TWO ALGORITHMS FOR LEADER ELECTION AND NETWORK SIZE ESTIMATION IN MOBILE AD HOC NETWORKS

A Thesis

by

NICHOLAS GERARD NEUMANN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2004

Major Subject: Computer Science

TWO ALGORITHMS FOR LEADER ELECTION AND NETWORK SIZE ESTIMATION IN MOBILE AD HOC NETWORKS

A Thesis

by

NICHOLAS GERARD NEUMANN

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

Jennifer Welch (Chair of Committee)	David Larson (Member)
Andreas Klappenecker (Member)	Valerie Taylor (Head of Department)

December 2004

Major Subject: Computer Science

ABSTRACT

Two Algorithms for Leader Election and Network Size
Estimation in Mobile Ad Hoc Networks. (December 2004)
Nicholas Gerard Neumann, B.S., Texas A&M University
Chair of Advisory Committee: Dr. Jennifer Welch

We develop two algorithms for important problems in mobile ad hoc networks (MANETs). A MANET is a collection of mobile processors ("nodes") which communicate via message passing over wireless links. Each node can communicate directly with other nodes within a specified transmission radius; other communication is accomplished via message relay. Communication links may go up and down in a MANET (as nodes move toward or away from each other); thus, the MANET can consist of multiple connected components, and connected components can split and merge over time.

We first present a deterministic leader election algorithm for asynchronous MANETs along with a correctness proof for it. Our work involves substantial modifications of an existing algorithm and its proof, and we adapt the existing algorithm to the asynchronous environment. Our algorithm's running time and message complexity compare favorably with existing algorithms for leader election in MANETs.

Second, many algorithms for MANETs require or can benefit from knowledge about the size of the network in terms of the number of processors. As such, we present an algorithm to approximately determine the size of a MANET. While the algorithm's approximations of network size are only rough ones, the algorithm has the important qualities of requiring little communication overhead and being tolerant of link failures.

ACKNOWLEDGMENTS

I would like to thank Jennifer Welch and Evelyn Pierce for their insightful comments and mentoring throughout the development of the leader election portion of this work. I would also like to thank Andreas Klappenecker for his suggestions and comments on early versions of the network size estimation portion of this work.

This work was supported in part by an NSF Graduate Research Fellowship, NSF Grant 0098305, and Texas Higher Education Coordinating Board Grant ARP-00512-0091-2001.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

A. Mobile ad hoc networks (MANETs)

Problems such as leader election, mutual exclusion, and consensus, have been extensively studied in many different traditional parallel and distributed computing models. Many of the problems that have been extensively studied in parallel or distributed computing models have natural analogs in a relatively new model: the mobile ad hoc network (MANET).

A mobile ad hoc network (MANET) is a collection of mobile processors ("nodes") which communicate via message passing over wireless links. Each node can communicate directly with other nodes within a specified transmission radius; other communication is accomplished via message relay. For our work, we assume all nodes have identical transmission radii; this immediately gives that communication links in our MANETs will be undirected.

In a MANET, communication links may go up and down (as nodes move toward or away from each other); thus, the MANET can consist of multiple connected components, and connected components can split and merge over time. The ease with which link changes may occur in a MANET can make developing algorithms for MANETs significantly more involved than developing algorithms for similar problems in more traditional distributed computing models.

A good deal of research has been done on solving the mobile ad hoc analogs of distributed computing problems such as leader election, mutual exclusion, and

---

The journal model is *IEEE Transactions on Automatic Control*.

consensus. However, these problems have not been studied as extensively as their corresponding versions in more traditional parallel or distributed models. Rather, much of the work on MANETs has focused on lower-level issues, such as routing and media access control (MAC). As such, the study of the higher-level problems in MANETs is still an active research area with several opportunities for contribution.

B. Leader election

Leader election is an extensively studied problem in distributed systems, and is a useful building block in such systems, especially in environments (like mobile ad hoc networks) where failures may occur. Leader election can be used to assist with mutual exclusion, group communication, and other standard problems in distributed systems. Additionally, solving leader election in MANETs is useful for other reasons, such as assisting in complex motion planning algorithms ([6]) and serving as a building block for creating particular tree communication structures ([1]).

The traditional definition of leader election in static distributed systems requires that the processors in the system eventually elect a unique leader. In a MANET, though, link changes are common and may cause the network to split into multiple connected components or cause some components to become separated from the leader. Additionally, two connected components, each with its own leader, may merge. Thus, the definition of the leader election problem has to be adapted to the mobile ad hoc environment. As in [17], we define the leader election problem in MANETs as the problem of guaranteeing that "*any component of the mobile ad hoc network whose topology is static sufficently long will eventually have exactly one leader.*"

C. Network size estimation

Many algorithms to solve traditional distributed computing problems in the MANET environment either require some sort of *a priori* knowledge about the number of nodes in the MANET. For example, [10] develops a group communication algorithm and uses knowledge about the number of nodes in a MANET to determine how long a processor should wait for a token used by the algorithm to return.

[15] describes additional circumstances in which knowing an estimate of network size is useful. It mentions a routing algorithm in which knowing network size allows for "short-cutting" long distances, along with peer-to-peer network algorithms which need only the size of the network and local information to operate.

Additionally, many algorithms for MANETs use data structures whose sizes are functions of the number of nodes in the MANET. Such algorithms could benefit from having an estimate of network size by using the estimate to more efficiently allocate space for data structures.

Because of the relevance of network size to MANET algorithms, a mechanism to provide information about the size of the network to its nodes is of obvious interest.

CHAPTER II

DETERMINISTIC LEADER ELECTION IN ASYNCHRONOUS MANETS

A. Previous work

While the problem of leader election has not been studied as extensively in mobile ad hoc systems as in more traditional settings, several algorithms have been developed for the problem in various mobile ad hoc settings.

We should note that many of these algorithms were not explicitly developed for the MANET setting, but rather as self-stabilizing[1] leader election algorithms for dynamic asynchronous networks. In a dynamic network, links may go up and down arbitrarily, but nodes are not mobile. However, the models of dynamic networks and MANETs are similar enough that algorithms for leader election in dynamic networks can be applied to MANETs.

One such algorithm is presented in [9]. This algorithm, like many leader election algorithms, constructs a spanning tree and selects its root as leader. The algorithm employs randomization and uses shared memory (as opposed to the message passing model used by most of the other works reviewed in this section). Its convergence time (time from when the last link change or node failure/recovery occurs until every component has a unique leader) is bounded by $O(dD)$, where d is the maximal degree of a node in the MANET and D is the largest diameter attained by some connected

[1] A self-stabilizing algorithm is one which can tolerate link and node failures, as well as corruption of processor states or memory. Because a self-stabilizing algorithm can tolerate memory corruption, it also has the property that it can be run on a network started in an arbitrary state; as long as memory corruption and node/link failures eventually cease, the algorithm will still "work" and converge to a correct solution.

component of the MANET during the algorithm's execution.

In [2], the ideas of [9] are extended to improve the convergence time of the leader election algorithm to $O(D)$. A deterministic version of the algorithm is also developed, but has the drawback that variables may grow arbitrarily long as links go up and down and nodes leave and join the MANET.

The paper [5], which deals primarily with the maintenance of replicated information, also develops an algorithm to maintain spanning trees. This algorithm uses the message passing model and takes time $O(n^2)$ to converge to the spanning tree, where n is the number of nodes in the MANET. This algorithm was later improved in [16] to have convergence time $O(A \log^3 A)$, where A is (roughly) the number of nodes in the largest connected component of the MANET.[2]

Also, the algorithm in [16] merges trees to eventually obtain a spanning tree. In this algorithm though, a node u must send a request to its neighbor v to join its tree, and then this request must be sent to the root of the tree and the response sent back through the tree. Our algorithm uses a similar "tree merging" approach, but the decision of whether u can join v's tree is made locally by v, which yields faster recovery from many scenarios, including single link failures.

The algorithm of [17] maintains a directed acyclic graph with a single sink, which is the leader, and adapts the ideas of [13, 19] to the MANET environment. Unfortunately, [17] is only proved correct for a completely synchronous system with a single link change.

The algorithm of [1] develops the innovative idea of "power supply" to perform self-stabilizing leader election. Unlike other works on leader election in MANETs, this

[2] A may be substantially smaller than n in certain networks; A must actually be carefully defined since the number of nodes in a connected component changes over time in a MANET. For a formal definition of A, we refer the reader to [16].

work allows for unidirectional communication links between nodes. The algorithm has convergence time $O(n)$, along with an extremely involved proof of correctness. Also, the algorithm, unlike ours, requires nodes to periodically send certain messages to their neighbors. While our algorithm is not self-stabilizing, it has the advantage that it eventually stops sending messages (if the network topology stops changing).

Like several of those described above, the algorithm of Arora and Singhai ([3, 4]) is a rooted spanning tree maintenance algorithm for a dynamic network. This algorithm uses the idea of coloring a node green or red to indicate whether or not it has detected that its parent is red or no longer a neighbor; merging of trees by adding a tree edge between neighboring nodes is only allowed when neighboring nodes are both green. This coloring scheme allows for a relatively simple algorithm for tree merging. The algorithm only works for a completely synchronous system where every node has instantaneous read access to all of its neighbors' variables.

The algorithm we develop in this paper is also a rooted spanning tree maintenance algorithm. It is deterministic and works in asynchronous MANETs, with convergence time $O(A)$. More details are given in the following sections.

A quick summary of the features of the above algorithms is given in Figure 1.

B. Our leader election algorithm

Our algorithm adapts the strategy of Arora/Singhai to an asynchronous system. At any point during the algorithm's execution (even after link changes and node failures/recoveries occur), the graph formed by drawing all nodes along with directed edges from each node to its parent node, consists of a forest of rooted trees. We perform leader election by generating a single rooted tree for each connected component of the communication graph, and selecting the root of each such tree as the leader of

Algorithm	**Deterministic/ Randomized?**	**Synchronous/ Asynchronous**	**Self-stabilizing?**	**Convergence Time**	**Miscellaneous**
[9]	Randomized	Asynchronous	Yes	$O(dD)$	Uses shared memory model
[2]	Both	Asynchronous	Yes	$O(D)$	Variables can grow without bound
[5]	Deterministic	Asynchronous	No	$O(n^2)$	
[16]	Deterministic	Asynchronous	No	$O(A \log^3 A)$	Complicated tree merging
[17]	Deterministic	Synchronous	No	$O(D)$	Only proved correct for a single link change
[1]	Deterministic	Asynchronous	Yes	$O(n)$	Unidirectional links; never stops sending messages
[3, 4]	Deterministic	Synchronous	No	$O(n)$	Assumes immediate read access to neighbor's variables
Our work	Deterministic	Asynchronous	No	$O(A)$	

Fig. 1. Summary of leader election algorithms for MANETs

the component.

