Leader Election Algorithms for Mobile Ad Hoc Networks*

Navneet Malpani
Dept. of Computer Science
Texas A&M University
College Station, TX
77843-3112
n0m4119@cs.tamu.edu

Jennifer L. Welch
Dept. of Computer Science
Texas A&M University
College Station, TX
77843-3112
welch@cs.tamu.edu

Nitin Vaidya
Dept. of Computer Science
Texas A&M University
College Station, TX
77843-3112
vaidya@cs.tamu.edu

ABSTRACT

We present two new leader election algorithms for mobile ad hoc networks. The algorithms ensure that eventually each connected component of the topology graph has exactly one leader. The algorithms are based on a routing algorithm called TORA [5], which in turn is based on an algorithm by Gafni and Bertsekas [3]. The algorithms require nodes to communicate with only their current neighbors, making it well suited to the ad hoc environment. The first algorithm is for a single topology change and is provided with a proof of correctness. The second algorithm tolerates multiple concurrent topology changes.

1. INTRODUCTION

A mobile ad hoc network is a network wherein a pair of nodes communicates by sending messages either over a direct wireless link, or over a sequence of wireless links including one or more intermediate nodes. Only pairs of nodes that lie within one another's transmission radius can directly communicate with each other. Wireless link "failures" occur when previously communicating nodes move such that they are no longer within transmission range of each other. Likewise, wireless link "formation" occurs when nodes that were too far separated to communicate move such that they are within transmission range of each other. Developing distributed algorithms for ad hoc networks is a very challenging task since the topology may change very frequently and unpredictably. In this paper, we present a *mobility aware* leader election algorithm.

Leader election is a useful building block in distributed systems, whether wired or wireless, especially when failures can occur. For example, if a node failure causes the token to be lost in a mutual exclusion algorithm, then the other nodes can elect a new leader to hold a replacement token. Leader election can also be used in group communication protocols, to choose a new coordinator when the group membership changes. The standard definition of the leader election problem for static networks [1] is that

1. eventually there is a leader and
2. there should never be more than one leader.

However, complications arise because partitions can occur in an ad hoc network. In such a case, some applications require that every component of the partition must have a unique leader. Thus we consider a modified definition of the leader election problem: *Any component whose topology is static sufficiently long will eventually have exactly one leader.* There could be a period when a component has no leader, occurring when a component is partitioned. However, the algorithm must guarantee that exactly one unique node will be elected as a leader in the new component that was separated from the old leader. Similarly, there could be a period when there are two or more leaders, caused by two or more components merging. But the algorithm must guarantee that only one unique leader survives.

We present two leader election algorithms based on TORA [5], which is a routing algorithm for mobile ad hoc networks. TORA in turn is based on a loop-free routing algorithm of Gafni and Bertsekas [3]. In the algorithms in [3], each node keeps a value, called its height, from a totally ordered set (typically, a tuple of integers), and links are logically considered to be directed from higher to lower heights. The heights are manipulated when topology changes occur in such a way that the graph converges to a directed acyclic graph (DAG) in which the destination is the only sink (node with no outgoing links). the resulting DAG is called *destination-oriented.* TORA adds a clever mechanism to detect network partitions such that nodes that no longer have a path to a particular destination learn this fact and cease sending useless messages.

Our leader election algorithms modify these ideas in the following ways.

1. Instead of having a single destination-oriented DAG, we ensure that each component eventually forms a leader-oriented DAG.
2. When a partition from the current leader is detected (using the TORA mechanism), a new leader is elected and its id is propagated throughout the component.

*Supported in part by NSF grant CCR-9972235.

3. When two components merge, a contest takes place between the leaders so that the winner's id is propagated and wipes out the loser's id.

4. When multiple topology changes occur, additional complications arise. This is due to the fact that while a new leader's id is being propagated changes could occur in the component and the process of electing a leader may be repeated.

Although the leader election problem does not specifically require any sort of DAG structure to be imposed on components, our algorithms do so, as a byproduct of being based on a routing algorithm.

We believe that the proof of correctness of our first algorithm, under the assumption that only one topology change occurs at a time, aids in understanding not only our algorithm, but also TORA [5]. It should also provide a solid basis for proving correctness in more complex situations.

The next section discusses related work. In section 3, we describe our system assumptions and define the problem in more detail. Brief reviews of the GB algorithms and TORA are presented in sections 4.1 and 4.2. We present our leader election algorithm for a single topology change in sections 4.3 through 4.5. We sketch a proof of correctness of this algorithm under simplifying assumptions in section 4.6. Section 5 presents modifications to the first algorithm for multiple concurrent link failures and formations. Section 6 presents our conclusions.

