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
{svasu, kurose, towsley}@cs.umass.edu

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.

I. 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 [9], routing coordination [23], sensor coordination [16], and general control [15], [20]. 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 [19] 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.

This work was supported in part by the National Science Foundation under grants ANI-0085848 and EEC-0313747001, and by the Defense Advanced Projects Research Agency under contract N66001-99-C-8616.

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. [14], [22], [8], [16], [13], [26], [1]), or assume that topological changes stop before an election starts (e.g. [3], [24]) or assume an unrealistic communication model such as a message-order preserving network [6]. While there are some proposals for leader election in mobile networks [20], 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 existing work on leader election, an

important contribution of this paper is that it presents a very systematic and methodical evaluation of our leader election algorithm based on simulations in a mobile, wireless setting. Our simulation study provides useful insights relating to the design of our algorithm and the signaling used. As we will see, we exploit these insights to develop a very efficient leader election algorithm.

Our proposed algorithm uses the concept of diffusing computations [10] 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 primary 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 significantly improve 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 rest of the paper is organized as follows. In Section II, we discuss related work. Section III describes our model assumptions and objectives. In Section IV, we describe our election algorithm. Simulation setting and the performance metrics are described in Section V. Section VI describes the lessons learned during algorithm design. In Section VII, we formally specify the various correctness properties of our algorithm. In Section VIII, we discuss some simulation results and conclude in Section IX.

II. 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 [14], [22]. 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 [8], [16], [13], [26], [1]. 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 [6], [3], [24] have been proposed for wired networks that assume process crashes and link failures and are therefore closely related to our work. However, in [3], [24] process failures are assumed to occur before election starts while in [6] the election algorithms make strong assumptions such as 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 [11] 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 for a shared memory model have been proposed in [4], [2], [12]. These algorithms assume a shared-memory model and are not suitable for a message-passing system such as an ad hoc network. In [25] a spanning tree algorithm for a message-passing system is proposed and is based on the algorithm in [4]. However, the algorithms in [4], [25] require nodes to know an a priori upper bound on the number of nodes in the network to detect illegal states. Also, the stabilization time is proportional to this bound. In a mobile, ad hoc network, with frequent partitions and merges such information will not be usually available.

Leader election algorithms for mobile ad hoc networks have been proposed in [20], [15]. As noted earlier, we are interested in an extrema-finding algorithm, because for the applications discussed in Section I, it is desirable to elect a leader with some system-related attributes such as maximum battery life or maximum computation power. The algorithms in [20] are not extrema-finding and cannot be extended to perform extrema-finding. Although, extrema-finding leader election algorithms for mobile ad hoc networks have been proposed in [15], 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. [18], [5]), but these algorithms elect clusterheads only within their single hop neighborhood.

The main contributions of this paper are thus a design

of an efficient asynchronous, leader election algorithm that can adapt to arbitrary topological changes as seen in a mobile, ad hoc network. 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 existing work on leader election is that we present a careful and systematic simulation-based study of our election algorithm in a mobile, wireless environment. We present several interesting insights culled from our experiences in simulating these algorithms in mobile environments and exploit them to design a highly efficient leader election algorithm. Our simulations show that the algorithm works very well, with each node having a leader for 97%-99% of time. 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.

III. 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) **Node Value:** Each node has a **value** associated with it. The value of a node indicates its "desirability" as a leader of the network and can be any performance-related attribute such as the node's battery power, computational capabilities etc.
2) **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.
3) **Links:** Links are bidirectional and FIFO, i.e. messages are delivered in order over a link between two neighbors.
4) **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.
5) **Node-to-Node Communications:** A message delivery is guaranteed only when the sender and the receiver remain connected (not partitioned) for the entire duration of message transfer.
6) **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.

IV. Leader Election Algorithm

Our leader election algorithm is based on the classical termination-detection algorithm for diffusing computations by Dijkstra and Scholten [10]. 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.