In addition to modifying [3, 4] to deal with asynchrony, we add some further modifications, in the vein of ideas from [13, 17, 19] to improve the time to converge to a single rooted tree for each component after a link change or node failure/recovery.

The basic strategy behind the algorithm is as follows. Each node has a parent variable, which is a pointer to another node. Consider a graph H with a vertex for each processor and a directed edge from vertex u to vertex v if v is the parent of u. Even when link changes and/or node failures/recoveries occur during the execution of the algorithm, we want the graph H to remain a forest of rooted trees. We maintain this forest of rooted trees, and once link changes and node failures/recoveries have stopped, the algorithm converges to a state where H consists of a single rooted tree for each connected component of the MANET. Once this stable state is reached, each node considers its leader to be the root of the tree in which it finds itself.

A color is assigned to each node from the set $\{red, green\}$. The color red indicates that the process has detected that its parent is red or no longer a neighbor. Green

nodes may petition green neighbors for adoption or may themselves adopt green neighbors.

Our algorithm for leader election, while inspired by [3, 4], is significantly different due to asynchrony. Additionally, while the proofs of correctness for both our algorithm and [3, 4] use the idea that an invariant is preserved during algorithm execution, our invariant and the proof of its invariance are notably different due to the asynchrony being dealt with and our modifications to improve our algorithm's convergence time.

We now give an outline of the rest of the paper. In section 2, we present necessary definitions. Section 3 contains a description of the algorithm. Section 4 contains correctness proofs for the algorithm. Section 5 contains performance analysis, and we conclude the paper in section 6. Detailed pseudocode for the algorithm is presented at the end of the paper.

C. Preliminaries

1. System Assumptions

We consider a completely asynchronous mobile ad hoc system. The system consists of mobile nodes, and each node has a (possibly changing) set of neighbors. Each node u has a unique integer identifier that is fixed throughout the node's lifetime. Also, every node has in-order reliable message delivery to its current set of neighbors. However, if a link fails while messages are in transit on it, then these messages are lost. Additionally, complete asynchrony in our system means that while a message sent between two nodes will be delivered if the link it is traveling on does not fail, the delay between the sending and receiving of the message can be arbitrarily long.

We assume that the communication channels are bidirectional. That is, for two nodes u and v, u is a neighbor of v iff v is a neighbor of u.

For our correctness proofs, we need to carefully formalize how u and v are notified of the failure of one of their bidirectional communication channels. We do this as follows. If u and v cease to be neighbors, then both u and v are informed via a *link-down*(u, v) message that their communication link has ceased to exist. Due to asynchrony, u may receive this message before or after v does.

If u and v become neighbors, then they are both informed via a *link-up*(u, v) message that a communication link has been established. However, we require that if the link between u and v has previously failed, then neither u nor v receives *link-up*(u, v) until both u and v have processed the *link-down*(u, v) message associated with this failure. (This sort of behavior of the link notification system could be achieved by requiring both u and v to communicate before agreeing to reestablish the link between each other as being "up.")

In addition, any given node may fail; we view such a failure as the failure of the links between the node and each of its neighbors. Node recovery is also allowed and requires that the recovering node initialize itself with a particular state (discussed later).

2. Definitions

During the execution of the algorithm, it is important that in following the chain of parent pointers from one node to the next, a cycle does not occur. With this in mind, we define what it means for the system's state to have property NC (no cycle).

Consider the directed graph $H = (V, E)$ where V is a set corresponding to the nodes of the system, and an edge $(u, v) \in E$ if and only if v is u's parent and v believes u is a neighbor. (It is possible, if v receives *link-down*(u, v) before u, that u has v as parent even though v no longer believes u is one of its children.) **If $H = (V, E)$ is a forest of rooted trees, then we say that the system's state has property**

NC.

We also define $G = (V, C)$ to be the *un*directed graph with an edge $(u, v) \in C$ if and only if u and v both believe they are neighbors of each other. That is, G, in a sense, is a rough snapshot of the processors' communication graph.

Next, we define a special constant ϕ which nodes can use as "parent" if the link to their current parent goes down. ϕ is special in that for any node j, ϕ is not considered to be an adjacent node by j. (That is, saying a node's parent is ϕ is a convenient way of saying that a node's parent is undefined because the node lost its link to its parent and has not yet recovered from this loss.) For technical convenience, we also say that ϕ's leader id is ∞.

We say that a node has an "invalid parent" if its parent is red or ϕ, or the link between it and its current parent is in the process of failing. (We say a link is in the process of failing if a *link-down* message is in transit between it and its parent (in either direction).)

Finally, we define an asynchronous round of algorithm execution in order to be able to quantify the convergence time of our algorithm. Given an execution of an algorithm by nodes in a MANET as a sequence of processors taking steps (e.g., $p_{i_1}, p_{i_2}, \ldots$), we define the first asynchronous round of the execution to be the smallest prefix of this sequence such that every processor in the MANET executes every step of its algorithm at least once in the prefix. We define the next asynchronous round as the first asynchronous round of the execution with its first asynchronous round removed.

D. The algorithm

1. Intuition and code description

The general idea behind the algorithm is that, even when link changes and/or node failures/recoveries occur, the graph H should remain a forest of rooted trees. The forest is maintained in such a way that (eventually) the root of each tree is the node in the tree with the largest identifier. Once link changes and node failures/recoveries have ceased, the algorithm converges to a state where H consists of a single rooted tree for each connected component of the communication graph G. Each node considers its leader id to be the root of the tree in which it is located in H. Thus, each connected component of G will eventually have the node with the largest identifier (henceforth, the largest node) as its leader.

There are two issues that must be addressed in maintaining parent pointers to accomplish leader election. First, there may be more than one rooted tree corresponding to some connected component of G. In this case, we need a mechanism with which we can merge trees. Second, a node's parent may fail, move away and cease to be a neighbor, or color itself red (since the parent may also have an issue with its own parent). To deal with this issue, we must devise a mechanism to correct the parent pointer. In [3, 4], if j was informed that it had an invalid parent, j would immediately color itself red and inform all of its neighbors of the change. Instead of directly adapting the mechanism proposed by [3, 4] to deal with this issue, we do the following, in the vein of part of the strategy in [13, 17, 19].

Every node j is given a distance field which is an estimate of its distance, through parent relations, to its current root. When a green node j learns that its parent is invalid, it first checks whether it has some neighbor k that has the same leader id and is currently "closer" (has smaller distance field) to this leader than j is. If such a k

exists, j will NOT color itself red, but instead try to set its parent to be k. If j does not succeed in finding a new parent, it will give up and color itself red.

When a red node j finds out that it no longer has any neighbors as children, it colors itself green, makes itself its own parent, and adopts its own id as its root value. The net effect of this is that when a node colors itself red, eventually all of its children either are adopted by a new parent or also color themselves red. Eventually, every descendant of j either is adopted by a new parent or colors itself red. When a red node finds out it no longer has any children, it will color itself green, make itself its own parent, and adopt its own id as its root value. Eventually, all descendants of j either adopt a new parent or become their own parent, and then j is able to recover as well. Once j recovers, it can again participate in tree merging.

The detailed pseudocode for the algorithm is presented in Figure 2. We now describe the steps of the algorithm depicted in Figure 2.

- **Steps 1-2:** Process j sends any updates and receives any messages. The $request.j$ and $response.j$ variables are used to communicate the fact that *join-request* or *join-response* messages have just been received, along with relevant parts of these messages, to later steps of the algorithm.

- **Steps 3-4:** Process j updates its $root.j$ and $dist.j$ variables if necessary. $root.j$ is the leader id of j (the root of j's tree), and $dist.j$ is an estimate of the distance from j to $root.j$ in the tree

1. Send values of all updated $col.j$,$root.j$,$par.j$, and $dist.j$ variables to neighbors

2. Receive messages $\{E_1, E_2, \ldots, E_m\}$

   for each message E_i **do**

   - **if** $E_i = \textit{link-down}(j,k)$ **then**
     - **if** $par.j = k$ **then** $par.j := \phi$
     - $adj.j := adj.j - \{k\}$
     - $kids.j := kids.j - \{k\}$
   - **if** $E_i = \textit{link-up}(j,k)$ **then**
     - $adj.j := adj.j \cup \{k\}$
   - **if** E_i is an update of k's local variables **then**
     - update each $prop_k.j$ accordingly
   - **if** E_i is an update of k's parent so that $par_k.j \neq j$ **then**
     - $kids.j := kids.j - \{k\}$
   - **if** $E_i = \textit{join-request}(k, color, root, dist, type)$ **then**
     - $request.j := request.j \cup \{(k, type)\}$
     - $color_k.j, root_k.j, dist_k.j := color, root, dist$
   - **if** $E_i = \textit{join-response}(k, color, root, dist, type, accept)$ **then**
     - $response.j := (k, type, accept)$
     - $color_k.j, root_k.j, dist_k.j := color, root, dist$

3. **if** $col.j = green \wedge par.j \in adj.j \wedge root_{par.j}.j > root.j \wedge \neg waiting.j$ **then**

   - $root.j := root_{par.j}.j$
   - $dist.j := dist_{par.j}.j + 1$

4. **if** $col.j = green \wedge col_{par.j}.j = green \wedge par.j \in adj.j \wedge dist.j > dist_{par.j}.j + 1 \wedge \neg waiting.j$ **then**

   - $dist.j := dist_{par.j}.j + 1$

5. **if** $col.j = green \wedge (par.j \notin adj.j \cup \{j\} \vee col_{par.j}.j = red) \wedge (|tried.j| = 0 \wedge \exists k : k \notin tried.j \wedge k \in adj.j \wedge col_k.j = green \wedge dist_k.j < dist.j \wedge root.j = root_k.j) \wedge \neg waiting.j$ **then**

   - Send to k: $\textit{join-request}(j, color.j, root.j, dist.j, 1)$
   - $tried.j := tried.j \cup \{k\}$
   - $response.j := (k, 0, 0)$
   - $waiting.j := true$

