A Leader Election Algorithm for Mobile Ad Hoc Networks

Sudarshan Vasudevan, Neil Immerman, Jim Kurose, Don Towsley

Department of Computer Science,
University of Massachusetts,
Amherst, MA 01003
{svasu,immerman,kurose,towsley}@cs.umass.edu

UMass Computer Science Techincal Report 03-01

January 13, 2003

Abstract

We present a leader election algorithm that can accommodate arbitrary (possibly concurrent) topological changes and is therefore well-suited for use in mobile ad hoc networks. The algorithm is based on finding an extrema and uses diffusing computations for this purpose. We show that the algorithm is "weakly" self-stabilizing and terminating, and present a proof of correctness using linear-time temporal logic.

Keywords: leader election, mobile ad hoc networks, diffusing computations, self-stabilization, distributed algorithms, formal verification,linear temporal logic.

1 Introduction

Leader election algorithms find many applications in both wired and wireless distributed systems. In group communication protocols, for example, a new coordinator must be elected when a group coordinator crashes or departs the system. The problem of leader election in distributed systems has been well-studied and there is a large body of literature for this problem; good surveys can be found in [1, 2]. Most of these works describe election algorithms for wired distributed systems and elect a unique node from among a set of candidate-nodes.

A mobile ad hoc network is a collection of wireless, mobile devices that make communication possible by routing packets to one another. Each node has a limited transmission radius and can communicate directly with the "neighboring" nodes that fall within this radius. Communication with other nodes is made possible by routing packets through its neighboring nodes. Since nodes are mobile, the network topology can change as nodes move in and out of transmission range of one another.

Recently, there has been considerable interest in using leader-election algorithms in wireless environments for key distribution [4], routing coordination [11], sensor coordination [16], and general control [7, 6]. Here, node mobility may result in frequent leader election, making the process a critical component of system operation. Designing distributed algorithms for such dynamically changing networks is a very challenging task.

The classical definition of the leader election problem [2] is to *eventually elect a unique leader* from among the nodes in a network. We will want to specialize this definition in two important ways. Our first modification arises from the fact that in many situations, it may be desirable to elect a leader with some system-related characteristic rather than simply electing a "random" leader. For example, in a mobile ad hoc network it might be desirable to elect the node with maximum remaining battery life, or the node with a maximum number of neighbors, as leader. Leader election based on such an ordering among nodes fits well with the class of leader election algorithms that are known as "extrema-finding" leader-election algorithms. The second modification is motivated by the need to accommodate frequent topology changes - changes that can occur during the leader election process itself. Network partitions can form due to node movement; multiple partitions can also merge into a single connected component. Given these considerations, the requirements for our leader election algorithm are: *after topological changes stop sufficiently long, every connected component will eventually have a unique leader with maximum identifier from among the nodes in that component*. It is important to realize that it is impossible to guarantee a unique leader at all times. When a network becomes partitioned, a component will be without a leader until the leader-election process terminates. Similarly, when components merge together there will temporarily be two leaders in the merged component. Thus, the modified problem definition requires that **eventually** every connected component has a unique leader. Furthermore, we require that the algorithm be able to operate in the face of node and link failures and additions, as well with network partitioning and merging.

Our proposed algorithm uses the concept of diffusing computations [8] to perform leader election. Informally, the algorithm operates as follows. Nodes periodically poll their leader. When a node is disconnected from its leader, the node detecting this event start a fresh diffusing computation to determine the new leader. We will see that several diffusing computations can be in progress concurrently, but that a node will participate only in a single diffusing computation at any one time. Eventually, when a diffusing computation terminates, the node that initiated that computation informs other nodes of the maximum node identifier which participated in that computation.

We formally specify our election algorithm and prove its correctness by showing that it is "weakly" self-stabilizing, i.e., that the algorithm can recover from an arbitrary (but finite number of) topological changes and converge to a state where every node has a unique, maximum-identity-node as its leader. This result is established using linear time temporal logic as a proof technique.

The remainder of the paper is organized as follows. In Section 2 we discuss related work. Section 3 describes our model assumptions, algorithms, and messaging in detail. Section 4 gives a brief introduction to temporal logic and the algorithm's proven properties. Conclusions and future work are discussed in Section 6. Finally, the appendix contains the detailed proof of correctness of our leader election algorithm.

2 Related Work

The problem of leader election has been widely studied in the context of wired distributed systems and recently there has been some work in the context of wireless networks. There have been several clustering and hierarchy construction schemes that can be adapted to do leader election [15, 14, 16, 10]. But all of these algorithms assume static networks and are not applicable when topology changes can occur frequently (in particular) during the election

process, as might happen in ad hoc mobile networks. Leader election algorithms for mobile ad hoc networks have been proposed in [6, 11]. As noted earlier, we are interested in an extrema-finding algorithm because for many applications it is desirable to elect a leader with some system-related characteristics. The algorithms in [6, 11] are not extrema-finding and neither of these algorithms has been extended to do extrema-finding. Extrema-finding leader election algorithms for mobile ad hoc networks have been proposed in [7]. However their protocols are not well-suited to the applications discussed in the previous section, as they require nodes to meet and exchange information in order to elect a leader.

There has been considerable work on leader election and spanning tree construction in the domain of self-stabilizing systems [19] that is important to our work. Informally, *self-stabilizing systems* are those systems that can recover from any arbitrary global state and reach a desired global state within finite time. Furthermore, this desired global state is stable, i.e., once it is reached, execution of a program action in the stable state will leave the system in the same stable state. A good survey on self-stabilization can be found in [22, 23]. We believe that much of the work from the self-stabilization literature can be leveraged to solve problems in ad hoc networks. Self-stabilizing leader election algorithms have been proposed in [20, 24, 25]. However, these algorithms assume a shared-memory model and are hence not applicable to the message-passing systems that we are interested in. In [21], however, the authors prove that for message-passing systems, problems like leader election and spanning tree construction do not admit solutions that are both *terminating* and *self-stabilizing*. An algorithm is said to be terminating if it reaches a *fixpoint* state i.e., a state in which all program actions are disabled. As we will see, our leader election algorithm achieves a slightly "weaker" form of stabilisation. As failures are not always arbitrary, it is sometimes useful to consider recovery from a restricted set of states, instead of any arbitrary state as required by stricter definition of self-stabilisation. This restricted set of states is a set of states that arise from link failures, node crashes, network partitioning and merging. We show that our leader election algorithm recovers from these states to a stable state. We also show that it is terminating and that, upon termination, our algorithm ensures that all nodes have reached an agreement on who their leader is.

Several leader election algorithms have been proposed for static networks that assume frequent process crashes and link failures that are closely related to our work. The authors in [28] propose several extrema-finding leader election algorithms for broadcast networks and which can tolerate arbitrary process failures. Every message in these algorithms is assumed to be reliably broadcast to all other nodes in the network. In their algorithms, every node that participates in election broadcasts its own identifier to all other nodes. A node upon receiving an identifier smaller than itself, in turn broadcasts its own identifier to all other nodes. If a node does not receive any other identifier for a time interval, it assumes itself to be the leader. Their algorithm is indeed applicable in a wireless ad hoc network. However, as every message has to be reliably broadcast to every other node, their algorithms can be expected to place an enormous strain on bandwidth in a wireless environment. In [26], a self-stabilizing leader election algorithms for a completely connected message-passing system has been proposed. In their model, process crashes are assumed to be permanent and no additions take place to the set of processes participating in the election after the election is initiated.

The main contribution of this paper is thus to provide a provably correct, distributed asynchronous extrema finding algorithm for a highly dynamic network. Our algorithm can tolerate arbitrary, concurrent node and link crashes,

network partitioning and merging. As a result, our algorithm is very well-suited for ad hoc networks. Furthermore, our algorithm does not make any assumptions about the underlying topology and works for an arbitrary, multi-hop network. Our algorithm is "weakly " self-stabilising and terminating - both extremely important and desirable properties.

3 Objectives, Constraints and Assumptions

In developing a leader election algorithm, we first define our system model, assumptions, and goal. Our ad hoc network is a multihop, wireless network of mobile nodes and is modeled as an undirected graph dynamically changing over time as nodes move. The vertices in the graph correspond to mobile nodes and an edge between a pair of nodes represents the fact that the two nodes are within each other's transmission radii and, hence, can directly communicate with one another. The graph can become disconnected if the network is partitioned due to node movement. We make the following assumptions about nodes and system architecture:

1. **Unique and Ordered Node IDs:** All nodes have unique identifiers. They are used to identify participants during the election process. A node ID could represent a performance attribute (e.g., remaining battery life, number of directly connected neighbors) or it could be a more traditional identifier (e.g., an address). Unique performance based identifiers can be obtained by concatenating a performance attribute with a unique traditional identifier.

2. **Links:** Links are bidirectional and FIFO, i.e., links do not reorder packets.

3. **Node Behavior:** Nodes can crash arbitrarly at any time and can come back up again at any time. Node crashes are implicitly modeled in our election algorithm as a node becoming disconnected from the network. The addition of a new node or the reboot of a previously down node is implicitly modeled by new link formations with its neighbors. When a node starts up after a crash, the election process in that node is bootstrapped again.

4. **Node-to-Node Communications:** Communication between nodes takes place using a reliable transport protocol. When a node sends a packet to another node, the sender knows whether the packet has been received by that node or not.

5. **Partition Detection:** There is a mechanism that tells whether a node is disconnected from another node. Examples of such a mechanism include polling and heartbeats.

6. **Local Connectivity Information:** Each node knows its current list of neighbors. Later in Section 4.2.2, we will describe a neighbor discovery mechanism used by each node to discover and subsequently maintain the list of its neighbors.

7. **Buffer Size:** Each node has a large enough receive buffer so to avoid buffer overflow at any point in node's lifetime.

The objective of our leader election algorithm is to ensure that:

"after a finite number of topological changes, it holds that *eventually* each node *i* has a leader which is the maximum identity node from amongst all nodes in the connected component to which *i* belongs."

4 Leader Election Algorithm

Our leader election algorithm is based on the classical termination-detection algorithm for diffusing computations by Dijkstra and Scholten [8]. In this section, we first provide an informal discussion of our election algorithm. Then we present a formal specification of the algorithm and discuss its operation in detail.

4.1 Overview

We first describe our election algorithm in the context of a static network, under the assumption that nodes and links never fail. We assume that nodes have unique identifiers and that all links are bidirectional. The algorithm operates by first "growing" and then "shrinking" a spanning tree that is rooted at the node that initiates the algorithm. A node initiates the algorithm in response to a trigger indicating that it has become disconnected from its leader. We refer to this computation-initiating node as the *source node*. As we will see, when the spanning tree shrinks completely, the source node will have adequate information to determine the maximum identity node and will then broadcast this identity to the rest of the nodes in the network.

The algorithm uses three messages.

- **Election.** Election messages are used to "grow" the spanning tree. Upon detecting leader departure, the source node, s, will start a *diffusing computation* by sending an *Election* message to all its immediate neighbors, denoted by the set N_s. Each node, *i*, other than the source, will designate the neighbor from which it first receives an *Election* message as its *parent* in the spanning tree. The parent of node *i* is denoted by the variable p_i. Upon setting its parent pointer, node *i* will propagate the received *Election* message to all its neighboring nodes (children) except its parent, i.e., the set of nodes $N_i \backslash \{p_i\}$ and may receive *Election* messages from multiple neighbors, but will have only one parent.

- **Ack.** When node i receives an *Election* message from a neighbor that is not its parent, it immediately responds with an *Ack* message. Node i will not return immediately an *Ack* message to its parent. Instead, node i will maintain a "pending *Ack*" for its parent, which it will send only after it has received an *Ack* from all of its children. As we will see shortly, the *Ack* message sent by i to its parent will contain leader-election information based on the *Ack* messages i has received from its children.

