Open access • Book Chapter • DOI:10.1007/978-3-642-16023-3_6

Slf-stabiliezing leader election in dynamic networks — Source link

Ajoy K. Datta, Lawrence L. Larmore, Hema Piniganti

Institutions: University of Nevada, Las Vegas

Published on: 20 Sep 2010 - International Conference on Stabilization, Safety, and Security of Distributed Systems

Topics: Leader election, Distributed algorithm, Connected component and Self-stabilization

12-2010

Self-stabilizing leader election in dynamic networks

Hema Piniganti
University of Nevada, Las Vegas

SELF-STABILIZING LEADER ELECTION IN DYNAMIC NETWORKS

By

Hema Piniganti

Bachelor of Engineering, Computer Science
Osmania University, India
2008

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
December 2010

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Hema Piniganti

entitled

Self-Stabilizing Leader Election in Dynamic Networks

be accepted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
School of Computer Science

Ajoy K. Datta, Committee Chair

Lawrence L. Larmore, Committee Member

Yoohwan Kim, Committee Member

Emma E. Regentova, Graduate Faculty Representative

Ronald Smith, Ph. D., Vice President for Research and Graduate Studies
and Dean of the Graduate College

December 2010

ABSTRACT

SELF-STABILIZING LEADER ELECTION IN DYNAMIC NETWORKS

By

Hema Piniganti

Dr. Ajoy K. Datta, Examination Committee Chair
School of Computer Science
University of Nevada, Las Vegas

The leader election problem is one of the fundamental problems in distributed computing. It has applications in almost every domain. In dynamic networks, topology is expected to change frequently. An algorithm A is self-stabilizing if, starting from a completely arbitrary configuration, the network will eventually reach a legitimate configuration.

Note that any self-stabilizing algorithm for the leader election problem is also an algorithm for the dynamic leader election problem, since when the topology of the network changes, we can consider that the algorithm is starting over again from an arbitrary state. There are a number of such algorithms in the literature which require large memory in each process, or which take O(n) time to converge, where n is size of the network. Given the need to conserve time, and possibly space, these algorithms may not be practical for the dynamic leader election problem.

In this thesis, three silent self-stabilizing asynchronous distributed algorithms are given for the leader election problem in a dynamic network with unique IDs, using the composite model of computation. If

topological changes to the network pause, a leader is elected for each component. A BFS tree is also constructed in each component, rooted at the leader. When another topological change occurs, leaders are then elected for the new components. This election takes O (Diam) rounds, where Diam is the maximum diameter of any component.

The three algorithms differ in their leadership stability. The first algorithm, which is the fastest in the worst case, chooses an arbitrary process as the leader. The second algorithm chooses the process of highest priority in each component, where priority can be defined in a variety of ways. The third algorithm has the strictest leadership stability; if a component contains processes that were leaders before the topological change, one of those must be elected to be the new leader. Formal algorithms and their correctness proofs will be given.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to sincerely thank Dr. Ajoy K Datta for his guidance, support, encouragement, and invaluable support throughout my Masters program and his help during my TA work for four semesters.

I would like to express my sincere gratitude for Dr. Ajoy K. Datta and Dr. Lawrence L. Larmore for their guidance and technical contributions in my thesis research.

I would also like to thank Dr. Yoohwan Kim and Dr. Emma E. Regentova for their time in reviewing my report and their willingness to serve on my committee.

My special gratitude goes to my family and my friends for always being there with me, without whose support, faith, and encouragement this work would not have been possible.

CHAPTER 1

INTRODUCTION

In this thesis, we present three silent self-stabilizing asynchronous distribution algorithms for the leader election problem in a dynamic network with unique IDs. Thus this research covers several domains of distributed computing, such as mobile ad hoc networks, self-stabilizing systems.

The leader election problem is one of the fundamental problems in distributed computing. In static networks, this problem is to select a process among all the processes in the network to be the leader. In this paper, we deal with leader election in dynamic networks, where a fault could occur, i.e., data could be corrupted, or the topology could change, even causing the network to become disconnected. In a dynamic network, the problem is modified slightly in the following manner: The goal is elect a leader for each component of the network after any number of concurrent faults.

