Design and Analysis of a Leader Election Algorithm for Mobile Ad Hoc Networks

Sudarshan Vasudevan, Jim Kurose, Don Towsley

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

UMass Computer Science Techincal Report 03-20

Abstract

Leader election is a very important problem, not only in wired networks, but in mobile, ad hoc networks as well. Existing solutions to leader election do not handle frequent topology changes and dynamic nature of mobile networks. In this paper, we present a leader election algorithm that is highly adaptive to 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, using linear-time temporal logic, that the algorithm is "weakly" self-stabilizing and terminating. We also simulate the algorithm in a mobile ad hoc setting. Through our simulation study, we elaborate on several important issues that can significantly impact performance of such a protocol for mobile ad hoc networks such as choice of signaling, broadcast nature of wireless medium etc. Our simulation study shows that our algorithm is quite effective in that each node has a leader approximately 97-99% of the time in a variety of operating conditions.

1 Introduction

Leader election is a fundamental control problem in both wired and wireless systems. For example, in group communication protocols, the election of a new coordinator is required when a group coordinator crashes or departs the system. In the context of wireless networks, leader election has a variety of applications such as key distribution [3], routing coordination [17], sensor coordination [21], and general control [6, 5]. When nodes are mobile, topologies can change and nodes may dynamically join/leave a network. In such networks, leader election can occur frequently, making it a particularly critical component of system operation.

The classical statement of the leader election problem [1] is to *eventually elect a unique leader* from a fixed set of nodes. Indeed, several algorithms have been proposed to solve this problem. However, in the context of mobile, ad hoc networks this statement must be specialized in two important ways :

- The election algorithm must tolerate arbitrary, concurrent topological changes and should **eventually** terminate electing a unique leader.

- The elected leader should be the **most-valued-node** from among all the nodes within that connected component, where the **value** of a node is a performance-related characteristic such as remaining battery life, minimum average distance to other nodes or computation capabilities.

The first 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. It is important to realize that it is impossible to guarantee a unique leader at all times. For example, when a network partition occurs or when two components merge, it will take some time for a new leader to be elected. Thus, the modified problem definition requires that **eventually** every connected component has a unique leader. Our second 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 minimum average distance to other nodes, as the leader. Leader election based on such an ordering among nodes fits well in the class of leader election algorithms that are known as "extrema-finding" leader-election algorithms. The second modification to the statement of leader election problem, therefore, requires the elected leader to be the **most-valued-node** from the set of nodes in its connected component. Given the modifications described above, the requirements for leader election algorithm become: *Given a network of mobile nodes each with a value, after a finite number of topological changes, every connected component will eventually select a unique leader, which is the most-valued-node from among the nodes in that component.*

Existing solutions to the problem of leader election do not work in the highly dynamic environment found in mobile networks. Existing solutions to the leader election problem assume a static topology (e.g. [11, 12, 20, 21, 16, 22, 7, 8]), or assume that topological changes stop before an election starts (e.g. [28, 13]) or assume an unrealistic communication model such as existence of reliable broadcast and an message-order preserving network [32]. While there are some proposals for leader election in mobile networks [5], these algorithms are designed to perform random node election and cannot be modified to perform extrema-finding. We therefore propose an election algorithm to perform extrema-finding in a highly dynamic and asynchronous environment such as found in a mobile, ad hoc network. Unlike these previous works, this paper also presents a detailed simulation study of the proposed election algorithm in a mobile environment.

Our proposed algorithm uses the concept of diffusing computations [14] to perform leader election. Informally, the algorithm operates as follows. When an election is triggered at a node, the node starts a diffusing computation to determine its new leader. Several nodes can start diffusing computations in response to the departure of a leader and hence several diffusing computations can be in progress concurrently; however, a node participates in only one diffusing computation at a time. Eventually, when a diffusing computation terminates, the node initiating the computation informs other nodes of the identity of the elected leader. An election can be triggered at a node for a number of reasons such as disconnection from its leader or the value of the leader falling below some application-defined threshold. We emphasize that the operation of our election algorithm is generic and does not depend on how elections are triggered.

The contributions of this paper are the following:

- We present an extrema-finding leader election algorithm that operates asynchronously and accommodates arbitrary topological changes induced by node mobility. We prove using temporal logic that this algorithm achieves a “weak” form of stabilization, i.e., given that each process starts in a designated initial state, that after a finite number of topological changes the algorithm converges to a desired stable state in finite amount of time.

- We develop an improved understanding of how to design and implement a distributed algorithm, such as extrema-finding leader election, that accounts for the broadcast nature of wireless channels and the mobility found in an ad hoc network. In the context of leader election, we observe that the choice of signaling used in the protocol, accounting for the broadcast nature of wireless medium, and making subtle design changes in the leader election algorithm can greatly affect the performance of our algorithm. In our context, this results in an algorithm that ensures that a node has a leader over 97% of the time in a wide variety of operating conditions.
- We present a thorough study of the performance of the algorithm as a function of mobility, transmission range and node density.

The remainder of the paper is organized as follows. In Section 2, we discuss related work. Section 3 describes our model assumptions and objectives. In Section 4, we first provide an overview of our algorithm and then describe the actual algorithm in detail. In Section 6, we describe the simulation setting and performance metrics for evaluating our algorithm. Section 7 describes the lessons learned during algorithm design. In Section 5, we formally specify the various correctness properties of our algorithm. In Section 8, we discuss how our algorithm is affected by different parameters such as mobility and transmission range. Finally, we conclude in Section 9.

2 Related Work

Although leader-election is a fairly old problem, it has received surprisingly little attention in the context of mobile, ad hoc networks.

Leader election algorithms for static networks have been proposed in [11, 12]. These algorithms work by constructing several spanning trees with a prospective leader at the root of the spanning tree and recursively reducing the number of spanning trees to one. However, these algorithms work only if the topology remains static and hence cannot be used in a mobile setting. There have been several clustering and hierarchy-construction schemes that can be adapted to perform leader election [20, 21, 16, 22, 7, 8]. However, these algorithms either assume static networks or a synchronous system and therefore cannot be used in an asynchronous, mobile system.

Several leader election algorithms [32, 28, 13] have been proposed for wired networks that assume process crashes and link failures and are therefore closely related to our work. However, in [28, 13] process failures are assumed to occur before election starts while in [32] the election algorithms make strong assumptions such as requiring that every message be reliably broadcast to all other nodes and that the network be order-preserving i.e., a message *m* sent by a node *i* at time *t* is received by all nodes before another message *m*’ sent by node *j* at some instant $t' > t$. Such assumptions are very strong and make these solutions impractical in mobile environments.

There has been some work on spanning tree construction in the domain of self-stabilizing systems [23] that is related to our work. Informally, a *self-stabilizing system* is one that can recover from any arbitrary global state and reach a desired stable global state within finite time. Self-stabilizing spanning tree algorithms have been proposed in [24, 26, 27]. However, these algorithms assume a shared-memory model and are not suitable for a message-passing system such as an ad hoc network. In particular, in a message-passing system these algorithms cannot be *terminating* [25], i.e., they do not reach a state in which all program actions are disabled; instead nodes have to exchange infinitely many messages to detect that a stable state has been reached. We will later see that our election-algorithm achieves a "weaker" form of stabilization, but that it terminates once the stable state has been reached.

Leader election algorithms for mobile ad hoc networks have been proposed in [5, 6]. As noted earlier, we are interested in an extrema-finding algorithm, because for the applications discussed in Section 1, it is desirable to elect a leader with some system-related attributes such as maximum battery life or maximum computation power. The algorithms in [5] are not extrema-finding and cannot be extended to perform extrema-finding. In the algorithms in [5], a node that detects a partition in the network gets elected as the leader and a partition can be detected by any "random" node. Also, no proof of correctness of their algorithms has been provided for the case of concurrent topological changes. Although, extrema-finding leader election algorithms for mobile ad hoc networks have been proposed in [6], these algorithms are unrealistic as they require nodes to meet and exchange information in order to elect a leader and are not well-suited to the applications discussed earlier. Several clustering algorithms have been proposed for mobile networks(e.g. [9, 10]), but these algorithms elect clusterheads only within their single hop neighborhood.

Designing a leader election algorithm that can tolerate arbitrary, concurrent node and link crashes, network partitioning/merging and which executes in an asynchronous fashion is a difficult and challenging task. This is the focus of this paper. Using linear-time temporal logic, we prove that our algorithm stabilizes to a desired state despite arbitrary topological changes caused by node mobility. An important distinction of our work from previous work on leader election is that, we have performed a detailed simulation study of our leader election algorithm to understand its performance in a mobile, ad hoc setting. Our simulations show that the algorithm works very well, with each node having a leader for 97%-99% of time. Furthermore, we present several interesting insights culled from our experiences in simulating these algorithms in mobile environments. In particular, we observe that subtle and seemingly small changes in the election algorithm and choice of signaling can have significant performance consequences. These insights can be very useful in the design of other protocols for mobile, ad hoc networks.

3 Objectives, Constraints and Assumptions

In developing a leader election algorithm, we first define our system model, assumptions, and goal. We model an ad hoc network as an undirected graph that changes 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 the 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. Node IDs are used to break ties among nodes which have the same value.

2. **Links:** Links are bidirectional and FIFO, i.e. messages are delivered in order over a link between two neighbors.

3. **Node Behavior:** Node mobility may result in arbitrary topology changes including network partitioning and merging. Furthermore, nodes can crash arbitrarily at any time and can come back up again at any time.

4. **Node-to-Node Communications:** A message sent by a node is eventually received by the intended receiver, provided that the two nodes remain connected forever starting from the instant the message is sent.

5. **Buffer Size:** Each node has a sufficiently large receive buffer to avoid buffer overflow at any point in its lifetime.

The objective of our leader election algorithm is to ensure that after a finite number of topology changes, *eventually* each node *i* has a leader which is the most-valued-node from among 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 [14]. In this section, we describe a leader election algorithm based on diffusing computations. In later sections, we will discuss in detail how this algorithm can be adapted to a mobile setting.

4.1 Leader Election in a Static Network

We first describe our election algorithm in the context of a static network, under the assumption that nodes and links never fail. The algorithm operates by first "growing" and then "shrinking" a spanning tree rooted at the node that initiates the election algorithm. We refer to this computation-initiating node as the *source node.* As we will see, after the spanning tree shrinks completely, the source node will have adequate information to determine the most-valued-node and will then broadcast its identity to the rest of the nodes in the network.

The algorithm uses three messages, viz. *Election*, *Ack* and *Leader*.

Election. *Election* messages are used to "grow" the spanning tree. When election is triggered at a source node *s* (for instance, upon departure of its current leader), the node begins a *diffusing computation* by sending an *Election* message to all of its immediate neighbors. Each node, *i*, other than the source, designates the neighbor from which it first receives an *Election* message as its *parent* in the spanning tree. Node *i* then propagates the received *Election* message to all of its neighboring nodes (children) except its 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 does not, however, immediately return an *Ack* message to its parent. Instead, it waits until it has received *Ack*s from all of its children, before sending an *Ack* to its parent. As we will see shortly, the *Ack* message sent by i to its parent contains leader-election information based on the *Ack* messages i has received from its children.

Once the spanning tree has completely grown, the spanning tree "shrinks" back toward the source. Specifically, once all of i's outgoing *Election* messages have been acknowledged, *i* sends its pending *Ack* message to its parent node. Tree "shrinkage" begins at the leaves of the spanning tree, which are parents to no other node. Eventually, each leaf receives *Ack* messages for all *Election* messages it has sent. These leaves thus eventually send their pending *Ack* messages to their respective parents, who in turn send their pending *Ack* messages to their own parents, and so on, until the source node receives all of its pending *Ack* messages. In its pending *Ack* message, a node announces to its parent the identifier and the value of the most-valued-node among all its downstream nodes. Hence the source node eventually has sufficient information to determine the most-valued-node 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 identifier of the most-valued-node.

Example:

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 figure, and for the rest of the paper, thin arrows indicate the direction of flow of messages and thick arrows indicate parent pointers. These parent pointers together represent the constructed spanning tree. The number adjacent to each node in Figure 1(a) represents its value. As shown in Figure 1, node *A* is a source node that starts a diffusing computation by sending out *Election* messages (denoted as "E" in the figure) to its immediate neighbors, viz. nodes *B* and *C*, shown in Figure 1(a). As indicated in Figure 1(b), nodes *B* and *C* 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 *C* send *Election* messages to one another. These *Election* messages are immediately acknowledged since nodes *B* and *C* have already received *Election* messages from their respective parents. Note that 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 *D* and *F* send their pending *Ack* messages (denoted by "A") to their respective parent nodes in the spanning tree. Each of these *Ack* messages contains the identity of the most-valued-node (and its actual value) downstream to nodes *D* and *F*, 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 *C* in Figure 1(e) and then broadcasts the identity of the leader, *D*, via the *Leader* message (denoted by "L" in the figure) shown in Figure 1(f).

4.2 Execution Model:

The *LeaderElection* algorithm specified in Figure 2 runs on each node and is of the form:

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

The algorithm has a set of variables, an initialization section and a set of actions. Each variable in the variable

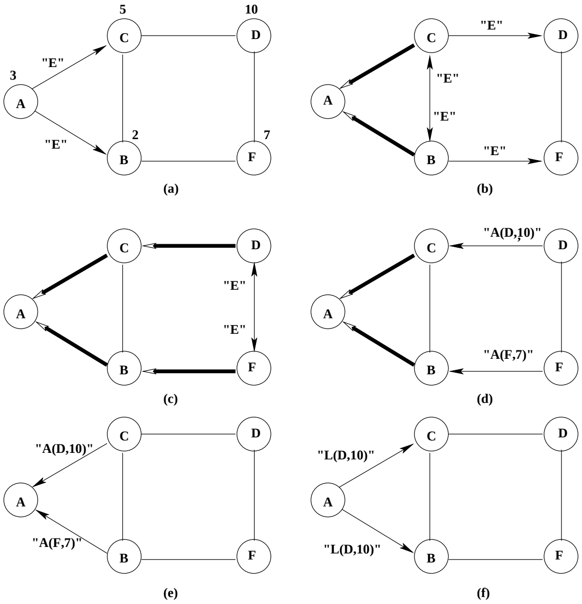


Figure 1: An execution of leader election algorithm based on Dijkstra-Scholten termination detection algorithm. Thin arrows indicate direction of flow of messages while the thick arrows represent the constructed spanning tree.