A. 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).

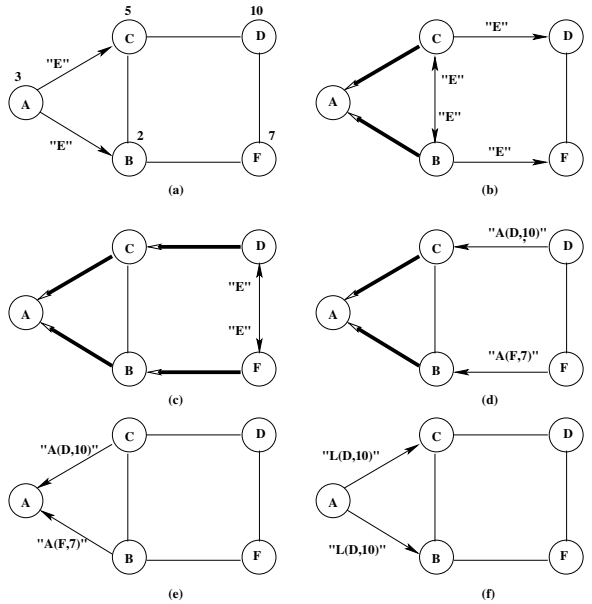


Fig. 1. An execution of leader election algorithm based on Dijkstra-Scholten termination detection algorithm. Thin arrows indicate direction of flow of messages while the thick arrows represent the constructed spanning tree.

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 I
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 II
LIST OF VARIABLES MAINTAINED BY A NODE *i* DURING THE ELECTION PROCESS.

B. 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 the 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.

1) **Variables and Message types:** The message types and variables used in the algorithm are shown in Table I and Table II respectively. The algorithm involves five message types: *Election*, *Ack*, *Leader*, *Probe* and *Reply*. The first three message types were described in Section IV-A. 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 IV-B.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.

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 [27].

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 2(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 2(b), and propagates the received *Election* message to nodes *B* and *C*.

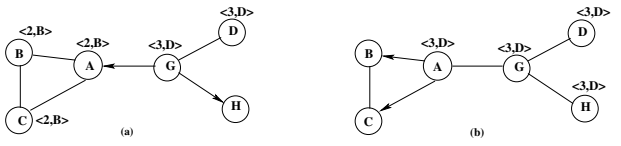


Fig. 2. Handling concurrent diffusing computations

C. 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 IV-A, 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 IV-A, *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 IV-A, 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.

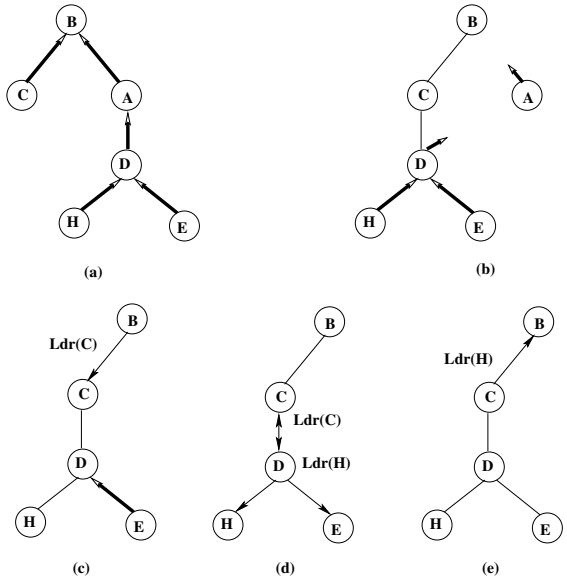


Fig. 3. Operation of Leader Election Algorithm in the face of partitions

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 3. Figure 3(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 3(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 3, 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 3(c).

When a node disconnects from its parent, it can no longer report an *Ack* message to its parent. Hence, it terminates 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 3(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 4(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.