  Once the spanning tree is completely grown via propagated *Election* messages, the spanning tree starts "shrinking" back towards the source. Specifically, once all of i's outgoing *Election* messages have been acknowledged, *i* will send its pending *Ack* message to its parent, p_i. Tree "shrinkage" begins at the leaves of the spanning tree, which are parents to no other node. Eventually, each leaf will receive *Ack* messages for all *Election* messages it has sent. These leaves will thus eventually send their pending *Ack* messages to their respective parents, who in turn will send their pending *Ack* messages to their own parents and so on, until the source node receives all of its pending *Ack* messages. In a pending *Ack* message, a node announces to its parent the maximum identity among all its downstream nodes, including itself. Hence the source node will eventually have sufficient information to determine the maximum identity from among all nodes in the network, since the spanning tree spans all network nodes.

- **Leader.** Once the source node for a computation has received *Ack*s from all of its children, it then broadcasts a *Leader* message to all nodes announcing the identity of the leader.

Let us illustrate a sample execution of the algorithm. We describe the algorithm in a somewhat synchronous manner even though all the activities are in fact asynchronous. Consider the network shown in Figure 1. In this example, node *A* is the source node and starts a diffusing computation by sending out *Election* messages to its immediate neighbors, viz. nodes *F* and *B*, shown in Figure 1(a). As indicated in Figure 1(b), nodes *F* and *B* set their parent pointers to point to node *A* and in turn propagate an *Election* message to all their neighbors except their parent nodes. Hence *B* and *F* send *Election* messages to one another. These *Election* messages are immediately acknowledged since nodes *B* and *F* have already received *Election* messages from their respective parents. Note that the immediate acknowledgments are not shown in the figure. In Figure 1(c), a complete spanning tree is built. In Figure 1(d), the spanning tree starts "shrinking" as nodes *C* and *D* send their pending *Ack* messages to their respective parent nodes in the spanning tree. Each of these *Ack* messages contains the maximum identity among the nodes downstream to nodes *C* and *D*, in this case the nodes themselves, since they are the leaves of the tree. Eventually, the source *A* hears pending acknowledgments from both *B* and *F* in Figure 1(e) and then broadcasts the identity of the leader i.e., *F* shown in Figure 1(f).

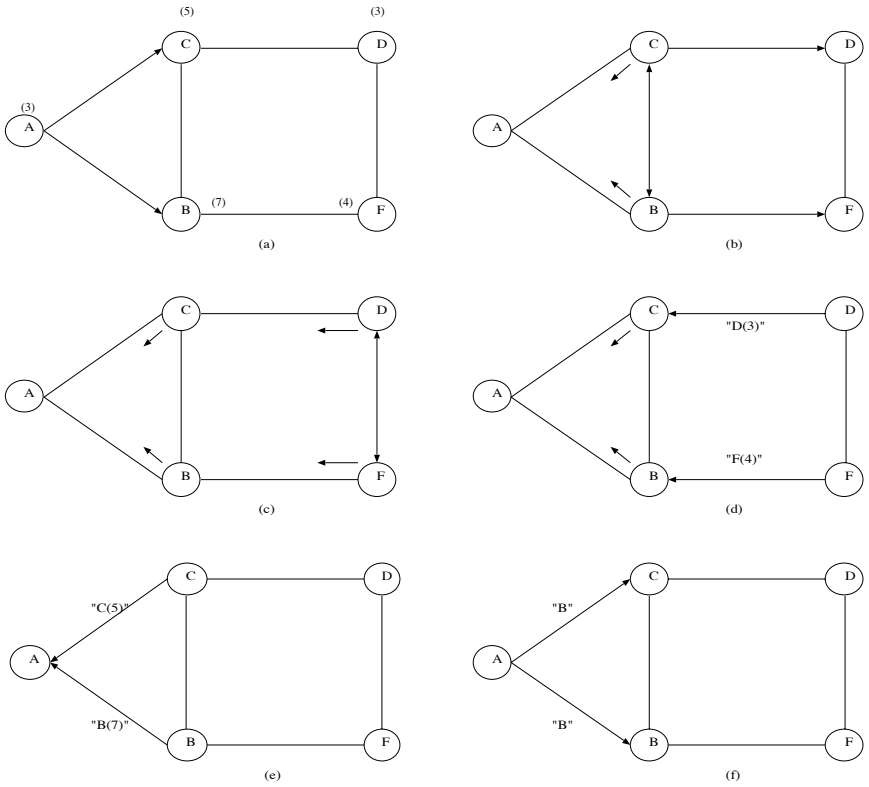

Figure 1: An execution of leader election algorithm based on Dijkstra-Scholten termination detection algorithm. Arrows on the edges indicate transmitted messages, while arrows parallel to the edges indicate parent pointers.

4.2 Detailed Discussion of Algorithm

Here we describe our leader election algorithm in detail. In the previous section, we provided an overview of the algorithm operation in a static network. But with the introduction of node mobility, node crashes, link failures,

network partitions and merging of partitions, the simple algorithm presented in the previous section is inadequate. Furthermore, we assumed in the previous section that an external input such as leader departure occurs only at a single node. In reality, many nodes can receive such inputs concurrently, with each of them starting a diffusing computation independently, due to lack of knowledge of other computations that have been started by other nodes. Before we formally specify our algorithm and describe it in detail, we briefly introduce notation used in our algorithm specification and the execution model.

4.2.1 Programming Notation and Execution Model:

The *LeaderElection* module in each node is of the form

module <module name>
var <variable declarations>;
initialization <assignment statements>;
begin
<action> ||<action>|| ... || <action>
end

Each module is defined by a set of variables, an initialization section and a set of actions. Each variable in the variable declarations list is local only to the election module on a particular node and can be updated only by that module. Variables are initialized to appropriate values in the **initialization** part of the module. Each action in the action set is of the form

$$< \text{guard} > \longrightarrow < \text{command} >$$

Each guard is a boolean expression over variables in the module and some boolean predicates. "Command" represents a list of assignment statements and perhaps one or more primitives such as send message, remove a message from receive buffer etc.

We now introduce some additional terms and definitions which we shall use throughout the rest of the paper. A *system* is defined to be a collection of processes and interconnections between processes. The *state* of the system is an assignment of values to every variable and every predicate of every process in the system. An action whose guard evaluates to true in some system state is said to be *enabled* at that state. Multiple actions can be simultaneously enabled in the same system state. In such a case, any one of the enabled actions is non-deterministically chosen for execution and the command corresponding to the guard is executed. Also, if multiple actions are simultaneously enabled, then execution of one action in the current state can potentially disable other previously enabled actions in the next system state. A *computation* of the system is a maximal, fair sequence of steps : in each state, an enabled action in that state is executed, which takes the system into its next state. The *maximality* of computation requires that *no computation be a proper prefix of another computation* while the *fairness constraint* states that *every continuously enabled action is eventually executed* [31]. Also, all action executions are *atomic* operations.

4.2.2 Algorithm Operation

1. **Algorithm Overview:**

module $LeaderElection_i(i : 1 \ldots n)$

var $\delta_i, \Delta_i, flag_i$: **boolean**;
Num_i : **integer**;
$lid_i, p_i, max_i : 1 \ldots n$;
$src_i = \langle num, id \rangle : \langle$ **integer**, $1 \ldots n \rangle$;
$N_i, S_i, L_i : \subseteq \{1 \ldots n\}$;

initialization $\delta_i, \Delta_i, flag_i, Num_i, src_i, lid_i, max_i := 0, 0, 0, 0, \langle 0, \_ \rangle, \_, i$;

begin

1. $\{\delta_i = 0 \wedge d_{i,lid_i} = \infty$
$\vee \{\exists j : j \in S_i : d_{i,j} = \infty \wedge \Delta_i = 1\}$
$\vee \{d_{i,p_i} = \infty \wedge \Delta_i = 1\}$
$\vee \{\Delta_i = 0 \wedge \delta_i = 1 \wedge d_{i,src_i.id} = \infty\}$
$\vee \{rcv_{i,j}(L) \wedge \delta_i = 1 \wedge src_i = L.src \wedge d_{i,L.lid} = \infty\}$
$\longrightarrow$ $src_i, E.src := \langle Num_i, i \rangle, \langle Num_i, i \rangle$;
$L_i, S_i, \Delta_i, \delta_i, p_i, flag_i := N_i, N_i, 1, 1, i, 0$;
$(\forall j \in N_i) send_{i,j}(E)$;
$Num_i := Num_i + 1$;
$Remove(L)$ if present;

2. $\{rcv_{i,j}(NL) \wedge j \in N_i \wedge j \notin L_i \wedge$
$\wedge src_i \neq NL.src \wedge (\delta_i = 1 \vee NL.\delta = 1)\}$
$\longrightarrow$ $Num_i := max(src_i.num, NL.src.num) + 1$;
all commands as in previous action;
$Remove(NL)$;

3. $\{rcv_{i,j}(E) \wedge (\delta_i = 0 \vee (\delta_i = 1 \wedge E.src \succ src_i))\}$
$\longrightarrow$ $src_i := E.src$;
$L_i, S_i, \Delta_i, \delta_i, p_i, flag_i := N_i \backslash \{j\}, N_i \backslash \{j\}, 1, 1, j, 0$;
$(\forall k \in N_i \backslash \{j\}) send_{i,k}(E)$;
if $(Num_i <= E.src.num)$
then $Num_i := E.src.num + 1$;
$Remove(E)$;

4. $\{rcv_{i,j}(E) \wedge \Delta_i = 1 \wedge src_i = E.src \wedge d_{i,j} \neq \infty\}$
$\longrightarrow$ $A.src, A.flag := src_i, 0$;
$send_{i,j}(A)$;
$Remove(E)$;

5. $\{rcv_{i,j}(A) \wedge \Delta_i = 1 \wedge src_i = A.src \wedge d_{i,j} \neq \infty\}$
$\longrightarrow$ $S_i := S_i \backslash \{j\}$;
if $(A.flag = 1$ **and** $A.id > max_i)$
then $max_i := A.id$;

6. $\{S_i = \{\} \wedge src_i.id \neq i \wedge \Delta_i = 1 \wedge d_{i,p_i} \neq \infty\}$
$\longrightarrow$ $A.src, A.flag, A.id, \Delta_i := src_i, 1, max_i, 0$;
$send_{i,p_i}(A)$;
Remove (E and A msgs w src $\prec src_i$*)*;

7. $\{S_i = \{\} \wedge src_i.id = i \wedge \Delta_i = 1\}$
$\longrightarrow$ $lid_i := max_i$;
$L.src, L.lid, \Delta_i, \delta_i := src_i, lid_i, 0, 0$;
$(\forall k \in N_i) send_{i,k}(L)$;
Remove (E and A msgs w src $\prec src_i$*)*;

8. $\{rcv_{i,j}(L) \wedge d_{i,L.lid} \neq \infty \wedge ((\delta_i = 1 \wedge \Delta_i = 0 \wedge L.src = src_i)$
$\vee (\delta_i = 0 \wedge lid_i < L.lid))\}$
$\longrightarrow$ $lid_i, \delta_i, flag_i := L.lid, 0, 1$;
$(\forall k \in N_i \backslash \{j\}) send_{i,k}(L)$;
$Remove(L)$;

9. $\{NewLinkFormed(i,j) \wedge j \in N_i\}$
$\longrightarrow$ $NL.src, NL.\delta, NL.flag, NL.lid :=$
$src_i, \delta_i, flag_i, lid_i$;
$send_{i,j}(NL)$;
$NewLinkFormed(i,j) := false$;

10. $\{rcv_{i,j}(NL) \wedge NL.flag = 1 \wedge NL.\delta = 0 \wedge$
$((\delta_i = 1 \wedge src_i = NL.src) \vee$
$(\delta_i = 0 \wedge d_{i,lid_i} \neq \infty \wedge d_{i,NL.lid} \neq \infty \wedge lid_i < NL.lid))\}$
$\longrightarrow$ $lid_i, \delta_i, flag_i := NL.lid, 0, 1$;
$L.src, L.lid := NL.src, lid_i$;
$(\forall k \in N_i \backslash \{j\}) send_{i,k}(L)$;
$Remove(NL)$;

end

Figure 2: Leader Election Algorithm Specification

The overall idea of our leader election algorithm specified in Figure 2 is to elect a leader by "growing" and "shrinking" a spanning tree as explained in Section 4.1. But because of node movement, nodes in the spanning tree can become disconnected from other nodes in the tree. Also two partitions each running a different diffusing computation can merge. Such pathological events can affect the correctness and termination properties of leader election algorithm. Our election algorithm adapts to such pathological events by having the ndes detecting these events take appropriate corrective measures. The corrective measure involves either starting a new computation or exchanging current state between neighboring nodes. Since we assume that there are a finite number of topological changes in the network, our algorithm ensures that eventually all nodes within a network component agree on a unique leader that belongs to that component, and that this leader is the maxi! mum identity node in that component.