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*. Also, all action executions are *atomic* operations.

```
module          $LeaderElection_i(i : 1 \dots n)$

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

initialization  $\delta_i, \Delta_i, lid_i, N_i := 0, 0, 0, \{\}$

begin

/*Start a new computation*/
1. $\{\delta_i = 0 \land d_{i,lid_i} = \infty\}$                         $\longrightarrow$  $src_i, , Num_i =< i, Num_i >, Num_i + 1;$
                                                                     $E.src, E.lid := i, lid_i;$
                                                                     $(\forall j \in N_i) send_{i,j}(E);$
                                                                     $S_i, \Delta_i, \delta_i, p_i, max_i := N_i, 1, 1, i, i;$

/*Join the computation*/
2. $\{rcv_{i,j}(E) \land (\delta_i = 0 \lor (\delta_i = 1 \land E.src > src_i))$   $\longrightarrow$  $src_i := E.src;$
   $\land (E.lid = lid_i)\}$                                              $(\forall k \in N_i \backslash \{j\}) send_{i,k}(E);$
                                                                     $S_i, \Delta_i, \delta_i, p_i, max_i := N_i \backslash \{j\}, 1, 1, j, i;$
                                                                     $Deque(E);$

/* Already in computation; or I still have my leader */
3. $\{rcv_{i,j}(E) \land \Delta_i = 1 \land src_i = E.src\}$               $\longrightarrow$  $A.src, A.flag := src_i, 0;$
   $\lor \{rcv_{i,j}(E) \land E.lid \neq lid_i\}$                            $send_{i,j}(A);$
                                                                     $Deque(E);$

/* Update list of nodes to be heard from*/
4. $\{rcv_{i,j}(A) \land \Delta_i = 1 \land src_i = A.src\}$               $\longrightarrow$  $S_i := S_i \backslash \{j\};$
   $\lor \{j \in S_i \land d_{i,j} = \infty\}$                               if $(rcv_{i,j}(A)$ and $A.flag = 1$ and $A.id > max_i)$
   $\lor \{j \in S_i \land rcv_{i,j}(R) \land (R.src \neq src_i \lor R.\Delta = 0 \lor R.flag = 0)\}$   then $max_i := A.id;$
                                                                     if $rcv_{i,j}(m)$ then $Deque(m);$

/*Report pending Ack to parent*/
5. $\{S_i = \{\} \land src_i.id \neq i \land \Delta_i = 1\}$              $\longrightarrow$  $\Delta_i := 0;$
                                                                     $A.src, A.flag, A.id := src_i, 1, lid_i;$
                                                                     $send_{i,p_i}(A);$
                                                                     if $rcv_{i,j}(E)$ then $Deque(E);$

/*Terminate computation, announce leader*/
6. $\{S_i = \{\} \land src_i.id = i \land \Delta_i = 1\}$                 $\longrightarrow$  $\Delta_i, \delta_i, lid_i := 0, 0, max_i;$
   $\lor \{\Delta_i = 0 \land \delta_i = 1 \land d_{i,p_i} = \infty\}$              $L.src, L.lid := src_i, lid_i;$
   $\lor \{rcv_{i,j}(L) \land L.lid < lid_i \land \delta_i = 0\}$                $(\forall k \in N_i) send_{i,k}(L);$
   $\lor \{rcv_{i,p_i}(R) \land \Delta_i = 0 \land (R.src \neq src_i \lor R.\delta = 0)\}$   if $rcv_{i,j}(m)$ then $Deque(m);$

/*Adopt a new leader*/
7. $\{\Delta_i = 0 \land \delta_i = 1 \land rcv_{i,j}(L) \land max_i < L.lid\}$   $\longrightarrow$  $lid_i, \delta_i, src_i := L.lid, 0, L.src;$
   $\lor \{\delta_i = 0 \land rcv_{i,j}(L) \land lid_i < L.lid\}$                 $(\forall k \in N_i) send_{i,k}(L);$
                                                                     $Deque(L);$

/*Announce my leader to a new neighbor*/
8. $\{Nlf(i,j) \land j \in N_i \land \delta_i = 0 \land d_{i,lid_i} < \infty\}$   $\longrightarrow$  $L.src, L.lid := src_i, lid_i;$
                                                                     $send_{i,j}(L);$
                                                                     $Nlf(i,j) := false;$

/*Send reply in response to received Probe message*/
9.  $\{rcv_{i,j}(P)\}$                                                $\longrightarrow$  $R.src, R.\delta, R.lid := src_i, \delta_i, lid_i;$
                                                                     $send_{i,j}(R);$
                                                                     $Deque(P);$

/* Deque message if no other action is enabled */
10.  $\{\forall m : rcv_{i,j}(m) \land \neg(guard1 \lor \dots \lor guard9)\}$    $\longrightarrow$  $Deque(m);$

end
```

Figure 2: Leader Election Algorithm Specification

Message	Purpose
Election	for growing a spanning tree
Ack	to acknowledge receipt of an *Election* msg
Leader	to announce the new leader
Probe	to determine if a node is still connected
Reply	sent in response to a *Probe* msg

Table 1: Message Types used in the Election Algorithm.

Variables	Meaning
δ_i	a binary variable indicating if *i* is currently in an election or not
p_i	*i*'s parent node in the spanning tree
Δ_i	a binary variable indicating if *i* has sent an *Ack* to p_i or not
lid_i	*i*'s leader
N_i	*i*'s current neighbors
S_i	set of nodes from which *i* is yet to hear an *Ack* from
src_i	*i*'s computation-index

Table 2: List of Variables Maintained by a node *i* during the Election Process.

4.3 Leader Election in a Mobile, Ad Hoc Network

We now describe the operation of our leader election algorithm in the context of a mobile, ad hoc network. In the previous section, we provided an overview of the algorithm's 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 is inadequate. Furthermore, we assumed in the previous section that only one node triggers an election. In reality, many nodes may concurrently trigger leader elections, with each of them independently starting a diffusing computation, due to lack of knowledge of other computations started by other nodes.

We note that throughout the discussion of our algorithm's operation and for the rest of the paper, we assume that the value of the node is same as its identifier. We emphasize that this assumption has been made only for simplicity of presentation and results in no loss of generality.

Before we describe how our algorithm accommodates node mobility, we describe the variables and messages used by the algorithm.

4.3.1 Variables and Message types

The message types and variables used in the algorithm are shown in Table 1 and Table 2 respectively. The algorithm involves five message types: *Election*, *Ack*, *Leader*, *Probe* and *Reply*. The first three message types were described in Section 4.1. We will discuss the use of *Probe* and *Reply* messages while describing our algorithm's operation.

Each node *i* maintains a boolean variable δ_i, whose value is 0 if node *i* has a leader, and 1 if it is in the process of electing one. The variable src_i contains the *computation-index* of the diffusing computation in which node *i* is currently participating. As we will see in Section 4.3.3, this *computation-index* uniquely identifies a computation and is required to handle multiple, concurrent computations. During a diffusing computation, node *i* keeps track of its parent, p_i. Variable Δ_i is set to 0 if node *i* has sent its pending *Ack* message to its parent and 1 if it has not (i.e., it is still in the spanning tree). Each node *i* maintains its current leader in lid_i. N_i is the list of *i*'s current neighbors (maintained by periodic exchange of messages between neighbors) and, S_i represents the set of nodes that *i* has yet to hear an *Ack* message from. It is updated each time *i* receives an *Ack* message.

4.3.2 Bootstrapping the Election Process

Each node starts execution by initializing the different variables of the leader election algorithm. After the initialization, the algorithm in each 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. Formal specification of the algorithm and the execution model are presented in Figure 2 and Section 4.2 respectively.

4.3.3 Handling Multiple, Concurrent Computations

The leader of a connected component periodically sends heartbeat messages to other nodes. The absence of a heartbeat message from its leader for a predefined timeout period triggers a fresh leader election process at a node. It should be noted that more than one node can concurrently detect leader departure and each node can initiate diffusing computations independently, leading to concurrent diffusing computations. We handle multiple, concurrent diffusing computations by requiring that each node participate in only one diffusing computation at a time. In order to achieve this, each diffusing computation is identified by a *computation-index*. This computation-index is a pair, viz. $\langle num, id \rangle$, where id represents the identifier of the node that initiated that computation and num is an integer, which is described below.

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 priority than another diffusing computation *B* iff *computation-index*$_A$ $\succ$ *computation-index*$_B$

A given source always starts a diffusing computation with num greater than that of any other computation it previously initiated, while the *source-id* field is used to break ties among concurrent diffusing computations with different sources but the same num value. As a result, there is a total ordering on computation-indices. The variable num is incremented each time a node starts a fresh diffusing computation. When a node participating in a diffusing computation "hears" another computation with a higher computation-index, the node stops participating in its current computation in favor of the higher computation-index. For instance, in Figure 3(a), node *G* sends an *Election* message with computation-index, $\langle 3, D \rangle$, to node *A* whose current computation-index is $\langle 3, B \rangle$. Upon receiving this *Election* message, node *A* stops participating in its current computation, sets its computation-index to $\langle 3, D \rangle$, as shown in Figure 3(b), and propagates the received *Election* message to nodes *B* and *C*.

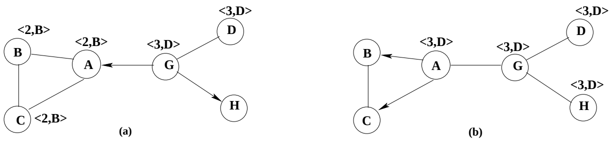


Figure 3: Handling concurrent diffusing computations

4.4 Algorithm Performed by the Nodes

The main idea of our algorithm is to "grow" and "shrink" a spanning tree during the election process and announce the leader after the tree shrinks completely. However, if node movement results in changes to this spanning tree, then nodes detect these changes and take appropriate actions. In this section, we describe through examples, how our election algorithm accommodates arbitrary changes in topology induced by node mobility.

Initiate Election: Node *i* begins the election process in response to the departure of its current leader. As described in Section 4.1, node *i* starts the process of "growing" a spanning tree by propagating *Election* messages to its neighbors, informing them of the start of an election of a new leader. In triggering a fresh election, node *i* sets its variable δ_i to 1 to indicate that it is currently involved in an election. As described in Section 4.1, *i* announces a leader only after it hears *Ack* messages from all the nodes to which it sends an *Election* message. The list S_i is, therefore, initialized to N_i, *i*'s current neighbors.

Spanning Tree Construction: Node *j*, upon receiving an *Election* message from node *i*, say *E*, joins the spanning tree by setting its parent pointer, $p_j = i$, and in turn propagates *Election* messages to its own neighbors in the set N_j. As described in Section 4.1, these *Election* messages are propagated forward to all nodes and eventually a spanning tree of nodes is constructed.

Handling Node Partitions: Once node *i* joins an election, it must receive *Ack* messages from all nodes in list S_i before it can report an *Ack* message to its parent node. However, because of node mobility, it may happen that node *j*, which has yet to report an *Ack* message, gets disconnected from node *i*. Node *i* must detect this event, since otherwise it will never report an *Ack* message to its parent and, therefore, no leader will be announced.

Consider a scenario in which a parent-child pair becomes disconnected during the election process, i.e. the condition $d_{i,j} = \infty$ is true for some $j \in S_i$, as illustrated in Figure 4. Figure 4(a) shows an example topology where the parent pointers represent the constructed spanning tree. Because of node mobility, node *A* becomes disconnected from the rest of the nodes and the topology changes to that shown in Figure 4(b). In order to detect such events, each node in the spanning tree sends periodic *Probe* messages to every node *j* in its list S_i. A node which receives a *Probe* message responds with a *Reply* message. The absence of a *Reply* message from a node *j* for a certain timeout period causes node *i* to remove *j* from list S_i and to no longer wait for an *Ack* message from node *j*. As shown in Figure 4, node *B*, which has already received an *Ack* message from node *C* but has yet to hear an *Ack* message from node *A*, eventually infers, using *Probe* messages, that node *A* has departed. Node *B* therefore removes *A* from the list S_B. Node *B* now has no more *Ack* messages to wait for and broadcasts a *Leader* message announcing *C* as the leader as illustrated in Figure 4(c).

When a node disconnects from its parent, it can no longer report an *Ack* message to its parent. Hence, it terminates

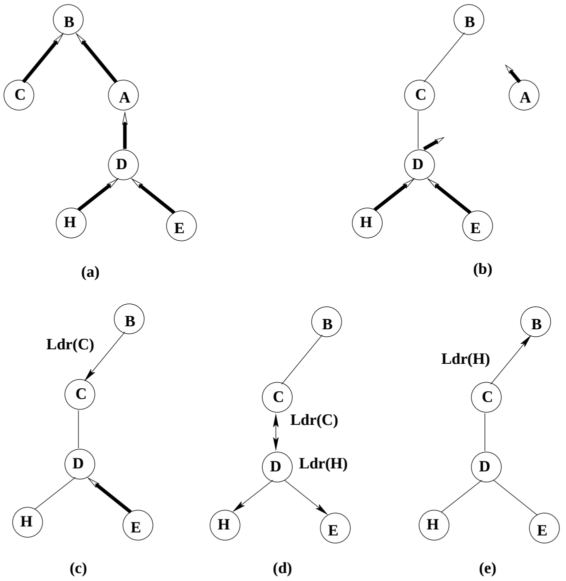


Figure 4: Operation of Leader Election Algorithm in the face of partitions

the diffusing computation by announcing its maximal downstream node as the leader. In our example, node *D*, which has *A* as its parent, eventually receives *Ack* messages from all its immediate children. As shown in Figure 4(d), node *D* subsequently detects node *A*'s departure and terminates the computation by broadcasting a *Leader* message, announcing *H* as the leader. In essence, node *D*, in the absence of a parent node, reports its maximal downstream node through its current neighbors.

Finally, node *C*, whose current leader is itself, propagates the new leader, *H*, upstream to node *B*. Thus, all nodes eventually have node *H* as their leader. Node *A* also eventually detects the departure of node *D* and its parent, node *B*. In this case, node *A* announces itself to be the leader.

Handling Partition Merges: Node mobility can also cause partitions to merge. There are several possibilities. The simplest case, as shown in Figure 5(a), involves two connected components, each with a unique leader, merging together by the formation of a new link between nodes *A* and *U* (indicated by a dashed line). Nodes *A* and *U* then exchange their leader identities over the newly formed link. Since node *U* has a higher-identity-leader (*W*) than *A* (*C*), *A* adopts *W* as its own leader and then broadcasts the new leader to the rest of the nodes in its component.

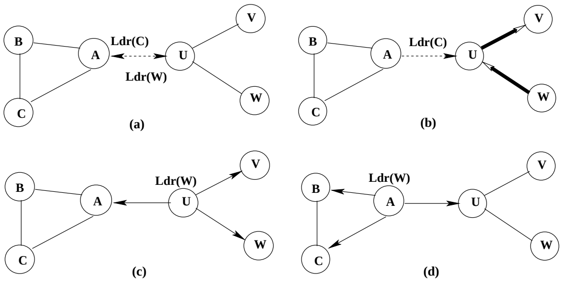


Figure 5: Operation of Leader Election Algorithm in the face of merges

Another possibility is that one or both of the components merging together are without a leader and are involved in a computation. As shown in Figure 5(b), nodes *U*, *V*, *W* are involved in a computation and merge with nodes *A*, *B* and *C* which have *C* as their leader. Our algorithm handles this case by allowing the ongoing computation to terminate before the exchange of leader identities takes place. In Figure 5(b), node *A*, upon detecting a new link formation announces its leader identity to node *U*. Upon termination of the ongoing computation, node *U* announces its leader (node *W*) to node *A*, which adopts *W* as its new leader and propagates this information to nodes *B* and *C*. The case when both merging components have an ongoing computation is also handled similarly.

Handling Node Crashes and Restarts: Our algorithm also tolerates arbitrary node crashes and recoveries. A node failure is treated as an instance of network partitioning and appropriate actions are taken, as described earlier. For our algorithm to tolerate node recoveries, we assume that when a node recovers from a crash, it first bootstraps the election process as described in Section 4.3.2. At the end of the bootstrap phase, the recovered node is without a leader and therefore starts a new election to find its leader. In essence, node crashes are treated as occurrences of partitions while the event of a node recovering from a failure is treated as the merging of two components.