2. RELATED WORK

Leader election algorithms for mobile ad hoc networks are presented in [4]. Compared to these algorithms, our algorithm is simpler and more practical. The algorithms in [4] are classified into *Non-Compulsory* protocols, which do not affect the motion of the nodes, and *Compulsory* protocols, which determine the motion of some or all the nodes. In both the protocol classes, it is assumed that the mobile nodes move in a bounded three-dimensional space S, where S is quantized by some regular polyhedron. In order for these algorithms to work, the mobile nodes should know in advance the type and dimensions of the polyhedron that is used for the quantization of S; furthermore, the nodes must be able to measure the distance that they cover when they move. All this adds to the complexity of the algorithm. Also, the *Non-Compulsory* protocols might never elect a unique leader and the *Compulsory* protocols force the nodes to perform a random walk. Neither of the protocol classes addresses the issue of creation of new components due to partitioning and merging of components

In our algorithm, the space S is not bounded and the nodes need not keep track of their physical location in S. The algorithm also does not impose any form of restricted motion on the nodes. As stated before, the algorithm is capable of handling formation of new components as well as merging of two or more components. The algorithm will eventually always elect a unique leader for each component.

The multicast operation of the Ad-hoc On-Demand Distance Vector (AODV) routing protocol [6, 7] performs leader election to elect a new multicast group leader when a partition occurs. After the multicast tree becomes disconnected due to a network partition, there are two group leaders. If the components reconnect, the multicast operation of the AODV protocol ensures that only one of the group leaders eventually becomes the leader of the reconnected tree. Thus we see that the problem definition for leader election in [6, 7] is quite similar to our problem definition. However, the approach that our algorithms take to solve this problem is very different.

The dynamic network model, which describes wired networks whose links are subject to frequent failures and recoveries, bears some important similarities to the mobile ad hoc network model. Algorithms have been devised for the dynamic network model to maintain a rooted spanning tree (e.g., [2]). These algorithms can be viewed as maintaining a leader (the root), but unlike our algorithm, which imposes a DAG structure on the topology, they impose a spanning tree structure on the topology. These algorithms do not handle partitions as well.

3. DEFINITIONS

3.1 System Model and Assumptions

The system contains a set of n independent mobile nodes, communicating by message passing over a wireless network. The network is modeled as a dynamically changing, not necessarily connected, undirected graph, with nodes as vertices and edges between vertices corresponding to nodes that can communicate. Assumptions on the mobile nodes and network are:

1. The nodes have unique node identifiers.

2. Communication links are bidirectional, reliable and FIFO. Unidirectional links, if any, are not used and ignored.

3. A link-level protocol ensures that each node is aware of the set of nodes with which it can currently directly communicate by providing indications of link formations and failures.

4. For the algorithm that we present in section 4, we assume that only one change (either a link failure or a link formation) can occur at a time. The next change occurs only after the entire network has recovered from the previous change. (The algorithm to handle multiple changes occurring concurrently is presented in section 5.)

3.2 Problem Statement

Each node i in the system must have a local variable lid_i that holds the identifier of the node currently considered to be the leader of i's component.

We require that in every execution with a finite number of topology changes, eventually it holds that:

- For every connected component C of the topology graph, there is a node l in C such that $lid_i = l$ for all nodes i in C.

An additional requirement, which might be useful in some applications, and is satisfied by our algorithm, is that each edge has a direction imposed on it by the endpoints such that eventually (after all the topology changes)

- Each connected component is a directed acyclic graph with the leader as the single sink (called a *leader-oriented or l-oriented* DAG).

The assumption in the precise problem statement that there is only a finite number of changes is technically convenient. However, it is equivalent to the more informal, and more practical, assumption that topology changes cease "sufficiently" long.

4. LEADER ELECTION ALGORITHM FOR A SINGLE TOPOLOGY CHANGE

Our algorithm is a modification of the TORA [5] routing algorithm, which in turn is based on a routing algorithm by Gafni and Bertsekas (GB) [3]. In this section we first provide informal descriptions of the GB algorithm, then TORA, and finally our algorithm. Detailed pseudocode of our algorithm is presented and some examples of algorithm operation. Finally, a proof of correctness is given.

4.1 Overview of the GB Algorithm

Gafni and Bertsekas [3] describe two algorithms for constructing a *destination-oriented DAG* in a network subject to link failures. Both algorithms work by assigning a unique *height* to each node, which is drawn from a totally ordered set; each link between two nodes is considered to be directed from the node with the higher height to the node with the lower height. The goal is for the directions on the links to form a DAG in which the destination is the only sink. To achieve the goal, whenever a node that is not the sink loses all its outgoing links, either because of a failure or because of a change in a neighbor's height, it calculates a new height for itself.