2. **Variables and Message types:**

   The algorithm involves four message types: *Election*, *Ack*, *Newlink* and *Leader*, abbreviated as *E,A,NL and L* respectively. The *E*, *A* and *L* messages have the same functionality as explained in Section 4.1. When a new link is formed between two neighboring nodes, they exchange their current state via *NL* messages. The various fields in each of these messages is shown in Appendix A.

   Each node participating in the election has a unique identifier drawn from the range $[1 \ldots n]$. The boolean variable δ_i is 0 if node *i* is not currently participating in any diffusing computation, and 1 if it is. Once δ_i is set to 1, it is reset to 0 only after it receives an *L* message from the source of its current diffusing computation or after sending an *L* message in case *i* itself is the source of the computation. The variable src_i contains the *computation-index* of the diffusing computation in which node *i* is currently participating. This *computation-index* uniquely identifies a computation; it is needed since there can be multiple, concurrent computations. While in a diffusing computation, node *i*'s parent in the spanning tree is stored in the variable p_i. The variable Δ_i is set to 0 if node *i* has sent its pending *A* message to its parent and 1 if it has not, i.e. it is stil! l in the spanning tree. It is easy to see that $\Delta_i = 1 \Rightarrow \delta_i = 1$, but the implication does not hold in the opposite direction. Also $\delta_i = 0 \Rightarrow \Delta_i = 0$. The variable max_i is used to hold the identity of the maximum downstream node from *i* in the spanning tree and is included in *i*'s *A* message to its parent p_i. The variable Num_i is set to one more than the *num* field of the maximum computation index it has seen so far. $flag_i$ is a boolean variable that is set to 1 if node *i* has a leader and 0 if it does not. At the end of the computation node *i* stores its leader in the variable lid_i.

   Node *i* maintains its current list of neighbors in the variable N_i using a neighbor discovery mechanism that is explained later in this section. If *i* is currently in a diffusing computation, then L_i represents the list of nodes to which *i* sent an *E* message. The list S_i represents the set of nodes from whom *i* is waiting to hear an *A* message from. When *i* receives an *A* message from a node *j*, it removes *j* from the list S_i. When S_i becomes empty, *i* sends an *A* message to its parent p_i. It should be noted that in the algorithm specification, predicate $rcv_{i,j}(m)$ holds true as long as there is a message m originated by node *j* in *i*'s receive buffer.

3. **Algorithm Performed By The Nodes:**

In this section, we describe the exact algorithm performed by an arbitrary node *i*. This exact specification is shown in Figure 2. The *LeaderElection* module on every node loops forever and on each iteration checks if any of the actions in the algorithm specification are enabled, executing at least one enabled action on every loop iteration. The bootstrapping of election module involves assigning values to variables as specified in the **initialization**[1] part of the *LeaderElection* module.

Each node *i* periodically polls its leader, lid_i, to check if it is still alive. The election process is triggered in node *i* when it notices departure of its leader, as denoted by $d_{i,lid_i} = \infty$ in the algorithm specification. It is obvious that more than one node can concurrently detect leader departure and each of them can initiate diffusing computations independently leading to concurrent diffusing computations. Since each of these computations have the same goal i.e. to elect a new maximum identity leader, we need to minimise this duplication of effort. Furthermore, the outcome of election is not affected by the identity of the node that initiated the computation and a node has to unnecessarily maintain a large amount of state if it participates in multiple diffusing computations at the same time. We, therefore, handle multiple, concurrent diffusing computations by requiring that each node participate in only a single diffusing computation at any given time. In order to achieve this, each diffusing computation is assigned, what we call, a *computation-index*. This computation-index is a pair, viz. $\langle num, id \rangle$. id represents the identifier of the node which initiated that computation and num is like a logical clock and is set to one more than the maximum of all nums the source has seen upto the point it initiated the computation. As mentioned in Section 4.1, this num is maintained in a variable, Num_i, which is initialized to 0. It should be noted that there is a total ordering on computation indices, since a source cannot start two different computations with same num value and *source-id* field is used to break ties amongst diffusing computations with different sources but same num value.

Definition: $\langle num_1, id_1 \rangle \succ \langle num_2, id_2 \rangle \iff ((num_1 > num_2) \vee ((num_1 = num_2) \wedge (id_1 > id_2)))$

A diffusing computation *A* is said to have higher priortiy than another diffusing computation *B* iff :

$$\textit{computation-index}_A \succ \textit{computation-index}_B$$

When a node participating in a diffusing computation "hears" another computation with a higher priority, then the node stops participating any further in its current computation in favor of the higher priority computation. This is shown in Figure 3 in which all nodes are assumed to be currently involved in a diffusing computation. In Figure 3(a), node D sends an *Election* message with a higher priority computation-index, <3,D>, to node C whose current computation-index is <2,B>. Upon receiving this *Election* message, node C stops participating in its current computation and sets its computation-index to <3,D>, as shown in Figure 3(b), and propagates the received *Election* message to nodes A and B.

A new diffusing computation is initiated by a node only by execution of action 1 or action 2. The guards in actions 1 and 2 capture the various pathological conditions that cause node *i* to trigger a new diffusing computation. The various pathological conditions are:

[1]Note that in the **initialization** part, we assign $lid_i = \_$. We mean this to imply that lid_i is disconnected from *i* and hence $d_{i,lid_i} = \infty$.

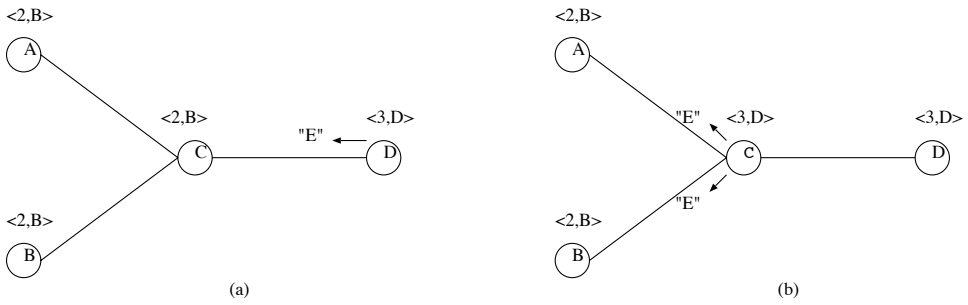

Figure 3: Computation indices and priorities between them

(a) a node *i*'s leader has departed in which case the condition $\delta_i = 0 \wedge d_{i,lid_i} = \infty$ will hold.

(b) a node *i* is currently in the spanning tree but its parent in the spanning tree has become disconnected, i.e. $\Delta_i = 1 \wedge d_{i,p_i} = \infty$.

(c) a node *i* is in the spanning tree, but a node from whom it is waiting to hear an *A* message has become disconnected, i.e. $\Delta_i = 1 \wedge j \in S_i \wedge d_{i,j} = \infty$.

(d) a node *i* is not in any spanning tree, but it is still waiting to hear a *L* message from its source and its source has become disconnected, i.e. $\Delta_i = 0 \wedge \delta_i = 1 \wedge d_{i,src_i.id} = \infty$.

(e) a node *i* is not in any spanning tree and it has just heard a *L* message from its source, but the leader in the *L* message has become disconnected after the source broadcasted the *L* message, i.e. $\Delta_i = 0 \wedge \delta_i = 1 \wedge d_{i,L.lid} = \infty$.

(f) a node *i* has received an *NL* message from one of its new neighbors which has a different computation-index than itself and which does not belong to L_i, the list of nodes to whom *i* sent an *E* message with its current source, and atleast one of *i* or its neighbor is participating in a diffusing computation. This condition is given by the guard in action 2.

While conditions (a)-(e) are self-explanatory, we shall motivate condition (f) with a simple example. Consider the example network shown in Figure 4. In Figure 4(a), all nodes are participating in the diffusing computation that is initiated by node E and node A has node E as its parent node. Assume that node D is waiting to hear an *Ack* message for an *Election* message that it sent to node B.

In Figure 4(b), node E becomes disconnected from node A and hence A starts a new diffusing computation by sending out an *Election* message to node B, as shown in Figure 4(c). At the same time nodes C and D start moving left as indicated by the dotted arrows, and before node A's *Election* message could reach node B, nodes C and D lose their links with node B and node C becomes A's neighbor instead, as shown in Figure 4(d). Thus, node D will wait forever for an *Ack* message from node B. However, following (f), nodes A and C will exchange their current state via *Newlink* messages and node A will then start a new diffusing computation thereby preventing node D from having to wait forever.

As stated in Section 3, for conditions (a)-(e), we use polling to determine whether node *i* is disconnected from another node. In executing action 1 or action 2, node *i* sets its δ_i to 1 indicating that it is currently participating

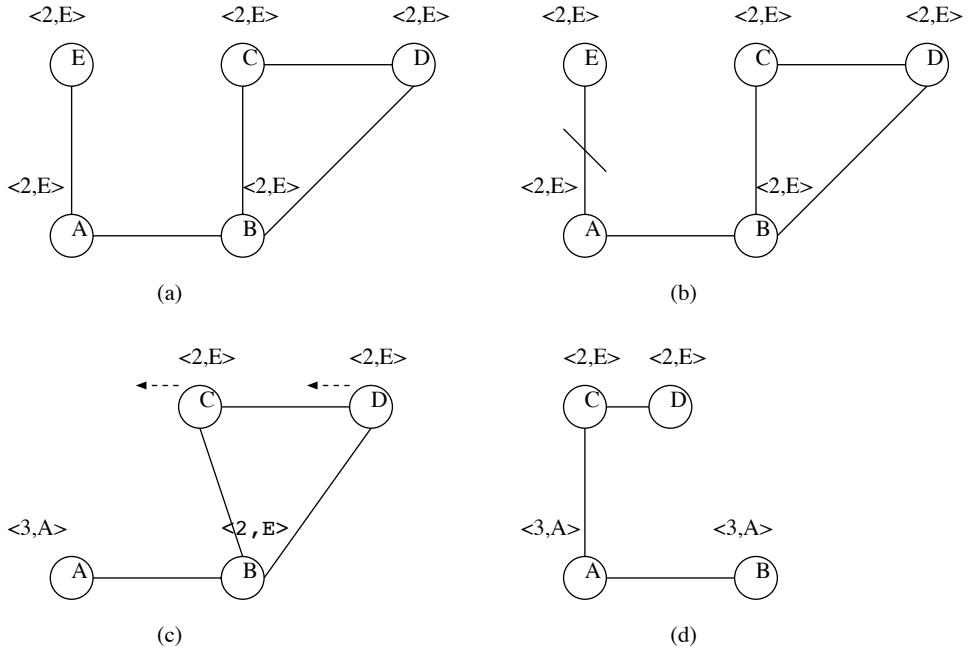

Figure 4: One of the conditions triggering a new diffusing computation

in a diffusing computation. *i* sets its parent pointer p_i to point to itself and sets its variable Δ_i to 1. Both L_i and S_i are set to N_i, *i*'s current list neighbors. The num field in src_i is set to the value contained in the variable Num_i and the id field is set to its own identifier. Node *i* then starts the process of "growing" a spanning tree by sending out *E* messages to all its neighboring nodes, given by the list N_i.

A node *i* upon hearing an *E* message with source $E.src$, starts participating in computation $E.src$ only if it is not in any computation i.e. $\delta_i = 0$ or if the newly arrived *E* message belongs to a higher priority computation than the one it is currently in, in other words $\delta_i = 1 \wedge E.src \succ src_i$ holds true. This is captured by action 3 of algorithm specification. Node *i* enters into this new computation by setting its parent pointer to the sender of *E* message and itself propagates this *E* message to all its neighbors except the one it just heard the *E* message from, thus searching for new nodes to be admitted into the spanning tree. The different variables are appropriately set as in the commands of actions 1 and 2. However, *i* does not immediately send an *A* message to its parent. Any other *E* messages that node *i* receives such that $E.src = src_i$ are immediately acknowledged by *i* ! as specified in action 4 of the algorithm.