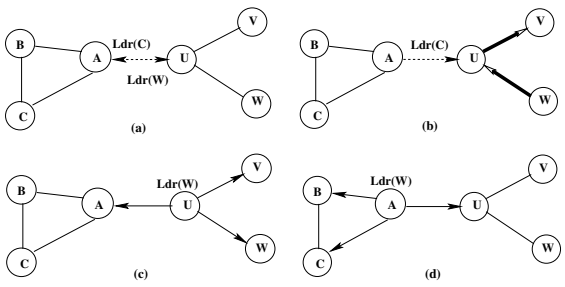


Fig. 4. Operation of Leader Election Algorithm in the face of merges

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 4(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 4(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 IV-B.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.

V. Simulation Setting

We simulate our algorithm using GloMoSim [30], 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.

A. 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 3. In that example, nodes *B*, *C* and *H* all participate in the same diffusing computation, which is initiated by *B* and they all have identical *computation-indexes*. 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.

B. Simulation Environment

Nodes are randomly placed in a 2000m $\times$ 2000m obstacle-free 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 [29], 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 [23]. We would like to point out that we also simulated our algorithm with DSR [17] as the routing protocol and observed very little change in the results shown in this paper. We refer the interested reader to [27].

In our simulations, a leader node periodically broadcasts *Beacon* messages to all 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.

VI. 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.

In Figure 5, 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.

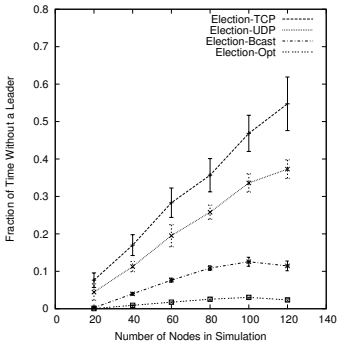


Fig. 5. "Evolution" of Leader Election Algorithm

From Figure 5, 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 5 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 5, 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 5, 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.

1) **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.
2) **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 6.

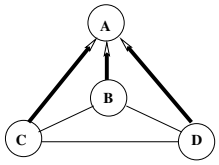


Fig. 6. An example network and the corresponding spanning tree

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* 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 6, 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. 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 5, 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 does not join the election and suppresses forwarding of the *Election* message. But it adopts a new leader if the newly elected leader has higher identity than its current leader.

Example:

Consider the scenario in Figure 7, 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.

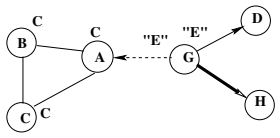


Fig. 7. Optimization : Avoiding unnecessary elections

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 of our *Election-Opt* algorithm using temporal logic and subsequently study, using simulations, its performance in a wide variety of operating conditions.

VII. Formal Verification of Algorithm

Using linear-time temporal logic [21], we formally verify the correctness of the *Election-Opt* algorithm described in the previous section. Due to space limitations, we do not present the proofs here. Detailed proofs are available in [27].

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'), 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.

In [27], we show that starting in a designated initial state, the system is guaranteed to reach a state satisfying predicate *P* after an arbitrary (but finite) 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 correctness of our election algorithm involves establishing the *Safety Property* and *Progress Property* described below:

- *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:

  $$G \ \Rightarrow \ \Diamond \ P \tag{1}$$

- *Progress Property:* This property states that eventually predicate *G* holds true i.e., there are no more diffusing computations in the network and all diffusing computations stop. Formally, we prove that:

  $$\Diamond \ G \tag{2}$$

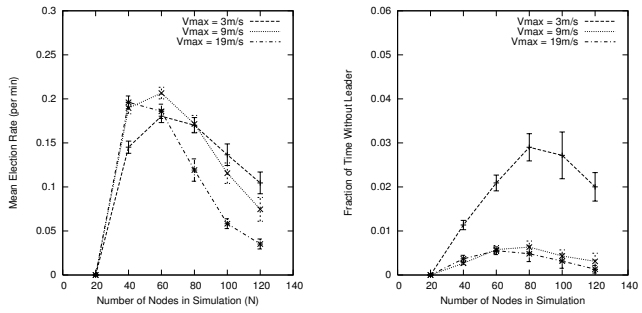
The *Safety* and *Progress* properties together ensure that the system eventually converges to a desired stable state specified by predicate *P*. Together, the *Safety* and *Liveness* properties guarantee that our algorithm also satisfies the *Termination Property* which states that eventually none of the program actions in our algorithm are enabled.