6. **if** $waiting.j \wedge (response.j = (k, type, answer) \wedge type = 1) \vee k \notin adj.j)$ **then**

   - $waiting.j := false$
   - $response.j = (0, 0, rejected)$
   - **if** $answer = accepted$ **then**
     - $root.j, par.j, dist.j = root_k.j, k, dist_k.j + 1$
     - $tried.j := \{\}$

Fig. 2. The algorithm code

7. **if** $col.j = green \wedge (par.j \notin adj.j \cup \{j\} \vee col_{par.j}.j = red) \wedge (|tried.j| > 0 \vee (\nexists k : k \notin tried.j \wedge k \in adj.j \wedge col_k.j = green \wedge dist_k.j < dist.j \wedge root.j = root_k.j)) \wedge \neg waiting.j$ **then**

   - $col.j := red$
   - $tried.j = \{\}$

8. **if** $col.j = red \wedge (\forall k : k \notin adj.j \vee k \notin kids.j) \wedge \neg waiting.j$ **then**

   - $col.j, root.j, par.j, dist.j := green, id.j, j, 0$

9. **if** $(col.j = green) \wedge (k \in adj.j) \wedge (col_k.j = green) \wedge (root.j < root_k.j) \wedge (\forall h \in adj.j : root_k.j \geq root_h.j) \wedge (\neg waiting.j)$ **then**

   - Send to k: $join\text{-}request(j, color.j, root.j, dist.j, 2)$
   - $response.j := (k, 0, 0)$
   - $waiting.j := true$
   - $tried.j := \{\}$

10. **if** $waiting.j \wedge (response.j = (k, type, answer) \wedge type = 2) \vee k \notin adj.j)$ **then**

    - $waiting.j := false$
    - $response.j = (0, 0, rejected)$
    - **if** $answer = accepted$ then
      - $root.j, par.j, dist.j := root_k.j, k, dist_k.j + 1$

11. **for** each $(k, type) \in request.j$ with $type = 1$ **do**

    - $request.j = request.j - \{(k, type)\}$
    - **if** $col.j = green \wedge root.j = root_k.j \wedge dist.j < dist_k.j$ **then**
      - Send to k: $join\text{-}response(j, color.j, root.j, dist.j, 1, accepted)$
      - $kids.j := kids.j \cup \{k\}$
    - **else**
      - Send to k: $join\text{-}response(j, color.j, root.j, dist.j, 1, rejected)$

12. **for** each $(k, type) \in request.j$ with $type = 2$ **do**

    - $request.j = request.j - \{(k, type)\}$
    - **if** $col.j = green \wedge root.j > root_k.j$ **then**
      - Send to k: $join\text{-}response(j, color.j, root.j, dist.j, 2, accepted)$
      - $kids.j := kids.j \cup \{k\}$
    - **else**
      - Send to k: $join\text{-}response(j, color.j, root.j, dist.j, 2, rejected)$

Fig. 2. Continued

- **Steps 5-7:** These steps help j resolve the issue of having an invalid parent. Step 5 is an attempt to recover from an invalid parent by petitioning a neighbor for adoption. The variable $tried.j$ is a set of neighbors j has already tried to have adopt it, and is used to limit the number of petitions for adoption j can issue to correct an invalid parent. (The pseudocode actually limits the number of petitions to 1, but this limit is easily adjusted.) $waiting.j$ ensures j has no more than one petition for adoption outstanding at a time. Step 6's body is performed by j when it receives a response to the request it issued in step 5 in a previous iteration of the algorithm. Step 7's body is executed by j when it finds out it has an invalid parent *and* it can no longer issue new parent petitions. In this case, the node simply colors itself red.

- **Step 8:** If process j is red and has no children, it disowns its parent and colors itself green.

- **Steps 9-10:** These steps help resolve the situation where there is more than one rooted parent relation tree corresponding to some connected component of G. Step 9 is an attempt to merge trees with different roots as described above. This step's body executes for process j whenever j is green and has a green neighbor k with larger leader id. Also, j must not have any other adoption petitions outstanding.

- **Steps 11-12:** These steps send responses to those nodes that have petitioned to become children of node j. Step 11 is the response to a petition sent in step 5 by another node k; the request is accepted by step 11 as long as j is green and really does have the same root and lower distance than k. Step 12 is the response to a petition sent in step 8 by another node j, and j accepts such a request from a node k if j is green and really does have higher root than k.

2. Variables and messages

a. Variables

Each process or node j has several variables, which are depicted in Figure 3.

Name	Description	Size	Updates sent
$adj.j$	the set of current neighbors of j, initially empty	$O(n \log n)$	No
$id.j$	the id of node j, fixed throughout j's execution	$O(\log n)$	No
$color.j$	the color of the processor j, taken from the set $\{red, green\}$; initially $green$	$O(1)$	Yes
$par.j$	the process that j currently considers to be its parent; initially j	$O(\log n)$	Yes
$root.j$	the id of the process j believes to be its leader; initially $id.j$	$O(\log n)$	Yes
$kids.j$	the set of neighbors of j which have j as parent; initially empty	$O(n \log n)$	No
$waiting.j$	a boolean indicating that j is waiting for a neighbor's response to a request to have the neighbor become its parent; initially $false$	$O(1)$	No
$dist.j$	the distance j believes itself to be from the process it considers to be its leader; initially 0	$O(\log n)$	Yes
$tried.j$	a set used to store those neighbors of j which have already been contacted in trying to get a new parent to replace its invalid parent; initially the empty set	$O(n \log n)$	No
$request.j$	a set of pairs corresponding to those nodes currently trying to get j to accept them as children; each pair is of the form $(k, type)$ and corresponds to node k trying to become a child of j through a request of type $type$; $type = 1$ if the merge request was issued by j in step 5, and $type = 2$ if the request was issued by j in step 9; initially the empty set	$O(n \log n)$	No
$response.j$	a triple corresponding to the response node j receives from a node it petitions to become a child of; the triple is of the form $(k, type, answer)$ where node j is petitioning for adoption by k through a request of type $type$; $answer = accepted$ if the request was accepted, and $answer = rejected$ if the request was rejected; initially $(0, 0, rejected)$	$O(\log n)$	No
$prop_k.j$	variables for each $k \in adj.j$, $prop \in \{color, par, root, dist\}$; represents node j's view of the relevant variable of neighbor k	$O(n \log n)$	No

Fig. 3. Processor variables

Node j sends out updates of $color.j$, $par.j$, $root.j$, and $dist.j$ to all nodes in $adj.j$ whenever these values change. Also, whenever a node k becomes a new neighbor of j, these values are sent to k. These are **the only** variable values that are exchanged in update messages, so an update message is of maximum size $O(\log n)$. For neighbors j and k, we denote by $prop_k.j$ j's most recent view of k's variable $prop.k$. For example, $col_k.j$ is the last value of $col.k$ received by j from k. This notation is designed to be consistent with local variable notation, i.e., to indicate that $prop_k.j$ is a variable

stored by j, just like $prop.j$. Finally, we require that when a node starts executing or recovers from a crash, it must initialize itself with the default variable values given in Figure 3.

b. Messages

Aside from the update messages and *link-up* and *link-down* messages, there are two additional messages that can be sent between neighbors. These are:

- *join-request*$(j, color, root, dist, type)$; this is the message sent by node j with $color := color.j$, $root := root.j$, $dist := dist.j$ and $type$ the type of join request being sent; $type = 1$ if the request is sent by j in step 5, and $type = 2$ if the request is sent by j in step 9.

- *join-response*$(j, color, root, dist, type, answer)$; this is the message sent by node j to a node k's *join-request* message; when sent, $color := color.j$, $root := root.j$, $dist := dist.j$, $type$ is the type of request being responded to, and $answer = accepted$ if the request was accepted, $answer = rejected$ otherwise.

Finally, we note that when a processor starts, it is not aware of any of its neighbors. It will be notified by *link-up* events of those processors which are within its communication radius.

E. Proof of correctness

We now show that the algorithm is correct; that is, that after the last link change or node failure/recovery has occurred, the algorithm will eventually reach a state where every up node is green, and the forest of rooted trees consists of a single tree for each connected component of G. Without loss of generality, once link changes and node

failures/recoveries have ceased, we can consider the algorithm's execution when G has a single connected component.

The strategy behind the proof is to specify a property T and show that if the algorithm starts in a state where T holds, then every step taken by the algorithm preserves T. It is then shown that the property NC is similarly preserved. It is important to note that we need invariance of T to prove preservation of NC, but NC is not required to show preservation of T. We also show that both T and NC are preserved by every node and link failure and recovery.

We then show that after the last link change or node failure/recovery, the algorithm progresses towards a state S where every up node is green and each connected component of G corresponds to a single tree in the parent relation forest H. Additionally, every node in a particular component will have the same *root* value. Thus, as long as our algorithm starts in a state where T and NC hold (which is true if all processors start with initial values given in the variables section), the algorithm will converge to S.

We begin by specifying T. T is similar to the T used in [3, 4]; one difference is that $col.par.j$ (the color of node $par.j$) is replaced with $col_{par.j}.j$, since sometimes a node j may believe its parent has a certain color when the node has actually changed its color but the message notifying j of the change has not yet reached j. Also, part (v) of T is added because of the modifications we made to improve convergence time. Additionally, [3, 4] includes NC in T. Asynchrony forces us to develop a substantially different proof that T and NC hold, and in our different proof strategy that our T does not include NC. Rather, we prove invariance of T without NC and then use the invariance of T to prove the invariance of NC.

Let us now present T.