There are several leader election algorithms for dynamic networks. However, the only self-stabilizing solutions we are aware of are presented in [49, 13]. The algorithm of [13] is simpler (both the pseudo-code and proof of correctness) and more efficient in terms of messages and message size than the solution in [49]. However, both solutions suffer from the same drawback, which is, the use of a global clock or the assumption of perfectly synchronized clocks. The following is quoted

from [13]: “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.” One goal of this paper is to solve the above open problem.

1.1 Contributions

Our algorithms have the following combination of features – they are asynchronous, self-stabilizing, and silent, converge in O(Diam) time, and use no global clock. They also use only O(1) variables per process; however, one of the variables is an unbounded integer, meaning that if the algorithm runs forever, the size of that integer grows without bound. However, as a practical matter, this is of little importance, since the size of that unbounded integer increases by at most one per step, and thus should not overflow a modest size memory, even if the algorithm runs for years.

All three of our algorithms elect a leader for each component of the network, and also build a BFS tree rooted at that leader. In each case, after a fault, each component of the network elects a leader within O(Diam) rounds, where Diam is the maximum diameter of any component, provided the variables are not corrupted.

The space and time complexities of our first algorithm, DLE, are smaller than those of [13], as DLE requires fewer variables, and converges in only Diam + 1 rounds from an arbitrary configuration,

approximately one third the time as the algorithm of [13]. As in [13], DLE picks an arbitrary process to be the leader of each component.

Our second algorithm, DLEP, picks the highest priority process of each component to be its leader. Priority of a process can be defined as a function of its ID, its local topology, or any data the process obtains from the application layer. Since the choice of leader is not arbitrary, DLEP should have greater leadership stability that DLE, i.e., there should be a tendency for leaders to remain the same after small topological changes.

Our third algorithm, DLEND, ensures even greater leadership stability than DLEP. If the network has reached a stable configuration, and if a topological change, however great, occurs in a given step, and no variables are corrupted, and then no fault occurs thereafter, every component (under the new topology) will elect an incumbent, i.e., a leader that was a leader before the topological change, if possible. In cases where a component contains no incumbent, or more than one incumbent, DLEND makes a choice based on priority in the same manner as DLEP.

DLEND has an additional stability feature, which we call no dithering. If the network has reached a stable configuration, and if a topological change, however great, occurs in a given step, and then no fault thereafter, then no process will change its choice of leader more than once.

1.2 Outline

In Chapter 2, we give an overview of the Distributed Systems, Mobile ad hoc networks (MANET), and Self-stabilizing systems.

In Chapter 3, the detailed introduction to Leader Election problem in Dynamic networks is given and then we discuss what Link Reversal algorithms are and why this technique is adopted for leader election problem. In latter sections of the Chapter we discuss few LRR algorithms like GB algorithm, TORA. In Section 3.3 and 3.4 we give a brief overview of Non Self Stabilizing and Self Stabilizing algorithms in the literature that are similar to our algorithms, we also mention other related work.

In Chapter 4, we discuss the algorithms that we proposed in our thesis.

In Chapter 5 we describe DLE the first proposed algorithm for Leader election problem, In 5.1 we give the overview of the algorithm. In the next section we present the definition of DLE that includes the variables of DLE, functions of DLE, Legitimate state of the DLE algorithm and actions of DLE. In section 5.3 we explain the algorithm DLE with sample execution. In section 5.4 we give the proofs for DLE.

In Chapter 6, we describe the Dynamic Leader election with Priority. DLEP picks the highest priority process of each component to be its leader. In section 6.1 we give the overview of DLEP. In latter sections we give the definition of DLEP in which we describe the variables, functions, configurations and actions of DLEP. In section 6.3 we explain

the algorithm with an example computation. In section 6.4 we give the proofs of DLEP.

In Chapter 7, we describe the algorithm DLEND. It ensures greater leadership stability than DLEP. Then we give the overview of DLEND. In section 7.2 we give the definition of DLEND in which we describe the variables, functions, actions of DLEND. In section 7.3 we explain DLEND with an example computation. In section 7.4 we give the proofs of DLEND.

In Chapter 8, we discuss the proofs of algorithms and in Chapter 9, concludes the thesis.