Upon receiving an *A* message from a node *j* such that $A.src = src_i$, *i* removes *j* from the list, S_i, of nodes from which it is waiting to hear an *A* message. This is captured in action 5. If the received *A* message has $A.flag$ set to 1, then it means that the sender is *i*'s child in the spanning tree and the $A.id$ field has the identity of the maximum downstream node from the sender. Node *i* updates its max_i variable if $A.id$ is greater than its

current max_i. Action 6 dictates that if a node *i* has no more *A* messages to be heard and if it is not the source of the diffusing computation, then it sends an *A* message to its parent, thus causing the spanning tree to "shrink". In this message, *i* sets $A.flag = 1$ to indicate that $A.id$ field is valid and sets $A.id$ to max_i, the maximum identity from amongst all nodes downstream to it in the spanning tree. The leaves of the spanning tree have no do! wnstream nodes and hence they set $A.id$ field to their own identity. If, on the other hand, node *i* were the source of the diffusing computation and it has heard all its pending *A* messages, then *i* executes action 7 and announces the maximum identity node in the *L* message, which is flooded throughout the network. In executing actions 6 and 7, node *i* removes all *E* and *A* messages with source less than its current source in src_i. This is a "clean-up" step which purges all *E* and *A* messages belonging to computations with lower priority than src_i. Otherwise, the *E* messages remain in the receive buffer even after δ_i has gone to 0, thereby enabling action 3 and unnecessarily starting a new diffusing computation.

Action 8 implies that if a node *i* which is currently in a computation ($\delta_i = 1$) receives an *L* message whose source is same as src_i and if the leader in the *L* message has a finite distance to node *i*, then node *i* stops participating in that computation by adopting that leader and propagates the *L* message to all its neighbors except the one it received from. In executing action 8, node *i* checks to see if it is still connected to the leader given the *L* message since it is possible that the leader has become disconnected after the source of computation sends out *L* message. Similarly, if *i* currently has a leader but it receives a higher identity leader in *L* message, then it adopts the higher identity node as its new leader and propagates the *L* message to all its neighbors. This could happen, for instance, when two partitions merge and each has a unique leader. Then the partition with smaller leader! accepts the leader of the other partition as its own leader by flooding *L* messages within its partition.

Action 9 is enabled when a node *i* has a newly arrived neighbor *j*, as indicated by predicate $Newlinkformed(i, j)$ which is set to true by our neighbor discovery mechanism. Action 9 when executed will induce node *i* to send *NL* message to node *j*. In this message, *i* sends its current values for variables $src_i, \delta_i, lid_i, flag_i$. The variable $flag_i$ is a binary variable that indicates whether the lid field in the message is valid (i.e. $\delta_i = 0 \wedge d_{i,lid_i} \neq \infty$). Action 9 enables new neighbor pair to know what the other neighbor's current state is, and subsequently to check if action 2 or action 10 is enabled. Upon receiving a *NL* message with a "valid" leader and whose identity is greater than *i*'s current leader, *i* executes action 10 by adopting the higher identity node as leader and propagating new leader further to its neighbors in a *L* message. This condition captures the case when! two partitions, each with a unique leader, merge. This action enables the two leader identities to be exchanged over the newly formed link(s) and eventually the maximum of the two identities gets accepted as the leader for the entire merged network.

4. **Neighbor Discovery Mechanism :**

   While specifying the algorithm, we assumed that the node executing the election algorithm knows its list of neighbors. In practice, each node *i* maintains a list of neighbors using a discovery mechanism. After every timer interval, each node *i* broadcasts a *Beacon* message containing its identity to all nodes that are within its transmission range. Associated with each neighboring node *j*, *i* maintains the time when it last heard the

beacon message from node *j*, τ_{ij}. Upon receiving a *Beacon* message from a node *j*, node *i* checks to see if node *j* is already in N_i, *i*'s current list of neighbors. If node *j* is not already in N_i, *i* adds *j* to N_i and sets the predicate $Newlinkformed(i, j)$ to true. τ_{ij} is set to *i*'s current local time. If node *j* is already in *i*'s list of neighbors, only τ_{ij} is updated to node *i*'s current local time. Absence of a *! Beacon* message from a neighbor *j* for three successive *Beacon Intervals* causes node *i* to remove node *j* from N_i.

4.3 Formal Verification of Algorithm

One of the main contributions of this paper is to provide a formal proof of correctness of our leader election algorithm. We use linear time temporal logic as a formal tool for this purpose. An extensive introduction to temporal logic and its use to verify communication protocols can be found in [30]. Temporal logic has also been discussed in [29].

A temporal formula consists of predicates, boolean operators ($\vee, \wedge, \neg, \Rightarrow, \iff$), quantification operators ($\forall, \exists$) and temporal operators like $\Box$ ('at every moment in the future'), $\Diamond$ ('eventually'), $\blacklozenge$ ('at some moment in the past'), $\blacksquare$ ('at every moment in the past'), $\odot$ ('at next time instant'), $\mathcal{U}$ ('until'), $\mathcal{W}$ ('unless'), $\mathcal{S}$ ('since'), $\mathcal{J}$ ('just'). If φ and ψ are arbitrary formulas, then $\Box\,\varphi$ means φ is true at every moment in the future. $\Diamond\,\varphi$ means φ will be true at some moment in the future. $\varphi\,\mathcal{U}\,\psi$ means that ψ will eventually be true and φ will be contiunously true until that moment. $\mathcal{W}$ is a "weak until" operator, i.e. $\varphi\,\mathcal{W}\,\psi$ means that either φ holds indefinitely or $\varphi\,\mathcal{U}\,\psi$ holds. $\blacksquare\,\varphi$ means that at every moment in the past φ holds true. $\blacklozenge\,\varphi$ means that at some moment in the past φ holds. $\varphi\,\mathcal{S}\,\psi$ means that ψ has been true at some moment in the past and φ has been continuously true since that moment. $\odot\,\varphi$ means that at the next time instant φ will hold true while $\mathcal{J}\,\varphi$ means that φ has just become true. In our proofs, we introduce another temporal operator $\blacklozenge_\tau$. Thus $\blacklozenge_\tau\,\varphi$ means that φ was true at some moment in the past after time τ.

We show in Appendix B that starting at any state that satisfies predicate I and assuming a finite number of topological changes, the *LeaderElection* algorithm is guaranteed to reach, within a finite number of steps, a state satisfying the state predicate G, where

$$I \equiv (\exists i :: guard_1 \vee guard_9)$$
$$G \equiv \Box(\forall i :: \delta_i = 0 \wedge d_{i,lid_i} \neq \infty \wedge lid_i = max\{k | d_{i,k} \neq \infty\})$$

and $guard_n$ represents the boolean guard of action n from the algorithm specification.

The predicate I captures the arbitrary failures that can occur in the system and which can affect the correctness and termination of the election algorithm. Since G is a stable predicate, the *LeaderElection* algorithm is said to be I stabilizing to G or $I \rightsquigarrow G$.

The proof of this result is divided into two parts and is shown in Appendix B:

- *Safety Property:* If diffusing computations stop in the network, then eventually all nodes will have a unique leader which is the maximum identity node in the network. More formally we prove that,

$$\Box(\forall i :: \delta_i = 0 \wedge d_{i,lid_i} \neq \infty) \Rightarrow \Diamond\Box(\forall i :: lid_i = max\{k | d_{i,k} \neq \infty\})$$

- *Progress Property:* We also show that eventually there are no more diffusing computations in the network.

$$I \rightsquigarrow \Diamond\Box(\forall i :: \delta_i = 0 \wedge d_{i,lid_i} \neq \infty)$$

- *Termination Property:* Eventually the algorithm terminates i.e. none of the program actions are enabled.

The *Safety* and *Progress* properties together ensure that the system satisfies the property $I \rightsquigarrow G$. The *Termination* property is achieved as a by-product of the *Safety* and *Progress* properties.

5 Conclusions and Future Work

In this paper, we have proposed an asynchronous, distributed extrema finding algorithm for mobile, ad hoc networks. This algorithm guarantees that after a finite number of topological changes, every network component has a unique leader and this leader is the maximum identity node in that component. We also observe that our algorithm is weakly self-stabilising and terminating. We formally establish the correctness of our leader election algorithm using linear time temporal logic.

We are currently working on demonstrating the effectiveness of our election algorithm using simulations. We have simulated the algorithm in GloMoSim [9], a mobile ad hoc network simulator. The metrics that we are particularly interested in are : fraction of the time each node is with a leader and message complexity of the algorithm. In the leader election algorithm that we proposed, we assume a reliable transport mechanism. But this can be a constraint in wireless networks in which communication links are bandwidth limited. We would like to study election algorithms that can tolerate message losses and can tradeoff algorithm correctness for performance gains. The theory of self-stabilizing systems looks very promising for designing correct, distributed algorithms for ad hoc networks. We believe that a lot of work from self-stabilizaing systems can be applied to designing distributed algorithms for ad hoc networks. Our future work will concentrate on applying this theoretical framework to other ad hoc networking problems.

References

[1] G. Tel. Introduction to Distributed Algorithms. Second Edition, Cambridge University Press.

[2] N. Lynch. Distributed Algorithms. ©1996, Morgan Kaufmann Publishers, Inc.

[3] C. Wong, M. Gouda and S. Lam. Secure Group Communication using Key Graphs. In *Proceedings of ACM SIGCOMM '98*, September 1998.

[4] B. DeCleene *et al.* Secure Group Communication for Wireless Networks. In *Proceedings of MILCOM 2001, VA*, October 2001.

[5] H. Harney and E. Harder. Logical Key Hierarchy Protocol. *Internet draft*, draft-harney-sparta-lkhp-sec00.txt, March 1999.

[6] N. Malpani, J. Welch and N. Vaidya. Leader Election Algorithms for Mobile Ad Hoc Networks. In *Fourth International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications, Boston, MA*, August 2000.

[7] K. Hatzis, G. Pentaris, P. Spirakis, V. Tampakas and R. Tan. Fundamental Control Algorithms in Mobile Networks. In *Proceedings of 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 251-260, 1999.

[8] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. In *Information Processing Letters*, vol. 11, no. 1,pp. 1-4, August 1980.

[9] X. Zeng, R. Bagrodia and M. Gerla. GloMoSim: a Library for Parallel Simulation of Large-scale Wireless Networks. In *Proceedings of 12th Workshop on Parallel and Distributed Simulations, Alberta, Canada*, May 1998.

[10] D. Estrin,R. Govindan,J. Heidemann and S. Kumar. Next Century Challenges : Scalable Coordination in Sensor Networks. In *Proceedings of ACM MobiComm*,August 1999.

[11] E. Royer and C. Perkins. Multicast Operations of the Ad Hoc On-Demand Distance Vector Routing Protocol. In *Proceedings of Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, pages 207-218, August 15-20, 1999.

[12] A. Rosenstein, J. Li and S. Tong. MASH : The Multicasting Archie Server Hierarchy. In *ACM Computer Communication Review*, 27(3), July 1997.

[13] D. Thaler and C. Ravishankar. Distributed top-down hierarchy construction. In *Proceedings of the IEEE INFOCOMM*, 1998.

[14] D. Coore, R. Nagpal and R. Weiss. Paradigms for Structure in an Amorphous Computer. *Technical Report 1614*, Massachussetts Institute of Technology Artificial Intelligence Laboratory, October 1997.

[15] P. Tsuchiya. The Landmark Hierarchy : A new hierarchy for routing in very large networks. In *Proceedings of the ACM SIGCOMM*, 1988.

[16] W. Heinzelman,A. Chandrakasan and H. Balakrishnan. Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In *Proceedings of Hawaiian International Conference on Systems Science*, January 2000.

[17] V. Park and M. Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. In *Proceedings of IEEE INFOCOM*, April 7-11, 1997.