Let $T \equiv (\forall j : j$ is up $\Rightarrow$

$$
\begin{array}{lll}
(col.j = red & \Rightarrow (par.j \notin adj.j \cup \{j\} \vee col_{par.j}.j = red)) \wedge & (i) \\
(par.j = j & \Rightarrow root.j = id.j) \wedge & (ii) \\
(par.j \neq j & \Rightarrow root.j > id.j) \wedge & (iii) \\
(par.j \in adj.j & \Rightarrow (root.j \leq root.par.j \vee & \\
 & \quad \text{link between } j, par.j \text{ has failed but only } par.j & \\
 & \quad \text{has processed } \textit{link-down}(j, par.j))) \wedge & (iv) \\
(par.j \in adj.j \wedge root.j = root.par.j & \Rightarrow (dist.j > dist.par.j \vee & \\
 & \quad \text{link between } j, par.j \text{ has failed but only } par.j & \\
 & \quad \text{has processed } \textit{link-down}(j, par.j)))) & (v)
\end{array}
$$

The ideas behind the conjuncts of T are as follows. The first conjunct guarantees that if a node is red, it is because it has an invalid parent. The second and third conjuncts are simple consistency conditions on what a node j can believe its leader to be. The fourth conjunct says that if $par.j$ is still considered by j to be j's neighbor, then the parent must have a leader at least as big as j's leader, unless $par.j$ no longer believes j to be a neighbor. This implies that following edges from node to parent in H yields monotonically nondecreasing leader values. Likewise, the fifth conjunct implies that following edges from node to parent in H will yield decreasing distance values as long as the leader values of the nodes do not change along the path.

1. Algorithm preserves T

We first prove that each step of the algorithm preserves T.

We proceed using a standard proof by induction on each step of the algorithm and each conjunct of T. We go through each conjunct and look at the effects the different steps of the algorithm can have on the specific conjunct. We show that if the conjunct of T holds before the step is executed, then it will also hold after the step is executed. By showing this is true for each conjunct and step, we have that if

T holds before any step, it will hold after any step. Note that steps 1, 5, 9, 11, and 12 have no effect on T so we need not consider them for any of the conjuncts.

Theorem 1. *T is preserved by each step of the algorithm.*

Before proceeding with the proof of this theorem, we need a few lemmas to help simplify some steps in the proof.

Lemma 1. *If a processor has $waiting.j = true$, the variables $col.j$, $root.j$, and $dist.j$ cannot be changed by the processor.*

Proof. The proof is by inspection of the detailed psuedocode in Figure 2. □

Lemma 2. *The only step of the algorithm that can decrease $root.j$ is step 8. Further, execution of the body of step 3 or step 10 with $answer = accepted$ increases $root.j$.*

Proof. The only steps of the algorithm that can change $root.j$ are steps 3, 6, 8, and 10. Step 3 clearly can only increase $root.j$.

For step 6, there are two important things to note. First, step 6 will only find $answer = accepted$ if k, in step 11, had $root_j.k = root.k$. Second, $root.j$ cannot change between j's execution of the body of step 5 and corresponding execution of step 6 (by Lemma 1, since $waiting.j = true$). Combining this yields that step 6, if it receives $answer = accepted$, will not change $root.j$.

For step 10, a similar argument as that for step 6 gives that execution of the body of step 10 with $answer = accepted$ will increase $root.j$. □

Lemma 3. *The variable $dist.j$ can only increase after a step of the algorithm executes if the variable $root.j$ also increases.*

Proof. The only steps of the algorithm that can change $dist.j$ are steps 3, 4, 6, 8, and 10. Step 3 increases $root.j$ so the lemma holds for step 3 trivially. Step 4 can

only decrease $dist.j$, so the lemma holds for step 4. Step 6 can only affect $dist.j$ if it receives an accept, in which case $dist_k.j < dist.j$ when step 6 executes, so $dist.j$ cannot increase. Step 8 sets $dist.j$ to 0 so it never increases $dist.j$. Step 10 will only affect $dist.j$ if it receives an accept, in which case $root.j$ is increased. So the lemma holds. □

Lemma 4. *$par.j \in adj.j$ unless $par.j = \phi$.*

Proof. This can be seen by simply examining the code. If $par.j$ is removed from $adj.j$ in step 2, then immediately before step 2 sets $par.j = \phi$. And when any step of the algorithm changes $par.j$, it is always changed to some processor in $adj.j$. □

Proof of Theorem 1. We know by 6 that no conjunct of T We go through each conjunct of T and show that no conjunct is violated by any step of the algorithm. More specifically, for the statements which are **for** or **if** statements (all but step 2), we need only consider what happens if the body of the **if** or **for** statement is executed.

- **Conjunct (i):** $col.j = red \Rightarrow (par.j \neq j \wedge (par.j \notin adj.j \vee col_{par.j}.j = red))$

  The only steps of the algorithm that could violate (i) are the steps of j.

  - **Step 2 by j** could only violate (i) if it caused the right side of the implication to be false. Since step 2 cannot result in $par.j = j$, it could only make the right side false by changing the values of $par.j$, $adj.j$, and/or $col_{par.j}.j$.

    If it sets $par.j$ (in the first bullet of the step), then it sets $par.j$ to ϕ and by definition $col_{par.j} = col_\phi = red$, so the conjunct is still satisfied.

    Step 2 cannot change the truth value of $par.j \in adj.j$ from false to true, since if it did we would have $par.j \neq \phi$ and $par.j \notin adj.j$ before step 2 executes, contradicting Lemma 4.

Finally, suppose step 2 changes $col_{par.j}.j$ from red to green. We will argue that this can never happen. By way of contradiction, suppose it does. Let k be the node that j has as $par.j$ before step 2 executes. At some point before step 2 executes, $col_k.j$ is *green*, since j can only set $par.j = k$ if in the same step it sets $col_k.j = green$. (This follows by inspecting the code.) Call the last point in time before step 2 when $col_k.j = green$ by α. At some point after α, $col_k.j$ is set to *red*. Call the time when this happens by β. $col_k.j$ is set to *red* because k sends j a message indicating that its color has changed to *red*. Call the time when k sends this message by γ. Now, in order for j to receive an update in step 2 to change $col_k.j$ to *green*, k must send an update of its color to j at some point after γ; call this time δ. Now, k is *red* at time γ and green at time δ. The only way k can change color from *red* to *green* is by executing step 8, which means that at some time τ between γ and δ, k had $j \notin kids.k$. Thus, a *link-down*(j, k) message must be sent to j between time γ and τ. So j receives this *link-down*(j, k) at some time after β (by FIFO message delivery). This causes j to set $par.j = \phi$ at some time ν after α but before step 2 executes. Now, j must have $par.j = k$ immediately before step 2 executes, but this necessitates that $col_k.j$ be set to *green* at some point between ν and execution of step 2. But $\nu > \alpha$, so this contradicts α being the last point in time before step 2's execution when $col_k.j = green$.

- **Step 4 by *j*** cannot violate (i).
- **Step 6 by *j*** can only set $col.j = green$. (This can be seen by examining the code of step 11, which sends the message from k that step 6 of j processes.) Setting $col.j = green$ causes (i) to be trivially satisfied.

- **Step 7 by j** can only set $col.j = red$ if the right side of (i) is true, since this is a precondition for the execution of step 7's body. Thus, (i) will not be violated.
- **Step 8 by j** can only affect (i) if it sets $col.j := green$, which trivially satisfies (i).
- **Step 10 by j** can only set $col.j = green$. (This can be seen by examining the code of step 12, which sends the message from k that step 10 of j processes.) Setting $col.j = green$ causes (i) to be trivially satisfied.

Thus, (i) will not be violated.

- **Conjuncts (ii) and (iii):** $par.j = j \Rightarrow root.j = id.j$ and $par.j \neq j \Rightarrow root.j > id.j$

  The only steps of the algorithm that can violate (ii) or (iii) are steps of node j.

  - **Step 2 by j** can only affect (ii) or (iii) by assigning $par.j := \phi$, which can only happen if $par.j \neq j$ previously. So (ii) and (iii) will both be preserved.
  - **Step 3 by j** can only increase a node's root value if the node is not its own parent, thus satisfying (iii) (and trivially satisfying (ii)).
  - **Step 4 by j** has no effect on (ii) or (iii).
  - **Step 6 by j** can only affect (ii) or (iii) if its body is executed. For the body to be executed, $par.j \neq j$ (since step 6 only occurs after the body of step 5 executes, and step 5 has the precondition that $par.j \neq j$ and sets $waiting.j = true$. $par.j$ cannot be set back to j by any step except step 8, which cannot execute while $waiting.j = true$.) Since $par.j \neq j$

immediately before step 6 executes, T guarantees that $root.j > id.j$ at this time. Step 6 never decreases $root.j$ (by Lemma 2), and can only change $par.j$ if $par.j \neq j$ before the step is executed. So (iii) will be preserved and (ii) will not be affected if the body of step 6 executes, and neither (ii) or (iii) will be affected if it does not.

- **Step 7 by j** has no effect on (ii) or (iii).
- **Step 8 by j**, if the **if,then** body executes, satisfies (ii) and trivially makes (iii) true (since $par.j = j$ afterwards).
- **Step 10 by j**, if the request to become a child is granted, increases the root of j (see Lemma 2), and since before step 10 was executed we had $T \Rightarrow root.j \geq id.j$, we then have $root.j > id.j$ if the request to become a child is granted. Thus, if the request is granted, $par.j \neq j \Rightarrow root.j > id.j$, and (ii) and (iii) will be satisfied. If the request is not granted, (ii) and (iii) are not affected.

Thus, (ii) and (iii) will not be violated.

- **Conjunct (iv):** $(par.j \in adj.j \Rightarrow root.j \leq root.par.j \vee$ link between $j, par.j$ has failed but only $par.j$ has received/processed *link-down*$(j, par.j))$

  The only steps of the algorithm that can violate (iv) are steps of j and $par.j$.

  - **Step 2 by j** can only affect (iv) by resulting in $par.j = \phi$, which makes (iv) trivially hold.
  - **Step 3 by j** could only violate (iv) if $root.par.j$ had decreased prior to 3's execution but j had not been informed of such a decrease. This could only happen if $par.j$ executed the body of step 8, which requires either $j \notin adj.par.j$ or $j \notin kids.par.j$, which can only happen if the link between $j, par.j$ has failed but only $par.j$ has received/processed *link-down*$(j, par.j)$.

Thus, if step 3 violates the first disjunct of (iv) the second disjunct of (iv) must be satisfied, so (iv) will not be violated.