VIII. Sensitivity Analysis : Results and Discussion

A. Election-Rate and Fraction of Time Without Leader

We first study the impact of node mobility and transmission range of nodes on the *Election-Rate* (*R*) and *Fraction of Time Without Leader* (*F*). We run each of our simulations for a duration of 400 minutes while discarding the data obtained from the first 150 minutes (corresponding to the 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 8, show the *Election-Rate* and *Fraction of Time Without Leader* for three different values of V_{max} viz. 3m/s, 9m/s, 19m/s.



(a) Mean Election Rate (b) Fraction (F)

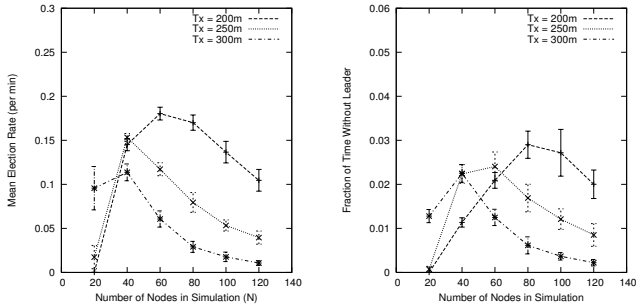
Fig. 8. Performance Vs V_{max}. Here $T_x = 200m$, $V_{min} = 1m/s$ and $P_t = 10s$.

The first conclusion we draw based on 8(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* is small (e.g., $N = 20$), most of the nodes can be expected to be isolated (i.e. not connected to any other node) and remain so for long durations. 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, the high node density means that components remain connected for longer durations and, hence, *Election Rate* drops. The second observation from Figure 8(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 [7] that higher speeds 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 re-connected to their leaders, thereby avoiding a fresh election.

From Figure 8(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 due to the increase in *Election-Rate* and also due to the fact that elections can be expected to be longer when there are more nodes. However, for $N \geq 100$, longer election durations are counterbalanced by a sharp decrease in *Election-Rate* and this accounts for the slight drop in *F*. Also, with an increase in V_{max}, *F* drops still further because of the decrease in *Election-Rate* as described earlier. As seen from Figure 8(b), *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 on our algorithm's performance. We observed that pause times had very little impact on *F* and *Election-Rate*. We refer the interested reader to [27].

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 *F* for three different choices of T_x, viz. 200m, 250m and 300m. From Figure 9(a), we see that



(a) Mean Election Rate (b) Fraction (F)

Fig. 9. Performance Vs T_x. Here $V_{max} = 3m/s$, $V_{min} = 1m/s$ and $P_t = 10s$.

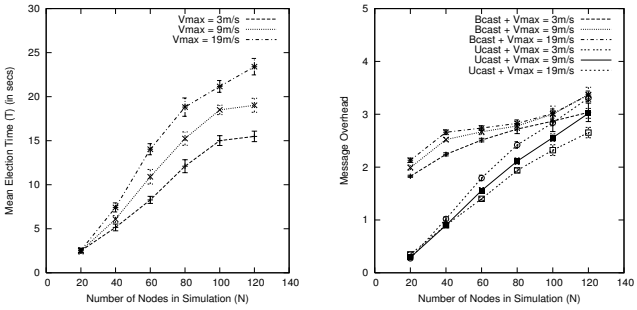
an increased transmission range of nodes leads to a higher *Election-Rate* when *N* is small (i.e. $N = 20$). Intuitively, this is because for a large value of T_x, there are fewer isolated nodes, but each component still consists of only a small number of nodes. For large values of *N* ($N \geq 60$), the *Election-Rate* becomes smaller with an 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 9(b), we see that *F* increases with T_x for small values of *N*, but for large values of *N*, it decreases with T_x because of corresponding decrease in *Election-Rate*. Again, we see that *F* is very low: always less than 0.03 and almost 0 when $T_x = 300m$ and $N = 120$.

