potential(bl^{s}) \leq nB$.

By Observation 16 $remainder(m^s, bl^s) \leq n+1$.
$remainder(m^w, bl^w) = remainder(m^s, bl^s) + \Sigma_s^{w-1}(remainder(m^{i+1}, bl^{i+1}) - remainder(m^i, bl^i)) \leq n+1+(-1)*(nB+n+3)+(+1)*(nB+1) = -1 < 0$. Contradiction, since by Observation 16 $remainder(m^w, bl^w) \geq 0$. □

Let s_2 be the state where the potential of every phony block equals zero.

Let q_m be a state, after state s_2, and let q_p be a state which is two time units after q_m. Let $bl^m bl^{m+1} \ldots bl^p = bl$ be the progression of a phony block bl from state q_m to state q_p.

Lemma 21 *If the number of nodes in bl^p is k, then the number of nodes in bl^m at state q^m is at least $k+1$.*

Proof: Let v be the tail node in bl^m at state q_m. Clearly $parent_v \neq nil$, since otherwise $\texttt{selected}_\texttt{v} = zero_v$ (by Line 1), which is a contradiction to the fact that v belongs to a phony block. Let $parent_v = in\_port_v(u)$, where u is an outgoing neighbor of v.

In order to prove the lemma we show that maximum 2 time units after q_m there exists an operation such that $\texttt{selected}_\texttt{v}$ changes. This is sufficient, since no new nodes can be captured by any phony block after state s_2.

Let p be the previous element to the tail of block bl^m at q_m. By property 6 of the block definition $p \cup bl^m$ is not a block.

There are two cases to consider: (1) p is a message m that is transfered from u to v, or (2) p is node u.

(1) In case p is a message then after a maximum of one time unit the first node in the block receives m from its parent such that $msg(m) \neq \texttt{selected}_\texttt{v}$ (since m can not be added to the block) and $\texttt{selected}_\texttt{v}$ changes to $zero_v$ by Lines 6-7.

(2) In case p is a node then after at most one time unit from q_m, u sends a message m by timeout. Message m can not be added to the phony block, and after one time unit node u still cannot be added to the block (no new phony nodes can be captured after s_2). Hence, we are again in case (1). □

Lemma 22 *$O(n)$ time units after s_2, there are no phony blocks.*

Proof: $O(n)$ time units after s_2, the block contains only a weak message sequence (no nodes at all). This follows from the following facts: (1) By lemma 21 in each $O(1)$ time units after state q_m, the number of nodes in a block is reduced by at least one. (2) There are at most n nodes in a phony block to start with. (3) No new nodes can be added to a phony block after state s_2 (since the potential of a phony block equals zero).

$O(1)$ time units after that state, the weak message sequence also disappears. This follows from the following statements: (1) weak messages can be send only in a time-out. (2) There is no node in the phony block to send them. □

Let s_3 be the global state in which there are no phony blocks.

Definition 16 *$Selected_v^i$, the* `selected` *value of node v at state q_i, is a derivative of $zero_u$ if there is a u to v path, $\{v_1 v_2 \ldots v_k\}$, where $u = v_1$ and $v = v_k$ and an assignment to the variables of all the* *nodes along the path that defines a block on the path s.t.,* $\mathtt{selected_u} = zero_u$ *and* $\mathtt{selected^i_{v_{j+1}}} = next(\mathtt{selected^i_{v_j}}, out\_port_{v_j}(v_{j+1}))$, *where* $0 \leq j \leq k-1$. *The path is called the derivation path from* u *to* v.

Observation 23 *The selected value of any node in a state after* s_3 *is a derivative of some zero value of a node in the network.*

The observation follows since there are no phony blocks in the network.