The two algorithms differ in the rule for calculating a new height. Both algorithms are special cases of a generic algorithm described in [3]. A correctness proof for the generic algorithm is given, which is quite abstract.

We now describe the *partial reversal* algorithm in [3], upon which both TORA and our algorithm are based. The height of a node i is a triple (α_i, β_i, i) of integers; the last component is the node's id in order to assure uniqueness. Triples are compared lexicographically. If i loses all its outgoing links, it chooses its new height to be (α'_i, β'_i, i), where α'_i is one larger than than the smallest α component among all its neighbors' heights. If i has a neighbor whose α height component is equal to α'_i, then β'_i is set to be one less than the smallest β value among all neighbors of i whose α height component equals α'_i. Otherwise the β component of i's height is unchanged.

The rule for setting α'_i ensures that node i will have at least one outgoing link, i.e., that (α'_i, β'_i, i) will be larger than the height of at least one neighbor, the one with the smallest height. The rule for setting β'_i tries to limit the number of links incident on i that will have their direction reversed, by keeping i's height smaller than that of any neighbors whose α height component is not smaller than α'_i. Reducing the number of links whose direction changes limits the propagation of height changes.

4.2 Overview of TORA

Park and Corson [5] adapted the GB algorithm for routing in mobile ad hoc networks, calling the result TORA (for Temporally Ordered Routing Algorithm). Their biggest addition was a mechanism for detecting when a piece of the network has been partitioned so that the destination is no longer reachable. The original GB algorithms would cause an infinite cycle of messages in that case. No correctness proof of TORA is given; instead an appeal is made to the generic proof in [3].

In TORA, the height of node i is a 5-tuple, $(\tau_i, oid_i, r_i, \delta_i, i)$. As before, the last component is the node's id, in order to ensure uniqueness.

The first three components form a *reference level*. A new reference level is started by node i if it loses its last outgoing link due to a link failure. τ_i is set to the time when this event occurs[1] and oid_i is set to i, the *originator* of this reference level. The third component r_i modifies the reference level. Initially, it is equal to 0, the *unreflected* reference level. As we explain shortly, sometimes it can be changed to 1, indicating a *reflected* reference level, which is instrumental in detecting partitions.

The δ_i components, together with the tie-breaking node ids, induce the directions on the links among all the nodes with the same reference level so as to help form a destination-oriented DAG. The originator of a new reference level sets its δ value to 0.

When a new reference level is created, say by node i, it is larger than any pre-existing reference level, since it is based on the current time. The originator notifies its neighbors of its new height. As we prove below in section 4.6 in the context of our leader election algorithm, this change eventually propagates among all nodes for whom i was on their only path to the destination. These are the nodes that must either form new paths to the destination or discover that, due to partitioning, there is none.

A node i can lose all its outgoing links due to a neighbor's height change under a number of different circumstances, which are now explained.

- If the neighbors of i do not all have the same reference level, then i sets its reference level to the largest among all its neighbors and sets its δ to one less than the minimum δ value among all neighbors with the largest reference level (a partial reversal).
- If all of i's neighbors do have the same reference level and it is an unreflected one, then i starts a reflection of this reference level by setting its reference level to

[1] See [5] for a detailed discussion concerning mechanisms for measuring time and their impact on the algorithm.

the reflected version of its neighbors' (with $r_i = 1$) and its δ to 0.

- If all of i's neighbors have the same reflected reference level with i as the originator, then i has detected a partition and takes appropriate action.
- If all of i's neighbors have the same reflected reference level with an originator other than i, then i starts a new reference level. This situation only happens if a link fails while the system is recovering from an earlier link failure.

4.3 Overview of Leader Election Algorithm

We made the following changes to TORA.

The height of each node i in our algorithm is a 6-tuple, $(lid_i, \tau_i, oid_i, r_i, \delta_i, i)$. The first component is the id of a node believed to the leader of i's component. The remaining five components are the same as in TORA.

The reference level $(-1, -1, -1)$ is used by the leader of a component to ensure that it is a sink.

In TORA, once a partition has been detected, the node that first detected the partition sends out indications to the other nodes in its component so that they cease performing height changes and sending useless messages. In our algorithm, the node that detected the partition elects itself as the leader of the new component. It then transmits this information to its neighbors, who in turn propagate this information to their neighbors and so on. Eventually all the nodes in the new component will become aware of the change in leader. When two or more components meet due to the formation of new links, the leader of the component whose id is the smallest will eventually become the sole leader of the entire new component.