CHAPTER 2

BACKGROUND

2.1 Distributed Systems

A number of definitions have been proposed in the literature to explain the meaning of distributed systems. A distributed system is a communication network, collection of independent computers that appears to its users as a single coherent system, and it can even be a single multitasking computer [21]. Although the processors in distributed systems are autonomous in nature, they may need to communicate with each other to coordinate their actions and achieve a reasonable level of cooperation [2]. A program composed of executable statements are run by each computer. Each execution of a statement changes the computer's local memory content, hence the state of the computer. Consequently, a distributed system is modeled as a set of n state machines that communicate with each other. There are mainly two models for communications between machines; message passing and shared memory. In the message passing model, machines communicate with each other by sending and receiving messages. While in the shared memory model, communication is carried out by writing in and reading from the shared memory.

The major goals of distributed systems are:

- Distribution Transparency.
- Connecting resources and users.

- Scalability.
- Openness.

2.2 MANET

Mobile ad hoc networks are formed dynamically by an autonomous system of mobile nodes that are connected via wireless links without using the existing network infrastructure or centralized administration. In fact two or more nodes can form the mobile ad hoc network by being in their transmission range. In this type of network, communication between mobile nodes is peer-to-peer, so each node has direct communication with another. Nodes also act as relay nodes to forward data packets. This is very important part of communication technology that supports truly pervasive/ubiquitous computing, because in many contexts, information exchange among mobile units cannot rely on any fixed network infrastructure but on the rapid configuration of wireless connections on the fly [3]

MANETs are gaining momentum because they help realize network services for mobile users in areas with no pre-existing communications infrastructure, or when the use of such infrastructure requires wireless extension. Ad hoc nodes can also be connected to a fixed backbone network through a dedicated gateway device enabling IP networking services in the areas where Internet service is not available due to the lack of a preinstalled infrastructure.

Minimal configuration and quick deployment make ad hoc networks suitable for emergency situations like natural or human-induced disasters, military conflicts, emergency medical situations, etc.

2.2.1 Characteristics

- Mobile ad hoc networks involve all networking layers, ranging from the physical to application layer.
- Nodes in the ad hoc network are free to move while communicating with other nodes.
- The bandwidth available is of the order of 1Mbps, an order of magnitude less than that of wired networks.
- The communication in the network is a broadcast, which means broadcast is no more expensive than unicast.
- Mobile nodes have limited battery power.
- Wireless links are much more error prone compared to wired links.
- The topology of ad hoc network is dynamic in nature due to constant movement of the participating nodes, causing the intercommunication patterns among the nodes to change continuously.
- Every node may not be within the communication range of every other node. So, multiple hops may be needed, for this the nodes should serve as routers for other nodes in the network so that data packets can be forwarded to their destinations.

2.2.2 Issues with Ad hoc Networking

In general, mobile ad hoc networks are formed dynamically by an autonomous system of mobile nodes that are connected via wireless links without using the existing network infrastructure or any centralized administration. These networks can be called as multi-hop wireless ad hoc networks because routes between nodes in ad hoc networks may include multiple hops.

In MANET wireless link "failures" occur when previously communicating nodes move such that they are no longer within transmission range of each other. Wireless link "formation" occurs when nodes those are not in communication range move within the transmission range of each other.

Ad hoc wireless networks inherit the traditional problems of wireless communications and wireless networking (IEEE P802.11/D10, January 14, 1999.) as described below:

- The wireless medium has neither absolute, nor readily observable boundaries outside of which nodes are known to be unable to receive network frames.
- The channel is unprotected from outside signals.
- The wireless medium is significantly less reliable than the wired media.
- The channel has time-varying and asymmetric propagation properties.

- Hidden-terminal and exposed-terminal phenomena may occur.

To these problems and complexities, the multi-hop nature and the lack of fixed infrastructure add a number of characteristics, complexities, and design constraints that are specific to ad hoc networking [4, 5], and are described below:

- MANET does not depend on any established infrastructure or centralized administration. Each node operates in distributed peer-to-peer mode, acts as an independent router, and generates independent data. Network management has to be distributed across different nodes, which brings added difficulty in fault detection and management.
- Multi-Hop Routing is required, No default router is available. Every node acts as a router and forwards each other's packets to enable information sharing among mobile nodes. Routing protocols are self-starting, adapt to the changes in network conditions, and also offer multi-hop paths from a source to a destination across the network. Routing protocols designed for ad hoc networks can be adopted to greatly improve the scalability of routing protocols designed for use in the global Internet, which would be an enormous payoff for ad hoc network research. More detailed information on routing in MANET is given in [6].
- Dynamically Changing Network Topologies. In mobile ad hoc networks, nodes can move arbitrarily. So the network topology,

which is multi-hop, can change frequently and unpredictably, resulting in route changes, frequent network partitions, and possibly packet losses.