- **Step 4 by j** has no effect on (iv).
- **Step 6 by j** could only violate (iv) if $root.par.j$ had decreased but j had not been informed of such a decrease. By the same argument as that for step 3, (iv) cannot be violated.
- **Step 7 by j** has no effect on (iv).
- **Step 8 by j**, if the body is executed, will yield $par.j = j$ so that (iv) is satisfied.
- **Step 10 by j** could only violate (iv) if $root.par.j$ had decreased but j had not been informed of such a decrease. By the same argument as that for step 3, (iv) cannot be violated.
- **Step 2 by $par.j$** can only affect (iv) by making the second disjunct of (iv) true.
- **Step 3 by $par.j$** serves only to increase $root.par.j$, so (iv) will not be violated.
- **Step 4 by $par.j$** has no effect on (iv).
- **Step 6 by $par.j$** does not change $root.par.j$ so (iv) is not violated.
- **Step 7 by $par.j$** has no effect on (iv).
- **Step 8 by $par.j$** can only have its body executed if $j \notin adj.par.j$ or $j \notin kids.par.j$, which can only happen if the link between $j, par.j$ has failed but only $par.j$ has received/processed $link\text{-}down(j, par.j)$. Thus, execution of the body of step 8 will occur only if the second conjunct of (iv) holds and thus (iv) will be satisfied after execution.

- **Step 10 by *par.j*** serves only to increase $root.par.j$, so (iv) will not be violated.

Thus, (iv) will not be violated.

- **Conjunct (v):** $(par.j \in adj.j \wedge root.j = root.par.j \Rightarrow dist.j > dist.par.j \vee$ link between $j, par.j$ has failed but only $par.j$ has received/processed *link-down*$(j, par.j))$

  The only steps that can violate (v) are the steps of j and $par.j$.

  - **Step 2 by *j*** can only affect (v) by resulting in $par.j = \phi$, which makes (v) trivially hold.
  - **Step 3 by *j*** cannot violate (v) unless $par.j$ has increased its distance and j has not processed the message from $par.j$ about the increased distance. By Lemma 3, the only way $par.j$ can increase its distance is by increasing its root. There are two possibilities:

    1. If $par.j$ increased its root to be $root.j$ before j executed step 3, then $root.par.j < root.j$ before 3 executed, which in combination with (iv) means the link between $j, par.j$ has failed but only $par.j$ has received/processed *link-down*$(j, par.j)$. Thus, (v) will hold when step 3 executes.
    2. If $par.j$ increased its root to be bigger than $root.j$ before j executed step 3, then $root.par.j > root.j$ at some point before 3 executes. If $par.j$ does not decrease its root between this point and immediately before 3 executes, then the $root.par.j > root.j$ immediately after 3 executes so (v) is satisfied. If $par.j$ does decrease its root, then it executes step 8, and $j \notin adj.par.j$ or $j \notin kids.par.j$ which again means the link between $j, par.j$ has failed but only $par.j$ has received/processed

link-down$(j, par.j)$. So (v) is again satisfied.

- **Step 4 by *j*** cannot violate (v) by the same argument made for step 3.
- **Step 6 by *j*** cannot violate (v) by the same argument made for step 3.
- **Step 7 by *j*** does not affect (v).
- **Step 8 by *j***, if the body of the **if,then** executes, yields $par.j = j$ so $par.j \notin adj.j$ and (v) is trivially satisfied.
- **Step 10 by *j*** cannot violate (v) by the same argument made for step 3.
- **Step 2 by *par.j*** can only affect (v) by making the second disjunct of (v) true.
- **Step 3 by *par.j*** could only violate (v) if it increased $root.par.j$ to be equal to $root.j$. But this would mean, before step 3 executed, that $root.j > root.par.j$. Thus, to violate (v), we would have to have $par.j \in adj.j$ when 3 executed, which combined with (iv) holding before 3 and (v) not holding after yields that the first conjunct of (iv) must hold before 3, or $root.j \leq root.par.j$. This contradicts $root.j > root.par.j$. So step 3 cannot violate (v).
- **Step 4 by *par.j*** serves only to decrease $dist.par.j$ so it will not violate (v).
- **Step 6 by *par.j*** cannot increase $dist.par.j$. This can be seen by combining the fact that step 6 does not change $root.par.j$ (Lemma 2) and the fact that $dist.par.j$ can only increase if $root.par.j$ increases (Lemma 3). Thus, step 6 will not violate (v).
- **Step 7 by *par.j*** does not affect (v).
- **Step 8 by *par.j*** cannot increase $dist.par.j$, so it will not violate (v).

– **Step 10 by *par.j*** could only violate (v) if it increased $root.par.j$ to be equal to $root.j$. By the same argument as for when $par.j$ executes step 3, this cannot happen, so step 10 cannot violate (v).

Thus, (v) will not be violated

We now have that each step of the algorithm preserves T.

□

2. Algorithm preserves NC

We next show that the algorithm preserves the no cycle property NC.

Lemma 5. *NC is preserved by the algorithm.*

Proof. Suppose to the contrary that NC is violated at some point during algorithm execution. Then some node j is the last node to execute step 6 or 10 and form the cycle in H, and does so by setting $par.j = k$ when there exists a chain of parent pointers in H from k to j. So there are two cases: either the cycle is formed by j's execution of step 6 or the cycle is formed by j's execution of step 10. We present the argument for step 10 first since it can be used in the argument for step 6:

- **Step 10:** By property (iv) of T, following the chain of parent pointers from k to j, we have $root.k \leq root.j$ immediately before j executes step 10. Before j executes step 10, k executes step 12, at which point $root.k > root.j$. So between k's execution of step 12 and j's execution of step 10, either $root.k$ decreases or $root.j$ increases. By Lemma 1, $root.j$ cannot increase during this time, since $waiting.j = true$. The only way $root.k$ can decrease after execution of step 12 is through k's execution of step 8, which can only occur if $j \notin adj.k$ or $j \notin kids.k$. But this happens only if *link-down*(j, k) has been processed by k and not j. But

then by the definition of H, even though j may have $par.j = k$, the edge (j, k) is not in H when step 10 completes execution, contradicting the assertion that it is the execution of step 10 by j that forms the cycle in H.

- **Step 6:** By property (iv), following the chain of parent pointers from k to j, we have $root.k \leq root.j$ immediately before 6 executes. Before 6 executed, k executed step 11, and then $root.k = root.j$.

  There are two possibilities for immediately before j executed step 6. Either $root.k = root.j$ or $root.k < root.j$. By argument similar to the proof for step 10, the case $root.k < root.j$ is impossible. So then $root.k = root.j$ immediately before 6 executes, and thus immediately after 6 executes (since 6 doesn't change $root.j$). Then $root.u = root.par.u$ for every node on the cycle incident on k and j right after 6 executes. By (v), $dist.j > dist.k$ but also $dist.k < dist.j$, the necessary contradiction.

□

3. Link and node failures and recoveries preserve T and NC

We next note that both T and NC are preserved whenever a link comes up or goes down, or a node fails or recovers. While the proof is fairly trivial, this fact is needed to guarantee that the MANET stays in a state where T and NC hold while the algorithm is executing, even if link and node failures and recoveries occur.

Lemma 6. *Failures and recoveries of nodes and links preserve T and NC.*

Proof. That link changes preserve T and NC follows immediately by the fact that the algorithm processes such link changes and we have already shown that the algorithm's actions do not violate T or NC.

The failure of a node j when T and NC hold can only fragment the graph of the parent relation. Thus, node failure preserves NC, and since T is trivially true for any node not up, node failure preserves T. Failures of links are handled by our algorithm, which we already showed preserves T and NC.

When a process j repairs itself, it sets its variables to the initial values listed in the previous section. Simple inspection reveals that this too does not violate T or NC. □

4. Convergence to a fixed point

The next step of the correctness proof is to show that, starting from a state in $T \cap NC$, the algorithm (if not interrupted by link changes and node failures/recoveries) eventually converges to a fixed point in which the graph of the parent relation yields a single rooted spanning tree for each connected component of G, every up node is colored green, and every up node in a given connected component of G has the same value for its root. The proof is structured as follows. We first show that eventually every up node is permanently colored green, and then show that once this happens, eventually (for each connected component of G) the algorithm will converge to the desired rooted spanning tree with every node having the largest id as root. Putting these facts together yields the correctness of our algorithm.

(Unlike our proofs that T and NC hold, this is fairly similar to the proof of convergence in [3, 4]. The main difference is the construction of the set F which actually allows two steps of the proof in [3, 4] to be combined into one.)

Lemma 7. *If the algorithm starts in a state in $T \cap NC$ and link changes and node failures/recoveries eventually stop, then eventually every up node is permanently colored green.*

Proof. Consider the set $F \equiv \{j : j \text{ is up} \wedge col.j = red \wedge par.j \notin adj.j \cup \{j\}\}$. At some point, link changes and node failures/recoveries will cease to occur. Once this happens, examination of the code reveals that after every *link-down* message has been processed and every node has completed its attempts to recover from any invalid parents caused by these *link-down* messages, the size of the set F can never increase. Now, after this point (once the size of F can't increase), suppose F is nonempty and thus there is some $j \in F$. We have that $col.j = red$ and $par.j \notin adj.j \cup \{j\}$.

Now, all of j's children eventually find out about j being colored red, and either color themselves red or disown their parent when they are adopted by a new parent. This propagation continues until nodes with no children discover that their parents are red. These nodes color themselves green and disown their parents, and then their parents color themselves green and disown their parents, and so on, until eventually node j no longer has any children and can color itself green. When this happens, j is no longer in F, and thus the size of F has decreased. Thus, eventually, F will have no members. Combining this with the fact that T holds, we will eventually have that if a node j is up and $col.j = red$, then $col_{par.j}.j = red$. Further, once F is empty, we know that for every node j, $par.j \neq \phi$, since if $par.j = \phi$ then $col.j = red$ and $par.j \notin adj.j \cup \{j\}$, or $j \in F$.

Now, if a node j believes its parent to be colored red, then its parent must truly be colored red, for $par.j$ cannot change its color from red to green until j disowns $par.j$ as its parent. Thus, we have that, eventually, if a node j is up and $col.j = red$, then $col.par.j = red$. Since $col.j = red \Rightarrow par.j \neq j$, we have that $par.k \neq k$ for any node k colored red.

Following this logic, once we have reached a state where F is empty, we know if a node is red, then it cannot be its own parent and its parent (a real node, not ϕ) is also red. If some node were red, then by following the chain of parent pointers, since

NC holds, we would eventually reach a node which was red but had no red node as a parent. From this contradiction, we can conclude that eventually we reach a state where every up node is colored green and remains green as long as no link changes or node failures/recoveries occur. □

Now that we know that eventually every up node is colored green, we can use this in proving convergence to the single rooted spanning tree.