Having described the operation of our election algorithm and its ability to adapt to arbitrary topological changes, we next study its performance through simulation in a mobile, ad hoc network under a variety of operating conditions.

5 Formal Verification of Algorithm

We formally verify the correctness of the election algorithm, titled *Election-Opt* in the previous section. We use linear time temporal logic as a formal tool for this purpose. An extensive introduction to temporal logic is available in [33]. We just present a sketch of our proofs here, while detailed proofs are presented in the Appendix.

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'), ⧫ ('at some moment in the past'), that are used to reason about past and the future. We use temporal logic to formally specify algorithm properties and establish invariants of our leader election algorithm.

We show that starting in an initial state (specified in the initialization part of the formal algorithm specification in Figure 2, the system is guaranteed to reach a state satisfying predicate *P* after an arbitrary number of changes and that it will forever remain in a state satisfying *P*, where

$$P \equiv (\forall i\ \exists l : \Box(\delta_i = 0 \wedge lid_i = l \wedge l = max\{j | d_{i,j} < \infty\}))$$

In words, predicate *P* describes the set of all states in which a node *i*'s leader (lid_i) is *l*, the maximum-identifier-node in *i*'s connected component, and that *l* remains *i*'s leader forever. Thus, we achieve a weaker form of stabilization with our algorithm, in which stabilization is guaranteed provided each process starts execution in a designated initial state. The proof of our algorithm is divided into two parts:

- *Safety Property:* If diffusing computations stop in the network, then eventually all nodes will have a unique leader from within their connected component which is the maximum-identifier-node in that component. More formally:

$$\text{If } G \equiv \Box(\forall i : \delta_i = 0)$$

then we prove that

$$G \Rightarrow \Diamond P$$

- *Progress Property:* We also show that eventually there are no more diffusing computations in the network, i.e., $\Diamond G$ holds true.
- *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 eventually reaches a stable state. The *Termination* property is achieved as a consequence of the *Safety* and *Progress* properties.

6 Simulation Setting

We simulate our algorithm using GloMoSim [15], an event-driven, packet-level simulator. The main objective of our simulations was to gain a better understanding of how to design and implement leader election algorithms for ad hoc networks and also study in detail how various simulation parameters impact the performance of our algorithm. The performance metrics which we consider in our simulations are the *Fraction of Time Without Leader*, *Message-Overhead*, *Election-Time* and *Election-Rate* defined below.

6.1 Performance Metrics

We now define the various performance metrics considered. *Fraction of Time Without Leader* (*F*) is the fraction of simulation time that a node is involved in an election (as indicated by $\delta = 1$). *Election-Rate* (*R*) is defined as the average number of elections that a node participates in per unit time (i.e. the average rate at which node *i* goes from $\delta_i = 0$ to $\delta_i = 1$). *Election-Time* (*T*) is defined as the mean time elapsed between the instant at which a node begins participating in an election process (corresponds to $\delta_i = 1$ in our algorithm) and the instant at which it knows the identity of its leader ($\delta_i = 0$). In some cases, node partitions occur during the election process, as shown in Figure 4. In that example, node *B* first chose *C* as its leader and then subsequently it chose node *H* as its leader. *Election-Time* corresponds to the time elapsed from the instant at which node *B* starts participating in the election until the time at which *B* chose *H* as its leader. *Message-Overhead* (*M*) is defined as the average number of messages sent by a node per election.

In all of our simulations, we study the behavior of the various performance metrics as a function of the number of nodes (*N*) in the simulation. For a given *N*, the results are averaged over all nodes in each simulation run and over 10 different simulation runs. We plot the 95% confidence intervals on the graphs.

6.2 Simulation Environment

Nodes are randomly placed in a 2000m × 2000m terrain. For all network sizes, nodes move according to the Random Waypoint mobility model. The parameters of this model are the minimum node speed (V_{min}), maximum node speed

(V_{max}) and node pause time (P_t). In accordance with suggestions made in [31], we set the minimum node speed to a positive value (1m/s in our case) throughout our simulations. Throughout our simulations, we use the IEEE 802.11 MAC protocol and Free Space Propagation path-loss model. For the results reported in this paper, the underlying routing protocol used is AODV [17]. We note that we simulated our algorithm with DSR [18] as the routing protocol and observed very little change in the results shown in this paper.

In our simulations, a leader node periodically broadcasts *Beacon* messages to other nodes. Absence of some number (indicated by *max-beacon-loss*) of *Beacon* messages from its leader causes a node to start a fresh election. In our simulations, we set the value of *beacon-interval* to 20 seconds and of *max-beacon-loss* to 6. This means that a node triggers an election, if it does not receive a heartbeat from the leader for a duration of two minutes. Note that *max-beacon-loss* is arbitrarily chosen and can be set according to application requirements.

7 Design Issues and Lessons Learned

We now describe various issues involved in designing an efficient leader election algorithm with particular emphasis on the fact that the election algorithm operates in a mobile, wireless ad hoc network. Using simulations, we illustrate how subtle changes in the algorithm and signaling methods it uses result in dramatic differences in its performance. We believe that the lessons learned from our simulations can be useful in other protocol designs.

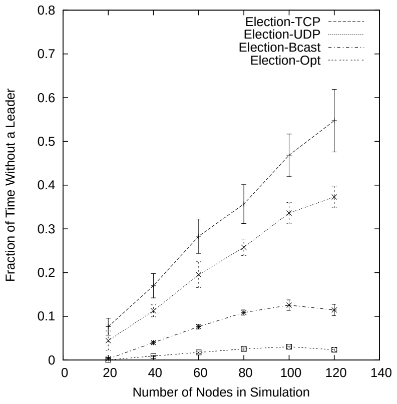


Figure 6: "Evolution" of Leader Election Algorithm

In Figure 6, we plot the fraction of time a node is without a leader (*F*) against number of nodes (*N*) for four different implementations of our election algorithm, which we call *Election-TCP*, *Election-UDP*, *Election-Bcast* and *Election-Opt*. Each of these versions will be described very shortly. The curves were obtained from a scenario in which nodes moved according to Random Waypoint model with $V_{min} = 1m/s$, $V_{max} = 3m/s$ and $P_t = 150s$. The node transmission range was 200m and each simulation was run for 100 simulation minutes. Each point is obtained by averaging over 10 different runs. The main purpose of this graph is merely to depict the dramatic improvements in performance by careful design choices.

From Figure 6, several interesting insights can be gleaned as we change from one algorithm to another, improving the algorithm each time based on the insights obtained. We explain how each change to the algorithm improves upon the previous case, until we achieve a really efficient algorithm.

1. **Election-TCP:** The uppermost-most curve in Figure 6 represents the *Election-TCP* version of our election algorithm. In this version, all messages (except the *Leader* message which is always flooded) are sent using TCP. This curve can be regarded as the most naive implementation of our election algorithm and serves as a baseline against which the other versions can be compared.

   From Figure 6, we see that the fraction of time that the node is without a leader is 0.54, when $N = 120$. There are several reasons that *Election-TCP* performs very poorly. Firstly, each leader-election message incurs the additional overhead of a three-way handshake before it is actually sent and a connection teardown phase after it is sent. This introduces a significant overhead on the wireless link bandwidth. Secondly, the large TCP timeout values for connection set-up, introduce a significant delay before node disconnections are detected by our algorithm, thereby resulting in an increased election duration. We therefore conclude that TCP is not a suitable choice for signaling for our election algorithm.

2. **Election-UDP:** The next curve immediately below *Election-TCP* in Figure 6, represents the *Election-UDP* version. In this version, all algorithm messages are sent point-to-point using UDP. If the message delivery fails after a fixed number of trials, the destination is assumed to be disconnected.

   We observe from the graph that the fraction F drops significantly from 0.54 to 0.37, when $N = 120$. This confirms our conclusion that TCP is not suitable for messaging in wireless networks, especially for distributed algorithms such as leader election. Although *Election-UDP* shows a significant improvement over *Election-TCP*, we will soon see that we can make further improvements in the performance of our algorithm.

3. **Election-Bcast:** The curve immediately below *Election-UDP* is labeled *Election-Bcast* and represents the version in which *Election* messages are sent using UDP broadcast. We will see shortly that broadcasting *Election* messages can help reduce not only the *Election* messages but also the number of *Ack* messages.

   (a) **Reduction in number of *Election* messages**: Recall from the algorithm description that, on joining an election, each node sends *Election* messages to all of its neighbors. In *Election-UDP*, a node unicasts an *Election* message to each of its immediate neighbors. However, because of the broadcast nature of wireless medium, a single broadcast *Election* message is sufficient to reach all neighbors.

   (b) **Reduction in number of *Ack* messages**: One interesting "side-effect" of using broadcast to send *Election* messages is that nodes need not maintain a list of their neighbors, as is done in the *Election-TCP* and *Election-UDP* versions of our algorithm. In *Election-TCP* and *Election-UDP* versions of our algorithm, node i reports an *Ack* message for each *Election* message it receives. This means that the number of *Ack* messages that a node has to send increases with the number of neighbors.

      Example:

      Consider an example network as shown in Figure 7.

      Node A is the source of the computation and nodes B, C and D are its children. Based on our algorithm description, nodes B, C and D would each receive three *Election* messages. Since each node sends an *Ack*

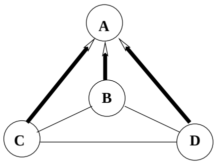


Figure 7: An example network and the corresponding spanning tree

message for every *Election* message it receives, a total of 9 *Ack* messages are sent in the *Election-TCP* and *Election-UDP* versions of our algorithm.

We can reduce the number of *Ack* messages by observing that a child-node needs to send an *Ack* message only to its parent node and can "ignore" *Election* messages received from other nodes. In Figure 7, upon receiving an *Election* message from node *A*, nodes *B*, *C* and *D* each report back an explicit *Child* message to node *A*, accepting *A* as their parent. This step is necessary, since in *Election-Bcast* version, nodes do not maintain a list of their neighbors. Based on the received *Child* messages, each parent knows precisely who its children are in the spanning tree. Meanwhile, node *A*, after sending an *Election* message, starts a timer called `CHILD-TIMEOUT`, to receive *Child* messages from its children and upon expiry of `CHILD-TIMEOUT`, *A* knows that nodes *B*, *C* and *D* are its children. Nodes *B*, *C* and *D* in turn propagate *Election* messages to one another. Since each of these nodes already has node *A* as its parent, none of them report *Ack* messages to one another. Eventually, nodes *B*, *C* and *D* report their pending *Ack* message to their parent-node *A*. Thus, with the proposed modification nodes *B*, *C* and *D* send 2 messages (1 *Child* + 1 *Ack*) each and therefore the total number of messages is reduced from 9 to 6. This modification can greatly reduce the number of *Ack* messages in a densely connected network, where each node has a large number of neighbors and consequently experiences contention for the shared wireless medium.

As seen in Figure 6, use of the above optimizations causes a further decrease in *F* from about 0.37 to 0.11 when there are 120 nodes. Thus we see that the reduction in message overhead also translates into a reduction in fraction *F*. The key insight we obtain from *Election-Bcast* is that the broadcast nature of wireless medium should be exploited not only for efficient messaging, but also in the form of optimizations to the proposed algorithm itself.

4. **Election-Opt:** In the *Election-TCP*, *Election-UDP* and *Election-Bcast* versions of the algorithm, whenever a node receives an *Election* message, it immediately joins the election by propagating the *Election* message to its own immediate neighbors. However, if a node currently has a leader that is not the same as departed leader (as indicated in the received *Election* message), then a node need not join the election. But it adopts a new leader if the newly elected leader has higher identity than its current leader.

   Example:

   Consider the scenario in Figure 8, when a node *G*, which is without a leader, starts a new computation and almost simultaneously merges (represented by a dashed line in the figure) with another connected component which has a leader, viz node *C*. Node *A*, upon receiving an *Election* message from node *G* does **not** propagate *G*'s *Election* message any further and immediately reports back an *Ack* message to node *G*. But eventually when node *H* is elected as the leader (by nodes *D*, *G* and *H*), *G*'s *Leader* message propagates to node *A*. Since *H* has a higher identifier than node *A*'s current leader (node *C*), *A* adopts *H* as its leader and in turn propagates a *Leader* message to nodes *B* and *C*, which eventually adopt *H* as their leader.

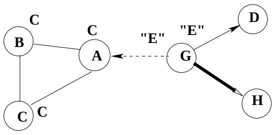


Figure 8: Optimization : Avoiding unnecessary elections

With this optimization, the fraction *F* again shows another significant decrease from 0.11 in case of *Election-Bcast* to about 0.025 when *N* = 120. The lowest curve, in particular, demonstrates the efficiency of our algorithm, in that each node has a leader up to 97.5% of the time.

From this section, we observe that careful signaling choices and algorithm optimizations can result in a very efficient algorithm design. We next formally specify the various correctness properties (detailed proofs in Appendix) of our *Election-Opt* algorithm using temporal logic and subsequently study, using simulations, its performance in a wide variety of operating conditions.

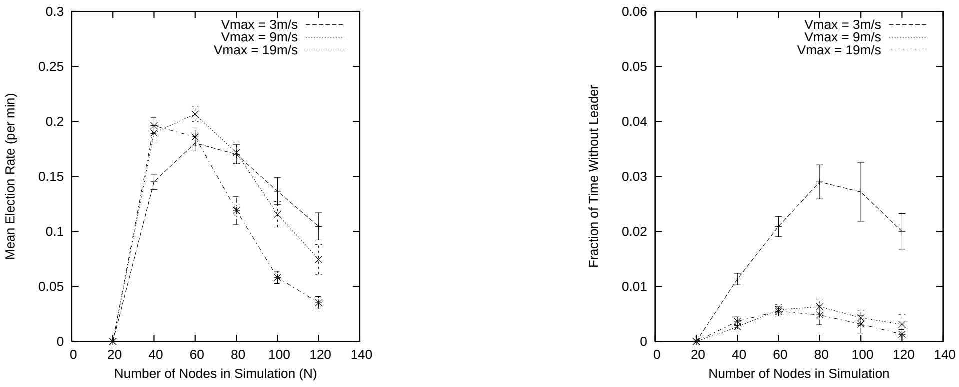
8 Sensitivity Analysis : Results and Discussion

8.1 Election-Rate and Fraction of Time Without Leader

We will first study the impact of node mobility and transmission range of nodes on *Election-Rate* and *Fraction of Time Without Leader*. We run each of our simulations for a duration of 400 minutes while discarding the data obtained from the first 150 minutes (corresponding to initial transient phase).

1. **Impact of Node Mobility :** In order to study the impact of node mobility, we vary V_{max}, the maximum node speed while keeping pause times and minimum node speed fixed. The graphs in Figure 9, show the *Election-Rate* and *Fraction of Time Without Leader* for three different values of V_{max} viz. 3m/s, 9m/s, 19m/s.