4.4 The Algorithm

Here we describe the code executed by node i. Each step is triggered either by the notification of the failure or formation of an incident link or by the receipt of a message from a neighbor. Node i stores its neighbors' ids in local variable N_i. When an incident link fails, i updates N_i. When an incident link forms, i updates N_i and sends an Update message over the link with its current height.

The only kind of message sent is an Update message, which contains the sender's height. Immediately upon receipt of an Update message, i updates a local data structure that keeps track of the current height reported for each of its neighbors. Node i uses this information to determine the direction of its incident links. References in the pseudocode below to variables $lid_j, \tau_j, oid_j, r_j$, and δ_j for a neighbor j of i actually refer to the information that i has stored about j's height, in variable $height_i[j]$.

At the end of each step, if i's height has changed, then it sends an Update message with the new height to all its neighbors.

The pseudocode below explains how and when node i's height is changed. Parts B through D are executed only if the leader id in the received Update message is the *same* as lid_i.

A. When node i has no outgoing links due to a link failure:
1. if node i has no incoming links as well then
2. $\quad lid_i := i$
3. $\quad (\tau_i, oid_i, r_i) := (-1, -1, -1)$
4. $\quad \delta_i := 0$
5. else
6. $\quad (\tau_i, oid_i, r_i) := (t, i, 0)$ // t is the current time
7. $\quad \delta_i := 0$

B. When node i has no outgoing links due to a link reversal following reception of an Update message and the reference levels (τ_j, oid_j, r_j) are not equal for all $j \in N_i$:
1. $(\tau_i, oid_i, r_i) := \max\{(\tau_j, oid_j, r_j) | j \in N_i\}$
2. $\delta_i := \min\{\delta_j | j \in N_i \text{ and } (\tau_j, oid_j, r_j) = (\tau_i, oid_i, r_i)\} - 1$

C. When node i has no outgoing links due to a link reversal following reception of an Update message and the reference levels (τ_j, oid_j, r_j) are equal with $r_j = 0$ for all $j \in N_i$:
1. $(\tau_i, oid_i, r_i) := (\tau_j, oid_j, 1)$ for any $j \in N_i$
2. $\delta_i := 0$

D. When node i has no outgoing links due to a link reversal following reception of an Update message and the reference levels (τ_j, oid_j, r_j) are equal with $r_j = 1$ for all $j \in N_i$ and $oid_j = i$:
1. $lid_i := i$
2. $(\tau_i, oid_i, r_i) := (-1, -1, -1)$
3. $\delta_i := 0$

E. When node i receives an Update message from neighboring node j such that $lid_j \neq lid_i$:
1. if $lid_i > lid_j$ or ($oid_i = lid_j$ and $r_i = 1$) then
2. $\quad lid_i := lid_j$
3. $\quad (\tau_i, oid_i, r_i) := (0, 0, 0)$
4. $\quad \delta_i := \delta_j + 1$

In part E, if the new id is smaller than yours, then adopt it. If the new id is larger than yours, then adopt it, but only if it is the case that the originator of a new reference level has detected a partition and elected itself.

4.5 Examples of Algorithm Operation

The example in figure 1 shows the working of the algorithm under 2 conditions:

1. When a node detects a partition, it declares itself as the leader of the new component and propagates the information to the other nodes in the new component.
2. When two components meet due to the formation of a new link, the leader of one of the components which has the lower identification number eventually becomes the sole leader of the new component.

The respective heights are shown adjacent to each node (recall that the last tuple entry is the node's id). Lexicographical ordering (where $0 < 1 < 2...$ and $A < B < C...$) is used