- Variation in Link and Node Capabilities. Each node may be equipped with one or more radio interfaces that have varying transmission/ receiving capabilities and operate across different frequency bands [7]. This heterogeneity in node radio capabilities can result in asymmetric links. In addition, each mobile node might have different hardware/software configuration, resulting in variability in processing capabilities. Designing network protocols and algorithms for this heterogeneous network can be complex, requiring dynamic adaption to the changing conditions.
- Energy Constrained Operation. Batteries carried by each mobile node have limited power supply, processing power is limited, which in turn limits services and applications that can be supported by each node. This becomes a bigger issue in mobile ad hoc networks because as each node is acting as both an end system and router at the same time, additional energy is required to forward packets from other nodes.
- Network Scalability. Currently, popular network management algorithms were mostly designed to work on fixed or relatively small wireless networks. Many mobile ad hoc network applications involve large networks with tens of thousands of nodes, for

example sensor networks and tactical networks [6]. Scalability is critical to the successful deployment of these networks. A network with large number of nodes and limited resources involve many challenges that are yet to be solved, such as addressing, routing, location management, configuration management, interoperability, security, high capacity wireless technologies, etc.

2.2.3 Applications of MANET

MANET has many applications, ranging from large scale mobile and highly dynamic networks, to small and static networks that are constrained by power sources. Typical application domains of MANET include commercial sector, military, battlefield, civilian environments, emergency operations, and personal area network (PAN). Some of the specific applications are mentioned below [6]:

- Conferencing. When mobile computer users gather outside their normal office environment, the business network infrastructure is often missing. The whole point of the meeting might be to make some further progress on a particular collaborative project. As it turns out, the establishment of an ad hoc network for collaborative mobile computer users is needed even when the Internet infrastructure support already exists.
- Home Networking. Consider the scenario that will result if wireless computers become popular at home. These computers will probably be taken to and from the office work environment and on

business trips. Such computers will not have topologically related IP addresses to each wireless node for identification purposes would add an administrative burden, and the alternative of deploying ad hoc networks seems more attractive.

- Emergency Services. A mobile ad-hoc network can also be used to provide crisis management services applications, such as in disaster recovery, where the entire communication infrastructure is destroyed and resorting communication quickly is crucial. By using a mobile ad-hoc network, an infrastructure could be set up in hours instead of weeks, as is required in the case of wired line communication.
- Personal Area Networks. The idea of a personal area network (PAN) is to create a much localized network populated by some network nodes that are closely associated with a single person. When people meet in real life, their PANs are likely to become aware of each other. Mobility becomes more important when interactions between several PANs are needed. Since people usually do not stay in a fixed location with respect to each other for a long time, dynamic nature of this inter-PAN communication is obvious. Ad hoc networks can be used to establish communications between node on spate PANs.
- Embedded Computing Applications. Some researches predict a world of ubiquitous computing [9], in which computers will be

around us, constantly performing mundane tasks to make our lives a little easier. These ubiquitous computers will often react to the changing environment in which they are situated and will themselves cause changes to the environment in ways that are, we hope, predictable and planned. These capabilities can be provided with or without the use of ad hic networks, but ad hoc networking is likely to be more flexible and convenient that the continual allocation and reallocation of endpoint IP address whenever a new wireless communication link is established.

- Sensor Dust. Consider a situation in which some hazardous chemicals were dispersed in an unknown manner because of an accident or explosion. Instead of sending persons who might be subjected to lethal gas and forced to work in unwieldy protective clothing, it would be better to distribute sensors containing wireless transceivers [10] [11]. The sensors could then form an ad hoc network and help in gathering the information about the accident and chemical concentrations.
- Automotive/PC Interaction. Ad hoc networks can be used to provide interactions between automotive computers and laptops or PDAs that may accompany us as we travel in our cars.
- Educational Applications. Setup ad hoc communication during conferences, meetings, or lectures.
- Commercial Environments.