[18] E. Gafni and D. Bertsekas. Distributed Algorithms for generating loop-free routes in networks with frequently changing topology. In *IEEE Transactions on Communications*, C-29(1):11-18, 1981.

[19] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. In *Communications of the ACM*, 17:634-644,1974.

[20] A. Arora and M. Gouda. Distributed Reset. In *IEEE Transactions on Computers*, 43(9), 1026–1038, 1994.

[21] A. Arora and M. Nesterenko. Unifying stabilization and termination in message-passing systems. *21st International Conference on Distributed Computer Systems (ICDCS'01), Phoenix*, 2001.

[22] A. Arora. Stabilization. Invited chapter, to appear in *Encyclopedia of Distributed Computing*, edited by Partha Dasgupta and Joseph E. Urban, Kluwer Academic Publishers, 2000.

[23] M. Schneider. Self-Stabilization. In *ACM Computing Surveys*, 25(1), 45-67, 1993.

[24] Y. Afek, S. Kutten and M. Yung. Local Detection for Global Self Stabilization. In *Theoretical Computer Science*, Vol 186 No. 1-2, 339 pp. 199-230, October 1997.

[25] S. Dolev, A. Israeli and S. Moran. Uniform dynamic self-stabilizing leader election part 1: Complete graph protocols. Preliminary version appeared in *Proceedings of 6th International Workshop on Distributed Algorithms, (S. Toueg et. al., eds.)*, LNCS 579, 167-180, 1992), 1993.

[26] M. Aguilera, C. Gallet, H. Fauconnier, S. Toueg Stable leader election. In *LNCS 2180*, p. 108 ff.

[27] C. Perkins and E. Royer Ad-hoc On Demand Distance Vector Routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, New Orleans, LA, February 1999, pages 90-100.

[28] J. Brunekreef, J. Katoen, R. Koymans and S. Mauw. Design and Analysis of Leader Election Protocols in Broadcast Networks. In *Distributed Computing*, vol. 9 no. 4, pages 157-171, 1996.

[29] Z. Manna and A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems - Specification. In *Springer-Verlag*, New York,1992.

[30] R. Gotzhein. Temporal logic and its applications - a tutorial. In *Computer Networks and ISDN systems*, 24:203-218, 1992.

[31] N. Francez. Fairness Springer-Verlag, 1986.

APPENDIX

A Messages and Message Structure

The various message types used in the algorithm are:

- Election
- Ack
- Newlink
- Leader

Node i's Election message has the following fields:

- src : set to computation-index of source
- $destid$: set to destination node identifier

Node i's Ack message has the following fields:

- src : set to computation-index of source
- $destid$: set to destination node identifier
- $flag$: set to 1 if $destid = p_i$, otherwise 0
- id : set to identifier of maximal downstream node (valid if flag = 1)

Node i's Newlink message has the following fields:

- src : set to src_i
- $destid$: set to destination node identifier
- δ : set to δ_i
- $flag$: set to $flag_i$
- lid : set to lid_i, valid only if $\delta_i = 0 \wedge flag_i = 1$

Node i's Leader message has the following fields:

- src : set to computation-index of source
- lid : set to identifier of leader

B Proof of correctness for Election module

B.1 Notation used and their semantics

- $rcv_{i,j}(msg)$: a predicate that is true as long as node *i* has *msg* sent by node *j* in its receive buffer.
- $rcv_i(msg)$: a predicate that is true as long as node *i* has *msg* in its receive buffer.
- $send_{i,j}(msg)$: a predicate that is true when node *i* has sent a message *msg* to node *j*.
- msg_x : a message *msg* whose computation-index is *x*.
- $msg(d)$: a message *msg* containing data *d*.
- $msg.f$: field *f* of message *msg*.
- def_i : number of acknowledgements that node *i* is waiting to hear.
- $D_{i,m}$: set of descendants (including itself) of node *i* in the spanning tree corresponding to the diffusing computation with computation-index *m*.

B.2 Proof of correctness:

Let M denote the guard in action 1 and N denote the guard in action 9 of *LeaderElection* module.

$$\text{Let } I \equiv M \vee N, \text{ and}$$
$$G \equiv \Diamond\Box(\forall i :: \delta_i = 0 \wedge d_{i,lid_i} \neq \infty \wedge lid_i = max\{k | d_{i,k} \neq \infty\})$$

Lemma 1: If a diffusing computation is guaranteed to "shrink" the spanning tree completely, then the initiator of that computation outputs the extremum of all nodes that participate in that computation.

$\forall k, m :: ((\delta_k = 0 \wedge src_k = m \wedge k = m.id) \Rightarrow$
$\quad \forall i :: ((\delta_i = 0 \wedge src_i = m \wedge \blacklozenge\, rcv_i(E_m)) \Rightarrow lid_i = lid_k = max\{p | k \cup \blacklozenge\, rcv_p(E_m)\}))$

Proof: Let *m* and *k* be arbitrary computation-index and node identifier respectively such that eventually :

$$P(m,k) \equiv (\delta_k = 0 \wedge src_k = m \wedge k = m.id)$$

First of all, we claim that all nodes that receive E_m participate in computation *m* until it gets over. Stated formally,

Claim 1:

$$(\Diamond P(m,k)) \Rightarrow \Box(\forall i :: rcv_i(E_m) \Rightarrow \Diamond((\Delta_i = 1 \wedge src_i = m)\, \mathcal{U}\, (\Delta_i = 0 \wedge src_i = m)))$$

Proof: We shall first prove that:

$$(\Diamond P(m,k)) \Rightarrow \Box(\forall i :: rcv_i(E_m) \Rightarrow \Diamond(\Delta_i = 0 \wedge src_i = m))$$

The proof is by contradiction. Let us assume that:

$$(\Diamond P(m,k)) \wedge \Diamond(\exists i, j : i \neq j : rcv_{i,j}(E_m) \wedge \Box\neg(\Delta_i = 0 \wedge src_i = m)) \tag{1}$$

Since computation *m* is exactly a single Dijkstra-Scholten algorithm initiated by node *k*, we shall make use of some proven properties of Dijkstra-Scholten Algorithm. Specifically, we use a property (Lemma 19.1 (6), pg 624 [2]) which implies that if $P(m,k)$ holds true, then none of the channels and node buffers have E_m message. From the algorithm specification, it should be noted that the variable Num_i in our algorithm is always strictly increasing between two successive computations initiated by node *i*. Hence, *i* cannot initiate two computations with same computation-indices. Hence, if $P(m,k)$ holds true in the current state, then by Lemma 19.1(6) [2] and the fact that Num_i is strictly increasing, at no moment in the future can there be a node *i* for which $\Delta_i = 1 \wedge src_i = m$ holds true.

$$\therefore P(m,k) \Rightarrow \Box(\forall i :: \neg(\Delta_i = 1 \wedge src_i = m))$$

Thus, it trivially follows that:

$$\therefore P(m,k) \Rightarrow \Box(\forall i :: ((\Delta_i = 1 \wedge src_i = m) \Rightarrow ((\Delta_i = 1 \wedge src_i = m)\, \mathcal{U}\, (\Delta_i = 0 \wedge src_i = m)))) \tag{2}$$

The same property(Lemma 19.1 (6), pg 624 [2]) also implies that if $P(m,k)$ holds true in the current state, then for each node *i* that was in the spanning tree associated with computation-index *m*, $\Delta_i = 0 \wedge src_i = m$ must have been

true at some moment in the past. Also, from the algorithm actions it is obvious that for $\Delta_i = 0 \wedge src_i = m$ to hold true, action 6 or 7 of the algorithm needs to be executed for which $\Delta_i = 1 \wedge src_i = m$ is a precondition.

From the above discussion and (2), we have

$$(\Diamond P(m,k)) \Rightarrow \Box(\forall i :: ((\Delta_i = 1 \wedge src_i = m) \Rightarrow ((\Delta_i = 1 \wedge src_i = m)\, \mathcal{U}\, (\Delta_i = 0 \wedge src_i = m)))) \quad (3)$$

It is easy to see that,

$$\forall i :: \Box\neg(\Delta_i = 0 \wedge src_i = m) \Rightarrow \Box\neg((\Delta_i = 1 \wedge src_i = m)\, \mathcal{U}\, (\Delta_i = 0 \wedge src_i = m))$$

Hence from (3) we infer that,

$$\forall i :: ((\Diamond P(m,k) \wedge \Box\neg(\Delta_i = 0 \wedge src_i = m)) \Rightarrow \Box\neg(\Delta_i = 1 \wedge src_i = m))$$

Substituting the statement above in (1), we get

$$(\Diamond P(m,k)) \wedge \Diamond(\exists i, j : i \neq j : rcv_{i,j}(E_m) \wedge \Box\neg(\Delta_i = 1 \wedge src_i = m))$$

$$\blacksquare(\Diamond P(m,k) \wedge \Diamond(\exists i, j : i \neq j : rcv_{i,j}(E_m) \wedge \Box\neg(\Delta_i = 1 \wedge src_i = m)))$$

Let us assume without loss of generality that $send_{j,i}$ was true at some moment in the past, but that node *i* is yet to receive E_m. The reasoning that follows will be very similar even if we assume $send_{j,i}$ will eventually be true.

$$\therefore \blacklozenge(\Diamond P(m,k) \wedge (\exists i, j : i \neq j : send_{j,i}(E_m) \wedge \Diamond(rcv_{i,j}(E_m) \wedge \Box\neg(\Delta_i = 1 \wedge src_i = m))))$$

But node *i* cannot send node *j* an A_m message until it receives an E_m message from node *j*.

$$\therefore \blacklozenge(\ \Diamond P(m,k) \wedge (\exists i, j : i \neq j : (\Delta_j = 1 \wedge src_j = m) \wedge send_{j,i}(E_m) \wedge (\neg send_{i,j}(A_m)\, \mathcal{U}\, (rcv_{i,j}(E_m) \wedge \Box\neg(\Delta_i = 1 \wedge src_i = m))))$$

$$\therefore \blacklozenge(\ \Diamond P(m,k) \wedge (\exists i, j : i \neq j : (\Delta_j = 1 \wedge src_j = m) \wedge send_{j,i}(E_m) \wedge \Box\neg send_{i,j}(A_m)))$$

$$\therefore \blacklozenge(\ \Diamond P(m,k) \wedge (\exists j :: (\Delta_j = 1 \wedge src_j = m) \wedge \Box\neg(def_j = 0 \wedge src_j = m))$$

$$\therefore \blacklozenge(\ \Diamond P(m,k) \wedge (\exists j :: (\Delta_j = 1 \wedge src_j = m) \wedge \Box\neg(\Delta_j = 0 \wedge src_j = m)))$$

Hence at some moment in the past the property stated in (3) is violated. Hence our assumption in (1) is wrong.

$$\therefore (\Diamond P(m,k)) \Rightarrow \Box(\forall i :: rcv_i(E_m) \Rightarrow \Diamond(src_i = m \wedge \Delta_i = 0))$$

But from the algorithm actions, it is evident that for a node that receives E_m, $\Delta_i = 1 \wedge src_i = m$ is a precondition for $\Delta_i = 0 \wedge src_i = m$ to happen, i.e. by execution of action 6 or action 7.

$\therefore (\Diamond P(m,k)) \Rightarrow \Box(\forall i :: rcv_i(E_m) \Rightarrow \Diamond((src_i = m \wedge \Delta_i = 1)\, \mathcal{U}\, (src_i = m \wedge \Delta_i = 0)))$

Hence **Claim 1** is proved.

For the rest of the proof of **Lemma 1**, let us assume that $(\Diamond P(m,k))$ holds true. We know from our algorithm specification that if a node *i* is already in a spanning tree associated with an arbitrary computation-index *x*, i.e. $\Delta_i = 1 \wedge src_i = x$ and if it subsequently hears a message E_x from a node *j*, then action 4 is continuously enabled as long as $src_i = x$. Using **Claim 1** we can infer that,

$$(\forall i, j :: (rcv_{i,j}(E_m) \wedge src_i = m \wedge \Delta_i = 1) \Rightarrow \Diamond send_{i,j}(A_m)) \tag{4}$$

If the right side of statement (4) were not to hold, then deficit of node *j*, which has sent E_m to node *i* would never be zero thus violating **Claim 1**.