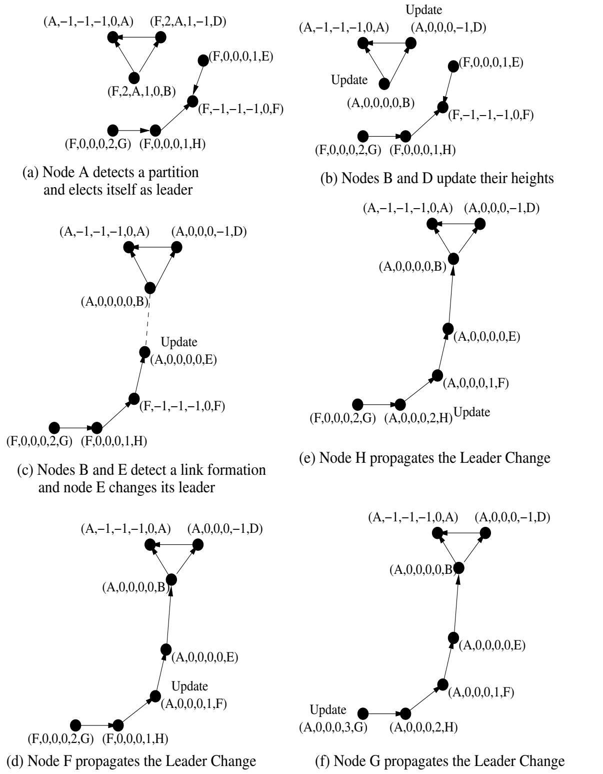

Figure 1: Operation of the Leader Election Algorithm (last element of tuple is node id)

to direct links. In figure 1(a), node A detects a partition and declares itself as the leader of the new component. Figure 1(b) shows the propagation of the message about the new leader to the other nodes in the new component. Figure 1(c)-(f) depicts the situation when two components meet due to a new link formation. Node A, which is the leader of one of the components, eventually becomes the sole leader of the entire component, since $A < F$, F being the leader of the other component.

4.6 Correctness

We assume that each connected component is a leader-oriented DAG originally and that only one change (either a link failure or a link formation) can occur at a time. The next change occurs only after the entire network has recovered from the previous change. We also assume that the system is synchronous, i.e., the execution occurs in lock step rounds. Messages are sent at the beginning of each round and are received by the nodes to whom they were sent before the end of each round.

THEOREM 1. *The algorithm ensures that each component eventually has exactly one unique leader.*

PROOF. We consider the following three cases (the remaining cases cause no changes):

Case 1: A link disappears at time t, causing node i to lose its last outgoing link but not disconnecting the component.

Case 2: A link appears at time t, joining two formerly separate components.

Case 3: A link disappears at time t, causing node i to lose its last outgoing link and disconnecting the component.

In each case we show that eventually each component in the resulting graph is a leader-oriented DAG.

Case 1: A link disappears at time t, causing node i to lose its last outgoing link but not disconnecting the component.

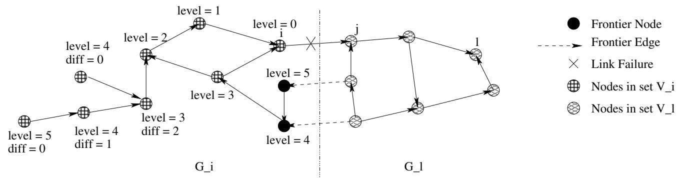

Figure 2: Example for Case 1.

Let G be the directed graph representing the resulting topology (of the component). Let l be the leader of the component. Then the component was an l-oriented DAG before the link was lost. Let V_l be the set of nodes that still have a path to l. At time t, the remaining nodes have a path to i; let this set be V_i. Let G_l be the graph induced by V_l and G_i be the graph induced by V_i. See Figure 2.

DEFINITION 1. *The* frontier nodes *of V_i are nodes that are adjacent to nodes in V_l; the edges between V_i and V_l are the* frontier edges.

Let k be any node in V_i.

DEFINITION 2. level(k) *is the length of the* **longest** *path in G_i from k to i.*

Note that level is defined with respect to the fixed G_i. Even though the direction of edges changes as the algorithm executes, the levels do not change.

LEMMA 1. *If k is on a path in G_i from a frontier node to i, then k's final height is $(l, t, i, 0, -level(k), k)$. Otherwise, k's final height is $(l, t, i, 1, -diff(k), k)$, where $diff(k) = \max\{level(h) | h \in V_i$ and k is reachable from h in $G_i\} - level(k)$.*

PROOF. We will show by induction on the number of rounds r after t that at the end of round r:

(a) If $r < level(k)$, then k's height is the same as it was at time t.

(b) If k is on a path from a frontier node to i and $r \geq level(k)$, then k's height is $(l, t, i, 0, -level(k), k)$.

(c) If k is not on a path from a frontier node to i and $level(k) \leq r < level(k) + 2 \cdot diff(k)$, then k's height is $(l, t, i, 0, -level(k), k)$.

(d) If k is not on a path from a frontier node to i and $r \geq level(k) + 2 \cdot diff(k)$, then k's height is $(l, t, i, 1, -diff(k), k)$.

Basis: $r = 0$. At the end of round t, clearly property (a) holds.

Induction: Assume the statement is true at the end of round $r-1$. We will show it is true at the end of round r.

(a) Suppose $r < level(k)$.