The first conclusion we draw based on 9(a) is that, irrespective of actual value of V_{max}, the *Election Rate* of a node first increases with *N* and then starts decreasing with any further increase in *N*. This is because when $N = 20$, most of the nodes can be expected to be isolated (i.e. not connected to any other node) and remain so for long durations. But as *N* increases, there will be a few components each with a few nodes. Node mobility results in frequent leader departures and hence an increased *Election Rate*. But after a certain threshold, the node density (nodes per unit area) becomes very high and most of the nodes belong to a large connected component. Although nodes move around, high node density means that components remain connected for longer durations and, hence, *Election Rate* drops. The second observation from Figure 9(a) is that *Election-Rate*, rather interestingly, **decreases** with the increase in node speeds for large values of *N*. We explain this behavior based on an observation made in [30] that higher speeds



(a) **Mean Election Rate**

(b) **Fraction (F)**

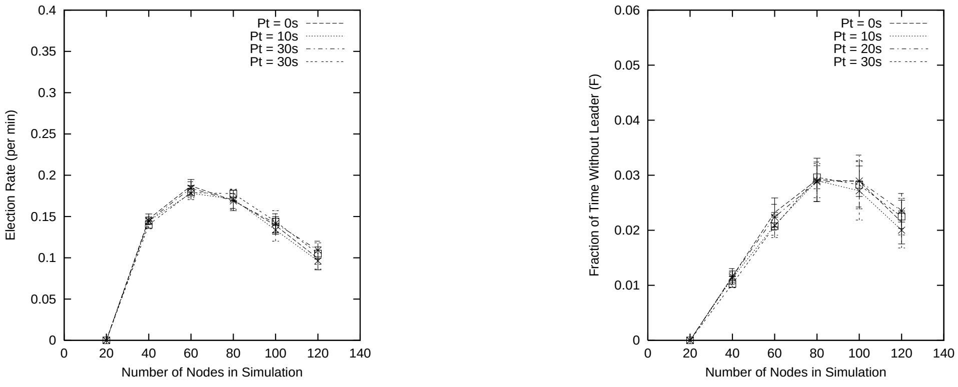
Figure 9: **Performance Vs V_{max}. Here $T_x = 200m$, $V_{min} = 1m/s$ and $P_t = 10s$.**

lead to a shorter lifetime of small components. In our case, what this means is that even though nodes might get disconnected from their leaders, at higher speeds they are disconnected only for very short durations. Hence, before *max-beacon-loss* becomes 6 these disconnected nodes get connected back to their leaders, thereby avoiding a fresh election.

From Figure 9(b), the *Fraction of Time Without Leader*(F) of a node initially increases with increase in *N* but then eventually drops slightly with further increase in *N*. This behavior can be described based on the trends observed in *Election-Rate*. Initially, with increase in *N*, *F* also increases because of increase in *Election-Rate* and also because elections can be expected to be longer when there are more nodes. However, for $N >= 100$, longer election durations are more than compensated by a sharp decrease in *Election-Rate* and this accounts for the slight drop in *F*. Also, with an increase in V_{max}, the fraction *F* drops still further because of the decrease in *Election-Rate* as described earlier. As seen from Figure 9(b), the fraction *F* is always below 3% and is very close to 0 when V_{max} = 19m/s. This means that each node always has a leader 97 to almost 100% of the time.

We also studied the impact of pause times (P_t) on *Election-Rate* and *Fraction*(F). We observed that pause times cause very little change in the performance metrics. In Figure 10, keeping V_{max} fixed at 3 m/s, we plot various performance metrics for four different pause times 0s, 10s, 20s and 30s. As can be seen in Figure 10, the curves match each other very closely. Repeating the experiment for $V_{max} = 9m/s$ and $V_{max} = 19m/s$ also showed that pause times do not affect *Election-Rate* and *Fraction*(F).

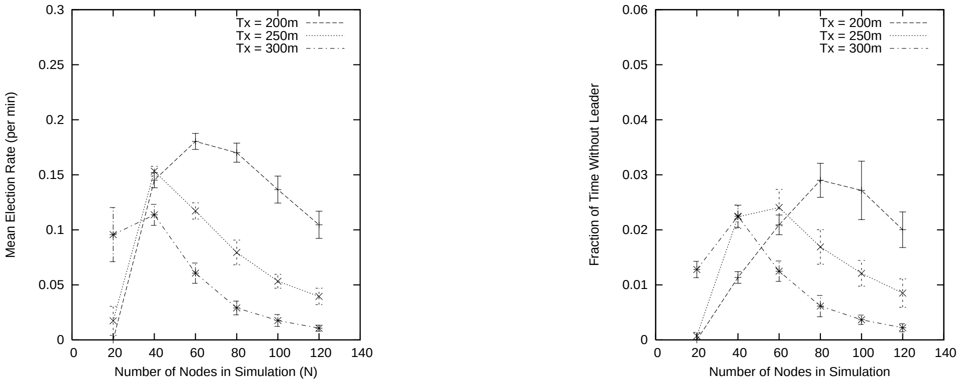
2. **Impact of Transmission Range (T_x) :** Keeping the node speeds and pause times fixed, we study the impact of T_x on *Election-Rate* and fraction *F* for three different choices of T_x, viz. 200m, 250m and 300m. From Figure 11(a), we see that increased transmission range of nodes leads to a higher *Election-Rate* when *N* is small (i.e. $N = 20$). Intuitively, this is because for a large value of T_x, there are fewer isolated nodes, but each component still has only a few nodes. For large values of *N* ($N >= 60$), the *Election-Rate* becomes smaller with increase in T_x. This is because for a given *N*, the component sizes are larger for large values of T_x and partitions occur less frequently. From Figure 11(b), we see that the fraction *F* increases with T_x for small values of *N*, but for large values of *N*, it



(a) **Election Rate**

(b) **Fraction(F)**

Figure 10: **Performance Vs P_t. Here $T_x = 200m$, $V_{min} = 1m/s$ and $V_{max} = 3m/s$.**



(a) **Mean Election Rate**

(b) **Fraction (F)**

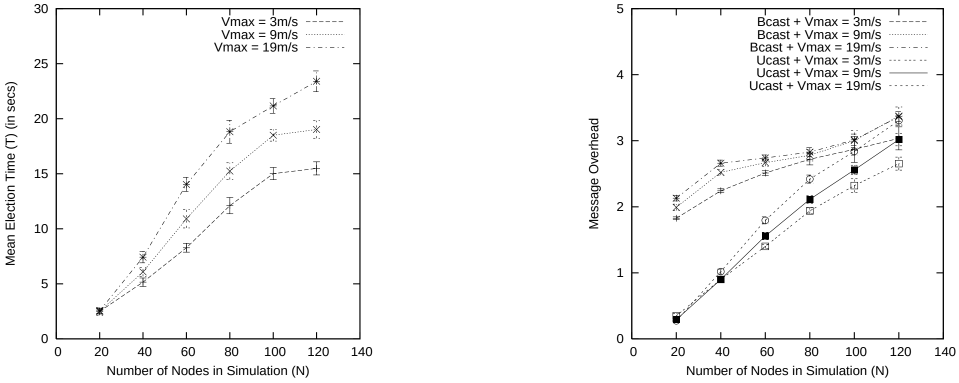
Figure 11: **Performance Vs T_x. Here $V_{max} = 3m/s$, $V_{min} = 1m/s$ and $P_t = 10s$.**

decreases with T_x because of corresponding decrease in *Election-Rate*. Again, we see that the fraction *F* is very low : always less than 3% and almost 0% when $T_x = 300m$ and $N = 120$.

8.2 Election-Time and Message-Overhead

We observed in Section 8.1 that the *Election-Rate* in some scenarios is very low (almost 0) and therefore, to get meaningful estimates of *Election-Time* and *Message Overhead* we would have had to run the simulations for very long durations. Therefore, in order to study *Election-Time* and *Message-Overhead*, we perform simulations in which elections are triggered at periodic intervals of time. Each simulation is run for a duration of 200 minutes and we discard the data from the first 50 minutes allowing for the nodes to converge to a constant average speed.

1. **Impact of Node Mobility** : As in Section 8.1, we plot the performance curves for three different choices of V_{max}, viz. 3m/s (low speed), 9m/s (medium) and 19m/s (high speed), and are shown in Figure 12. The pause time is fixed at 10 seconds for all node speeds.



(a) **Mean Election Time**

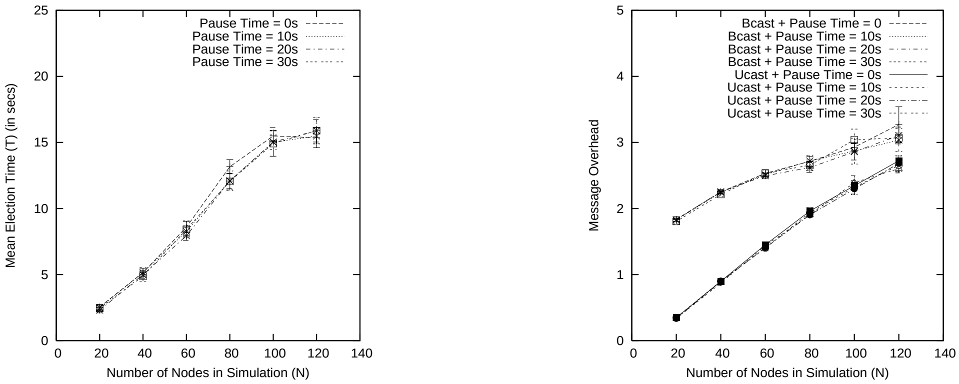
(b) **Message Overhead**

Figure 12: **Performance Vs** V_{max}**. Here** $T_x = 200m$**,** $V_{min} = 1m/s$ **and** $P_t = 10s$.

We first observe from Figure 12(a) that, irrespective of V_{max}, *Election-Time* increases with *N*. This is intuitive since as *N* increases, both the node density and the number of nodes involved in an election are expected to increase. This leads to greater contention for channel bandwidth and higher message delays. Furthermore, for a given *N*, the mean *Election-time* of a node **increases** with increase in node speed. This, most likely, is due to increased message delays incurred by the unicast *Ack* and *Child* messages. At higher node speeds, link breaks occur more frequently, therefore increasing, both the routing overhead (in terms of number of control packets) and unicast message delays. From Figure 12(a), the *Election-Time* ranges from 15 seconds when $V_{max} = 3m/s$ to about 23 seconds when $V_{max} = 19m/s$.

The *Message Overhead* is shown in Figure 12(b). Broadcast message overhead and unicast message overhead are shown separately. Recall from Section 7 that, any node (except for the source) in the spanning tree sends at least one unicast *Child* message and one *Ack* message to its parent. In addition, it sends at least 2 broadcast messages, viz. one *Election* message upon joining the election and one *Leader* message upon termination. From Figure 12(b), we see that when $N = 20$, (i.e.there are many isolated nodes), each of which just sends 2 broadcast messages, 1 *Election* + 1 *Leader* per election. But with the increase in N, both the broadcast message overhead and unicast message overhead increase, irrespective of actual value of V_{max}. This is because as N increases, components become larger and several nodes initiate elections concurrently when a leader departs. The broadcast overhead increases because each node sends one broadcast *Election* message for every computation it joins, while the unicast overhead increases since each node sends a unicast *Child* message for every computation it joins. The broadcast *Message-Overhead* shows very little difference with increase in V_{max}, while the unicast *Message-Overhead* increases only slightly. This is because, as elections get longer (with increasing V_{max}), the parent and child nodes in the spanning tree are more likely to exchange *Probe* and *Reply* messages, leading to increase in unicast overhead. However, it is evident from the graphs that this increase is very small. We thus conclude from Figure 12(b) that the *Message-Overhead* incurred by our algorithm is very small, ranging from 2-3 broadcast messages and 0-3 unicast messages per node per election.

As in the case of *Election-Rate* and *Fraction*(F), we observe that pause times cause very little change in both *Election-Time* and *Message-Overhead*. In Figure 13, keeping V_{max} fixed at 3 m/s, we plot various performance metrics for four different pause times 0s, 10s, 20s and 30s. As can be seen in Figure 13, the curves match each other very closely.

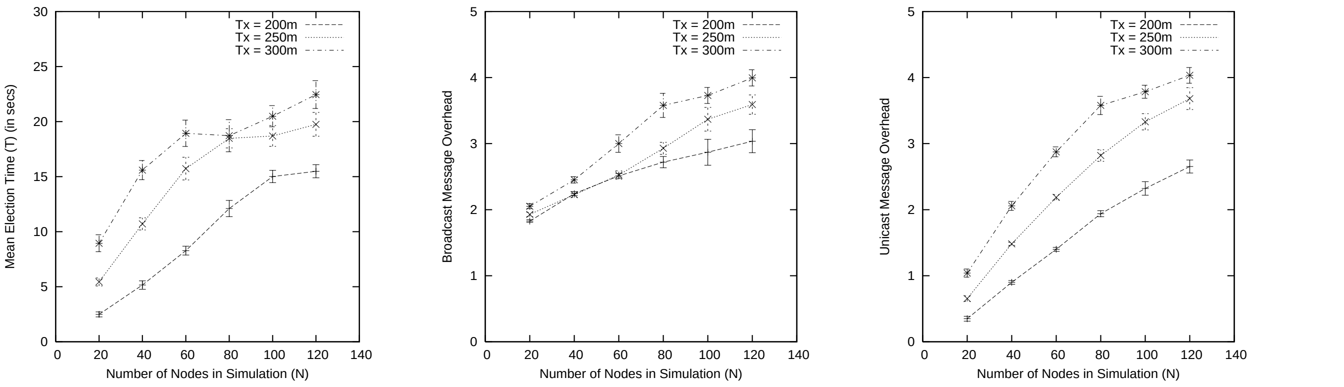


(a) **Mean Election Time**

(b) **Message Overhead**

Figure 13: **Performance Vs P_t. Here $T_x = 200m$, $V_{min} = 1m/s$ and $V_{max} = 3m/s$.**

2. **Impact of Transmission Range of Nodes :** We next study the effect of transmission range (T_x) of individual nodes on *Election-Time* and *Message-Overhead*. Keeping all other parameters fixed, we plot *Election-Time* and *Message-Overhead* against N for different values of T_x. Our study showed that for a given N, the *Election-Time* increases with an increase in T_x. This is intuitive, since the increase in transmission range leads to increased node density (average number of neighbors for a given node) and larger component sizes. Hence, there will be larger numbers of nodes participating in any given election. Increased node density also translates into greater contention for the channel bandwidth leading to greater message delays. The *Message-Overhead* was again observed to be fairly small. However, it showed a slight increase with increase in T_x. This is because, as components grow larger in size and fewer in number, higher will be the number of concurrent elections triggered on leader departure, leading



(a) **Mean Election Time**

(b) **Broadcast Message Overhead**

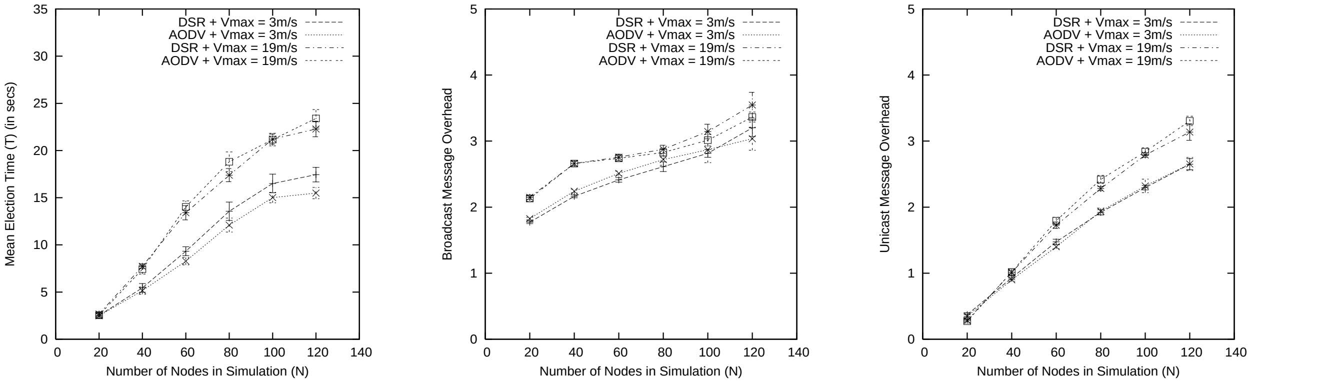
(c) **Unicast Message Overhead**

Figure 14: **Performance Vs T_x. Here $P_t = 10s$, $V_{min} = 1m/s$ and $V_{max} = 3m/s$.**

to a higher message overhead as explained earlier.

