An Asynchronous Leader Election Algorithm for Dynamic Networks

B.Tech Seminar report

by

Sabitha C.B
ETAHECS068

Department of Computer Science And Engineering
Government Engineering College, Thrissur
December 2010

Acknowledgment

I would like to acknowledge and extend my sincere gratitude to the following persons who have made the completion of this Seminar possible:

Head of the Department of Computer Science and Engineering **Prof. Manoj Kumar** for permitting me to present this seminar.

Prof.K S Valsaraj, for providing me with timely guidance and motivation.

Mr.Ajay James, lecturer and **Mrs.Baby Syla** lecturer, for the constant reminders and encouragement during various stages of this Seminar.

All Computer Science and Engineering Department faculty members and Staff.

Most especially to my classmates for their support.

And to God almighty, who made all things possible.

Sabitha C.B
December 2010
Govt. Engineering College, Thrissur

Contents

1 Introduction **1**
1.1 Organization Of the Report 3

2 Preliminaries **4**
2.1 System Model 4
2.2 Modeling Asynchronous Dynamic Links 4
2.3 Configurations and Executions 5
2.4 Problem Definition 6

3 Leader Election Algorithm **8**
3.1 Overview of Algorithm 10

4 Advantages and disadvantages **14**

5 Conclusion **16**

References **16**

List of Figures

2.1 State diagram for status variable of Link { u, v } 5

3.1 Code triggered by Update message. 10
3.2 Code triggered by link changes 11
3.3 Subroutines. 11
3.4 Simple execution when leader H becomes disconnected (a), with time increasing from (a)-(h). With no other link changes, every node in the connected component will eventually adopt G as its leader. 13

Abstract

This is a leader election algorithm, used in dynamic networks. In this algorithm what pattern of topology change occur, when the topology change is over then every connected component in the network will contain a unique leader. Dynamic network is a network whose communication topology changes frequently. This algorithm is applicable in asynchronous networks. The algorithm is developed using ideas from a routing algorithm named TORA. TORA (Temporally Ordered Routing Algorithm) is a routing algorithm for mobile ad hoc networks. TORA uses a Wave algorithm and a height-based mechanism for reversing the logical direction of communication links [6]. In this algorithm if a node loses its last outgoing link it will start search for the leader, if it cannot get a path to the leader it will elect itself as a leader. If it finds the old leader there will not be any change in the leader. This algorithm can elect any node as the leader, involves fewer types of messages than other algorithms, and uses point-to-point communication rather than broadcasting. The strategy for breaking ties between competing leaders makes this algorithm compact and elegant, as messages sent between nodes carry only the height information of the sending node, and every message is identical in content. This algorithm is also suited to an arbitrary communication topology. It is proved that in certain well-behaved situations, a new leader is not elected unnecessarily.

Chapter 1

Introduction

Leader election is important in distributed computing, it is used as a subroutine for any application that requires the selection of a unique processor among multiple candidate processors. Applications that need a leader range from the primary-backup approach to replication based fault-tolerance to group communication systems , and from video conferencing to multi-player games . A core activity for a network leader is communication. Clear and transparent communication around the networks aims, values and activities are crucial to building ownership and participation. A network needs data, information and intelligence around which to plan, work and learn. A network leader will be proactive in identifying and accessing knowledge sources within and beyond the network. A network leader brokers and sustains the relationships, carefully building trust and security as a foundation for innovation and experimentation.

In a dynamic network, communication links go up and down frequently. Wireless mobile networks are one example of dynamic networks, since node mobility changes the communication topology continuously. Even if nodes do not move, wireless communications are subject to more interference than in the wired case, but wired networks can also experience frequent topology changes. Recent research has focused on porting some of the applications mentioned above to dynamic networks, including wireless and sensor networks. For instance, Wang and Wu propose a replication-based scheme for data delivery in mobile and fault-prone sensor networks . Thus there is a need for leader election algorithms that work in dynamic networks.