Then k originally has an outgoing link to a node h whose level is at least r. At the end of round $r-1$, by induction, h still has its original height, as does k, so the edge between k and h is still directed toward h. During round r, h might receive a message causing it to change its height, but even if this happens, h's Update message is not received by k until round $r+1$. So at the end of round r, k still has its original height.

(b) Suppose k is on a path from a frontier node to i and $r \geq level(k)$.

When $r = level(k)$, all outgoing neighbors of k have $level < r$. By induction, by the beginning of round r, all outgoing neighbors h of k will have reported their new heights $(l, t, i, 0, -level(h), h)$ to k, causing k to raise its height to $(l, t, i, 0, -level(k), k)$.

When $r > level(k)$, the height of k will not change since when $r = level(h)$, where h is a frontier node, the direction of the frontier edge will change and a path from i to l will be established and no messages will be reflected back.

(c) Suppose k is not on a path from a frontier node to i and $level(k) \leq r < level(k) + 2 \cdot diff(k)$.

When $r = level(k)$, k gets the last Update message from a (formerly) outgoing neighbor and thus k loses its last outgoing link. Then it raises its height to $(l, t, i, 0, -level(k), k)$. This causes all the links that used to come into k to go out from k. (If k is a frontier node, that would include reversing the frontier edges, as seen in the previous case).

When $r > level(k)$ but still less than $level(k) + 2 \cdot diff(k)$, there is no change in the level of k since node k has at least one outgoing edge and it has not received any Update messages from its outgoing neighbors.

Actually, there is a case when $level(k) = level(k) + 2 \cdot diff(k)$, when $diff(k) = 0$. In this case, no nodes of V_i initially were incoming to k. Node k, on receiving the last Update message from a (formerly) outgoing neighbor, loses its last outgoing link. It now raises its height to $(l, t, i, 1, 0, k)$. In essence, node k starts the reflection.

When $diff(k) > 0$, since $r < level(k) + 2 \cdot diff(k)$, there has not been enough time for k to receive reflected messages from all its neighbors. Thus, the height of k remains unchanged.

(d) Suppose k is not on a path from a frontier node to i and $r \geq level(k) + 2 \cdot diff(k)$.

When $r = level(k) + 2 \cdot diff(k)$, k gets the last reflected message from its neighbors and updates its height to $(l, t, i, 1, -diff(k), k)$. When $r > level(k) + 2 \cdot diff(k)$, the height of k remains unchanged since a path from i to l has been or will be established as shown in case 2.

□

Thus, Lemma 1 implies that the resulting graph is an l-oriented DAG, since all nodes in G_i now have paths to frontier nodes. The frontier edges are now directed from V_i to V_l because the τ-component in the heights of nodes in V_i is larger than for V_l (since the algorithm has access to synchronized or at least logical clocks).

Case 2: A link appears at time t, joining two formerly separate components C_1 and C_2 into component C.

Let l_1 be the leader of C_1 and l_2 the leader of C_2. Assume without loss of generality that $l_1 < l_2$. Suppose a link appears at time t between k_1, a node in C_1, and k_2, a node in C_2.

Lemma 2. *Eventually l_1 becomes the leader of component C and C is an l_1-oriented DAG.*

Proof. Let r be the number of rounds after t.

At $r = 0$, k_1 and k_2 send Update messages to each other. Since k_1's leader l_1 is smaller than k_2's leader l_2, k_2 updates its height to $(l_1, 0, 0, 0, \delta_{k_1} + 1, k_2)$ and obtains an outgoing link to k_1.

Let the value of $dist(k)$ for any node k in $partition_2$ be the **shortest** path distance from that node to node k_2 (the path distance is in terms of number of links between them).

When $r < dist(k)$, the height of k remains unchanged since it has not yet received the Update message regarding the change in leadership. When $r = dist(k)$, k (including l_2) changes its height to $(l_1, 0, 0, 0, \delta_{k_1} + dist(k) + 1, k)$. Thus k now has a route to k_1 and its leader id has also changed to indicate a change in leadership. When $r > dist(k)$, the height of k remains unchanged.

Thus we see that when $r = dist(k)$, such that k is the farthest node from k_2, all the nodes in $partition_2$ have updated their heights and have a route to k_1. The resultant graph (for the merged component) will be an l_1-oriented DAG, since k_1 is a node in $partition_1$ which is an l_1-oriented DAG. □

Case 3: A link disappears at time t, causing node i to lose its last outgoing link and disconnecting the component.

The proof for case 3 is very similar to case 1, except that there will be no path from node i to a frontier node. The following condition will arise which is different from the conditions in case 1.