We use a property of Dijkstra-Scholten termination detection algorithm (Lemma 19.1(5) [2])that the parent pointers of all nodes participating in computation *m* do not form cycles in ancestor-descendant relationships. Since there are a finite number of nodes in the network, there exist nodes participating in computation *m* which are not "parent" of any other node. Let *S* be the set of all such nodes.

$$S = \{i | (\neg \exists j :: src_i = src_j = m \wedge p_j = i \wedge \Delta_i = 1)\}$$

From **Claim 1** and definition of S it follows that,

$$(\forall i, j : j \in S : rcv_{i,j}(E_m) \Rightarrow \Diamond(src_i = m \wedge \Delta_i = 1 \wedge p_i \neq j))$$

$$\therefore (\forall i, j : j \in S : rcv_{i,j}(E_m) \Rightarrow \Diamond(src_i = m \wedge \Delta_i = 1 \wedge rcv_{i,j}(E_m)))$$

Using (4) in the right side of the above implication, we get

$$(\forall i, j : j \in S : rcv_{i,j}(E_m) \Rightarrow \Diamond send_{i,j}(A_m))$$

But the statement above means that $\Delta_j = 1 \wedge src_j = m$ was true at some moment in the past and since *j* has not yet heard A_m from node *i*, $def_j = 0 \wedge src_j = m$ cannot hold true in the current state and in fact can never be true at least until it receives A_m from node *i*.

$$\therefore (\forall i, j : j \in S : rcv_{i,j}(E_m) \Rightarrow ((\blacklozenge(\Delta_j = 1 \wedge src_j = m) \wedge \neg(def_j = 0 \wedge src_j = m))\ \mathcal{W}\ rcv_{j,i}(A_m)))$$

$$\therefore (\forall i, j : j \in S : rcv_{i,j}(E_m) \Rightarrow ((\blacklozenge(\Delta_j = 1 \wedge src_j = m) \wedge \neg(\Delta_j = 0 \wedge src_j = m))\ \mathcal{W}\ rcv_{j,i}(A_m))) \tag{5}$$

Since we have assumed $\Diamond(P(m,k))$ holds true, from (3) it follows that:

$$(\forall i, j : j \in S : (\Delta_j = 1 \wedge src_j = m) \Rightarrow ((\Delta_j = 1 \wedge src_j = m)\ \mathcal{U}\ (\Delta_j = 0 \wedge src_j = m))))$$

Substituting the statement above in (5),

$$\therefore (\forall i, j : j \in S : rcv_{i,j}(E_m) \Rightarrow ((\blacklozenge(\Delta_j = 1 \wedge src_j = m) \wedge \neg(\Delta_j = 0 \wedge src_j = m))\ \mathcal{W}\ rcv_{j,i}(A_m))\ \mathcal{U}\ (\Delta_j = 0 \wedge src_j = m))$$

$$\therefore (\forall i, j : j \in S : rcv_{i,j}(E_m) \Rightarrow ((\blacklozenge(\Delta_j = 1 \wedge src_j = m) \wedge \neg(\Delta_j = 0 \wedge src_j = m))\ \mathcal{W}\ rcv_{j,i}(A_m))\ \mathcal{U}\ (def_j = 0 \wedge src_j = m))$$

$$\therefore (\forall i,j : j \in S : rcv_{i,j}(E_m) \Rightarrow ((\blacklozenge(\Delta_j = 1 \wedge src_j = m) \wedge \neg(\Delta_j = 0 \wedge src_j = m))\, \mathcal{W}\, rcv_{j,i}(A_m))\, \mathcal{U}\, (rcv_{j,i}(A_m)))$$

$$\therefore (\forall i,j : j \in S : rcv_{i,j}(E_m) \Rightarrow \Diamond rcv_{j,i}(A_m)) \tag{6}$$

$$\therefore (\forall j : j \in S : \Diamond(def_j = 0 \wedge src_j = m)) \tag{7}$$

Statement(7) and **Claim 1** imply that action 6 will be continuously enabled and by fairness constraint must eventually be executed. Otherwise, $\Delta_j = 0 \wedge src_j = m$ will never hold thus violating **Claim 1**. Thus the "shrinking" process of spanning tree will eventually begin.

$$(\forall j : j \in S : \Diamond(\Delta_j = 0 \wedge send_{j,p_j}(A_m(j)))) \tag{8}$$

Define $U_l = \{i | src_i = m \wedge \Delta_i = 1 \wedge (\exists j : j \in U_{l-1} : p_j = i) \wedge (\forall j, t :: (p_j = i \wedge j \in U_t) \Rightarrow t < l)\}$, and $U_0 = S$

Thus U_l represents the set of all nodes at level l of the spanning tree, where the leaves are at level 0 and the root at level h.

We shall now prove by induction that :

$$N_l \equiv (\forall j : j \in U_l : \Diamond send_{j,p_j}(A_m(max\{i | i \in D_{k,m}\}))), \forall l :: 0 \leq l \leq h-1$$

Proof : The proof is by induction on level l of a node in the spanning tree.

Base Case : $l = 0$.

Using Definition of U_l in (8), we get

$$(\forall j : j \in U_0 : \Diamond(\Delta_j = 0 \wedge send_{j,p_j}(A_m(j)))) \tag{9}$$

From the algorithm specification, the leaves of the spanning tree send their own identities in their *A* messages to their parent nodes since they have no downstream nodes in the tree. Thus, by definition of $D_{j,m}$, the following trivially holds true:

$$\therefore (\forall j : j \in U_0 : \Diamond(\Delta_j = 0 \wedge send_{j,p_j}(A_m(max\{i | i \in D_{j,m}\}))))$$

Inductive Hypothesis: Let us assume that N_l holds true for all l such that $0 \leq l \leq n$. We shall now prove that N_l holds true for $l = n+1$.

By definition of U_{n+1} it follows that:

$$(\forall j, i, t : j \in U_{n+1} : p_i = j \wedge i \in U_t \Rightarrow t < n+1)$$

By inductive hypothesis,

$$(\forall j, i : j \in U_{n+1} : p_i = j \Rightarrow \Diamond send_{i,j}(A_m(max\{x | x \in D_{i,m}\}))) \tag{10}$$

Restating (9),

$$(\Diamond P(m,k)) \Rightarrow (\forall i,j :: (rcv_{i,j}(E_m) \wedge src_i = m \wedge \Delta_i = 1) \Rightarrow \Diamond send_{i,j}(A_m))$$

Since we have assumed that $(\Diamond P(m,k))$ holds true,

$$(\forall i,j : j \in U_{n+1} : (rcv_{i,j}(E_m) \wedge src_i = m \wedge \Delta_i = 1) \Rightarrow p_i \neq j \wedge \Diamond send_{i,j}(A_m)) \tag{11}$$

From **Claim 1**, it follows that

$$(\forall i,j : j \in U_{n+1} : (rcv_{i,j}(E_m) \Rightarrow \Diamond(src_i = m \wedge \Delta_i = 1))) \tag{12}$$

On receiving E_m from a node *j*, either node *i* starts participating in computation *m* by adopting *j* as its parent and removing E_m from its receive buffer, or it is the case that node *i* has received E_m from some other node who it chooses as its parent.

$$(\forall i,j : j \in U_{n+1} : rcv_{i,j}(E_m) \Rightarrow \Diamond((src_i = m \wedge \Delta_i = 1 \wedge p_i = j \wedge \neg rcv_{i,j}(E_m)) \vee (src_i = m \wedge \Delta_i = 1 \wedge p_i \neq j \wedge rcv_{i,j}(E_m)))) \tag{13}$$

Substituting from (10), (11) and (12) in (13), we get

$$(\forall i,j : j \in U_{n+1} : rcv_{i,j}(E_m) \Rightarrow \Diamond send_{i,j}(A_m))$$

Reasoning as we did to prove (6), we get

$$(\forall i,j : j \in U_{n+1} : rcv_{i,j}(E_m) \Rightarrow \Diamond rcv_{i,j}(A_m))$$

$$\therefore (\forall j : j \in U_{n+1} : \Diamond(def_j = 0 \wedge src_j = m))$$

By **Claim 1**, action 6 will remain continuously enabled and will eventually be executed.

$$(\forall j : j \in U_{n+1} : \Diamond send_{j,p_j}(A_m(max\{i | i \in D_{j,m}\})))$$

Therefore, N_{n+1} holds true , which completes the induction.

Since the node *k* is the only node at level *h*, it is the parent of all nodes at level *h* - 1. Therefore we get:

$$(\forall j : j \in U_{h-1} : \Diamond send_{j,k}(A_m(max\{i | i \in D_{j,m}\})))$$

Therefore,$(\delta_k = 0 \wedge src_k = m \wedge k = m.id) \Rightarrow (lid_k = max\{i | i \in D_{k,m}\} \wedge (\forall j : j \in D_{k,m} : send_{k,j}(L_m(lid_k))))$

$$\therefore (\forall i :: (\delta_i = 0 \wedge src_i = m \wedge \blacklozenge(rcv_i(E_m))) \Rightarrow lid_i = lid_k)$$

The above implication holds because the antecedent in the above implication can hold true only by execution of action 8 of algorithm.

$$(\forall i :: (\delta_i = 0 \wedge src_i = m \wedge \blacklozenge(rcv_i(E_m))) \Rightarrow lid_i = max\{j | j \in D_{k,m}\})$$

Since nodes in $D_{k,m}$ are *k* and precisely the nodes that received E_m at some moment in the past, we get

$$P \Rightarrow (\forall i :: (\delta_i = 0 \wedge src_i = m \wedge \blacklozenge(rcv_i(E_m))) \Rightarrow lid_i = max\{j | k \cup \blacklozenge\,(rcv_j(E_m))\})$$

Since *k* and *m* are chosen arbitrarily, **Lemma 1** is proved.

Lemma 2:

$\Box(\forall i :: \delta_i = 0 \wedge d_{i,lid_i} \neq \infty) \Rightarrow \Diamond\Box(\forall i :: lid_i = max\{k | d_{i,k} \neq \infty\})$

Proof: Let $U \equiv \Box(\forall i :: \delta_i = 0 \wedge d_{i,lid_i} \neq \infty)$ and suppose that *U* holds true.

Since we have assumed that there are finite number of topological changes,

$$\text{let } V \equiv \Box \textit{ no more topological changes occur}$$

$$\text{Hence, } U \Rightarrow \Diamond\Box(U \wedge V)$$

Claim 2: $(U \wedge V) \Rightarrow (\forall i, j : j \in N_i : lid_i \neq lid_j \Rightarrow$
$(Nlf(i,j) \vee send_{i,j}(NL(lid_i)) \vee send_{j,i}(NL(lid_j)) \vee send_{i,j}(L(lid_i)) \vee send_{j,i}(L(lid_j)))$

Proof: The proof is by contradiction.

Let us assume that

$$(U \wedge V) \wedge (\exists i, j : j \in N_i : lid_i \neq$$
$$lid_j \wedge \neg Nlf(i,j) \wedge \neg send_{i,j}(NL(lid_i)) \wedge \neg send_{j,i}(NL(lid_j)) \wedge \neg send_{i,j}(L(lid_i)) \wedge \neg send_{j,i}(L(lid_j)))$$

holds true.

Let *s* and *t* be two neighboring nodes such that :

$$t \in N_s \wedge lid_s \neq lid_t \wedge \neg Nlf(s,t) \wedge \neg send_{s,t}(NL(lid_s)) \wedge \neg send_{t,s}(NL(lid_t)) \wedge \neg send_{s,t}(L(lid_s)) \wedge \neg send_{t,s}(L(lid_t)) \quad (14)$$

Let *m* and *n* be the last computations which node *s* and *t* participated in respectively. Since $U \wedge V$ holds true,

$$\blacklozenge((\delta_s = 1 \wedge src_s = m)\,\mathcal{U}\,(\delta_s = 0 \wedge src_s = m)) \wedge \Box\delta_s = 0 \quad (15)$$

$$\blacklozenge((\delta_t = 1 \wedge src_t = n)\,\mathcal{U}\,(\delta_t = 0 \wedge src_t = n)) \wedge \Box\delta_t = 0 \quad (16)$$

Let τ be the time when $Nlf(s,t)$ is set to false for the last time, i.e. nodes *s* and *t* could have been neighbors of each other on more than one occassion in the past and then moved out of range of each other and finally become neighbors again. It should be noted that nodes *s* and *t* will remain neighbors of each other forever since τ, since predicate V holds true.