This algorithm consider the problem as the one which ensures that, if link changes eventually cease, then eventually each connected component of the network has a unique leader, this is represented as the local leader election problem[3]. The algorithm is presented as an extension of the leader election algorithm in, which in turn is an extension of the MANET routing algorithm TORA[10]. TORA itself is based on ideas from two routing algorithms presented by Gafni and Bertsekas [4]based on the

notion of link reversal. In these algorithms, each node maintains a height variable, drawn from a totally ordered set; the link between two nodes is considered to be directed from the endpoint with larger height to that with smaller height. Whenever a node becomes a sink, i.e., has no outgoing links, due to a link failure or due to notification of a neighbors changed height, the node increases its height so that at least one of its incoming links becomes outgoing. In one of the algorithms, the height is a pair, while in the other the height is a triple; in both situations, heights are compared lexicographically and the least significant component is the nodes unique id. The algorithms cause an infinite number of messages to be sent if a portion of the graph is disconnected from the destination.

This drawback is overcome in TORA, through the addition of a clever mechanism by which nodes can identify that they have been partitioned from the destination. In this case, the nodes go into a quiescent state. In TORA, each node maintains a 5-tuple of integers for its height, consisting of, from left to right, a 3-tuple called the reference level, a delta component, and the nodes unique id. The height tuple of each node is lexicographically compared to the tuple of each neighbor to impose a logical direction on links (higher tuple toward lower.) The purpose of a non-zero reference level is to indicate when nodes have lost their path to the destination. Initially, the reference level is all zeroes. When a node loses its last outgoing link due to a link disappearing, it starts a new reference level by changing the first component of the triple to the current time, the second to its own id, and the third to 0, meaning that the search for the destination is started. Reference levels are propagated throughout a connected component, as nodes lose outgoing links, in a search for an alternate directed path to the destination. Propagation of reference levels is done using a mechanism by which a node increases its reference level when it becomes a sink; the delta value of the height is manipulated to ensure that links are oriented appropriately. If one section of the communication graph is a dead-end, then the third component of the reference level triple is set to 1. When this happens, the reference level is said to have been reflected, since it is subsequently propagated back toward the originator. If the originator receives reflected reference levels back from all its neighbors, then it has identified a partitioning from the destination.

The key observation in a Leader election algorithms for mobile ad hoc networks presented by N. Mopani, J. Welch, and N. Vaidya[6] is that TORA can be adapted for leader election: when a node detects that it has been partitioned from the destination (the old leader), then, instead of becoming quiescent, it elects itself. The information about the new leader is then propagated through the connected component. A sixth component was added to the height tuple to record the leaders id.

However, when multiple topology changes occur, this algorithm can fail. But this

new algorithm works in an asynchronous system with arbitrary topology changes. One new feature of this algorithm is to add a seventh component to the height: a timestamp associated with the leader id that record the time that the leader was elected. This algorithm also includes a new rule by which nodes can choose new leaders. A newly elected leader initiates a wave algorithm[9]: when different leader ids collide at a node, the one with the most recent timestamp is chosen as the winner and the newly adopted height is further propagated. This strategy for breaking ties between competing leaders makes our algorithm compact and elegant, as messages sent between nodes carry only the height information of the sending node and every message is identical in content.

1.1 Organization Of the Report

1. Chapter 2 describes Preliminaries for the algorithm
2. Chapter 3 describes Algorithm and its term
3. Chapter 4 describes Advantages and disadvantages comparing with other algorithm

Chapter 2

Preliminaries

2.1 System Model

In this algorithm the system is assumed as Directed Acyclic Graph (DAG) consisting of a set P of computing nodes as set of vertices and a set L of bidirectional communication links between nodes as set of edges. L consists of one link for each unordered pair of nodes, i.e., every possible link is represented. The nodes are assumed to be completely reliable. The links between nodes go up and down, due to the movement of the nodes. While a link is up, the communication across it is in first-in-first-out order and is reliable but asynchronous.

The whole system is modeled as a set of (infinite) state machines that interact through shared events (a specialization of the IOA model). Each node and each link is modeled as a separate state machine. The shared events are Link Up/Down notifications and receipt of messages, all of which are controlled and initiated by the link and responded to by the node. The sending of a message is also a shared event, but it is controlled and initiated by the node and responded to by the link; we are not explicitly modeling this. The next subsection gives more details about how links are modeled and specifies the initial states. The algorithm executed by the nodes and its initial states are described in Chapter 3.

2.2 Modeling Asynchronous Dynamic Links

This section specifies how communication is assumed to occur over the dynamic links, and how notification of a links status is synchronized at the two endpoints of the link. The state of a link Link { u, v } which models the bidirectional. Communication link between node u and node v, consists of a status variable and two queues of messages. The possible values of the status variable are Up, GoingDown_u, GoingDown_v,

Down, ComingUp_u, and ComingUp_v. The link transitions among different values of its status variable through LinkUp and LinkDown events.