Let r_1 be equal to $\max\{level(k)+2\cdot\ \mathit{diff}(k)\}$ for all k adjacent to i. At round r_1, the heights of all the adjacent nodes k will be $(l, t, i, 1, -\mathit{diff}(k), k)$ and node i will detect that a partition has occurred and will elect itself as the leader.

LEMMA 3. *At round r_1 a DAG with node i as the sink has already been formed.*

PROOF. We know from the proof of case 1 that, when $r > level(k) + 2\cdot\ \mathit{diff}(k)$ for any node k other than i, node k has changed its height to $(l, t, i, 1, -\mathit{diff}(k), k)$ and has no outgoing link towards node i. This height of k will not change when $r > level(k) + 2\cdot\ \mathit{diff}(k)$ and $r < r_1$. Also when $r = r_1 - 1$, one of the nodes k which is adjacent to i will change its height to $(l, t, i, 1, -\mathit{diff}(k), k)$ and have on outgoing link to node i. This node k will also be the last adjacent node of i to do so. $\square$

Thus at r_1, when node i detects the partition, it changes its height to $(i, -1, -1, -1, 0, i)$ and sends an Update message to its neighbors. This message is propagated throughout the new component. The resulting graph is an i-oriented DAG. The proof for this is the same as the proof for Lemma 2.

Thus we see from all the three cases that our algorithm will eventually ensure that each component has exactly one unique leader.

5. LEADER ELECTION ALGORITHM FOR CONCURRENT CHANGES

In this section we describe modifications to our algorithm from Section 4 to handle concurrent topological changes. By concurrent topological changes, we mean that after a change (link failure or link formation) occurs, another change occurs before the network has finished recovering from the previous change. Case E is replaced by the code given below and a new case (F) is introduced.

E. When node i receives an Update message from neighboring node j such that $lid_j \neq lid_i$:
1. if $lid_i > lid_j$ or ($oid_i = lid_j$ and $r_i = 1$) then
2. $\quad lid_i := lid_j$
3. $\quad$ if $lid_j = j$ then
4. $\quad\quad (\tau_i, oid_i, r_i) := (0, 0, 0)$
5. $\quad$ else
6. $\quad\quad (\tau_i, oid_i, r_i) := (\tau_j, oid_j, r_j)$
7. $\quad \delta_i := \delta_j + 1$
8. else if $lid_i < lid_j$ and this is **not** first Update received from j since the link formed then
// "precompute" j's height; j will be incoming to i
9. $\quad$ if $lid_i = i$ then
10. $\quad\quad height_i[j] := (lid_i, 0, 0, 0, \delta_i + 1, i)$
11. $\quad$ else
12. $\quad\quad height_i[j] = (lid_i, \tau_i, oid_i, r_i, \delta_i + 1, i)$
// remind j about i's height
14. $\quad$ send Update message containing i's height to j
// check whether i should start a new reference level

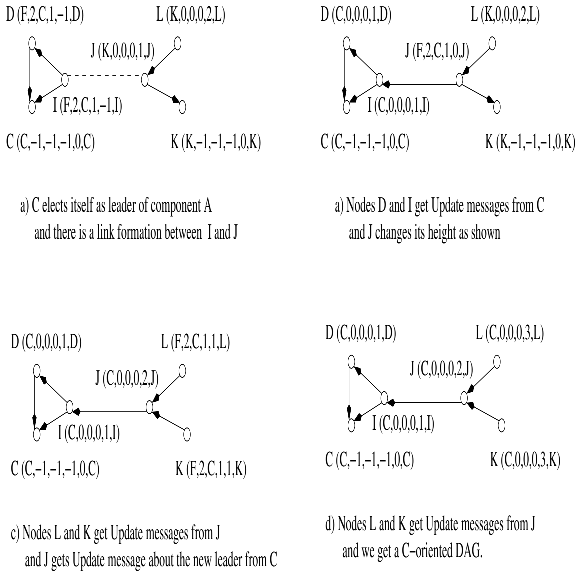

Figure 3: Example 1 for the second algorithm

15. $\quad$ if i has no outgoing links and $lid_i \neq i$ and $lid_k = lid_i$ for all $k \in N_i$ then
16. $\quad\quad (\tau_i, oid_i, r_i) := (t, i, 0)$ // t is current time
17. $\quad\quad \delta_i := 0$

F. When node i has no outgoing links due to a link reversal following reception of an Update message and the reference levels (τ_j, oid_j, r_j) are equal with $r_j = 1$ for all $j \in N_i$ and $oid_j \neq i$:
1. $lid_i := i$
2. $(\tau_i, oid_i, r_i) := (t, i, 0)$ // t is current time
3. $\delta_i := 0$

Lines 3-6 in case E are required to handle the case when two components, say A and B, meet. An example for this is shown in figure 3. Assume node i of component A has not yet received the Update message about the new leader of component A, but has a reflected reference level. Node i receives an Update message from node j of component B. If $lid_j > lid_i$ and $lid_j \neq j$, then the reference level of j is updated to the reference level of i. This enables node i to propagate the Update message about the new leader of component A throughout component B.