8.2.1 Choice of Routing Protocol

Finally, we study the performance of our algorithm using two popular routing algorithms in ad hoc networks, viz. DSR [18] and AODV. The motive here is to study whether the choice of routing protocols makes a significant difference to various performance metrics. The resulting graphs are shown in Figure 15. We show the mean *Election-Time* using AODV and DSR for $V_{max} = 3m/s$ and $V_{max} = 9m/s$.



(a) **Mean Election Time**

(b) **Broadcast Message Overhead**

(c) **Unicast Message Overhead**

Figure 15: **Performance Vs Routing Protocol. Here $T_x = 200m$, $V_{min} = 1m/s$**

From Figure 15, we conclude that there is no clear choice between AODV and DSR. However, we would like to point that the implementation of AODV in GloMoSim follows the specification in AODV Internet Draft (draft-ietf-manet-aodv-03.txt) and since then, many improvements to AODV have been proposed. It will be interesting to see whether our algorithm shows any improvements using the optimizations made to AODV and also by using other routing protocols such as GPSR [19].

8.3 Summary of Sensitivity Analysis

Our simulation study has shown that :

1. Our election algorithm shows very good performance in terms of *Fraction of Time Without Leader* (*F*). The fraction *F* was almost always less than 3% over a wide variety of operating conditions.

2. The *Election-Rate*, decreases with the increase in node speeds as well as transmission range.

3. The *Election-Time* increases as V_{max} and T_x increase.

4. The *Message-Overhead* of our algorithm is very small. Neither node speeds nor pause times significantly impact *Message Overhead*, but increase in transmission range of nodes leads to a small increase in *Message Overhead*.

5. We observed that pause times do not impact the performance metrics significantly. We also simulated our algorithm with DSR and observed that the results obtained were similar as that of AODV.

9 Conclusions and Discussion

In this paper, we proposed an asynchronous, distributed extrema finding algorithm for mobile, ad hoc networks and showed it to be "weakly" self-stabilizing. We formally established this property of our algorithm using temporal logic. Finally, we simulated our algorithm and through our study provide useful insights, based on our experiences in designing a leader election algorithm. We found that subtle changes to algorithm and the signaling methods it uses lead to dramatic improvements in our algorithm performance. We also studied in detail the impact of various parameters such as node mobility, transmission range, etc. on the various performance metrics of our algorithm.

Although, in this paper we described our algorithm as an extrema-finding one, our algorithm can be used in scenarios where just a unique leader is desired. Also, it might sometimes be useful to elect *top k* nodes in the network as opposed to just a single node with the extrema. This case can be trivially handled by modifying our algorithm to have each node report the *top k* downstream nodes in its *Ack* message to its parent during the election process. Also, in Section 3, we assumed that the links were bidirectional. However, the algorithm should work correctly even in the case of unidirectional links, provided that there is symmetric connectivity between nodes. We are currently working on the proof of correctness in the case of unidirectional links. We are also investigating on how our election algorithm can be adapted to perform clustering in wireless, ad hoc networks. Finally, we note that the concept of diffusing computations is quite generic in nature and can potentially be applied to other distributed problems in ad hoc networks.

References

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

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

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

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

[5] 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.

[6] 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.

[7] A. Amis, R. Prakash, T. Vuong, and D.T. Huynh. MaxMin D-Cluster Formation in Wireless Ad Hoc Networks. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*, March 1999.

[8] S. Banerjee and S. Khuller. A Clsutering Scheme for Hierarchichal Control in Multi-Hop Wireless networks. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM), Anchorage, Alaska*, Apr. 2001.

[9] C. Lin and M. Gerla. Adaptive Clustering for Mobile Wireless Networks. In *IEEE Journal on Selected Areas in Communications*, 15(7):1265-75, Sep 1997.

[10] P. Basu, N. Khan and T. Little. A Mobility based metric for clustering in mobile ad hoc networks. In *International Workshop on Wireless Networks and Mobile Computing*, Apr. 2001.

[11] R. Gallager, P. Humblet and P. Spira. A Distributed Algorithm for Minimum Weight Spanning Trees. In *ACM Transactions on Programming Languages and Systems*, vol.4, no.1, pages 66-77, January 1983.

[12] D. Peleg. Time Optimal Leader Election in General Networks. In *Journal of Parallel and Distributed Computing*, vol.8, no.1, pages 96-99, January 1990.

[13] G. Taubenfeld. Leader Election in presence of n-1 initial failures. In *Information Processing Letters*, vol.33, no.1, pages 25-28, October 1989.

[14] 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.

[15] 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.

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

[17] 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, pp. 90-100.

[18] D. Johnson and D. Maltz. DSR : The Dynamic Source Routing Protocol for Multi-Hop Wireless Ad Hoc Networks. In *Ad Hoc Networking*, edited by Charles E. Perkins, Chapter 5, pp. 139-172, Addison-Wesley, 2001.

[19] B. Karp and H. Kung. GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. In *Proceedings of ACM Mobicom*, August 6-11, 2000.

[20] 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.

[21] 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.

[22] S. Vasudevan, B. DeCleene, N. Immerman, J. Kurose and D. Towsley. Leader Election Algorithms for Wireless Ad Hoc Networks. In *Proceedings of IEEE DISCEX III*, April 22-24, 2003.

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

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

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

[26] 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.

[27] 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.

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

[29] 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.

[30] T. Chu and I. Nikolaidis. On the Artifacts of Random Waypoint Simulations. In *Proceedings of 1st International Workshop on Wired/Wireless Internet Communications (WWIC 2002), in conjunction with International Conference on Internet Computing (IC'02).*, 2002.

[31] J. Yoon, M. Liu and B. Noble. Random Waypoint Considered Harmful In *Proceedings of IEEE Infocom*, 2003.

[32] 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.

[33] Z. Manna and A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems - Specification.

APPENDIX

A Introduction to Temporal Logic

We use temporal logic as the formal tool for proving correctness of our algorithm. 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 continuously 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. For ease of representation, we introduce two temporal operators $\blacksquare_\tau$ and $\blacklozenge_\tau$. $\blacksquare_\tau\ \varphi$ means that φ was true at very moment in the past since time τ, while $\blacklozenge_\tau\ \varphi$ means at some moment in the past after τ, φ holds true.

B Notation Used in Proofs

Before proving the correctness, we will formally state the definitions of various symbols and predicates employed by our proofs.

- δ_i : a binary variable which is 0 if a node *i* is not currently in an election; otherwise it is 1.
- $d_{i,j}$: distance from node *i* to node *j*.
- lid_i : a variable which contains the value of *i*'s current leader if $\delta_i = 0$; else if $\delta_i = 1$ it contains the value of *i*'s last leader.
- N_i : node *i*'s current list of neighbors, as known to node *i*.
- src_i : computation-index of last computation node *i* participated in.
- $src_{i,k}$: $src_{i,k} = j \iff (src_i = k \wedge \blacklozenge(rcv_{i,j}(E_k) \wedge src_i \neq k \wedge \odot src_i = k))$ is true i.e., a node *j* that causes node *i* to begin participation in computation *k*.
- $Nlf(i,j)$: a predicate at node *i*, which is set to true when a new link is formed between nodes *i* and *j*, i.e. when a node *j* not previously in N_i has just been added to N_i. More formally,

  $$(\forall i, j : (j \notin N_i \wedge \odot(j \in N_i)) \Rightarrow \odot Nlf(i,j)) \tag{1}$$

  Once $Nlf(i,j)$ is set to true, it will continuously remain true until it is falsified by execution of action 8 in Figure 2.
- $snd_{i,j}(m)$: a predicate which is true at some time instant *t* if an action is executed at time *t* in which node *i* executes the **send** action to send a message *m* to node *j*. Note that the predicate $snd_{i,j}(m)$ is only true at that time instant *t* and is false immediately preceding and following *t*.
- $rcv_{i,j}(m)$: a predicate which is true at the instant when a message *m* sent by node *j* is at the head of node *i*'s receive buffer (and therefore ready to be processed by node *j*). It is falsified once the message is dequeued.

C Assumptions

We now state the various assumptions made in our proofs:

1. *Links:* We assume that links are bidirectional. Although we make this simplifying assumption for ease of proof of correctness, we strongly believe that our algorithm would still work correctly if links were unidirectional, as long as all nodes are connected (have a path) to each other. The proof in the case of unidirectional links is currently being worked on. Also, links are assumed to be FIFO, i.e. messages do not get reordered in the links.

2. *Receive Buffers:* The receiver buffer size is large enough so as not to cause buffer overflows at any instant in the execution of the algorithm. We also assume that for every message *m* in the receiver buffer *eventually* $rcv_{i,j}(m)$ will be true.

3. *Uniqueness of Messages:* Each message *m* sent via the **send** action is unique. A message *m* sent (by a **send** action) at some time instant *t* is distinct from a message *m'* sent (by **send** action) at some other time instant *t'*.

4. *Reliable Delivery:* We make a "weak" reliability requirement, in which a message sent will eventually be received provided that the receiver is guaranteed to remain connected to the sender forever. More formally,

$$(\forall i, j, m : (\Box\ (d_{i,j} < \infty) \wedge snd_{i,j}(m)) \Rightarrow (\Diamond\ rcv_{j,i}(m))) \tag{2}$$

D Proof of Correctness

As stated in Section 5, the proof of correctness of our leader election algorithm involves proofs of *Safety* and *Liveness* conditions. In particular, assuming each node *i* starts execution in a state satisfying predicate I_i, we will show that starting in a state satisfying predicate *B* that could have been reached after a finite number of arbitrary changes, *eventually* the system is guaranteed to reach a state satisfying predicate *P*, where

$I_i \equiv \delta_i = 0 \wedge \Delta_i = 0 \wedge Num_i = 0 \wedge lid_i = 0 \wedge N_i = \{\}$

B $\equiv$ a predicate that describes a state reached after a finite number of topological changes, assuming each node *i* starts execution in a state satisfying predicate I_i.

$P \equiv (\forall i\ \exists l : \Box(\delta_i = 0 \wedge lid_i = l \wedge l = max\{j | d_{i,j} < \infty\}))$

Note that a state satisfying predicate *P* is a **stable** state, i.e. once *P* holds true it will hold forever.

D.1 Safety Property

Let $G \ \equiv\ \Box(\forall i : \delta_i = 0)$

Then the *Safety Property* states that upon reaching a state satisfying the predicate G, the system is guaranteed to reach a state in which each node has a leader which is the maximum-identity-node in its connected component, i.e. eventually predicate *P* holds true.

The *Safety Property* can be stated more formally as:

$$G \ \Rightarrow\ \Diamond\ P \tag{3}$$

Proof of Safety Property:

In order to establish the *Safety Property*, we will first prove the following claim:

Claim 1:

$$G \Rightarrow (\forall i, j, l_1 : (j \in N_i \wedge lid_i = l_1) \Rightarrow (\blacklozenge snd_{i,j}(L(l_1)) \vee Nlf(i, j))) \tag{4}$$

Note that a node could have adopted a leader with identifier, l_1, at many different time instants in the past. But for the sake of convenience of notation, we will always use $L(l_1)$ to refer to the leader message sent after node *i* last adopted leader l.

Proof of Claim 1: **Claim 1** intuitively means that whenever a node *i* has a leader, it has either already sent this leader identity to all its neighboring nodes or it is the case a new neighboring node *j* has arrived and the predicate $Nlf(i, j)$ is true. As we will see, using **Claim 1** we will later establish that if *G* holds true, then all nodes within a connected component will *eventually agree on a common leader*. This will trivially lead to establishment of the *Safety Property*.

We will prove statement (4) by contradiction, i.e. let us assume that:

$$G \wedge (\exists i, j, l_1 : (j \in N_i \wedge lid_i = l_1 \wedge \blacksquare \neg snd_{i,j}(L(l_1)) \wedge \neg Nlf(i, j))) \tag{5}$$

Let us consider one such pair of nodes, *i* and *j*, satisfying (5).

Let *t* be the time instant at which node *i* last adopted its current leader l_1 and let *t'* be the time instant at which the predicate $Nlf(i, j)$ was last set to false by node *i*. Stated formally,

$$\blacksquare_t lid_i = l_1 \tag{6}$$

$$\blacksquare_{t'} \neg Nlf(i, j) \tag{7}$$

From (5), we have

$$j \in N_i \wedge \neg Nlf(i, j) \tag{8}$$

From (6), (7), and (8), it follows that :

$$\blacksquare_{t'} : j \in N_i \tag{9}$$

There are two only possibilities :

1. $t > t'$: This means that node *i* adopted its current leader, l_1 after *j* was last added to N_i. Therefore, node *i* would have executed action 6 or action 7 of the algorithm at the time *t* when it adopted leader, l_1. Since $t > t'$, from (9), it follows that :

   $$\blacksquare_t j \in N_i \tag{10}$$

   $$\therefore \blacklozenge_t snd_{i,j}(L(l_1)) \tag{11}$$

   But this contradicts our assumption 5.

2. $t \leq t'$: This means that node *i* adopted l_1 before node *j* became its neighbor. Recall that, the list N_i (node *i*'s neighbors) is initialized to empty set and by definition (1) will be set to true whenever node *j* is newly added to N_i. In accordance with assumption (5), the only way $Nlf(i, j)$ is set to false is by action 8 of the algorithm. But in executing this action, node *i* would have sent leader l_1 to node *j*. Hence, we have

   $$\blacklozenge_{t'} snd_{i,j}(L(l_1)) \tag{12}$$

   Again, this leads to a contradiction of our assumption (5).

Hence, our assumption (5) is wrong. This proves **Claim 1**.