- E-Commerce: e.g., Electronic payments from anywhere.
- Business: Dynamic access to customer files stored in a central location on the fly, Provide consistent database for all agents.
- Vehicular Services: Transmission of news, road condition, weather, music, road/accident guidance etc.

In spite of the various applications served by the ad hoc networks, they still have to overcome the defects such as the limited wireless transmission range, link quality, fading, noise, interference caused due to its broadcast nature, route changes and packet losses induced due to its broadcast nature, route changes and packet losses induced due to the node mobility, battery constraints, and potentially frequent network partitions. Security and interception problems are of a major concern, especially in military applications. Therefore, designing the protocol for MANET is very crucial, and these issues must be carefully examined before widespread commercial deployment.

2.3 Self-Stabilizing Systems

Now a day's Software systems are used everywhere. Thus commercially available software systems must be able to adjust to different inputs and handle different faults so that they can be used in many different environments.

Self-Stabilization is related to autonomic Computing, which entails several "self-*" attributes like; self-organized [12], self-configuration [13],

self-healing [14], and self-maintaining [15]. According to [16], research in a self-* system is “a direct response to the shift from needing bigger, faster, stronger computer systems to the need for less human-intensive management of the systems currently available. System complexity has reached the point where administration generally costs more than hardware and software infrastructure.” The goals of the self-* systems are reduction of human administration and maintenance, and an increase of reliability, availability and performance.

In 1973, Dijkstra introduced the term self-stabilization in the world of computer science [17, 18] which was a concept of fault-tolerance. Unfortunately, only a few people had become aware of its importance until Lamport endorsed this as “Dijkstra’s most brilliant work” and a “milestone in work on fault-tolerance in his invited talk at the ACM Symposium on Principles of Distributed Computing in 1983. Today it is one of the most active areas of research in the field of computer science.

A system is considered self-stabilizing if starting from any arbitrary state (possibly a fault state) it is guaranteed to converge to a legitimate state which satisfies its problem specification in a finite number of steps. Once it converges to a legitimate state, it must stay in that legitimate state thereafter unless a fault occurs. With respect to behavior, it can also be defined as a system starting from an arbitrary state, reaching a state in finite time from which it starts behaving correctly according to its

specification. This self-stabilization enables systems to recover from a transient fault automatically.

According to [19, 20], the self-stabilization can be defined in terms of two properties; closure and convergence. Closure means that if a system is in a correct (or legitimate) state, it is guaranteed to stay in a correct state, if no fault occurs. On the other hand, convergence means that starting from any arbitrary state, it is guaranteed that the system will eventually reach a correct state in finite steps. In order for a system to be self stabilizing it must satisfy both of these properties.

Self–stabilization has been extensively studied in the area of network protocols. Protocols like routing, sensor networks, high-speed networks, and connection management are just a part of many applications of self-stabilization. Also, there exist many self-stabilizing distributed solutions for graph theory problems. For example, spanning tree constructions, maximal matching, search structures, and graph coloring. Many self-stabilizing solutions for numerous classical distributed algorithms were also proposed. Those include mutual exclusion, token circulation, leader election, distributed reset, termination detection, and propagation of information with feedback [21].

In the study of self-stabilization, several aspects of models have been considered, such as the following:

- Interprocess Communication: shared registers or message passing.
- Fairness: weakly fair, strongly fair, or unfair.

- Atomicity: composite or read/write atomicity.
- Types of Daemon: central or distributed.

All together proving stabilization programs are quite challenging. Two techniques have been commonly used in research literature: convergence stair [22] and variant function [23] methods. Furthermore, many general methods of designing self-stabilizing programs have been proposed which include diffusing computation [24], silent stabilization [25], local stabilizer [26], local checking and local correction [27, 28], counter flushing [29], self-containment [30], snap-stabilization [31], super-stabilization [32], and transient fault detector [33].

Self-stabilization is a significant concept in the study of MANETs. Due to the dynamic nature of MANET topology, the protocols for setting up and organizing MANETs are desirable to self-stabilizing.