Figure 2.1 shows the state transition diagram for Link { u, v}.

The intuition is that if a LinkUp (resp., LinkDown) occurs at one endpoint of the link, then LinkUp (resp., LinkDown) must occur at the other endpoint before LinkDown (resp., LinkUp) can occur at either end.

Figure 2.1: State diagram for status variable of Link { u, v }

The other components of the links local state are the two message queues: $\text{mqueue}_{u,v}$ holds messages in transit from u to v and $\text{mqueue}_{v,u}$ holds messages in transit from v to u. An attempt by node u to send a message to node v results in the message being appended to $\text{mqueue}_{u,v}$ if the links status is either ComingUp_u or Up; otherwise there is no effect. If the status is ComingUp_u, then messages in transit from u to v are held in the queue until v has been notified that the link is Up. Once the link is Up, the event by which node u receives the message at the head of $\text{mqueue}_{v,u}$is enabled to occur. An attempt by node v to send a message to node u is handled analogously. Whenever a LinkDown_u or LinkDown_v event occurs, both message queues are emptied. Neither u nor v is alerted to which messages in transit have been lost due to the LinkDown. In an initial state of the link, both message queues are empty and the status is either Up or Down.

2.3 Configurations and Executions

The notion of configuration is used to capture an instantaneous snapshot of the state of the entire system. A configuration is a vector of node states, one for each node

in P, and a vector of link states, one for each link in L. Assume that the undirected graph G = (V,E) defines the initial communication topology of the system, where V is a set of vertices corresponding to the set P of nodes, and E, Is a set of edges corresponding to the set of communication links that are up. In an initial configuration with respect to G, each node is in an initial state each link corresponding to an edge in E is in an initial state with its status equal to Up, and every other link has its status equal to Down.

Define an execution as an infinite sequence C0,e1,C1,e2,C2, . . . of alternating configurations and events, starting with an initial configuration and, if finite, ending with a configuration, that satisfies the following safety conditions:

- C0 is an initial configuration (w.r.t. some initial topology G).
- The preconditions for event ei are true in Ci-1 for all i $\geq$ 1.
- Ci is the result of executing event ei on configuration Ci-1, for all i $\geq$ 1 (only the node and link involved in an event change state, and they change according to their state machine transitions).

An execution also satisfies the following liveness conditions:

- If a link remainsUp for infinitely long, then every message sent over the link is eventually delivered.
- For each link, if only a finite number of link events occur, then the link status after the last one is either Up or Down (not in between).

We also assign a positive real-valued global time gt to each event ei, i $\geq$ 1, such that gt(ei) $<$ gt(ei+1) and, if the execution is infinite, the global times increase without bound. Each configuration inherits the global time of its preceding event, so gt(Ci) = gt(ei) for i $\geq$ 1; we define gt(C0) to be 0. We assume that the nodes have perfect clocks, i.e., the nodes have access to gt.

2.4 Problem Definition

Each node u in the system has a local variable lid_u to hold the identifier of the node currently considered by u to be the leader of the connected component containing u.

In every execution that includes a finite number of topology changes, we require

that the following eventually holds: Every connected component CC of the final topology contains a node ℓ, the leader, such that $\text{lid}_u = \ell$ for all nodes u CC, including ℓ itself.

This algorithm also ensures that eventually each link in the system has a direction imposed on it by virtue of the data stored at each endpoint such that each connected component CC is a leader-oriented DAG, i.e., every node has a directed path to the leader.

Chapter 3

Leader Election Algorithm

This section explains the local variables used in our leader election algorithm. The pseudo code for the algorithm is presented in Figures 3.1, 3.2 and 3.3. An overview and sample execution is given in Section 3.1. In the analysis, variable v of node i will be indicated as v_i. Each node i keep an array of heights, heighti, with an entry for itself and for each of its neighbors, in which it stores the most recent height information that it has received for those nodes.
Each height is a 7-tuple, with the following components:

1. τ , a nonnegative timestamp that is either 0 or the time when the current search for an alternate path to the leader was initiated
2. oid, a nonnegative value that is either 0 or the id of the node that started the current search
3. r, a bit that is set to 0 when the current search is initiated and set to 1 when the current search hits a dead end
4. δ, an integer that is set to ensure that links are directed appropriately to neighbors with the same first three components
5. nlts, a nonpositive timestamp whose absolute value is the time when the current leader was elected
6. lid, the id of the current leader
7. id, the id of the node

Components (τ ,oid, r) are referred to as the reference level, or RL; (τ ,oid) alone are referred to as the reference level prefix; and (nlts, lid) is referred to as the

leader pair or LP. The components of entry k in height$_i$ are referred to as (τ_k,oid$_k$, r$_k$,δ_k,nlts$_k$, lid$_k$,k) in the pseudo code.