Depending on when $Nlf(s,t)$ is set to false there are two possibilities,viz:

1. If $Nlf(s,t)$ was set to false before $(\delta_s = 1 \wedge src_s = m)$, then $send_{s,t}(E_m)$ must be true and eventually $rcv_{t,s}(E_m)$ must be true.

2. If $Nlf(s,t)$ was set to false after $(\delta_s = 1 \wedge src_s = m)$, then $send_{s,t}(NL)$ must be true, as the only way $Nlf(s,t)$ becomes false is by execution of action 9.

Depending on these possibilities, we deduce that

$$\neg Nlf(s,t) \Rightarrow \blacklozenge_\tau(send_{s,t}(E_m) \vee (\delta_s = 1 \wedge send_{s,t}(NL_m)) \vee (\delta_s = 0 \wedge send_{s,t}(NL_m)))$$

Since *s* and *t* remain neighbors forever, any message sent by node *s* to node *t* will eventually be received by node *t*. Hence,

$$\therefore \neg Nlf(s,t) \Rightarrow ((\blacklozenge_\tau((rcv_{t,s}(E_m) \vee (rcv_{t,s}(NL_m) \wedge NL_m.\delta = 1) \vee (rcv_{t,s}(NL_m) \wedge NL_m.\delta = 0))))$$

$$\vee(\Diamond(rcv_{t,s}(E_m) \vee (rcv_{t,s}(NL_m) \wedge NL_m.\delta = 1) \vee (rcv_{t,s}(NL_m) \wedge NL_m.\delta = 0)))) \quad (17)$$

We shall first prove **Claim 2** for the case when $m \neq n$ and subsequently for $m = n$.

1. $m \neq n$: If $\blacklozenge_\tau rcv_{t,s}(E_m)$ is true, then by **Claim 1** $(\Delta_t = 1 \wedge src_t = m)\, \mathcal{U}\, (\Delta_t = 0 \wedge src_t = m)$. Since *n* is the last computation that *t* participates in, $send_{t,s}(E_n)$ will be true and again by **Claim 1** $(\Delta_s = 1 \wedge src_s = n)\, \mathcal{U}\, (\Delta_s = 0 \wedge src_s = n)$ will hold true thus violating the assumptiont that *m* was the last computation node *s* participated in. On the other hand, if $\Diamond rcv_{t,s}(E_m)$ is true, then eventually $\delta_t = 1 \wedge src_t = m$ will be true and will violate the assumption that $\Box(\forall i :: \delta_i = 0)$.

   If $\blacklozenge_\tau(rcv_{t,s}(NL_m) \wedge NL_m.\delta = 1)$ is true, then action 2 will remain continuously enabled for node *t* and will start a new computation with a computation-index $\succ m$. The case when $\Diamond(rcv_{t,s}(NL_m) \wedge NL_m.\delta = 1)$ holds true leads to the same consequence. Node *s* will be forced to participate in this newly started computation thus violating the assumption that *m* is the last computation that node *s* participated in.

   Eliminating the conflicting cases from (17),

$$\neg Nlf(s,t) \Rightarrow (\blacklozenge_\tau(rcv_{t,s}(NL_m) \wedge NL_m.\delta = 0) \vee \Diamond(rcv_{t,m}(NL_m) \wedge NL_m.\delta = 0)) \quad (18)$$

   Also, if $(rcv_{t,s}(NL_m) \wedge NL_m.\delta = 0)$ is true and if $\delta_t = 1$ then action 2 will remain continuously enabled for node *t* until $\delta_t = 0$. If node *t* executes action 2, then it will start a computation with an identifier $x \succ m$ and thereby forcing node *s* to participate in it. But this will violate our assumption that *m* is node *s*'s last computation. Hence,

$$\neg Nlf(s,t) \Rightarrow (\blacklozenge_\tau(rcv_{t,s}(NL_m) \wedge NL_m.\delta = 0 \wedge \delta_t = 0) \vee \Diamond(rcv_{t,s}(NL_m) \wedge NL_m.\delta = 0 \wedge \delta_t = 0)) \quad (19)$$

   After topological changes stop, then for all neighboring node pairs *i,j*, for which $Nlf(i,j)$ predicate is true, action 9 is continuously enabled and by fairness constraint will eventually be executed, setting $Nlf(i,j)$ to false.

$$\therefore V \Rightarrow \Diamond\Box(\forall i,j : j \in N_i : \neg Nlf(i,j)) \quad (20)$$

   Applying Universal elimination in (20),

$$V \Rightarrow \Diamond\Box((\neg Nlf(s,t) \wedge (\neg Nlf(t,s)) \quad (21)$$

It is easy to see that a similar implication as in (19) will hold for $\neg Nlf(t,s)$, assuming that τ' is the time when $Nlf(t,s)$ is set to false for the final time. Assuming V to hold true and substituting result (19) in (21), we get

$$\Diamond\Box((\blacklozenge_{\tau}(rcv_{t,s}(NL_m) \wedge NL_m.\delta = 0 \wedge \delta_t = 0)) \wedge (\blacklozenge_{\tau'}(rcv_{s,t}(NL_n) \wedge NL_n.\delta = 0 \wedge \delta_s = 0))) \quad (22)$$

But if $send_{s,t}(NL_m)$ is true, it means that action 9 was executed. Since, *s* and *t* remain neighbors forever after time τ, $\Diamond rcv_{t,s}(NL_m)$ is also true. But we know from (22) that $\blacklozenge_{\tau'}\delta_s = 0$. Let $lid_s = l_1$. As a result of execution of action 9, $send_{s,t}(NL(l_1))$ is true. It should be noted that $send_{s,t}(NL)$ will never again become true, since we know that after time τ, nodes *s* and *t* remain neighbors forever and so $Nlf(s,t)$ will remain false forever. Let us assume that $send_{s,t}(NL(l_1)) \wedge \Box\neg send_{s,t}(L)$ holds true. Similarly, at some moment after τ', $send_{t,s}(NL(l_2))$ will be true, after which $send_{t,s}(NL)$ will never again become true. Let us assume that $send_{t,s}(NL(l_2)) \wedge \Box\neg send_{t,s}(L)$ holds true. (22) will then be,

$$\Diamond\Box((\blacklozenge_{\tau}(rcv_{t,s}(NL(l_1)) \wedge \neg\Box rcv_{t,s}(L))) \wedge \blacklozenge_{\tau'}(rcv_{s,t}(NL(l_2)) \wedge \Box\neg rcv_{s,t}(L))) \quad (23)$$

Let us assume without loss of generality that $l_1 > l_2$. From the algorithm specification, it is clear that node *t* will adopt either l_1 or some other higher identity node as its leader, say $lid_t = l_k$. Naturally, *t* has received $L(l_k)$ or $NL(l_k)$ from a node different from node *s*. But then if *t* adopts l_k as its leader, then it can do so only by executing actions 8 or 10. But then each action ensures that $send_{t,s}(L(l_k))$ is true thus violating (23). Thus, (23) implies that eventually both *s* and *t* will have l_1 as their leader forever.

$$\Diamond\Box(lid_s = lid_t = l_1)$$

But this contradicts assumption (14) which implies that $\Diamond\Box(lid_s \neq lid_t)$. Thus our assumption (23) is wrong. Hence, either *s* or *t* changed its leader at least once and we know every leader change occurs through execution of action 8 or action 10.

$$\therefore lid_s \neq lid_t \Rightarrow send_{s,t}(L(lid_s)) \vee send_{t,s}(L(lid_t))$$

But this again contradicts our assumption (14). Thus, our assumption (14) is wrong and hence **Claim 2** is proved for the case when $m \neq n$.

2. $m = n$: From assumptions (15) and (16), it follows that *m* is the last computation that both *s* and *t* participate in. As in case (i), $lid_s \neq lid_t$ is possible only because of execution of action 8 or action 10 by either node *s* or node *t*.

$$\text{But,} lid_s \neq lid_t \Rightarrow send_{s,t}(L(lid_s)) \vee send_{t,s}(L(lid_t))$$

But this contradicts assumption (14). Thus **Claim 2** is proved for the case when $m = n$.

From the algorithm actions, it is evident that

$$(\forall i, j : j \in N_i : U \wedge V \wedge Nlf(i,j) \Rightarrow \Diamond(\neg Nlf(i,j) \wedge send_{i,j}(NL_{src_i}(lid_i))))$$

Substituting in **Claim 2**, we get

$$(U \wedge V \Rightarrow (\forall i, j : j \in N_i : lid_i \neq lid_j \Rightarrow \Diamond(send_{i,j}(NL_{src_i}(lid_i)) \vee$$
$$send_{j,i}(NL_{src_j}(lid_j)) \vee send_{i,j}(L_{src_i}(lid_i)) \vee send_{j,i}(L_{src_j}(lid_j))))) \quad (24)$$

Upon receipt of a L or NL message, a node changes its leader only when it finds that its current leader has a smaller identity than the one in the received message and also the leader in the received message has a finite path cost to the node. Since there are finite number of topological changes, finite number of diffusing computations are ever initiated. Hence only a finite number of computations actually return a leader. Hence, each node receives a finite number of L and NL messages. Thus, each node can change its leader only a finite number of times.Thus applying (24) to all pairs of neighboring nodes,

$$(U \wedge V \Rightarrow (\forall i, j : j \in N_i : lid_i \neq lid_j \Rightarrow \Diamond\Box((lid_i = lid_j) \wedge d_{i,lid_i} \neq \infty \wedge d_{j,lid_j} \neq \infty)) \quad (25)$$

Restating **Lemma 1**,

$$((\forall k, m :: \delta_k = 0 \wedge src_k = m \wedge k = m.id) \Rightarrow$$
$$((\forall i :: \delta_i = 0 \wedge src_i = m \wedge \blacklozenge rcv_i(E_m)) \Rightarrow lid_i = lid_k = max\{p | k \cup \blacklozenge rcv_p(E_m)\})))$$

Each diffusing computation that got over elected the maximum of all nodes that participated in that computation. From the algorithm actions, it is clear that a node changes its leader, only when it finds a higher identity leader in its component. Hence **Lemma 1** and (25) together, imply that

$$(U \wedge V \Rightarrow \Diamond\Box(\forall i :: lid_i = max\{k | d_{i,k} \neq \infty\}))$$

This proves **Lemma 2**.

Lemma 3: Given finite number of topological changes, the algorithm guarantees that eventually all nodes stop participating in diffusing computations for ever and have a leader. **Lemma 3** states the *Progress property* of our algorithm.

$$I \rightsquigarrow \Diamond\Box(\forall i :: \delta_i = 0 \wedge d_{i,lid_i} \neq \infty)$$

Proof: Observe from algorithm actions that the only way a new diffusing computation is initiated is by execution of either action 1 or 2, i.e when I holds true. Upon execution of either of these actions by a node, they are not enabled again for that node until another different topological change occurs. Since there are only a finite number of topological changes and only a finite number of nodes in the network, only a finite number of failures, which result in a state satisfying I, occur. As a result, only a finite number of computations are ever initiated since the election process is first triggered. Hence the following must be true :

1. Eventually no new computation is initiated hereafter, i.e. $\Diamond\Box\neg I$.

2. Eventually there are no more topological changes hereafter. (assumption).

Therefore, eventually both (1) and (2) must hold true forever. Let this condition be denoted by predicate W. Thus, $I \rightsquigarrow W$.