Definition 17 Stabilization proprieties
Each frame work that is suitable for our generic algorithm defines a set of legal global and final states such that the following proprieties hold:

1. *Each node has a stable and legal* `selected` *value. Node* v *has a legal* `selected` *value at state after* s_3, *if its derivation path,* $\{v_0, v_1, \ldots, v_k, v\}$, *denoted by legal derivation path of* v, *satisfies the following proprieties:*

   (a) *There is a unique node denoted* r, *the root of the tree, such that,* $v = r$ *and* r*'s legal* `selected` *value is* $zero_r$. *Otherwise, if* $v \neq r$, $selected_v$, *the legal value of node* v *is a derivative of* $zero_r$, *i.e.,* $v_0 = r$.

   (b) *For every* $0 \leq i \leq k$, $\{v_0, v_1, \ldots, v_i\}$ *is a legal derivation path for a legal* $selected_{v_i}$ *value.*

   (c) *A legal* `selected` *value of node* v *is selected as* $selected_v$ *over any other suggestion for a non legal* `selected` *value which is a derivative of some zero value of a node in the network.*

2. *Each node has a stable parent. The parent of* r *is nil, and the parent of each other node is a pointer to one of its incoming neighbors. The graph induced by the parent relationship is a tree.*

Definition 18 *A block is a legal derivation block of* v *laid over derivation path* dp, *if the block is on the path* dp *and the assignment of the* `selected` *values of all the nodes along* dp *is legal.*

Remark A node can have more than one legal `selected` value, hence there can be more than one legal derivation path and legal derivation block.

Observation 24 *If* v *has a legal* `selected` *value after* s_3 *then any block that contains* v *must be a block or a sub-block of a legal derivation block.*

The observation follows form prosperity (b).

Observation 25 *Let* bl *be a block or a sub-block of a legal derivation block then the source of* bl *is a block or sub-block of a legal derivation block.*

Lemma 26 *In any state* q_i *which is* $O(n)$ *or more time units after* s_3, *if* $selected_v$ *is legal then there exists a legal derivation block based on the derivation path from* r *to* v.

Before proving the lemma let us define:

Definition 19 *A block without power supply is a block whose tail's* `selected` *value is not a zero value.*

Observation 27 *A legal derivation block is a block with power supply.*

The proof of the lemma is based on the following two claims:

Claim 28 *Let bl be a block without power supply at a state after s_3, then after $O(n)$ time units the block disappear.*

The proof of the claim is similar to the proof of Lemma 22 and will not be repeated here. The intuition is that a phony block is also a block without power supply. The same properties because of which a phony block disappears also hold for a block without power supply. □

Consider a state q_i after s_3 in which $selected_u$ is a legal value and a legal derivation block is laid over ldp, the legal derivation path from r to u, then,

Claim 29 *At any state after q_i, u has a legal derivation block which is laid over ldp.*

Proof:

Assume to the contrary, then there is an operation π_w such that at state q_w, after q_i, there is a legal derivation block laid over ldp and in state q_{w+1} after this operation there is no legal derivation block on ldp.

Hence, some node from ldp changes its state in operation π_w. Let v be that node. By Propriety (b) of a legal derivation path v has a legal `selected` value. By Propriety (c) it can not be that in π_w v changes its state. Contradiction. □

Proof of Lemma 26 Assume to the contrary, that there exists a node v in state q_i, $O(n)$ after s_3 such that $selected_v$ is legal but there is no legal derivation block. Let bl' be a block that contains v. Block bl' must be a sub-block of a legal derivation block at state q_i (By Observation 24) hence by Observation 27 bl' is a block without power supply.

By Claim 28, the source of bl' at state q_m before s_3 is a block with power supply. Hence, there is an operation after s_3, denoted by π_p, because of which the block is without power supply. But this is impossible since at state q_p the source of bl' is a legal derivation block (By Observation 25) and by Claim 29 the source of bl' at state q_{p+1} is also a legal derivation block, hence a block with power supply (by Observation 27). Contradiction, hence the lemma. □

Lemma 30 *$O(n)$ time units after s_h the network enters state s_4 and a legal final state L such that in any state after s_4 the network is in state L. In the legal final state L a rooted tree as required spans the network.*

Proof:

The proof is based on an inductive assumption that at state q_k (which is at least $2k$ time units after s_h), every node that has at least one legal derivation path that contains k nodes its *selected* value is legal.