B. Election-Time and Message-Overhead

We observed in Section VIII-A that the *Election-Rate* is very low (almost 0) in some scenarios 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 VIII-A, 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), as shown in Figure 10. The pause time is fixed at 10 seconds for all node speeds.

We first observe from Figure 10(a) that, irrespective of V_{max}, *Election-Time* increases with *N*. This is intuitive since



(a) Mean Election Time (b) Message Overhead

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

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** as the node-speed increases. 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 AODV routing overhead (in terms of number of control packets) and unicast message delays. From Figure 10(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 10(b). Broadcast message overhead and unicast message overhead are shown separately. Recall from Section VI 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, each node sends at least 2 broadcast messages, viz. one *Election* message upon joining the election and one *Leader* message upon termination. From Figure 10(b), we see that when $N = 20$, (i.e., there are many isolated nodes), each node 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 each computation it joins. The broadcast *Message-Overhead* shows very little difference with an 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 an increase in unicast overhead. However, it is evident from the graphs that this increase is very small. We thus conclude from Figure 10(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.

We also studied the impact of pause times (P_t) on the

performance metrics. We observed that pause times have little effect on both *Election-Time* and *Message-Overhead*. Due to space limitations we do not present the results here and refer the interested reader to [27].

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*. Due to space limitations, we do not present the results here. But we refer the interested reader to [27].

Our study shows that for a given *N*, *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 channel bandwidth, leading to greater message delays. The *Message-Overhead* was again observed to be fairly small. However, it showed a slight increase with an increase in T_x. This is because, as components grow larger in size and fewer in number, the number of concurrent elections triggered on leader departure increases, leading to a higher message overhead (both unicast and broadcast messages) as explained earlier.

IX. 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 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 III, 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.

References

[1] A. Amis, R. Prakash, T. Vuong, and D.T. Huynh. MaxMin D-Cluster Formation in Wireless Ad Hoc Networks. In *Proc. of IEEE INFOCOM*, March 1999.

[2] 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.

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

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

[5] 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*, April 2001.

[6] 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.

[7] T. Chu and I. Nikolaidis. On the Artifacts of Random Waypoint Simulations. In *Proc. of 1st International Workshop on Wired/Wireless Internet Communications (WWIC 2002)*, 2002.

[8] 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.

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

[10] 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.

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

[12] S. Dolev, A. Israeli and S. Moran. Uniform dynamic self-stabilizing leader election part 1: Complete graph protocols. In *LNCS 579*, 1993.

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

[14] 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.

[15] K. Hatzis, G. Pentaris, P. Spirakis, V. Tampakas and R. Tan. Fundamental Control Algorithms in Mobile Networks. In *Proc. of 11th ACM SPAA*, pages 251-260, March 1999.

[16] W. Heinzelman,A. Chandrakasan and H. Balakrishnan. Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In *Proc. of HICSS*, 2000.

[17] 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.

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

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

[20] 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.

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

[22] 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.

[23] C. Perkins and E. Royer. Ad-hoc On-Demand Distance Vector Routing. In *Proc. of the 2nd IEEE WMCSA*, New Orleans, LA, February 1999, pp. 90-100.

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

[25] G. Varghese. Self-Stabilization by local checking and correction. Ph.D. Thesis, LCS Technical Report, 1992.

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

[27] S. Vasudevan, J. Kurose and D. Towsley. Design and Analysis of a Leader Election Algorithm for Mobile, Ad Hoc Networks. UMass CMPSCI Technical Report 03-20.

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

[29] J. Yoon, M. Liu and B. Noble. Random Waypoint Considered Harmful In *Proc. of IEEE INFOCOM*, 2003.

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