2.4 DAG

A directed acyclic graph (DAG) is a directed graph that contains no cycles. DAG is rooted at the destination means that is the node that will have only incoming links all other nodes that have incoming should have outgoing links. A rooted tree is a special kind of DAG and a DAG is a special kind of directed graph. For example, a DAG may be used to represent common sub expressions in an optimizing compiler.

CHAPTER 3

LEADER ELECTION IN DYNAMIC NETWORKS

In distributed computing leader election is an important primitive. It is useful for many applications that require the selection of a unique processor among multiple processors.

There are many applications for leader election algorithms; usually used as a primitive in other distributed algorithms.

- Primary-backup approach to replication based fault-tolerance.
- Group communication systems [35].
- For video conferencing.
- Multiplayer games [36].
- Leader election is required when a mutual exclusion application is blocked because of the failure of a token holding node.
- It is required in key distribution and management [37], and routing coordination [38, 39, 40].

Leader Election Problem:

The leader election problem is one of the fundamental problems in distributed computing, whether wired or wireless, especially when failures are common.

The leader election problem for static networks is [41]: *Every network should eventually have a unique leader.*

However, in dynamic networks due to node mobility and link failures partitions can occur and leader will not be elected until a

partitioning is detected. And sometimes, two network components may merge and temporarily there may be two leaders in the newly formed network or there can be a period where there is no leader in the component. Thus the leader election problem definition in mobile ad hoc networks should be slightly modified: *Every connected component will eventually have a unique leader, even after any number of concurrent faults.*

There are many leader election algorithms given, but most of them are not self-stabilizing or few doesn't work when there are concurrent changes in the network.

Some leader election algorithms are based on TORA [42] and few are based on diffusing computation.

We discuss leader election algorithms based on TORA which is a routing algorithm for mobile ad hoc networks. TORA in turn is based on a loop-free routing algorithm of Gafni and Bertsekas, which are based on Link Reversal algorithms.

3.1 Link Reversal Routing:

Link reversal routing is a highly adaptive form of routing originally intended for use in networks with rapidly changing topologies [46]. A key concept behind LRR is the decoupling of far-reaching control message propagation from the dynamics of the network's topology. It will be appropriate for use in networks where the rate of topological change is

not as fast as to make flooding the only possible routing method and not so slow to make algorithms capable of supporting a shortest path computation applicable.

The main objective of LRR is to minimize the amount of routing overhead that must be exchanged between nodes when reacting to changes in a network topology to a greatest extent possible. This is possible by localizing the algorithm's reaction to topological changes. Instead of maintaining the shortest path routing computation for a destination in the network, LRR will maintain only the state sufficient to constitute a directed acyclic graph (DAG) rooted at the destination. DAG is a loop free routing and can provide nodes in the network with multiple, redundant routes to the destination. The nodes don't know the distance from the destination or anything about the nodes in the network other than their one-hop neighbors. This differentiates the LRR from other routing techniques. It is very difficult for a node to continuously estimate its shortest distance to the destination. So this algorithm may not give the shortest path route it may result in less optimal routing but it is very efficient in terms of routing overhead communication complexity, and hence it is very adaptive and scalable.

Link failures are very common in dynamic networks and we need to update the link state and find the new shortest path and must be communicated to all the nodes for which that link forms a part of their shortest path spanning tree. For LRR, the reaction to a link failure is

limited to the set of nodes that lost their last outgoing link to the destination because of the failed link. Thus the redundancy in the routing DAG minimizes the frequency and scope of algorithmic reactions due to link failures.

3.1.1 Gafni-Bertsekas Algorithm:

This is a highly adaptive loop-free multipath routing algorithm based on LRR. Given a connected, destination-disoriented DAG, transform it into a destination-oriented DAG by reversing the direction of some of its links. A DAG is destination oriented only when every node has a directed path originating at the node and terminated at the destination. Otherwise a DAG is destination disoriented if any node other than the destination node has no outgoing link.

There are 2 algorithms to solve this problem Full Reversal method and Partial Reversal method [44].

Full Reversal, in this if a node doesn't have outgoing link then it will reverse the direction of its entire links. Thus no list is required here.

Partial Reversal Method:

Every node i in the network other than the destination, will maintain the list of its neighboring nodes j that have reversed the direction of the corresponding links (i,j). At each iteration the node that doesn't have any outgoing link will reverse the direction of links (i,j) for all j that do not appear on its list and empties the list. If the list is full i.e.

there is no such j then node i reverses the direction of all incoming links and empties the list.