Lemma 8. *If the algorithm starts in a state in $T \cap NC$, eventually (for each connected component of G) every up node has the up node with largest id in its connected component as its root and is a member of the tree rooted at this largest up node.*

Proof. By the previous lemma, if the algorithm starts in a state in $T \cap NC$, eventually all up nodes will be green. We must show that eventually every up node adopts the maximum value of $id.j$ among all up nodes as its root value, and that every up node becomes a member of the tree rooted at this largest up node.

Let us denote by m the node for which $id.m$ is maximum among all up nodes. Note that (ii) and (iv) together guarantee that $root.j \leq id.m$ for all up nodes j. Further, by simply following parent relations, we can get, using (ii), that if $root.j = id.m$, then m is an ancestor of j in the graph of the parent relation.

From these facts, once all nodes are green, we can see that all nodes $j \in adj.m$ will have $root.j = id.m$ after one asynchronous round (defined in Section 2). Further, all nodes adjacent to those nodes in $adj.m$ will have $root.j = id.m$ after two asynchronous rounds. A simple inductive argument gives us that eventually all up nodes j will have $root.j = id.m$ and thus will have m as an ancestor. Thus, we will eventually have reached the desired state where we have a single leader and every node acknowledges this leader as its leader. □

Putting these facts together we have the correctness of our algorithm.

Theorem 2. *Our algorithm solves the leader election problem in asynchronous MANETs.*

Proof. If T and NC hold in the MANET when the algorithm starts executing, then by preservation of T and NC and the previous two lemmas, the algorithm solves the leader election problem in asynchronous MANETs. Since the initial values specified for the nodes satisfy T and NC, the algorithm solves leader election in MANETs. □

F. Performance

1. Convergence time and message complexity

The analysis of the performance of our algorithm is fairly straightforward. First, we note that our algorithm converges to a single rooted spanning tree, and thus performs the task of leader election, in time $O(A)$, where A is the maximum size of a connected component of G throughout the algorithm's execution. (The formal definition of A given in [16] is more complex, but reduces to the above in our system model.)

Lemma 9. *The convergence time of the algorithm is $O(A)$.*

Proof. To see why this holds, we first argue that once a node becomes red, it takes at most $O(A)$ time for all of its children to disown it. Note that a node u must wait for all of its children to disown it before u can disown its own parent and color itself green, so that if u is red, it may have to wait for all of its children's descendants to disown their children as parents before it no longer has any children. Once all of a node's children have disowned it, the node can become green and make itself its own parent. Combining this with the fact that it takes a leaf node in a tree of H $O(1)$ time to color itself green, a simple inductive argument yields the $O(A)$ time bound. Thus, every node is green within $O(A)$ time of the algorithm's execution.

Now, once all nodes in a connected component are green, let us look at the node with the largest id. Call this node v. Within 1 round, all neighbors of v will have

v as leader. Within 2 rounds, all neighbors of neighbors of v that have not already taken on v as leader will take on one of $adj.v$ as a parent. Thus, within $O(D_{comp})$ rounds, where D_{comp} is the maximum diameter of the connected components of G, every node in v's connected component will have v as leader. So the total time to converge to the solution of the leader election problem is $O(A + D_{comp}) = O(A)$. □

We note that if the algorithm is started in a state where all nodes are green, then it will converge to a solution in time $O(D_{comp})$. However, if the algorithm is started in an arbitrary state (where some nodes are red due to link or node failures that occurred), convergence may take as long as $O(A)$.

Next we note that our algorithm, in many cases, can deal with link failures in $O(1)$ time. Specifically, if the link between j and $par.j$ fails, j might be able to reestablish a path to $par.j$ in $O(1)$ time. This can happen as long as j has another neighbor at a shorter distance from $root.j$ or all of j's children have another neighbor closer to $root.j$ than j was.

We also note that the messages sent by our algorithm are of size $O(\log n)$. This can be seen by inspecting the messages and variables sent in the code.

2. A heuristic to improve performance

Finally, we briefly discuss a heuristic to improve the performance of our algorithm. When a node j discovers it has an invalid parent, it will petition a suitable neighbor for adoption, if one exists. The variable $tried.j$ is used to keep track of which nodes j has petitioned for adoption to deal with the current invalid parent. In the current pseudocode, if j's first request for adoption is denied, j immediately colors itself red. The code can be modified so that if j's first petition fails, it petitions a different neighbor, repeating until it finds a new parent or runs out of neighbors.

While this modification yields improved running time in many cases, in some situations it may take $O(A^2)$ time to converge. (Such convergence time could occur, for example, if a number of nodes petition each neighbor for adoption, but have every such petition rejected.) For this reason we did not include the modification in the pseudocode. Experiments might be useful to see if the worst case convergence time occurs frequently enough to counteract the advantages of the modification, or to tune this modification by finding the number of allowed failed petitions that would yield the best performance.

CHAPTER III

MANET SIZE ESTIMATION VIA RANDOMIZATION

A. The mobile ad-hoc setting and randomization

Before proceeding any further, let us more carefully formalize our MANET system assumptions for studying the network size estimation problem. Each node p_i in the network has a unique identifier denoted by id_i. Again, given a state of a MANET, we can create a corresponding undirected communication graph G. We assume that each node p_i automatically maintains a set adj_i which consists of those nodes it believes are in its communication zone.

We note that G can consist of more than one connected component. Many problems, in their MANET specification, require that the problem be solved for each connected component. While we have previously spoken of estimating the size of the MANET, we really mean and are interested in estimating the size of (equivalently, number of nodes in) a particular connected component of the MANET. We will elaborate on this further in the problem statement below.

1. Randomization

Because of the dynamic nature of MANETs, mechanisms must be created to deal with their unpredictability. Using algorithms that are themselves unpredictable due to randomization seems like a natural method to deal with such unpredictability. Indeed, several papers employ randomization in specific ways in the MANET setting. Random walks are used for group communication and token circulation in [10],[8] respectively. Randomization is used both to break symmetry and deal with link state

changes while constructing spanning trees in [2]. Randomization is also sometimes used to model the mobility and link status changes involved in a MANET ([7]).

B. Problem statement

We now provide an explicit statement of the problem we wish to study and solve. We would like to provide to each node p_i in the MANET an estimate of the number of nodes in its connected component in G. **We are concerned with the size of the connected component (CC) rather than the size of the MANET because the MANET may consist of several connected components unaware of the existence of each other. Throughout we will only be concerned with a particular connected component of the MANET, and this connected component will have size n.**

Additionally, we define the underlying communication graph corresponding to node p_i's connected component. G_i is the subgraph obtained from G by removing all nodes not in p_i's CC and any edges incident on such nodes. If two nodes p_i and p_j are in the same CC, then $G_i = G_j$.

C. Quick and dirty solutions to the problem

One could easily provide the size of a connected component (CC) of a MANET to every node in the CC via several methods. For example, all nodes could simply periodically broadcast lists of their neighbors, and these lists could be flooded throughout the network. Every node would just periodically count the number of nodes in its "view" of the network obtained via these broadcast messages.

Alternatively, one could use any of a number of algorithms for constructing rooted spanning trees in MANETs (e.g., [1, 2]). Each node could propagate up to its parent

its number of descendants in the rooted spanning tree, and then the root would eventually know the total number of nodes in its CC of the MANET. This number could then be distributed to all nodes.

While both methods discussed above provide information about CC size to the nodes of the CC, they have notable drawbacks. The first method incurs a large communication overhead which makes it impractical and unscalable to large networks. The second method has the drawbacks of taking significant amounts of time and communication resources to construct the rooted tree, and possibly requiring adjustments to the tree structure when link changes occur. We are interested in a method to compute CC size with extremely low overhead and high adaptability to link changes.

D. Previous work

In surveying the literature, we were able to find two algorithms which propose interesting and efficient solutions to estimating the size of MANET CCs.

1. Horowitz and Malkhi

Horowitz and Malkhi, in [15], present an algorithm to estimate CC size in dynamic wired networks using only local information. Their algorithm requires a node to only perform an action upon entering a CC or leaving a CC.

Upon joining a CC, a node uses randomization to uniformly at random pick any of the other nodes in the CC as its "successor." The successor and the joining node then communicate some information so that both nodes have an updated estimate of the size of the CC. This simple strategy has surprisingly good performance, as its expected accuracy is within the range $n/2$ to n^2, where the CC for which we are estimating size has size n.

Unfortunately, the algorithm also has some drawbacks. It requires that a joining node be able to pick any existing node in the CC as its successor with equal probability. This requires the joining node to communicate with nodes besides its neighbors, and forces the development of a routing algorithm to allow a node and its successor to communicate. Nodes may be as far as $O(n)$ hops apart, so each node joining or leaving the CC can incur high communication costs. These drawbacks can make the algorithm unsuitable for a MANET environment where frequent link changes can occur or communication limits may exist.

2. Dolev, Schiller, and Welch

Dolev, Schiller, and Welch, in [10], develop a random walk technique for group communication in MANETs. As a quick aside, [10] mentions a method to approximate the size n of the CC. The strategy is as follows.

The procedure uses a parameter l which can be varied to make the estimate of n more accurate. The algorithm starts a token circulating around the CC using a standard random walk. Whenever the token encounters a previously unvisited node, say the t'th node, the token will continue walking for an additional $2lt^3 \log t$ steps. If no new node is encountered during this process, then the total number of new nodes encountered by the token during its walk is reported as the estimate of n.

To examine the accuracy of this algorithm, we need to recall that the expected cover time for a random walk on a connected undirected graph is $4n^3/27 + o(n^3)$ [11]. Let us assume that the graph's topology is static during the execution of the algorithm so that n remains constant.

For large enough l, the probability that some processor is not visited during lt^3 token steps is less than $1/2$, since the expected cover time for a connected component of t nodes is $O(t^3)$. So the probability that some processor is not visited during

$2lt^3 \log t$ token steps is $< (1/2)^{2\log t} = 1/t^2$, and the probability of not visiting a node of the CC during the entire estimation procedure is bounded by $\sum_{i=2}^{n-1} 1/i^2 < 1/2$.

Thus, the algorithm, if not interrupted by link changes, will give an exact count of the number of nodes in the CC with probability $> 1/2$. This result is simple and rather impressive, and a candidate for solving the problem we are studying in a MANET with infrequent link changes.