A scenario when lines 8-17 in case E are required is as follows (see figure 4). Say a partition has been detected and the leader of the new component starts propagating the information about its leadership throughout the new component. Assume that node k loses its last outgoing link due to a link failure before it receives the Update message about the leadership change. Node k will then define its own reference level at time t and its leader id will be the leader id of the previous component it belonged to. Say node k now receives the leadership change message. Now if the leader id of k is smaller than the id of the new leader, node k will ignore the Update message. This will lead to a situation where node

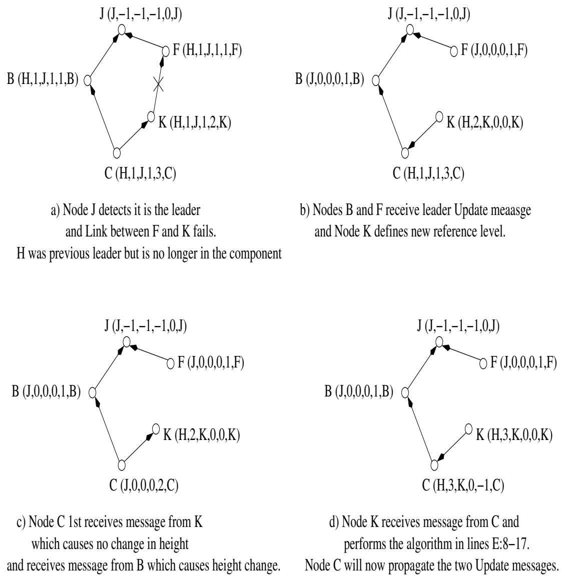

Figure 4: Example 2 for the second algorithm

k will be unaware of the new leader in the component, but will still be part of the component.

6. DISCUSSION

We have proposed two distributed and highly adaptive leader election algorithms, based on TORA [5], designed for operation in ad hoc networks. Both leader election algorithms guarantee that every connected component in the network will eventually have a unique leader. The first algorithm works when only a single topological change occurs. A proof of correctness is provided for this algorithm, which also provides insight into the workings of the TORA algorithm. The second algorithm handles multiple concurrent topological changes.

The initialization of our algorithms can be achieved by starting each node as the leader of its own component, i.e., each node i starts with its height to $(i, -1, -1, -1, 0, i)$ and is neighbor list N_i to empty.

For our algorithms to be tolerant to node failures, we assume that when a node recovers from a node failure, it restarts by declaring itself as the leader, i.e, setting its height to $(i, -1, -1, -1, 0, i)$.

Our future work will concentrate on simulating the algorithm and evaluating its performance. We also plan to provide the proof of correctness for the case when multiple concurrent topological changes occur.

Clearly, other algorithms can be conceived for leader election in mobile ad hoc networks. Different algorithms are expected to differ in the ease of implementation, message complexity, space usage, etc. Comparison of different algorithms is a topic for further work.

Acknowledgements

We thank Charles Perkins and Jennifer Walter for helpful discussions.

7. REFERENCES

[1] H. Attiya and J. L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* London, UK: McGraw-Hill, 1998.

[2] Chunhsiang Cheng and Srikanta P. R. Kumar. A Loop-Free Spanning-Tree Protocol in Dynamic Topology. *Proc. 27th Annual Allerton Conference on Communication, Control and Computing*, Sept. 1989, pp. 594-595.

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

[4] Kostas P. Hatzis, George P. Pentaris, Paul G. Spirakis, Vasilis T. Tampakas and Richard B. Tan. Fundamental Control Algorithms in Mobile Networks. *Proc. 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 251-260, 1999.

[5] Vincent D. Park and M. Scott Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. *Proc. IEEE INFOCOM*, April 7-11, 1997.

[6] Elizabeth M. Royer and Charles E. Perkins. Multicast Operations of the Ad-hoc On-Demand Distance Vector Routing Protocol. *Proc. Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, pages 207-218, August 15-20, 1999.

[7] Elizabeth M. Royer, Samir R. Das and Charles E. Perkins. Ad Hoc On-Demand Distance Vector (AODV) Routing (Internet-Draft). *Mobile Ad Hoc Network (MANET) Working Group*, 10 March, 2000 (work in progress).