Now the following statement trivially holds true:

$$G \Rightarrow \Box G \tag{13}$$

Substituting from (4) in (13), we get

$$G \Rightarrow \Box(\forall i, j, l_1 : (j \in N_i \wedge lid_i = l_1) \Rightarrow (\blacklozenge snd_{i,j}(L(l_1)) \vee Nlf(i,j))) \tag{14}$$

$\Box$

From (9), (11) and (12), we conclude that if $snd_{i,j}(L(l_1))$ was true at some moment in the past then node *j* was *i*'s neighbor ever since and including the instant when $snd_{i,j}(L(l_1)$ was true and will forever be *i*'s neighbor (as no more changes occur). Thus it follows that,

$$G \Rightarrow \Box(\forall i, j, l_1 : (j \in N_i \wedge lid_i = l_1) \Rightarrow (\Box(j \in N_i)\mathcal{S}(j \in N_i \wedge snd_{i,j}(L(l_1))) \vee Nlf(i,j))) \tag{15}$$

Since no more changes occur and $\Box\delta_i = 0$ holds true (as G is true), if $Nlf(i,j)$ is true for a pair of neighbors *i* and *j*, action 8 of the algorithm will remain continuously enabled and hence executed eventually. Therefore,

$$G \Rightarrow (\forall i, j : (j \in N_i \wedge Nlf(i,j)) \Rightarrow (\Diamond\neg Nlf(i,j))) \tag{16}$$

Since no more topological changes occur, $Nlf(i,j)$ will forever remain false. Hence,

$$G \Rightarrow (\forall i, j : (j \in N_i \wedge Nlf(i,j)) \Rightarrow (\Diamond\Box\neg Nlf(i,j))) \tag{17}$$

Substituting from (17) in (15), we get

$$G \Rightarrow \Diamond\Box(\forall i, j, l_1 : (j \in N_i \wedge lid_i = l_1) \Rightarrow (\Box(j \in N_i)\mathcal{S}(j \in N_i \wedge snd_{i,j}(L(l_1))))) \tag{18}$$

From Assumption (2) it follows that,

$$G \Rightarrow \Diamond\Box(\forall i, j, l_1 : (j \in N_i \wedge lid_i = l_1) \Rightarrow (\blacklozenge(rcv_{j,i}(L(l_1)) \wedge \Box(j \in N_i))) \vee \Diamond(rcv_{j,i}(L(l_1))))) \tag{19}$$

$$\therefore G \Rightarrow \Diamond\Box(\forall i, j, l_1 : (j \in N_i \wedge lid_i = l_1) \Rightarrow \Diamond\Box(\blacklozenge(rcv_{j,i}(L(l_1)) \wedge \Box(j \in N_i)))) \tag{20}$$

The proof *Safety Property* involves proving the following components:

- Eventually each node has a leader and it remains with that leader forever, i.e. leader changes stop.
- Eventually all nodes in a connected component agree on a unique leader.
- The elected leader is from within the connected component, and finally
- The elected leader is the maximum-identifier-node in that connected component.

We first argue that *eventually leader changes stop*. This is because all leader ids are finite and totally ordered. As each node changes its leader only in favor of a higher-identity-leader, each node can change its leader only a finite number of times.

$$\therefore G \Rightarrow \Diamond\ (\forall i \exists l : \Box\ (lid_i = l)) \tag{21}$$

We will now use (19) to prove **Claim 2** which states that *eventually all nodes in a connected component have the same leader*.

Claim 2:

$$G \Rightarrow \Diamond\Box(\forall i, j : (j \in N_i \wedge i \in N_j) \Rightarrow lid_i = lid_j) \tag{22}$$

Proof of Claim 2: In **Claim 2**, we show that eventually any two neighboring nodes always agree on a common leader. Because of Assumption (1) stating that links are bidirectional, **Claim 2** will trivially imply that all nodes in a connected component have the same leader.

The proof is by contradiction. Let us assume that

$$G \wedge \Box\Diamond(\exists i, j : i \in N_j \wedge j \in N_i \wedge lid_i \neq lid_j) \tag{23}$$

From (21) and (23), we infer

$$G \wedge \Box\ \Diamond\ (\exists i, j, l_1, l_2 : i \in N_j \wedge j \in N_i \wedge \Box(lid_i = l_1) \wedge \Box(lid_j = l_2) \wedge l_1 \neq l_2)) \tag{24}$$

Since no more topological changes occur, we can rewrite (24) to get

$$G \wedge \Box\ (\exists i, j, l_1, l_2 : \Diamond\Box(i \in N_j \wedge j \in N_i \wedge lid_i = l_1 \wedge lid_j = l_2 \wedge l_1 \neq l_2)) \tag{25}$$

$$\therefore G \wedge (\exists i, j, l_1, l_2 : \Diamond\Box(i \in N_j \wedge j \in N_i \wedge lid_i = l_1 \wedge lid_j = l_2 \wedge l_1 \neq l_2)) \tag{26}$$

$$P(i, j, l_1, l_2) \equiv (i \in N_j \wedge j \in N_i \wedge lid_i = l_1 \wedge lid_j = l_2 \wedge l_1 \neq l_2)$$

Substituting from (20) in (26),

$$G \wedge (\exists i, j, l_1, l_2 : \Diamond\Box(P(i, j, l_1, l_2) \wedge \blacklozenge(rcv_{j,i}(L(l_1)) \wedge \Box(j \in N_i)) \wedge \blacklozenge(rcv_{i,j}(L(l_2)) \wedge \Box(i \in N_j)))) \tag{27}$$

Let us assume without loss of generality $l_1 > l_2$ in (27). In (27), $rcv_{j,i}(L(l_1))$ could only have been true at instant before *j* last adopted l_2. Otherwise, node *j* would have chosen l_1 as its leader following its last adoption of l_2 by executing action 7 of the algorithm, thus violating (26) which states that l_2 will remain *j*'s leader forever since the instant it last adopted l_2.

This means that $rcv_{i,j}(L(l_2))$ could have been true only after $rcv_{j,i}(L(l_1))$ was true. From (27) we infer,

$$G \wedge (\exists i, j, l_1, l_2 : \Diamond\Box(P(i, j, l_1, l_2) \wedge \blacklozenge(rcv_{j,i}(L(l_1)) \wedge \Diamond rcv_{i,j}(L(l_2)) \wedge \Box(j \in N_i \wedge i \in N_j)) \tag{28}$$

$$G \wedge (\exists i, j, l_1, l_2 : \Diamond\Box(P(i, j, l_1, l_2) \wedge \blacklozenge(lid_i = l_1 \wedge rcv_{i,j}(L(l_2)) \wedge \Box(j \in N_i \wedge i \in N_j)))) \tag{29}$$

When a node *i* with a leader l_1 receives $L(l_2)$ such that $l_1 > l_2$, action 8 of the algorithm will eventually get executed. This means that node *i* will send its leader l_1 once again to node *j*. Substituting in (29) we get,

$$\therefore G\wedge(\exists i, j, l_1, l_2 : \Diamond\Box(i \in N_j \wedge j \in N_i \wedge lid_i = l_1 \wedge lid_j = l_2 \wedge l_1 \neq l_2 \wedge \blacklozenge(snd_{i,j}(L'(l_1)) \wedge \Box(j \in N_i \wedge i \in N_j)))) \tag{30}$$

The notation $L'(l_1)$ in (30) is used to distinguish from $L(l_1)$, since $L'(l_1)$ refers to the second time node *i* sends a leader message with identifier l_1 after it last adopted l_1 as its leader.

$$\therefore G\wedge(\exists i, j, l_1, l_2 : \Diamond\Box(i \in N_j \wedge j \in N_i \wedge lid_i = l_1 \wedge lid_j = l_2 \wedge l_1 \neq l_2 \wedge \blacklozenge(rcv_{j,i}(L'(l_1)) \wedge \Box(j \in N_i \wedge i \in N_j)))) \tag{31}$$

But $rcv_{j,i}(L'(l_1))$ could have been true only after *j* last adopted l_2, since $L'(l_1)$ was sent in response to $rcv_{i,j}(L(l_2))$. Hence by execution of action 7 of the algorithm, node *j* on receiving $L'(l_1)$ will be forced to adopt l_1 as its leader since $l_1 > l_2$, and thereby violating (25) which states that l_2 will remain *j*'s leader forever since the instant it last adopted l_2. We can reason in the same manner for the case when $l_1 < l_2$ and arrive at a contradiction. Hence, our initial assumption in (23) must be wrong. Thus **Claim 2** is proved.

Now predicate *G* and (22) trivially imply that eventually each node has a *leader from within its connected component*; otherwise action 1 of the algorithm will remain continuously enabled in every node *i* with that leader, since no more leader changes occur. This causes an election to be triggered in node *i* leading to the condition $\delta_i = 1$, which violates predicate *G*. More formally,

$$G \Rightarrow \Diamond\Box(\forall i, j : (j \in N_i \wedge i \in N_j) \Rightarrow (lid_i = lid_j \wedge d_{i,l} < \infty \wedge d_{j,l} < \infty)) \tag{32}$$

Finally, we claim that the elected leader is the *maximum-identifier-node from within the connected component.*

This is because, the max_i variable in a node i is initialized to its own identifier (in actions 1 and 2) and is updated only (by action 4) when an *Ack* message is received with a higher identifier than max_i's current value. When a node currently in a diffusing computation receives a *Leader* message, a node will adopt a leader only if the received leader identity is at least as large as max_i. When a node is not in a diffusing computation (i.e.it has a leader), it will adopt a leader only with a higher identity than its current one.

$$\therefore (\forall i : i \leq lid_i) \tag{33}$$

From (21), (32) and (33), we can infer that

$$G \Rightarrow \Diamond\, (\forall i \exists l : \Box\, (lid_i = l \wedge l = max\{k | d_{i,k} < \infty\})) \tag{34}$$

Thus the *Safety Property* stated in (3) is proved.

D.2 Liveness Property

The *Liveness Property* states that starting from a state satisfying predicate *B*, eventually (in a finite time) predicate *G* will hold true. Stated formally,

$$B \leadsto \Diamond\, \Box\, (\forall i : \delta_i = 0) \tag{35}$$

Proof of Liveness Property:
The proof of *Liveness Property* involves proof of the following two components :

- Once topological changes stop, only a finite number of diffusing computations are initiated thereafter.

- Once topological changes stop, every diffusing computation is guaranteed to terminate.

We will prove the first component of *Liveness Property*. Let *D* be the set of diffusing computations that are initiated once *B* holds true. More formally,

$$D \equiv \{\langle i, num\rangle | \exists i : \Diamond(src_i = \langle i, num\rangle \wedge \delta_i = 1)\} \tag{36}$$

Claim 3:

$$|D| < \infty. \tag{37}$$

Proof of Claim 3: Recall from Section 4.3.3 that each source can only initiate one diffusing computation with a given computation-index. Hence,(37) is sufficient to represent that "only finite number of diffusing computations are ever initiated."

In order to prove **Claim 3**, we first prove the following claim:

Claim 3.1

$$\Diamond\Box(\forall i, m : ((\Delta_i = 1 \wedge src_i = m)\mathcal{U}(\Delta_i = 0 \wedge src_i = m)) \Rightarrow \Diamond(\Delta_i = 0 \wedge src_i = m \wedge d_{i,max_i} < \infty)) \tag{38}$$

The above statement implies that once node *i* receives *Ack*s from all its children, the identifier it would report in its *Ack* message to its parent is guaranteed to be that of a node within its connected component.

Proof of Claim 3.1: We will show **Claim 3.1** to hold for an arbitrary diffusing computation *m* initiated after the last topological change occurred.

From the algorithm description, each node *i* for which $\Delta_i = 0$ is true, $snd_{i,p_i}(A(k))$ should have been true at the same instant when Δ_i is set to 0 (action 5 of the algorithm), where

$k = max_i = max(i, max\{id | \exists j : \blacklozenge p_j = i \wedge \blacklozenge rcv_{i,j}(A(id))\})$ i.e., *k* is the maximal downstream node from node *i*, which is also stored in variable max_i.

Since $\Delta_i = 0$, $\Delta_j = 0$ should have been earlier true $\forall j$ such that $p_j = i$ and which reported *Ack*s to node *i*. Stated formally,

If $P(i, j, m) \equiv (\Delta_i = 0 \wedge src_i = m \wedge \blacklozenge(p_j = i \wedge rcv_{i,j}(A)))$, and
$Q(i, j, m, k) \equiv (\exists k : \Delta_j = 0 \wedge src_j = m \wedge max_j = k \wedge snd_{j,i}(A(k)))$, we have

$$(\forall i, j, m : P(i, j, m) \Rightarrow \blacklozenge Q(i, j, m, k)) \tag{39}$$

We will next show that the identifier reported by each node *i* to its parent p_i in the *Ack* message to its parent is the identifier of a node from within *i*'s connected component. More formally, we will show that

$$(\forall i, j, m : P(i, j, m) \Rightarrow \blacklozenge(Q(i, j, m, k) \wedge d_{j,k} < \infty)) \tag{40}$$

We will use a result from ([1], Lemma 19.1) which states that for any diffusing computation *m* parent pointers are acyclic, i.e.,

$$(\forall m, n : (\neg \exists a_1, a_2, \ldots, a_n : p_{a1} = a2 \wedge p_{a2} = a3 \wedge \ldots \wedge p_{an} = a1 \wedge src_{a_1} = src_{a_2} = \ldots = src_{a_n} = m)) \tag{41}$$

Also, each node can participate in at most one diffusing computation at any given time. Hence,

$$(\forall n : (\neg \exists a_1, a_2, \ldots, a_n : p_{a1} = a2 \wedge p_{a2} = a3 \wedge \ldots \wedge p_{an} = a1) \tag{42}$$

Since there are only finite nodes in the network, (42) will imply that there will eventually be nodes which do not have any children, which we call *leaf nodes*. From the algorithm description, these *leaf nodes* report only their own identities in the *Ack*s to their parent nodes. Also, a node *i* updates the variable max_i based only on the *Ack*s received from its children, which are identified by $A.flag = 1$. Hence, every node *i* which is not a *leaf node* reports $A(k)$, such that $k = max(i, max\{id | \exists j : \blacklozenge p_j = i \wedge \blacklozenge rcv_{i,j}(A(id))\})$. Since the diffusing computation *m* was initiated only after the last topological change occurred, the identity reported by each node *i* in its *Ack* to its parent must be within *i*'s connected component. Thus (39), will become

$$(\forall i, j, m : P(i, j, m) \Rightarrow \blacklozenge(Q(i, j, k, m) \wedge d_{j,k} < \infty)) \tag{43}$$

Thus (40) is proved.

From (40), we can infer that

$$(\forall i, j, m : P(i, j, m) \Rightarrow d_{i,max_i} < \infty) \tag{44}$$

(38) trivially follows from (44).

This proves **Claim 3.1**.

From (38), it follows that,

$$\Diamond\Box(\forall i, m((\delta_i = 1 \wedge src_i = m)\, \mathcal{U}\, (\delta_i = 0 \wedge src_i = m)) \Rightarrow \Diamond(d_{i,lid_i} < \infty)) \tag{45}$$

This is because, from action 6 of the algorithm, the leader identifier reported by an initiator *i* of a diffusing computation *m* is same as the value in variable max_i. From (38), we know that $d_{i,max_i} < \infty$ is true. Hence, every node *j* that adopts max_i as its leader should also have $d_{j,max_i} < \infty$ holding true.

We now continue with our proof for **Claim 3**.