Partial reversal method is improved to a “height based” form from a list-based. In this algorithm each node maintains a height variable, drawn from a totally ordered set. Every node i will have height variable associated with it, which is a triple (α_i, β_i, i), where α_i and β_i are integers.

Let N denotes the set of nodes in the network. The initial set of triples $\{(\alpha_i^0, \beta_i^0, i) \mid i \in N\}$ satisfies $\alpha_i^0 = 0$ for all i, and for any link (i, j) we have $(\alpha_i^0, \beta_i^0, i) > (\alpha_j^0, \beta_j^0, j)$ if and only if link (i, j) in the initial DAG is directed from i to j.

Assuming that each node ID i is unique, the set of triples form a total order and the directed graph is loop free regardless of the values of α_i and β_i. The links are directed on the basis of the relative heights of neighboring nodes, i.e. from higher to lower. The parameter α_i represents the reference level and β_i and i are used to differentiate the heights with common reference level. The algorithm is triggered when ever one or more nodes lose all their outgoing links.

Let N_i denote the set of one-hop neighbors of node i. The kth iteration is implemented as follows:

A node i, other than the destination, for which $(\alpha_i^0, \beta_i^0, i) < (\alpha_j^0, \beta_j^0, j)$, $\forall j \in N_i$, increases $\alpha_i^k \{\alpha_j^k \mid j \in N_i\} + 1$

and sets $\beta_i^{k+1} = \min\{ \beta_j^k \mid j \in N_i, \alpha_i^{k+1} = \alpha_j^k\} - 1$

if there exists a neighbor j with $\alpha_i^{k+1} = \alpha_j^k$

otherwise, $\beta_i^{k+1} = \beta_i^k$.

The rule for setting α_i^{k+1} ensures that node i will have at least one outgoing link, i.e., that (α_i^{k+1}, β_i^{k+1}, i) will be larger than the height of at least one neighbor with the smallest height. The rule for setting β_i^{k+1} tries to limit the number of links incident on i that will have their direction reversed, by keeping i's height smaller than that of any neighbors whose α height component is not smaller than α_i^{k+1}. Reducing the number of links whose direction changes limits the propagation of height changes.

The GB algorithm is deadlock free and loop free at all times. The destination node is the lowest numbered node. As long as the network remains connected, the GB algorithm converges after a finite number of iterations but the algorithm is unstable and never converges in the network portions that are disconnected from the destination.

3.1.2 TORA

Park and Corson adapted the GB algorithm for routing in mobile ad hoc networks, calling it as TORA (Temporally Ordered Routing Algorithm) [42].

In TORA there is a mechanism for detecting the network partition where the destination is no longer reachable, where as GB algorithm would have caused infinite number of messages. The protocol is adaptive, reasonably efficient, and scalable, making it potentially well suited for use in large, dynamic networks.

In TORA the height of node i is a 5-tuple, (λ_i, oid_i, r_i, δ_i, i) from a totally ordered set and links between the nodes are logically considered to be directed from higher to lower heights. From left to right the first 3 components in the height form a reference level, the next one is a delta component, and the next is the node's id, which is unique for every node in the network.

A new reference level is started by node i if it loses its last outgoing link due to a link failure. As nodes lose outgoing links new reference levels are propagated throughout the connected component in search of the alternate directed path to the destination. Whenever a node becomes sink it will increase its reference level and propagates the reference level in the component.

The search for the destination is started by setting λ_i to the time when a node started the new reference level initially we assume that all nodes have synchronized clocks and oid_i is set to i, the originator of this reference level, this ensures that the reference levels can be totally ordered lexicographically, even if multiple nodes define new reference levels because of the link failures that occur simultaneously. The third component r_i is zero initially it is used to detect the partition in the network, when one section of the network is dead-end and it can't find the destination in that direction then it will set the r_i to 1 and the reference level is reflected back to towards the originator, When the originator receives the reflected reference levels back from all its

neighbors then it has identified a partition from the destination. δ value is used to order the nodes that have the common reference level.