We now claim the following :

Claim 3: $W \Rightarrow \Box(\forall i :: \delta_i = 1 \Rightarrow \Diamond(\exists m :: ((\delta_i = 1 \wedge src_i = m)\, \mathcal{U}\, (\delta_i = 0 \wedge src_i = m)) \wedge \Box(\delta_i = 0)))$

Proof: Let us assume for the rest of the proof that W holds true. The proof is by contradiction i.e let us assume that :

$$\neg(\Box(\forall i :: \delta_i = 1 \Rightarrow \Diamond(\exists m :: (\delta_i = 1 \wedge src_i = m)\, \mathcal{U}\, (\delta_i = 0 \wedge src_i = m) \wedge \Box(\delta_i = 0)))) \tag{26}$$

$$\therefore \Diamond(\exists i :: \delta_i = 1 \wedge \Box(\forall m :: (\neg(\delta_i = 0 \wedge src_i = m) \vee \Diamond(((\delta_i = 1 \wedge src_i = m)\, \mathcal{U}\, (\delta_i = 0 \wedge src_i = m)) \wedge \Diamond \delta_i = 1)))) \tag{27}$$

Now from algorithm specification it is clear that,

$$(\forall i, m :: ((\delta_i = 1 \wedge src_i = m)\, \mathcal{U}\, (\delta_i = 0 \wedge src_i = m)) \Rightarrow \blacklozenge(\exists k :: \delta_k = 0 \wedge src_k = m \wedge m.id = k \wedge d_{k,lid_k} \neq \infty)) \tag{28}$$

We use property (Lemma 19.1(6) [2]) of Dijkstra-Scholten Algorithm for a specific computation *m* such that $(\Delta_k = 0 \wedge src_k = m \wedge m.id = k)$ holds, then none of the channels have E_m or A_m messages.

Now restating **Claim 1**,

$$((\Diamond(\Delta_k = 0 \wedge src_k = m \wedge m.id = k)) \Rightarrow \Box(\forall i :: rcv_i(E_m) \Rightarrow \Diamond((\Delta_i = 1 \wedge src_i = m)\, \mathcal{U}\, (\Delta_i = 0 \wedge src_i = m))))$$

Thus if $\Diamond(\Delta_k = 0 \wedge src_k = m \wedge m.id = k)$ holds, then each node *i* which received E_m at some moment in the past will eventually have no E_m or A_m messages in its buffer, because it is guaranteed to reach a state where $(\Delta_i = 0 \wedge src_i = m)$, in accordance with **Claim 1**. When that state is reached, node *i* has sent out A_m for each of the E_m message it received, which means that all of the E_m messages will be deleted from the receive buffer by execution of either action 3 or action 4. Also, since node *i* sends out all its A_ms, it means that it has received all the A_ms it was supposed to hear and $S_i = \{\}$. Hence, all of A_m messages would have been removed from *i*'s buffer by execution of actions 5 or 6.

The above discussion leads to the following inference :

$$(\forall i, m :: ((\delta_i = 1 \wedge src_i = m)\, \mathcal{U}\, (\delta_i = 0 \wedge src_i = m)) \Rightarrow \Box\neg(\delta_i = 1 \wedge src_i = m))$$

Thus a node cannot participate in a completed computation more than once. At any given time, there are only a finite number of computations in progress and finite number of nodes in the system. Hence, if every node *i* always participates in a computation *m* that is guaranteed to terminate i.e. a computation *m* for which $\Delta_k = 0 \wedge src_k = m \wedge m.id = k$ is guaranteed to hold , then eventually all nodes will reach a state satisfying $\Box\delta_i = 0$, thus violating (27). Also, a node can "abort" a computation only when it finds a higher priority computation. Since there are finite number of them in progress, it can abort only a finite number of times. Eventually, it will never find a higher priority

computation than the one that it is currently participating in. Hence for (27) to hold, the followinf statement must hold true:

$$\Diamond(\exists i, m :: \Box(\delta_i = 1 \wedge src_i = m)) \tag{29}$$

We now claim the the following:

Claim 4:

$$(\forall i, j, m : j \in N_i : (\Box(\delta_i = 1 \wedge src_i = m) \wedge (((\delta_i = \delta_j = 1) \vee (\delta_i \neq \delta_j)) \wedge (src_i \neq src_j))) \Rightarrow \Diamond\Box(\delta_j = 1 \wedge src_j = m))$$

Proof: Let *i* and *j* be arbitrary nodes and let *m* be an arbitrary computation such that :

$X \equiv (j \in N_i \wedge (\Box(\delta_i = 1 \wedge src_i = m) \wedge (((\delta_i = \delta_j = 1) \vee (\delta_i \neq \delta_j)) \wedge (src_i \neq src_j))))$ holds true.

Let us assume that $Nlf(i, j)$ is set to false. If $Nlf(i, j)$ is true, then action 9 will remain continuously enabled and eventually executed setting $Nlf(i, j)$ to false.

Depending on when $Nlf(i, j)$ is set to false, there are two possibilities :

- If $Nlf(i, j)$ is set to false before $\delta_i = 1 \wedge src_i = m$, then $send_{i,j}(E_m)$ must be true and hence eventually $rcv_{j,i}(E_m)$ will be true.
- If $Nlf(i, j)$ is set to false after $\delta_i = 1 \wedge src_i = m$, then it means that action 9 was executed after node *i* started participating in computation *m* and hence $send_{i,j}(NL_m)$ must hold true and hence eventually $rcv_{j,i}(NL_m)$ will be true.

$$\therefore X \Rightarrow \blacklozenge(rcv_{j,i}(E_m) \vee rcv_{j,i}(NL_m)) \vee \Diamond(rcv_{j,i}(E_m) \vee rcv_{j,i}(NL_m)) \tag{30}$$

$$\text{Now,} \blacklozenge(rcv_{j,i}(E_m) \vee rcv_{j,i}(NL_m)) \Rightarrow (\blacklozenge(\delta_j = 0 \wedge src_j = m) \vee (\blacksquare(\neg(\delta_j = 0 \wedge src_j = m)) \wedge rcv_{j,i}(E_m)) \vee (\blacksquare(\neg(\delta_j = 0 \wedge src_j = m)) \wedge \neg rcv_{j,i}(E_m)))$$

It should be noted that the right hand side of the above implication is simply a tautology. Let us consider the case when $\blacklozenge(\delta_j = 0 \wedge src_j = m)$ holds true.

$$\text{But, } \blacklozenge(\delta_j = 0 \wedge src_j = m) \Rightarrow \blacklozenge(\exists h :: ((\delta_h = 1 \wedge src_h = m)\, \mathcal{U}\, (\delta_h = 0 \wedge src_h = m)))$$

From (28), we get

$$\therefore \blacklozenge(\delta_j = 0 \wedge src_j = m) \Rightarrow \blacklozenge(\exists k :: \delta_k = 0 \wedge src_k = m \wedge m.id = k \wedge d_{k,lid_k} \neq \infty)$$

From **Claim 1** we can infer that $\blacklozenge(\Delta_i = 0 \wedge src_i = m)$ must be true. But since $\blacklozenge(\delta_j = 0 \wedge src_j = m))$ is true, $\blacklozenge(send_{j,i}(L_m) \vee send_{j,i}(NL_m))$ must be true. In either case, the message will be eventually received because there are no more changes and we use reliable communications. This implies that action 8 or action 10 of our algorithm will remain continuously enabled and so either $\blacklozenge(\delta_i = 0 \wedge src_i = m)$ is true or $\Diamond(\delta_i = 0 \wedge src_i = m)$ will hold true, both of which violate (29).

Let us consider the case when $(\blacksquare(\neg(\delta_j = 0 \wedge src_j = m)) \wedge \neg rcv_{j,i}(E_m))$ holds true.

$$((\blacksquare(\neg(\delta_j = 0 \wedge src_j = m)) \wedge \neg rcv_{j,i}(E_m)) \Rightarrow (\blacksquare(src_j \succ m \vee \blacklozenge(src_j = m \wedge \Diamond(src_j \succ m \wedge \delta_j = 1))))))$$

$$((\blacksquare(\neg(\delta_j = 0 \wedge src_j = m)) \wedge \neg rcv_{j,i}(E_m)) \Rightarrow (\blacklozenge(send_{j,i}(NL_{x \succ m}) \vee send_{j,i}(E_{x \succ m}))))$$

But, when $NL_{x \succ m}$ or $E_{x \succ m}$ is received by node *i*, action 2 or 3 in node *i* will remain enabled continuously and eventually executed, violating (29).

Thus eliminating conflicting cases from (30),

$$\therefore X \Rightarrow ((\blacksquare(\neg(\delta_j = 0 \wedge src_j = m)) \wedge rcv_{j,i}(E_m)) \vee \Diamond(rcv_{j,i}(E_m) \vee rcv_{j,i}(NL_m)))$$

If $\Diamond\ rcv_{j,i}(NL_m)$ holds true, then action 2 for node *j* will remain continuously enabled ($\because\ src_j \neq m$) and hence will get eventually executed, unless src_j becomes equal to *m*. If action 2 gets executed, then *j* will start a computation with $src_j \succ m$ and hence eventually $src_i \succ m$ thus violating (26). The only way src_j becomes equal to *m* is if $rcv_{j,i}(E_m)$ eventually holds true.

$$\therefore X \Rightarrow ((\blacksquare(\neg(\delta_j = 0 \wedge src_j = m)) \wedge rcv_{j,i}(E_m)) \vee \Diamond rcv_{j,i}(E_m))$$

Now $\delta_j = 1 \Rightarrow src_j \not\succ m$. If $src_j = n \wedge n \succ m$, then the following will hold

$$\blacklozenge(rcv_{i,j}(E_n) \vee rcv_{i,j}(NL_n)) \vee \Diamond(rcv_{i,j}(E_n) \vee rcv_{i,j}(NL_n))$$

But this will violate (26). In fact for the same reason, $\Box\ (\delta_j = 1 \Rightarrow src_j \not\succ m)$.

$$\therefore X \Rightarrow \Diamond\Box(\delta_j = 1 \wedge src_j = m)$$

Since *i*,*j* and *m* were chosen arbitrary **Claim 4** is proved.

Applying **Claim 4** inductively, we get

$$\Diamond\Box(\forall i :: src_i = m \wedge \delta_i = 1) \tag{31}$$

The parent pointers, as we know from property (Lemma 19.1(5) [2]), form a directed, acyclic graph rooted at source *m*. Statement (31) describes a quiescence condition after which no *Election* messages are sent or received. Hence, the tree formed by parent pointers stops growing, stabilizing to a fixed tree *T* and the quiescence condition also implies that the tree does not shrink any further. But, since topological changes have stopped, every *Ack* message sent by execution of action 3 of the algorithm will eventually be received by the intended destination. Hence, the leaf nodes of the tree will eventually hear all pending *Ack* messages allowing their Δ to become zero. This means that each leaf node will eventually send out its pending *Ack* message to its parent causing the tree to shrink further, a contradiction to (31). In fact, the tree will continue to shrink until the root node hears all pending *Ack* messages, which causes its variable, Δ, to be set to 0.

$$\therefore (\forall i :: (src_i = m \wedge \delta_i = 1) \Rightarrow \Diamond\Box(src_i = m \wedge \delta_i = 0))$$

But this violates (31). Hence, our assumption in (26) is wrong. This proves **Claim 3**. From the algorithm specification, it is obvious that

$$(\forall i :: ((\delta_i = 1 \wedge src_i = m)\ \mathcal{U}\ (\delta_i = 0 \wedge src_i = m)) \Rightarrow (\delta_i = 0 \wedge src_i = m \wedge d_{i,lid_i} \neq \infty))$$

Now it is very easy to see that **Claim 3** implies **Lemma 3**.

Lemma 4 : Eventually the program terminates, i.e. all program actions are disabled.

Proof: It is esay to see that **Lemma 2** and **Lemma 3** ensure that all program actions will be eventually disabled. **Lemma 2** ensures that actions 2-7 will never be enabled. **Lemma 2** and **Lemma 3** together ensure that actions 8 and 10 will never be enabled. The assumption of finite topological changes ensures that eventually action 9 will never be enabled.

Theorem 1: $I \rightsquigarrow G$, i.e. I stabilises to G.

Proof: It is easy to see that **Lemma 2** and **Lemma 3** imply Theorem 1.