Nodes communicate over links during the algorithm execution by sending Update messages. Each message contains only the height tuple of the sending node. The link between node i and one of its neighboring nodes j is considered by i to be outgoing (directed from i to j) if and only if height$_i$[i]> height$_i$[j]. That is, node i uses the information in its local state concerning itself and node j to determine (its view of) the direction of the link to j. Because of message delays, it is not necessarily the case that i and j have consistent views of the direction of the link between them.

Other events that occur at a node are formations (LinkUps) and failures (LinkDowns) of links. Suppose the most recent indication that node i has received concerning the link between itself and node j is a LinkUp. If i has received a message from j since that LinkUp, then i considers j as one of its neighbors, and stores the id of j in its local variable Ni. If i has not yet received a message from j, then the link is considered as still forming, and i stores the id of j in its local variable forming$_i$; j is not considered a neighbor of i (yet).

Given an initial connected communication graph G = (V,E), with V corresponding to the set of nodes and E to the set of communication links that are up, the initial state of each node i is defined as follows.

- forming$_i$ is empty
- N$_i$ contains the id of every node j such that the vertices in V corresponding to i and j are neighbors in G
- height$_i$[i] = (0,0,0,δ_i,0, ℓ, i), where ℓ is the id of a fixed node in is connected component, the current leader
- for each neighbor j of i, height$_i$[j] = height$_j$[j] (i.e., i has accurate information about js height)

Furthermore, for each node i, δ_i equals the distance from i to ℓ; this condition ensures that every node has a directed path to ℓ.

Next we define the conditions under which a node considers itself to be a sink.

- SINK = ((LP$^j{}_i$ = LP$^i{}_i$ j N$_i$) and (height$_i$[i] < minheight$_i$[j] j N$_i$) and (lidii, i)). This predicate is true when, according to is local state, i is not a leader, has all neighbors with the same LP, and has no outgoing links. If node i has links to any neighbors with different LPs, i is not considered a sink, regardless of the

directions of those links.

3.1 Overview of Algorithm

We depict the network as a DAG in which each bidirectional communication link points from a node with lexicographically higher height to another node with lexicographically lower height. Nodes send algorithm messages only if initial knowledge of neighboring nodes is not available, then the algorithm could begin with each node in a singleton connected component.

When node i receives Update(h) from node j:

```
height[j] := h
forming := forming \ {j}
N := N ∪ {j}
myOldHeight := height[i]
if ((nlts^i, lid^i) = (nlts^j, lid^j)) // leader pairs are the same
    if (SINK)
        if (∃ (τ,oid,r) | (τ^k, oid^k, r^k) = (τ,oid,r) ∀ k ∈ N)
            if ((τ > 0) and (r = 0))
                REFLECTREFLEVEL
            else if ((τ > 0) and (r = 1) and (oid = i))
                ELECTSELF
            else
            // (τ = 0) or (τ > 0 and r = 1 and oid ≠ i)
                STARTNEWREFLEVEL
            end if
        else
        // neighbors have different ref levels
            PROPAGATELARGESTREFLEVEL
        end if
    // else not sink, do nothing
    end if
else //  leader pairs are different
    ADOPTLPIFPRIORITY(j)
end if
if (myOldHeight ≠ height[i])
     send Update(height[i]) to all k ∈ (N ∪ forming)
end if
```

Figure 3.1: Code triggered by Update message.

```
When LinkDown event occurs at node i for link to node j:
1.   N := N \ {j}
2.   forming := forming \ {j}
3.   if (N = ∅)
4.       ELECTSELF
5.   else if (SINK)
6.       STARTNEWREFLEVEL
7.       send Update(height[i]) to all k ∈ (N ∪ forming)
8.   end if

When LinkUp event occurs at node i for link to node j:
1.   forming := forming ∪ {j}
2.   send Update(height[i]) over new link
```