However, if some nodes are continuously leaving and joining a CC, one could imagine a scenario where the token, during its random walk, repeatedly encounters unvisited nodes, and so may never yield an estimate of n. For example, consider a CC which contains some location-fixed nodes which are always part of the CC, but also allows nodes passing by to temporarily join the CC. Even if we impose the restriction that a node is not allowed to leave the CC while it has the token, we can have the following situation. If a steady stream of nodes join and leave the CC, the token may continue to encounter new nodes even though the size of the CC is roughly constant.

Another drawback of this algorithm is its possibly large running time. It takes expected time $\Theta(n^3)$ to report the size of the CC.

While both algorithms presented in this section address the problem of approximating CC size, each has drawbacks which make it impractical in the presence of frequent link changes and/or communication limits. With this in mind, we present the following ideas for obtaining a rough estimate of CC size in a MANET.

E. Basic ideas of our algorithm

We propose a strategy in the vein of [10] which is more tolerant of link changes and does not take as long to give an estimate of CC size, but trades these advantages for the drawback of providing a less exact estimate of CC size. (Theorem 3 will

characterize how precise our estimate is and how long our algorithm must run to obtain the estimate.) Our strategy is as follows.

Each node p_k in our CC maintains a counter $tokens_k$ which is initialized to 0 and is the number of tokens p_k has generated during its existence. If some node p_i in the CC lacks recent information about the size of its network, it generates a new token $(id_i, tokens_i, steps, 0)$. The third entry of this token, $steps$, is the number of steps the token has taken. The fourth entry simply indicates that the token is being circulated on G_i. (Later we will introduce another token that is circulated on a modified version of G_i, and has 1 as its fourth entry.) This token is then circulated throughout p_i's CC using a traditional random walk on the underlying undirected communication graph G_i of the CC. (We require that each node in G_i has a self-loop to itself so that we can guarantee that the random walk (Markov chain) is aperiodic and will have a unique stationary distribution.)

As this token is circulated throughout the CC via the random walk, it carries a running sum of the number of steps it has taken during its walk. Assuming the underlying communication graph G_i is static, we know by the Fundamental Theorem of Markov Chains ([18]) that the token will visit each node in the CC infinitely often (with probability 1).

Now, let h_{ij} denote the expected number of steps a token takes in its random walk to travel from node p_i to node p_j. If $i = j$ then we define this as the time for the token to leave p_i and return to p_i. Let m be the number of edges in the CC and $d(k)$ be the degree (number of edges incident on p_k in G_k) of p_k.

Define the random variable R_k to be the number of steps a token takes in a random walk in G_k to leave p_k and return again to p_k. The theory of random walks tells us that $\mathbf{E}[R_k] = 2m/d(k)$, so we know that by simply observing the random walk of a token, we can gain some information about the global structure of G_k. (Namely,

we have a random variable whose expected value is a function of the number of edges in G_k, or m.)

With this in mind, define $X_k = R_k d(k)/2$. (X_k is simply R_k multiplied by a constant to force $\mathbf{E}[X_k] = m$, the number of edges in G_k.) We have the following lemma.

Lemma 10. $\mathbf{E}[X_k] = m$.

Proof. By Lemma 6.3 of [18], we have that for any node p_k in the CC

$$h_{kk} = \frac{2m}{d(k)}.$$

Thus,

$$\mathbf{E}[X_k] = \frac{d(k)}{2}\mathbf{E}[R_k] = \frac{d(k)}{2}h_{kk} = \frac{d(k)}{2}\frac{2m}{d(k)} = m,$$

and the result holds. □

Note that each node p_k in the CC, by simply observing how many steps it takes the token to make a round trip starting and ending at p_k, can obtain some global information about the graph. Namely, multiplying this number of steps by $d(k)/2$ will yield a random variable X_k whose expected value is m. By simply averaging the number of steps for repeated round trips starting and ending at p_k, node p_k can obtain an estimate of the number of edges in the graph. We will use this estimate of the number of edges in G_k to obtain an estimate of the number of nodes in G_k.

1. How to estimate n given an estimate of m

We have (at least informally) presented a simple method to provide an estimate of m to each node p_k in the CC. Now we present two simple ideas for obtaining an estimate of n by using similar ideas.

a. Adding self loops

The first idea is to, in addition to performing the random walk outlined above, also simultaneously perform a random walk with another token $(id_i, tokens_i, steps, 1)$. However, this random walk is done on the graph G'_k obtained by adding an additional self-loop to each node in G_k. (There are now two self-loops on each node in G'_k.) For a random walk on this modified graph, we let h'_{ij} denote the expected number of steps the random walk will take to travel from node p_i to node p_j.

Analogous to before, define the random variable R'_k to be the number of steps a token takes in a random walk in G'_k to leave p_k and return again to p_k. Define $d'(k) = d(k) + 1$ to be the degree of node k in G'_k. Further, let $X'_k = R'(k)d'(k)/2$. We have the following lemma.

Lemma 11. $\mathbf{E}[X'_k] = m + n$.

Proof. Note that the number of edges in G'_k is $m + n$. Then, just as before,

$$\mathbf{E}[X'_k] = \frac{d'(k)}{2}\mathbf{E}[R'_k] = \frac{d'(k)}{2}h'_{kk} = \frac{d'(k)}{2}\frac{2(m+n)}{d'(k)} = m + n,$$

and the result holds. □

Thus, multiplying the number of steps for a round trip of this token starting and ending at p_k by $(d(k) + 1)/2$ yields a random variable X'_k whose expected value is $(m + n)$.

Now, define $Z_k = X'_k - X_k$. Simple linearity of expectations ($\mathbf{E}[X'_k - X_k] = \mathbf{E}[X'_k] - \mathbf{E}[X_k]$) gives the following.

Lemma 12. $\mathbf{E}[Z_k] = n$.

Thus, by performing two random walks and observing their local behavior, each node p_k samples a random variable $Z_k = X'_k - X_k$ with expected value n. As long as we

sample this value enough times (i.e., allow the random walk to execute long enough), each node p_k will eventually have a reasonable estimate of n. We will quantify "long enough" and "reasonable estimate" with Theorem 3.

b. Employing average degree

The second approach to using the estimate of m to gain an estimate of n assumes some method of determining (at least approximately) the average degree of the nodes in p_i's CC. Let d_{avg} be the average node degree in p_i's CC. By definition, $d_{avg} = 2m/n$. Let Y be a random variable with expected value $d_{avg}/2$. Assume X_k and Y are independent. (This will be true as long as the methods used to obtain X_k and Y do not depend on each other.) Then setting $Z'_k = X_k/Y$, we can show the following.

Lemma 13. $\mathbf{E}[Z'_k] = n$

Proof.

$$\mathbf{E}[Z'] = \mathbf{E}[X_k/Y] = \mathbf{E}[X_k]/\mathbf{E}[Y] = \frac{m}{d_{avg}/2} = n.$$

□

Thus, having an estimate of d_{avg} can provide an estimate of n, given an estimate of m.

F. Technical details

Thus far, the description of our approach to solving the CC size estimation problem has ignored some important technical details, including how close our estimate of n gets to the actual value of n. We now address these issues.

1. Link changes

Up to this point, our algorithm has assumed that the underlying communication graph G is static during the random walks the tokens take. We note that as long as link changes do not remove a token from a CC, the link changes will simply modify transition probabilities for the random walks for those nodes incident on links going up or coming down. At least intuitively, one can expect that over time these link changes will only have a significant impact upon the random walks if they significantly change the size of a CC. It is precisely when the size of a CC changes drastically that we want the behavior of the random walks to change to yield a better estimate of CC size. When round trip times (on the average) start to change drastically, we may wish to discard all previously observed round trip times and start gathering round trip times anew. This will give us a new and better estimate of the new CC size.

For the remainder of the section on technical details, we assume G is static.

2. Token generation

In developing our algorithm so far, we have assumed that a specific node generated a token and that this token walked around the node's connected component forever. However, a general MANET poses some issues that we need to address. For example, the token generated by a node p_i may be lost from node p_i's CC due to link changes and/or node failures. Unfortunately, p_i will not be able to distinguish this scenario from the situation in which p_i's component is so large that the token is taking an unusually large amount of time to return to p_i.

One method to handle this is to have p_i generate tokens with decreasing frequency until the tokens start returning to p_i. For example, p_i could generate tokens, doubling the time interval between each token generation, until tokens start to return to p_i.

When a node p_k received a token from p_i, it would ignore and destroy any tokens previously generated by p_i encountered. (Previously created tokens can be identified by having a lower $tokens_i$ entry in the token.) This method could be used to give p_i a good idea of an appropriate timeout period at which to decide that the token is likely lost and a new token should be generated. If later p_i discovered that it generated a new token even though an old token still existed in the network, it could appropriately increase the timeout. Occasionally p_i might want to repeat this entire process to obtain a new timeout value so that the timeout value would not become excessively large.

Another issue we need to handle is determining which node(s) should be responsible for token generation. The problem of deciding upon a single node for token generation is equivalent to leader election, which is a well-studied and surprisingly complicated problem. One solution to the issue of deciding responsibility for token generation is to allow all nodes to generate tokens. To reduce the number of tokens present in the system, we assume that each node has access (via GPS or some other mechanism) to a synchronized clock, and have each node timestamp its tokens. Nodes only continue the random walk for a token with most recent timestamp, and destroy any tokens with "outdated" timestamps. By using this mechanism, eventually only one token (the last generated token) will exist in the CC, and this token will be used by all nodes in the CC to estimate n.

3. Goodness of estimate via adding self-loops

In the previous section, we presented a method to sample a random variable Z_k with expected value n. An important question is how wildly this random variable can vary.

If we sample Z_k a certain number of times and average the values obtained, then we will get a better estimate of n. In particular, let $Z_{k,1}, Z_{k,2}, \ldots, Z_{k,a}$ be identical

and independently distributed random variables with distribution identical to Z_k, and let Z_k^* be the average of these random variables. The following theorem answers our question about how wildly our estimates vary.

Theorem 3. *There is a polynomial $p(n)$ such that by setting $a = \lceil 8p(n)/n^2 \rceil$,* $\mathbf{Pr}[|Z_k^* - n| \geq n/2] \leq \frac{1}{2}$.