Define,

$$L = \{j | \exists i : d_{i,j} = \infty \wedge \Diamond(\delta_i = 0 \wedge lid_i = j)\} \tag{46}$$

In other words, set *L* represents the set of nodes which get elected as leaders but which are disconnected from the nodes which have chosen them as their leader. Thus, *L* represents the set of departed leaders. We will show that the set of departed leaders is a finite set. i.e.,

$$|L| < \infty \tag{47}$$

This is because there are always only a finite number of nodes in the network and therefore, only a finite number them have leaders. Since each node has at most one leader at any given time, there are only a finite number of leaders at any given time. Also from (45), it follows that eventually forever for any diffusing computation *m*, the leader identifier *l* announced by an initiator *i* upon termination of *m*, is that of a node within *i*'s connected component and therefore cannot belong to the set *L*. This means that *eventually* no more additions to the set *L* can take place. Therefore, $|L| < \infty$ must hold true.

So far we have established that the set *L* of departed leaders is a finite set. We will use this result to prove (37). The proof of (37) is by contradiction. i.e. let us assume that $|D| = \infty$, i.e, infinite number of diffusing computations are initiated. This means that there exists a node *i* that **initiates** infinite number of diffusing computations. A node *i* initiates a fresh diffusing computation only by action 1 of the algorithm, i.e. when $\delta_i = 0 \wedge d_{i,lid_i} = \infty$. Since $|L| < \infty$ (from (47)), for a node *i* to initiate infinite number of diffusing computations, there must exist a node $k \in L$ such that node *i* adopts *k* infinite number of times.

In other words, for $|D| = \infty$ to hold, the following condition must hold :

$$(\exists i, k : k \in L \wedge \Box\Diamond((\delta_i = 1 \wedge src_i.id = i \wedge lid_i = k)\,\mathcal{U}\,(\delta_i = 0))) \tag{48}$$

We rewrite (48) as,

$$\therefore (\exists i, k : k \in L \wedge R(i, k)) \tag{49}$$

where $R(i, k) \equiv \Box\Diamond((\delta_i = 1 \wedge src_i.id = i \wedge lid_i = k)\,\mathcal{U}\,(\delta_i = 0))$

Let *A* and *B* are two nodes such that $R(A, B)$ holds true. From (38), we know that every time node *A* participates in a diffusing computation max_A will eventually contain an identifier from within *A*'s connected component. Thus for (48) to hold, we must have

$$(\exists j : j \in N_A \wedge \Box\Diamond((\delta_A = 1 \wedge lid_A = B)\,\mathcal{U}\,(\Delta_A = 0 \wedge \Diamond(rcv_{A,j}(L(B)) \wedge \odot(\delta_A = 0 \wedge lid_A = B))))) \tag{50}$$

(50) informally means that for (48) to hold, there exists a neighboring node *j* from which *A* receives $L(B)$ message following which it adopts *B* as its leader.

Define,

$$H_{i,k} = \{j | \Box\Diamond(rcv_{i,j}(L(k)) \wedge k \in L \wedge \Delta_i = 0 \wedge \odot lid_i = k)\} \tag{51}$$

In other words, $H_{i,k}$ represents the set of *i*'s neighboring nodes that force node *i* to adopt a leader $k \in L$ infinite number of times. Define,

$$(\forall i, k : k \in L \wedge R(i, k)), M_{i,k} = \begin{cases} H_{i,k} \\ \\ \bigcup_{j \in M_{i,k}} H_{j,k} \end{cases} \tag{52}$$

Thus, $M_{A,B}$ represents the set of nodes that adopt leader *B* an infinite number of times, given that $R(A, B)$ holds true.

In order to complete the proof for **Claim 3**, we next prove the following claim:

Claim 3.2:

$$(\forall i, k : (k \in L \wedge R(i, k)) \Rightarrow (\exists m, k' : k' \in L \wedge m \in M_{i,k} \wedge k' > k \wedge R(m, k')) \tag{53}$$

The above statement means that there must exist a node *m* in the set $M_{i,k}$ that itself initiates an infinite number of computations by adopting a leader $k' \in L$ an infinite number of times and such a leader *k*' has a strictly higher identifier than node *k*. **Claim 3.1** and **Claim 3.2** will together prove **Claim 3**.

Proof of Claim 3.2: We will prove (53) by contradiction. Let us assume that

$$(\exists i, k : (k \in L \wedge R(i,k)) \wedge (\forall m, k' : \neg(k' \in L \wedge m \in M_{i,k}) \vee (k' \leq k \vee \neg R(m,k')))) \tag{54}$$

Let us assume for the rest of proof of **Claim 3.2** that A and B are nodes such that $B \in L \wedge R(A,B)$ is true. Then (54) can be restated as :

$$(\forall m, k' : (k' \in L \wedge m \in M_{A,B}) \Rightarrow (k' <= B \vee \neg R(m,k'))) \tag{55}$$

Thus, there are two possible cases in (55). Let us first consider that $\neg R(m,k')$ is true.

Now we know that

$$\neg(p\,\mathcal{U}\,q) \iff \neg q \mathcal{W}(\neg p \wedge \neg q) \tag{56}$$

Substituting from (56) in (54) for $\neg R(m,k')$, we get

$$(\forall m, k' : (k' \in L \wedge m \in M_{A,B}) \Rightarrow \Diamond\Box(E(m,k')\,\mathcal{W}\,(E(m,k') \wedge F(m,k'))) \tag{57}$$

where,

$E(m,k') \equiv \neg(\delta_m = 0)$, and

$F(m,k') \equiv \neg(\delta_m = 1 \wedge src_m.id = m \wedge lid_m = k')$

From (57), we get

$$(\forall m, k' : (k' \in L \wedge m \in M_{A,B}) \Rightarrow \Diamond\Box(E(m,k'))) \tag{58}$$

$$(\forall m, k' : (k' \in L \wedge m \in M_{A,B}) \Rightarrow \Diamond\Box(\delta_m = 1)) \tag{59}$$

By definition of $M_{A,B}$, since $m \in M_{A,B}$ is true , it means that $m \in H_{n,B}$, for some node *n*. This means that node *n* receives $L(B)$ from node *m* infinite number of times. This means that node *m* adopts leader *B* infinite number of times. But from (59), if $\Diamond\ \Box(\delta_m = 1)$ holds, then node *m* will eventually no longer adopt leader *B*. Hence $m \notin M_{A,B}$ must be true. But this leads to a contradiction. Hence $\neg R(m,k')$ cannot be true.

$$(\forall m, k' : (k' \in L \wedge m \in M_{A,B}) \Rightarrow R(m,k')) \tag{60}$$

Substituting from (60) in (55), we get

$$(\forall m, k' : (k' \in L \wedge m \in M_{A,B} \wedge R(m,k')) \Rightarrow (k' <= B)) \tag{61}$$

In order to complete the proof for **Claim 3.2**, we will prove four claims **Claim 3.2.1-Claim 3.2.4**. Let us go through the proofs of each of these four claims in turn.

Claim 3.2.1:

$$(\forall m : (m \in M_{A,B}) \Rightarrow \Box\Diamond((\delta_m = 1 \wedge lid_m = B)\mathcal{U}(\delta_m = 0))) \tag{62}$$

i.e, every node in $M_{A,B}$ will participate in an infinite number of computations initiated in response to departure of leader *B*.

Proof of Claim 3.2.1:

Assume that the negation of (62) holds true. Hence, we must have

$$\exists m : (m \in M_{A,B} \wedge \Diamond\Box((\delta_m = 1)\mathcal{W}((\delta_m = 1) \wedge \neg(\delta_m = 1 \wedge lid_m = B)))) \tag{63}$$

$$\therefore \exists m : (m \in M_{A,B} \wedge \Diamond\Box(\delta_m = 1)) \tag{64}$$

From (64), $m \notin M_{A,B}$ must be true, leading to a contradiction. Hence (62) must be true. This proves **Claim 3.2.1**.

Claim 3.2.2: We now claim that for every node *m* in $M_{A,B}$, eventually every time node *m* has a leader, its identifier will be at most as large as that of *B*.

$$(\forall m : (m \in M_{A,B} \Rightarrow \Diamond\Box(\delta_m = 0 \Rightarrow lid_m <= B)) \tag{65}$$

Proof of Claim 3.2.2:

Once again, we can prove (65) by assuming its negation holds true. i.e.,

$$(\exists m : (m \in M_{A,B} \wedge \Box\Diamond(\delta_m = 0 \wedge lid_m > B)) \tag{66}$$

But since $m \in M_{A,B}$ is true, from(66), we must have

$$(\exists m : (m \in M_{A,B} \wedge \Box\Diamond(\delta_m = 0 \wedge lid_m = B) \wedge \Box\Diamond(\delta_m = 0 \wedge lid_m > B))) \tag{67}$$

$$\therefore (\exists m : (m \in M_{A,B} \wedge \Box\Diamond(\delta_m = 0 \wedge lid_m = B) \wedge \Box\Diamond(\exists k : k \in L \wedge k > B \wedge R(m,k)) \tag{68}$$

But (68) contradicts (61). Hence our assumption in (66) is wrong. Therefore, (65) must be true. This proves **Claim 3.2.2**

From **Claim 3.2.2** we can infer that

$$(\forall m : (m \in M_{A,B} \Rightarrow \Diamond\Box((\delta_m = 0 \wedge lid_m = B) \Rightarrow ((\delta_m = 0 \wedge lid_m = B)\mathcal{U}(\delta_m = 1 \wedge lid_m = B)))) \tag{69}$$

This statement means that eventually, whenever a node *m* adopts *B*, it will continuously have *B* as its leader until it begins to participate in a computation started by a node that detected *B*'s departure. This follows from **Claim 3.2.2** which states that eventually *m* can only adopt a leader with identifier $\leq B$ and that $B \in L$.

Define,

$$W_{k,B} = \{i | i \in M_{A,B} \wedge (\Delta_i = 1 \wedge src_i = k \wedge lid_i = B)\mathcal{U}(\Delta_i = 0 \wedge src_i = k)\} \tag{70}$$

Thus, $W_{k,B}$ represents the set of nodes in $M_{A,B}$ that participate in diffusing computation *k* triggered in response to *B*'s departure.

From (38), we can infer that:

$$(\forall i, k : i \in W_{k,B} \Rightarrow \Diamond(\Delta_i = 0 \wedge src_i = k \wedge d_{i,max_i} < \infty)) \tag{71}$$

This means that for a node $i \in W_{k,B}$ to adopt leader *B* in the future (and it will have to, as it is in the set $M_{A,B}$), a node $j \notin W_{k,B}$ must send $L(B)$ to either node *i* directly or to some other node $m \in W_{k,B}$ which in turn "propagates" $L(B)$ either directly to node *i* or through a series of nodes along a path to node *i*.

Let us now formally represent the set of nodes which participated in diffusing computation *k* and which will receive $L(B)$ sent by a node that did not participate in *k*.

Define $(\forall j, k : j \notin W_{k,B})$,

$FWD_{j,k,B} = \{j\} \bigcup \{i | i \in W_{k,B} \wedge j \in N_i \wedge \Diamond(\Delta_i = 0 \wedge src_i = k \wedge \neg(\delta_i = 0 \wedge lid_i = B) \wedge rcv_{i,j}(L(B)) \wedge \odot(\delta_i = 0 \wedge lid_i = B))\}$

$FWD_{j,k,B} = FWD_{j,k,B} \bigcup \{i | i \in W_{k,B} \wedge \exists m : (m \in FWD_{j,k,B} \wedge lid_m = B \wedge m \in N_i \wedge \Diamond(\Delta_i = 0 \wedge src_i = k \neg(\delta_i = 0 \wedge lid_i = B) \wedge rcv_{i,m}(L(B)) \wedge \odot(\delta_i = 0 \wedge lid_i = B))\}$

(72)

In other words, $FWD_{j,k,B}$ represents the set of nodes in that participated in diffusing computation *k* and to which node *j* "propagated" leader *B*.

Also, eventually such a node *j* could only be a node in $M_{A,B}$; this is because each node $j \notin M_{A,B}$ can force its neighbors to adopt *B* only a finite number of times and will therefore eventually stop sending $L(B)$ messages. Hence, by definition of $M_{A,B}$,

$(\forall i, j : i \in M_{A,B} \wedge j \notin M_{A,B} : \Diamond\Box\neg(\Delta_i = 0 \wedge rcv_{i,j}(L(B)) \wedge \odot lid_i = B))$

Hence, we must have

$$\Diamond\Box(\forall j, k : j \in FWD_{j,k,B} \Rightarrow j \in M_{A,B}) \tag{73}$$

We next claim that :

Claim 3.2.3:

$$\Diamond\Box(\forall j, k, i : i \in FWD_{j,k,B} \Rightarrow \Diamond(\exists k' : k' \neq k \wedge (\delta_i = 1 \wedge lid_i = B \wedge src_i = k')\mathcal{U}(\delta_i = 0))) \tag{74}$$

Proof of Claim 3.2.3:

From definitions of $FWD_{j,k,B}$ and $W_{k,B}$ in (72) and (70) respectively and (73), we know that every node in $FWD_{j,k,B}$ is also a node in $M_{A,B}$.

Therefore from **Claim 3.2.1**,

$$(\forall j,k,i : i \in FWD_{j,k,B} \Rightarrow \Diamond(\exists k' : (\delta_i = 1 \wedge lid_i = B \wedge src_i = k')\mathcal{U}(\delta_i = 0)) \tag{75}$$

Thus **Claim 3.2.3** is proved.

Again from (38) and (74) it follows that,

$$(\forall j,k,i : i \in FWD_{j,k,B} \Rightarrow \Diamond(\exists k' : k' \neq k \wedge \Delta_i = 0 \wedge src_i = k' \wedge d_{i,max_i} < \infty)) \tag{76}$$

We next claim that all nodes in $FWD_{j,k,B}$ will participate in the same diffusing computation.i.e.,
Claim 3.2.4:

$$(\exists k'\forall j,k,i : i \in FWD_{j,k,B} \Rightarrow \Diamond((\Delta_i = 1 \wedge lid_i = B \wedge src_i = k')\mathcal{U}(\Delta_i = 0 \wedge src_i = k' \wedge d_{i,max_i} < \infty)) \tag{77}$$

Proof of Claim 3.2.4:
From(74), we state that:

$$\Diamond\Box(\forall j,k,i : i \in FWD_{j,k,B} \Rightarrow \Diamond(\exists k' : k' \neq k \wedge (\delta_i = 1 \wedge src_i = k' \wedge lid_i = B)) \tag{78}$$

From the algorithm specification, each node *i* participating in a computation, will report an *Ack* to its parent node **only** after it has received *Ack*s from each of its neighboring nodes *j*. Each node *j* will report an *Ack* to *i* only after receiving an *Election* message from *i*. This means that node *i*'s *Election* message is guaranteed to be received before *i* reports an *Ack* to its parent node. Formally stated,

$$\Diamond\Box(\forall i,j,k,k' : (\delta_i = 1 \wedge src_i = k \wedge src_{i,k} = k') \Rightarrow (j \in N_i\backslash\{k'\} \Rightarrow \Diamond(rcv_{j,i}(E(k'))))) \tag{79}$$

Because of our assumption that links are FIFO, if a node $i \in FWD_{j,k,B}$ begins participation in a computation k' triggered in response to *B*'s departure, then the *Election* message it propagates will be eventually be received by all of its neighbors *m* and this *Election* message will be received only after $rcv_{m,i}(L(B))$ is true.