Figure 3.2: Code triggered by link changes

```
ELECTSELF
1.   height[i] := (0,0,0,0,−t,i,i)
     // t is current time, negated for timestamp

REFLECTREFLEVEL
1.   height[i] := (τ,oid,1,0,nlts^i,lid^i,i)

PROPAGATELARGESTREFLEVEL
1.   (τ^i,oid^i,r^i) := max{(τ^k,oid^k,r^k)| k ∈ N}
2.   δ^i := min{ δ^k | k ∈ N and (τ^i,oid^i,r^i) = (τ^k,oid^k,r^k)} − 1

STARTNEWREFLEVEL
1.   height[i] := (t,i,0,0,nlts^i,lid^i,i)   // t is current time

ADOPTLPIFPRIORITY(j)
1.   if ((nlts^j < nlts^i) or ((nlts^i = nlts^j) and (lid^j < lid^i)))
2.       height[i] := (τ^j,oid^j,r^j,δ^j + 1,nlts^j,lid^j,i)
3.   end if
```

Figure 3.3: Subroutines.

when they change the contents of their height tuple. The contents of the height tuple at a particular node are changed only when the node elects itself a leader, when it changes its current leader, or when it loses its last outgoing link to its current leader. The network is quiescent when there is no message in transit on any link. Messages that do not cause a node to lose its last outgoing link to its current leader or to change its current leader result only in a change to the internal data that node keeps about its neighbors heights.

Figure 3.4 shows a sample execution of the algorithm.

Each part (a)-(h) is discussed below.

(a) A quiescent network is a leader-oriented DAG in which node H is the current leader. The height of each node is displayed in parenthesis. Link direction in this Figure is shown using solid-headed arrows and messages in transit are arrows with outlined heads superimposed on the links that point from message sender to receiver.

(b) When non-leader node G loses its last outgoing link due to the loss of the link to node H, G executes subroutine STARTNEWREFLEVEL and takes on RL (5,G,0) and $\delta = 0$. Then node G sends messages with its new height to all its neighbors. By raising its height in this way, G has started a search for leader H.
(c) NodesD, E, and F receive the messages sent from node G, messages that cause each of these nodes to take on RL (5,G,0) and set its δ to -1, ensuring that its height is low er than Gs but higher than the other neighbors. Then D, E and F send messages to their neighbors.
(d) Node B has received messages from both E and D with the new RL (5,G,0), and C has received a message from F with RL (5,G,0); as a result, B and C take on RL (5,G,0) with δ set to -2, and send messages.
(e) Node A has received message from both nodes B and C. In this situation, node A is connected only to nodes that are participating in the search started by node G for leader H. In this case, node A reflects the search by setting the reflection bit in the (5,G,*) reference level to 1, resetting its δ to 0, and sending its new height to its neighbors.
(f) Nodes B and C take on the reflected reference level (5,G,1) and set their δ to -1, causing their heights to be lower than As and higher than their other neighbors. Then they send their new heights to their neighbors.
g) Nodes D, E, and F act similarly as B and C did in part (f), but set their variables to -2. (h) When node G receives the reflected reference level from all its neighbors at time 14, it knows that its search for H is in vain and it elects itself. The new LP (-14, G) then propagates through the component assuming no further link changes occur; eventually each node has RL (0,0,0) and LP (-14,G), with D, E and having $\delta = 1$, B and C having $\delta = 2$, and A having $\delta = 3$.

Figure 3.4: Simple execution when leader H becomes disconnected (a), with time increasing from (a)-(h). With no other link changes, every node in the connected component will eventually adopt G as its leader.