Proof. As [12] notes, the variance of the time C it takes a token to visit every node in a graph during a random walk is bounded by a polynomial function of n. Since the time it takes for a token to return to a particular node is necessarily less than the time it takes for the token to visit all nodes (and thus also return to the particular node), we get that the variances of X_k and X'_k are bounded by polynomial functions of n. So there is a polynomial $p(n)$ such that $\mathbf{Var}[Z_k] < p(n)$. Let $a = \lceil 8p(n)/n^2 \rceil$.

Then by the law of large numbers ([20]),

$$\mathbf{Var}\Big[Z_k^* = \frac{\sum_{c=1}^{a} Z_{k,c}}{a}\Big] = \frac{\mathbf{Var}[Z_k]}{a} \leq \frac{p(n)}{8p(n)/n^2} = \frac{n^2}{8}.$$

We have, by Chebyshev's inequality ([18]),

$$\mathbf{Pr}[|Z_k^* - n| \geq n/2] \leq \frac{1}{2}$$

□

Thus, given an upper bound N on the CC size, by allowing the random walks to visit each node $\lceil 8p(N)/N^2 \rceil$ times we obtain a probability greater than 1/2 of obtaining a "good" estimate on n. (Here, "good" means within $n/2$ of n.) This number of visits may be quite large, but may be feasible in networks which can send tokens between nodes quickly. Additionally, for many graphs, it may be the case that far fewer node visits are necessary before a reasonable estimate of n is obtained.

4. Goodness of estimate using average node degree

Next we examine the goodness of the estimate on n that is obtained by using an estimate of average node degree. Let $p'(n)$ be an upper bound on $\mathbf{Var}[X_k]$, and N be an upper bound on the CC size. If we have the exact average node degree of nodes in p_i's CC, then analysis similar to that in the analysis of adding self-loops yields that allowing the random walk to visit each node $\lceil 32p'(N)/(N^2 d_{avg}^2) \rceil$ yields probability greater than $1/2$ of obtaining a "good" (within $n/2$ of n) estimate on n. (This makes use of the fact that $\mathbf{Var}[X_k/(d_{avg}/2)] = \mathbf{Var}[X_k]/(d_{avg}/2)^2$.)

For some MANETs, the average degree of a node may somehow be constrained, and one can make use of this information in obtaining an estimate on n. The constraints on average degree would need to be sufficiently strong to be of significant use.

Additionally, assuming that the average degree of a MANET is fairly constant over a MANET's lifespan, we could provide for a simple distributed algorithm that runs very occasionally to obtain an estimate of d_{avg}. Nodes would simply propagate their degrees to their neighbors, neighbors would collect this information and pass it on to their neighbors, and this would continue for a fixed number of "hops." Eventually all nodes would have average node degree for those nodes within a certain number of hops from them.

Finally, we can also use the token and its random walk to get an estimate of average node degree. We know that the Markov chain for the random walk converges to a stationary distribution (assuming no link changes), and also know the relative pairwise probabilities in this distribution. Let π_j be the probability the random walk is at node p_j in this stationary distribution. $\pi_j = d(j)/2m$, so $\frac{\pi_j}{\pi_k} = \frac{d(j)}{d(k)}$.

We can attach to the token two counters *sum_degrees* and *num_degrees*. When

the token visits a node p_j, it with probability $1/d(j)$ increments *sum_degrees* by $d(j)$ and *num_degrees* by 1. (This is done so that each node's degree has equal probability of being included in the sample.) It readily follows that, assuming the Markov chain is rapidly mixing and that the random walk continues for a long enough period of time, that $\frac{sum\_degrees}{num\_degrees}$ will be an estimate of d_{avg}. This estimate can then be used along with the estimate for m to obtain an estimate of n. Examining the accuracy of this estimate is quite complicated and beyond the scope of this work.

G. Advantages and disadvantages of our algorithm

The greatest advantage of our algorithm lies in its simplicity. It requires little effort from each node to carry out the token circulation to perform the random walk and eventually provides every node in the MANET with an estimate of CC size.

Additionally, our algorithm is more resilient than other algorithms to link failures. If we impose the restriction that a link can only fail if neither of its endpoint nodes currently holds a token, then tokens will never be lost and an estimate of CC size will be provided to each node.

Our algorithm also has disadvantages worth noting. Its main disadvantage is that it provides only a rough estimate of network size, and we were only able to prove (Theorem 3) that for some polynomial $p(N)$, if each node observes numerous $(\lceil 8p(N)/N^2 \rceil)$ round trips of tokens it can obtain a reasonable (within $n/2$) estimate of n with probability $\geq 1/2$.

CHAPTER IV

CONCLUSIONS

We have developed two new algorithms to solve two important problems in mobile ad hoc networks.

First, we have developed a deterministic leader election algorithm for asynchronous MANETs. The algorithm has good convergence time, relatively simple code and correctness proofs, and no need for randomization. It has at least one advantage (which varies with the algorithm it is being compared to) over each of the other leader election algorithms developed for MANETs.

The algorithm, among other things, shows that there is still room for improvement in developing leader election algorithms for MANETs. No single leader election algorithm developed so far can make the claim of being the "best" leader election algorithm, as each one has its advantages and disadvantages. Future work could address this issue and attempt to combine the advantages while eliminating the disadvantages.

Second, we have examined the problem of estimating connected component size in MANETs, surveyed existing algorithms for the problem, and presented our own algorithm for the problem. While our algorithm definitely has some drawbacks that make it impractical in certain situations, its flexibility and low overhead make it a candidate for estimating connected component size in MANETs with frequent link changes.

One thing to be addressed in future work is the polynomial $p(n)$ used in the results on the accuracy and running time of our algorithm. While we know that such a polynomial (which bounds the variance of Z_k) exists, our algorithm cannot

be implemented without knowing what this polynomial actually is. Future work could investigate the nature of this polynomial variance bound so that a constructive network size estimation algorithm could be produced.

Future work could also focus on the accuracy of our sampling method for determining an estimate of d_{avg}. Towards the end of our work, we began to find research that could provide better bounds on how long our algorithm (using the d_{avg} estimate) has to run to get a good estimate on network size([14]), and we plan to look into this. Additionally, simulations would be useful to examine the performance of our algorithm and compare it to the algorithm in [10]. It may be the case that, for many MANETs, our algorithm has accuracy significantly better than the bounds obtained in this work.

REFERENCES

[1] Y. Afek and A. Bremler. "Self-stabilizing unidirectional network algorithms by power supply," *Chicago Journal of Theoretical Computer Science*, December 1998.

[2] S. Aggarwal. "Time optimal self-stabilizing spanning tree algorithms," in *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, Bombay, India, 1994.

[3] A. Arora and A. Singhai. "Optimal, nonmasking fault-tolerant reconfiguration of trees and rings," OSU Technical Report CISRC-TR09-1994, Oklahoma State University, Stillwater, OK, 1994.

[4] A. Arora and A. Singhai. "Fault-tolerant reconfiguration of trees and rings in distributed systems," *High Integrity Systems* vol. 1, no. 4, 1996.

[5] B. Awerbuch, I. Cidon, and S. Kutten. "Optimal maintenance of replicated information," in *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science (FOCS 90)*, St. Louis, MO, pp. 492-502, October 1990.

[6] O. Bayazit, J. Lien, and N. Amato, "Better group behaviors in complex environments using global roadmaps," in *Proceedings of the 8th International Conference on the Simulation and Synthesis of Living Systems (Alife '02)*, Sydney, NSW, Australia, pp. 362-370, December 2002.

[7] C. Bettstetter. "Smooth is better than sharp: A random mobility model for simulation of wireless networks," in *Proceedings of the ACM Workshop on Modeling, Analysis, and Simulation of Wireless and Mobile Systems*, Rome, Italy, pp.19-27, 2001.

[8] Y. Chen and J. Welch. "Self-stabilizing mutual exclusion using tokens in mobile ad-hoc networks," in *Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, Atlanta, GA, vol. 1, pp. 34-42, 2002.

[9] S. Dolev, A. Israeli, and S. Moran. "Uniform dynamic self-stabilizing leader election," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 4, pp. 424-440, April 1997.

[10] S. Dolev, E. Schiller, and J. Welch. "Random walk for self-stabilizing group communication in ad-hoc networks," in *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, Monterey, CA, vol. 8, p. 259, 2002.

[11] U. Feige. "A tight upper bound on the cover time for random walks on graphs," *Random Structures and Algorithms*, vol. 6, no. 4, pp. 51-54, 1995.

[12] U. Feige and Y. Rabinovich. "Deterministic approximation of the cover time," *Israel Symposium on Theory of Computing Systems*, pp. 208-218, 1996.

[13] E. Gafni and D. Bertsekas. "Distributed algorithms for generating loop-free routes in networks with frequently changing topology," *IEEE Transactions on Communications*, vol. 29, no. 1, p. 1118, 1981.

[14] D. Gillman. "A Chernoff bound for random walks on expander graphs," *SIAM Journal on Computing*, vol. 27, no. 4, pp. 1203-1220, 1998.

[15] K. Horowitz and D. Malkhi. "Estimating network size from local information." Technical report available at http://leibniz.cs.huji.ac.il/tr/acc/2003/HUJI-CSE-LTR-2003-29_ipl2.ps, Accessed May 2004.

[16] S. Kutten and A. Porat. "Maintenance of a spanning tree in dynamic networks," in *Proceedings of the 13th International Symposium on Distributed Computing (DISC '99)*, Bratislava, Slovak Republic, September 1999.

[17] N. Malpani, J. Welch, and N. Vaidya. "Leader election algorithms for mobile ad hoc networks," in *Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communication*, Boston, MA, pp. 96-103, 2000.

[18] R. Motwani and P. Raghavan. *Randomized Algorithms*. New York: Cambridge University Press, 1995.

[19] V. Park and M. Corson. "A highly adaptive distributed routing algorithm for mobile wireless networks," *Proceedings IEEE INFOCOM*, Kobe, Japan, April 7-11, 1997.

[20] "Weak law of large numbers," *MathWorld*, http://mathworld.wolfram.com/WeakLawofLargeNumbers.html, Accessed May 2004.

VITA

Nicholas Gerard Neumann grew up in El Paso, TX. He attended Texas A&M University in College Station, TX where he graduated Summa Cum Laude with a B.S. in Mathematics in May 2002. He will begin work as a software engineer for National Instruments in Austin, TX in Summer 2004.

His permanent address is: Nicholas Gerard Neumann, 4609 R.L. Shoemaker, El Paso, TX, 79924.

The typist for this thesis was Nicholas Gerard Neumann.