Base: For $k = 1$, we should prove that the inductive assumption holds for r. By Propriety (a) of legal derivation path, $zero_r$ is selected as a legal value of $selected_r$ from all the derivative `selected` values for node r. Hence by line 2 and by Observation 23 the result.

Step: Assuming the inductive assumption holds for k, then we prove it for $k+1$. Let v be a node that has a legal derivation path with $k+1$ nodes. By Propriety (b) of a legal derivation block, node v has a neighbor w with a legal derivation path with k nodes and hence the inductive assumption holds for node w. Hence by Lemma 26 a legal derivation block exists for node w at state q_k which is at least $2k$ time units after s_h. Node w receives a strong message every time unit from its parent in the legal derivation block. This follows from the fact that the tail of a legal derivation block is r and $\texttt{selected}_r = zero_r$ (by Propriety (a) of a legal derivation block) and r sends a strong message on all of its outgoing-links every time unit (by procedure time-out Line 30). Hence, at q_{k+1}, the state which is at least $2(k+1)$ time units after s_h, v receives two strong messages from w. The body part of these messages is a suggestion for a legal `selected`. If those suggestions are selected as $selected_v$ then upon receiving the first message, $prev$ is updated (Line 14-21), and upon receiving the second message `selected` is updated (Line 11-13). Hence, at q_{k+1} $selected_v$ has a legal value. Otherwise, this suggestion for $selected_v$ is not selected as $selected_v$. By Propriety (c) of legal derivation block, the latter can happen only if $selected_v$ has already a legal `selected` value. Hence the lemma □.

This completes the proof of the main theorem.

Lemma 31 *The time complexity of the BFS is $O(D)$.*

Proof: The same inductive assumption as in Lemma 30 holds with some changes when the starting point is a state after s_1 and not after s_3. □

Remark: The proof ignored a remark that was made about the model in sections 2, and 4, thus assuming that the rate of which messages are processed is faster than the rate at which links transmit them. It is easy to modify the proof to hold with the remark. In effect the remark is equivalent to loosing some messages on the links which may only decrease the potential of phony blocks and does not disturb the power-supply principle. Notice that in the remark we introduce two buffers at each port thus preventing the possibility of few blocks uniting into one block, i.e., not effecting the correctness of Lemma 9.

Acknowledgments: We would like to thank Shlomi Dolev for several helpful discussions. This work was supported by the Broadband Telecommunications R&D Consortium administered by the Chief Scientist of the Israeli Ministry of Industry and Trade.

References

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

[AB93] 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.

[AEYH92] 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.

[AG94a] Y. Afek and E. Gafni. Distributed algorithms for unidirectional networks. *Siam J. on Computing*, 23(6):1152–1178, December 1994.

[AG94b] A. Arora and M. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:1026–1038, 1994.

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

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

[AL95] Y. Afek and T. Lev. Distributed synchronization protocols for SDH networks. Submitted for publication, November 1995.

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

[APSVD94] 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.

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

[BGM90] 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.

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

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

[CD94] Z. Collin and S. Dolev. Self-stabilizing depth first search. *Information Processing Letters*, 49:297–301, 1994.

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

[DIM91] 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.

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

[DKR82] 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.

[ELW90] 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.

[GA84] E. Gafni and Y. Afek. Election and traversal in unidirectional networks. In *Proc. of the 3rd Ann. ACM Symp. on Principles of Distributed Computing*, pages 190–198, August 1984.

[GK84] 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.

[GKA83] M. Gerla, L. Kleinrock, and Y. Afek. A distributed routing algorithm for unidirectional networks. In *Proc. GLOBCOM 83*, 1983.

[Her90] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35:63–67, 1990.

[IJ93] A Israeli and M Jalfon. Uniform self-stabilizing ring orientation. *Information and Computation*, 104:175–196, 1993.

[KP89] 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.

[Kut88] 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.

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

[MOY96] 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.