Chapter 4

Advantages and disadvantages

Several other leader election algorithms have been developed based on MANET routing algorithms. An efficient leader election algorithm for mobile ad hoc networks presented by P. Parvathipuram, V. Kumar, and G.-C. Yang[7] is based on the Zone Routing Protocol Z. Haas. A new routing protocol for the reconfigurable wireless networks[5]. A correctness proof is given, but only for the synchronous case assuming only one topology change. In a self-stabilizing leader election algorithm for highly dynamic ad hoc mobile networks, Derhab and Badache present a leader election algorithm for ad hoc wireless networks that, like this algorithm, is based on the algorithms(Leader election algorithms for mobile ad hoc networks) presented by Malpani. This algorithm is simpler and uses fewer message types and smaller messages than the algorithm presented by Derhab and Badache. Unlike Derhab and Badache, we can prove this algorithm is correct even when communication is asynchronous and multiple link changes and network partitions occur during the leader election process. Dagdeviren in a hierarchical leader election protocol for mobile ad hoc networks[1] and Rahman in an algorithm for performance analysis of leader election algorithms in mobile ad hoc networks[8] have recently proposed leader election algorithms for mobile ad hoc networks; these algorithms have been evaluated solely through simulation, and lack correctness proofs. A different direction is randomized leader election algorithms for wireless networks; this algorithm is deterministic. Fault-tolerant leader election algorithms have been proposed for wired networks. Representative examples are Mans and Santoros algorithm for loop graphs subject to permanent link failures and Singhs algorithm for complete graphs, Leader Election in the Presence of Link Failures subject to intermittent link failures.

In this algorithm, the algorithm stops the propagation of the older leader. This could be changed so that older leaders are sometimes chosen over the newer ones. However, there is a potential problem with selecting an older leader, as it is less

likely to be up-to-date as topology changes in the network are still occurring. During the implementation of the algorithm presented by A. Derhab and N. Badache. A preliminary comparison between the performance of the two algorithms on the same topology records shows that the algorithm presented by A. Derhab and N. Badache. is only marginally better with regards to the number of messages it uses to find a unique leader for each connected network component. The algorithm of Derhab and Badache uses fewer messages than the algorithm of Ingram; it does so at the expense of higher computational cost and higher algorithm complexity, and therefore may not be an efficient nor correct solution to the leader election problem on distributed networks with highly asynchronous message delays.

Chapter 5

Conclusion

We have described a leader election algorithm for asynchronous dynamic networks. This algorithm will work in networks having frequently changing topology is present. Also work in situations where several topology change occur simultaneously. It can break the ties when more than one node initiate the search and collide at a common node. Our algorithm is simpler and uses fewer message types and smaller messages than other algorithms. A set of circumstances were identified under which the algorithm does not elect a leader unnecessarily, but it remains to give a more complete characterization of such circumstances. The algorithm relies on the nodes having perfectly synchronized clocks; an interesting open question is to quantify the effect on the algorithm of approximately synchronized clocks. Future work can focus on a more thorough investigation of the performance and computational complexity and testing a modification of this algorithm.

References

[1] O. Dagdeviren and K. Erciyes. A hierarchical leader election protocol for mobile ad hoc networks. In Proc. 8th Intl Conf. on Computational Science, 2008.

[2] A. Derhab and N. Badache. A self-stabilizing leader election algorithm in highly dynamic ad hoc mobile networks. IEEE Trans. on Parallel and Distributed Systems, 2008.

[3] C. Fetzer and F. Cristian. A highly available local leader election service. IEEE Trans. on Software Engineering, 1999.

[4] E. Gafni and D. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. IEEE Trans. on Communications, 1981.

[5] Z. Haas. A new routing protocol for the reconfigurable wireless networks. In Proc. 6th IEEE Intl Conf. on Universal Personal Comm., 1997.

[6] N. Malpani, J. Welch, and N. Vaidya. Leader election algorithms for mobile ad hoc networks. In Proc. ACM DIAL-M Workshop, 2000.

[7] P. Parvathipuram, V. Kumar, and G.-C. Yang. An efficient leader election algorithm for mobile ad hoc networks. In Proc. 1st Intl Conf. on Dist. Computing and Internet Technology, 2004.

[8] David A. Patterson, John L. Hennessy, "Computer Architecture, A Quantitative Approach", Morgan Kaufmann Publications Inc., San Mateo, California, USA, 1990

[9] M. Rahman, M. Abdullah-Al-Wadud, and O. Chae. Performance analysis of leader election algorithms in mobile ad hoc networks. Intl J. of Computer Science and Network Security, 2008.

[10] G. Tel. Introduction to Distributed Algorithms, Second Edition. Cambridge University Press, 2000.

[11] V. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In Proc. INFOCOM9, 1997.

[12] Rebecca Ingram, Patrick Shields, Jennifer E. Walter, Jennifer L. Welch .An Asynchronous Leader Election Algorithm for Dynamic Networks Technical Report 2009-1-1, Department of Computer Science and Engineering, Texas A and M University