From (79) we state,
$\Diamond\Box(\forall j,k,i,k',m,n : (i \in FWD_{j,k,B} \wedge \Diamond(src_i = k \wedge lid_i = B \wedge \Diamond(src_i = k' \wedge src_{i,k'} = n))) \Rightarrow ((m \in N_i\backslash\{n\} \wedge m \in FWD_{j,k,B}) \Rightarrow \Diamond(rcv_{m,i}(L(B)) \wedge \Diamond rcv_{m,i}(E(k'))))$

(80)

where $E(k')$ represents an *Election* message with computation-index k'. This means that, every *Election* message that a node $i \in FWD_{j,k,B}$ receives, it is guaranteed to be received by all of *i*'s neighbors $\in FWD_{j,k,B}$ only after its $L(B)$ message has reached them.

From the definition of $FWD_{j,k,B}$, every node $i \in W_{k,B} \wedge i \in FWD_{j,k,B}$ must have at least one neighbor *m* $\in FWD_{j,k,B}$ that forces node *i* to adopt leader *B* after computation *k*. More formally,

$$\Diamond\Box(\forall i,j,k : (i \in FWD_{j,k,B} \cap W_{k,B}) \Rightarrow (\exists m : m \in N_i \cap FWD_{j,k,B} \wedge (\neg(\delta_i = 0 \wedge lid_i = B)\mathcal{U}(rcv_{i,m}(L(B)) \wedge \odot(\delta_i = 0 \wedge lid_i = B)))) \tag{81}$$

Let $Q(m,i,j,k,B) \equiv (m \in N_i \cap FWD_{j,k,B} \wedge (\neg(\delta_i = 0 \wedge lid_i = B)\mathcal{U}(rcv_{i,m}(L(B)) \wedge \odot(\delta_i = 0 \wedge lid_i = B))))$

Then substituting in (81), we get

$$\Diamond\Box(\forall i,j,k : (i \in FWD_{j,k,B} \cap W_{k,B}) \Rightarrow (\exists m : Q(m,i,j,k,B))) \tag{82}$$

From **Claim 3.2.1**, we know that eventually for every node $i \in M_{A,B}$, whenever *i* adopts *B* as its leader it will continuously have *B* as its leader until it begins participation in a computation k'. Hence, from definition of $FWD_{j,k,B}$, if $Q(m,i,j,k,B)$ is true for some node *m*, then there can be no node m' other than *m* such that $Q(m',i,j,k,B)$ is true.

$$\Diamond\Box(\forall i,j,m,k : (i \in FWD_{j,k,B} \wedge i \in W_{k,B} \wedge Q(m,i,j,k,B)) \Rightarrow (\forall m' : m' \neq m \Rightarrow (\blacksquare\neg Q(m',i,j,k,B) \wedge \Box\neg Q(m',i,j,k,B)))) \tag{83}$$

Also, it is obvious that

$$\Diamond\Box(\forall i,j,m,k : (i,m \in FWD_{j,k,B} \cap W_{k,B} \wedge Q(m,i,j,k,B)) \Rightarrow (\exists m' : m' \neq i \wedge \blacklozenge Q(m',m,j,k,B))) \tag{84}$$

This is because, before a node *m* forces node *i* to adopt *B* as its leader, *m* itslef should have adopted *B* as its leader from some other node m'.

From (83) and (84), we infer that

$$\Diamond\Box(\forall j,k,m_1,m_2,\ldots,m_n : (m_1,\ldots,m_n \in FWD_{j,k,B} \cap W_{k,B}) \Rightarrow \neg(Q(m_1,m_2,j,k,B) \wedge Q(m_2,m_3,j,k,B) \wedge \ldots \wedge Q(m_n,m_1,j,k,B))) \tag{85}$$

Statements (82) and (85) together informally imply that if $p_m = i \iff Q(i,m,j,k,B)$, then the parent pointers form an acyclic tree rooted at node *j*.

Restating (75), we get

$$(\forall j,k,i : i \in FWD_{j,k,B} \Rightarrow \Diamond(\exists k' : (\delta_i = 1 \wedge lid_i = B \wedge src_i = k')\mathcal{U}(\delta_i = 0)) \tag{86}$$

From (80), we know that *i*'s *Election* message will be received by all of its neighbors only after its $L(B)$ message. From the algorithm operation and our premise that no more topological changes occur, *i* must have received *Acks* from all of its neighbors before δ_i is set to 0. This means that for any $i \in FWD_{j,k,B} \wedge \delta_i = 1 \wedge src_i = k'$, its *Election* message, $E(k')$, will cause each of its neighboring nodes *m*, such that $Q(m,i,j,k,B) \vee Q(i,m,j,k,B)$ holds true, to begin participation in computation k' or *m* is already in some computation such that $src_m \succ k'$. In that case, we know from (80), *i* will eventually receive $E(src_m)$ from its neighbors.

We know that there exists a total ordering on computation indices and from (86), we know that a node *i* can change its computation-index only a finite number of times before δ_i is set to 0. We, therefore, infer that

$$\Diamond\Box(\forall j,k,i,m : (i \in FWD_{j,k,B} \wedge (Q(i,m,j,k,B) \vee Q(m,i,j,k,B))) \Rightarrow (\exists k' : \Diamond(((\delta_m = 1 \wedge src_m = k')\mathcal{U}(\delta_m = 0)) \wedge ((\delta_i = 1 \wedge src_i = k')\mathcal{U}(\delta_i = 0))) \tag{87}$$

The above statement means that eventually for every node $i \in FWD_{j,k,B}$ will participate in the same computation as the node *m* which forced it to adopt *B* and also the node m' which was forced by node *i* to adopt *B*.

From (82), (85) and (87), we infer that eventually all nodes in $FWD_{j,k,B}$ must participate in the same diffusing computation.

Hence,

$$(\exists k' \forall j,k,i : i \in FWD_{j,k,B} \Rightarrow \Diamond((\Delta_i = 1 \wedge lid_i = B \wedge src_i = k'))) \tag{88}$$

Substituting from **Claim 3.2.1** in (88),

$$(\exists k' \forall j,k,i : i \in FWD_{j,k,B} \Rightarrow \Diamond(((\Delta_i = 1 \wedge lid_i = B \wedge src_i = k'))\mathcal{U}((\Delta_i = 0 \wedge src_i = k' \wedge d_{i,max_i} < \infty)))) \tag{89}$$

Thus **Claim 3.2.4** is proved.

From the definition of $W_{k,B}$,

$$\therefore (\exists k' \forall j,k,i : i \in FWD_{j,k,B} \Rightarrow \Diamond(i \in W_{k',B})) \tag{90}$$

Also, we must have that

$$\Diamond\Box(\forall i,n,j,k : (i \in W_{k,B} \backslash FWD_{j,k,B} \wedge n \in FWD_{j,k,B}) \Rightarrow \Box\neg(src_i = k \wedge \Delta_i = 0 \wedge rcv_{i,n}(L(B)))) \tag{91}$$

The above statement means that a node *i* in the set $W_{k,B} \backslash FWD_{j,k,B}$ could not have received $L(B)$ from a node in $FWD_{j,k,B}$; otherwise by definition of $FWD_{j,k,B}$, node *i* should have been in $FWD_{j,k,B}$ as well.

From (38), we know that any node *i* in the set $W_{k',B}$, at the end of computation k' (i.e.$\Delta_j = 0 \wedge src_j = k'$), can adopt leader *B* only if a node $j \notin W_{k',B}$ "propagates" $L(B)$ to node *i*.

Let $D_B = W_{k,B} \bigcup FWD_{j,k,B}, \forall j,k : k \in D$

But from (72), we know that $FWD_{j,k,B} = W_{k',B}, k' \in D$.

Let D_B be initialized to $W_{k,B}$, for some *k*. Consider the set $D_B = W_{k,B} \bigcup W_{k',B}$, for some computation k' such that $W_{k',B} = FWD_{j,k,B}$. Now, it is clear that $|D_B| > |W_{k,B}|$. This is because, by definition of $FWD_{j,k,B}$, $W_{k',B}$ contains a node *j* which does not belong to $W_{k,B}$. Thus, any node $i \in D_B$ can adopt a leader *B* again only if a node $j \notin D_B$ "propagates" $L(B)$ to node *i*, i.e. only if $i \in FWD_{j,m,B}$, where $m = src_i = k \vee k'$.

Thus, after every computation k', triggered in response to departure of node *B*, the size of D_B has to grow at least by one, i.e. after every computation at least one new node is added to D_B every time. Since there are only finite number of nodes eventually D_B will include all nodes in the network. Hence the set D_B cannot grow any further. This means that there will be no more diffusing computations in the network contradicting our initial assumption that infinite number of computations are triggered or $|D| = \infty$.

Hence, our assumption in (54) is wrong. This proves **Claim 3.2**.

Let $l_{max} = max\{i|i \in L\}$

But, recursively applying **Claim 3.2**, we must have

$$(\forall i : (R(i, l_{max})) \Rightarrow (\exists m, k' : k' \in L \wedge m \in M_{i,k} \wedge k' > l_{max} \wedge R(m, k')) \tag{92}$$

This leads to a contradiction as l_{max} is the maximum-identifier-node in set L.

Hence our assumption that $|D| = \infty$ is wrong. Hence, **Claim 3** is proved.

Claim 4:

We will next prove the second component of *Liveness Property* which states that starting in a state satisfying predicate *B*, eventually all diffusing computations are terminated.

$$(\forall i : (\delta_i = 1) \Rightarrow \Diamond\ (\delta_i = 0)) \tag{93}$$

Proof: We know from the operation of the algorithm that when a node that is currently in a diffusing computation with computation-index *m*, "hears" another diffusing computation with a higher computation-index *m'*, it stops participating in computation *m* in favor of *m'*. But we proved earlier that only finite number of diffusing computations are ever initiated (i.e. $|D| < \infty$), once predicate *B* is satisfied. This means that each node can "change" its diffusing-computation at most finite number of times. Hence,

$$(\forall i : (\delta_i = 1) \Rightarrow \Diamond(\exists m : (\delta_i = 1 \wedge src_i = m)\ \mathcal{W}(\delta_i = 0 \wedge src_i = m)))) \tag{94}$$

Let us prove (93) by contradiction. Let us assume that

$$(\exists i : \Box\ (\delta_i = 1)) \tag{95}$$

Substituting from (94) in (93), we get

$$(\exists i, m : \Box\ (\delta_i = 1 \wedge src_i = m)) \tag{96}$$

This means that either node *i* has reported an *Ack* message to its parent node p_i or it is still

$$(\exists i, m : \Box\ (src_i = m \wedge (\Delta_i = 1 \vee (\Delta_i = 0 \wedge \delta_i = 1)) \qquad (97)$$

Let us consider the two cases in (97) in turn:

Case 1:

$$(\exists i, m : \Box\ (src_i = m \wedge \Delta_i = 1) \qquad (98)$$

We know from the algorithm operation that,

$$(\forall i, m : (src_i = m \wedge \Delta_i = 1) \Rightarrow (\exists j \in S_i : \blacksquare \neg rcv_{i,j}(A_m))) \qquad (99)$$

where A_m denotes *Ack* message with its src field set to *m*.

From (99) in (98), we get

$$(\exists i, j, m : \Box\ (src_i = m \wedge \Delta_i = 1 \wedge j \in S_i \wedge \blacksquare \neg rcv_{i,j}(A_m)) \qquad (100)$$

In this case, node *i* will eventually send *Probe* messages to node *j*. If $d_{i,j} = \infty$, then absence of any *Rep* message from node *j* will eventually cause node *i* to time out and remove *j* from S_i. However, if node *i* receives a *Rep (R)* message from node *j*, the following cases can arise:

1. $rcv_{i,j}(R) \wedge R.src \neq m \wedge src_i = m$: In this case, action 4 of the algorithm will be enabled and will cause node *i* to remove *j* from S_i.

2. $rcv_{i,j}(R) \wedge R.\Delta = 0 \wedge src_i = m$: Again by action 4, node *j* will be removed from S_i.

3. $rcv_{i,j}(R) \wedge R.\Delta = 1 \wedge R.src = m \wedge R.flag = 0 \wedge src_i = m$: Since $R.flag = 0$, node *i* is **NOT** *j*'s parent node. Hence, by action 4 of the algorithm node *j* will be removed from S_i.

4. $rcv_{i,j}(R) \wedge R.\Delta = 1 \wedge R.src = m \wedge R.flag = 1 \wedge src_i = m$: This means that $p_j = i$ and S_j in turn is non-empty.

Hence, from the above conditions we can infer that for (100)to hold,

$$(\exists i, j, m : \Box\ (src_i = m \wedge \Delta_i = 1 \wedge j \in S_i \wedge src_j = m \wedge p_j = i \wedge \Delta_j = 1) \qquad (101)$$

We will now use a well-known result ([1], Lemma 19.1) that for any single computation *m* the parent pointers p_i of all nodes *i* participating in computation *m* are acyclic. But if we recursively apply (101), we must have an infinitely long chain of parent pointers if (100) is to hold true. Since there are only finite nodes in the network, our claim in (98) must be incorrect.

$$\therefore (\forall i : \Delta_i = 1 \Rightarrow \Diamond(\Delta_i = 0)) \qquad (102)$$

Case 2: Let us consider the remaining case in (97).

$$(\exists i, m : \Box\,(src_i = m \wedge \Delta_i = 0 \wedge \delta_i = 1) \tag{103}$$

This means that node *i* has reported an *Ack* to its parent, but it is yet to hear a *Leader* message. In this case, node *i* will periodically send *Probe* messages to its parent node, p_i. The parent node in turn will respond with *Rep(R)* messages.

Now, if $d_{i,p_i} = \infty \vee R.src \neq m$ is true, then action 6 of the algorithm will eventually be executed and hence δ_i will be set to 0. Thus, for (103) to hold,

$$(\exists i, m : \Box\,(src_i = m \wedge \Delta_i = 0 \wedge \delta_i = 1 \wedge d_{i,p_i} < \infty \wedge \delta_{p_i} = 1 \wedge src_{p_i} = m) \tag{104}$$

Substituting from (102) in (104), we get

$$(\exists i, m : \Box\,(src_i = m \wedge \Delta_i = 0 \wedge \delta_i = 1 \wedge d_{i,p_i} < \infty \wedge \delta_{p_i} = 1 \wedge src_{p_i} = m \wedge \Delta_{p_i} = 0) \tag{105}$$

Arguing in the same way as in *Case 1*, recursive application of (105) and the fact that the pointers p_i are acyclic, will yield that there must be infinite nodes in the network. Again, this leads to a contradiction. Therefore, our claim in (103) must be wrong.

This means that our assumption (95) must be wrong. This proves **Claim 4**.

Thus, we have proved the two obligations, viz. **Claim 3** and **Claim 4**, required for the **Liveness Property** to hold true.

D.3 Termination Property

The *Termination Property* states that *eventually all program actions are disabled*. This follows trivially once predicate *P* is established. This is because once predicate *P* is established, from the algorithm actions it is clear that eventually no more messages are sent or received. Also the predicate $Nlf(i, j)$ will eventually be set to false for all nodes $j \in N_i$. It is easy to see that none of the program actions are enabled thereafter.