[Mul88] N. Multari. *Self-Stabilizing Protocols.* PhD thesis, Department of Computer Sciences, University of Texas, 1988. Ph.D. Thesis.

[OW95] 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.

[Pet82] 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.

[Taj77] W. P. Tajibnapis. A correctness proof of a topology information maintenance protocol for a distributed computer network. *CACM*, 20-7:477–485, 1977.

7 Appendix

A Lower bound of the synchronous algorithm

In Figure 6 a simple scenario in which our algorithm has $\Omega(n)$ time complexity is presented. In the figure, along side each node we placed the record [leader, distance from the leader], plus the node id in bold face. Let us look at the block of size $n-1$ with fake-id 1. To simplify we describe the scenario under the synchronous model. Every round only one node from the block of fake-id 1 disappears, the node which has the smallest distance in the block.

For example, in the first round the node with *id* 80 changes its state since it receives only one message [10,0] from node 10 and according to this message its leader should be 10 and not 1. Assuming this is the first time node 80 receives the change the node changes its state to [80,0], otherwise it changes to [10,1]. All other nodes in the block of fake-id 1 do not change their state since they get a message that reassures their state from their incoming neigbour in the block of fake-id 1. Similarly, in the next round node 70 changes its state since it receives the messages [80,0] or [10,1] from node 80 and [10,0] from node 10 that does not support its current leader 1. All other nodes in the block of fake-id 1 do not change their state since they get messages that reassure their state from their incoming neigbour in the block of fake-id 1. Hence, only after $n-1$ rounds the block of fake-id 1 disappears, even though the diameter of the network is 2.

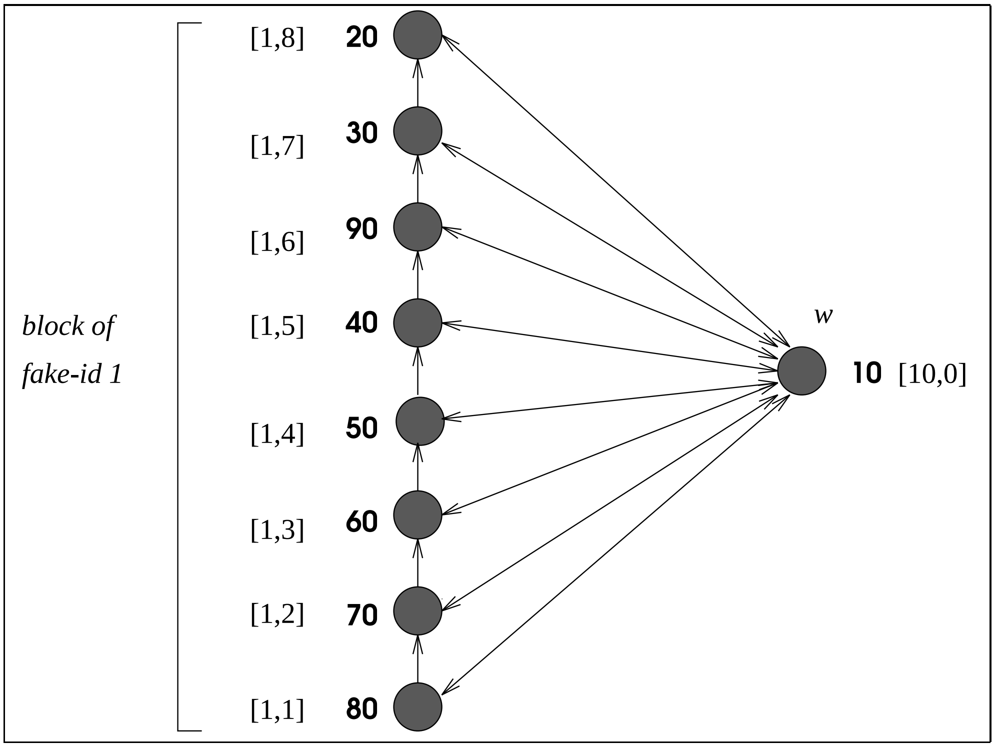

Figure 6: Example