Each node i maintains its height H_i with respect to the destination. Each node i that has no outgoing links will modify its height $H_i = (\lambda_i, oid_i, r_i, \delta_i, i)$ as follows.

Case 1: If, because of a link failure, node i has no outgoing links, it modifies its height as

$$(\lambda_i, oid_i, r_i) = (t, i, 0)$$

$$(\delta_i, i) = (0, i)$$

t is the time of the failure.

Case 2: If, because of a link reversal the node i lost it's all outgoing links and the reference levels of its neighbors are not equal. i.e. $((\lambda_j, oid_j, r_j)$ are not equal for all $j \in N_i$, then node i modifies its height as

$$(\lambda_i, oid_i, r_i) = \max \{((\lambda_j, oid_j, r_j) \mid j \in N_i\},$$

$$(\delta_i, i) = (\min \{\delta_j \mid j \in N_i \text{ with } (\lambda_j, oid_j, r_j) = \max \{(\lambda_j, oid_j, r_j)\}\} - 1, i)$$

Here node i will choose the highest reference level of all the neighbors and selects the height lower than that of all the neighbors with that reference level.

Case 3: If, because of a link reversal node i loses all the outgoing links and the reference levels of the neighbors are equal i.e. (λ_j, oid_j, r_j) for all $j \in N_i$ are equal with $r_j = 0$, then node i modifies its height as

$$(\lambda_i, oid_i, r_i) = (\lambda_j, oid_j, 1),$$

$$(\delta_i, i) = (0, i)$$

Here if a node receives same reference levels from all the neighbors then the node reflects back the same reference level by setting $r_i = 1$. The reflected reference level is propagated back towards the node that originally defined the reference level.

Case 4: If, because of a link reversal node i lost all the outgoing links and the reference levels

(λ_j, oid_j, r_j) are equal with $r_j = 1$ for all $j \in N_i$, and $oid_j = i$, then i modifies its height as

$$(\lambda_i, oid_i, r_i) = (\_, \_, \_),$$

$$(\delta_i, i) = (\_, i)$$

Here the node i has detected a partition and i must initiate the process of erasing the invalid routes, routes that are not routed at the destination.

Case 5: If, because of a link reversal node i lost all the outgoing links and the reference levels (λ_j, oid_j, r_j) are equal with $r_j = 1$ for all $j \in N_i$, and $oid_j \neq i$, i.e., node i did not define the level, then i modifies its height as

$$(\lambda_i, oid_i, r_i) = (t, i, 0),$$

$$(\delta_i, i) = (0, i)$$

Here node i did not define the reference level itself it experienced a link failure between the time it propagated the reference level and the time it got the reflected reference level from all the neighbors. This is not necessarily an indication of a partitioning of the component. Therefore, the node starts a new reference level. This case occurs only when a link fails while the system is recovering from an earlier link failure.

3.4 Non Self Stabilizing Leader Election Algorithms:

Malpani et al. [1] have adapted the Temporally Ordered Routing Algorithm (TORA) to elect a unique leader in each network component. They have proposed 2 leader election algorithms, the first one is for one topological change and the second one is for concurrent topological changes. One topological change means a new topological change occurs only after the algorithm has terminated its execution of current topology change and concurrent topology changes means changes can occur at any time. First algorithm is proved to elect a unique leader, if the network stabilizes for a sufficiently long time. Whereas the second algorithm doesn't have any proof of correctness.

Malpani modifies the TORA to leader election algorithm by changing few things as follows:

In Malpani's algorithm the height of each node is a 6-tuple,

(lid_i, λ_i, oid_i, r_i, δ_i, i). The first component is the id of the node considered to be the leader of i's component. The remaining 5 components are same as in TORA.

Instead of having a single destination-oriented DAG as in TORA, each component eventually forms a leader-oriented DAG. The reference level (- 1, - 1, - 1) is used by the leader of the component to ensure that it is a sink.

When a node has no outgoing links as well as no incoming links then that node elects itself as a leader. The height variable of a leader node is (i, -1, -1, -1, 0, i)

In TORA the node that detected the partition from destination sends out indications to the other nodes about the partition so that they will cease performing height changes and stops sending useless messages to reach the destination. In Malpani's algorithm, the node that detects the partition elects itself as the leader of the new component and it transmits this information to its neighbors, who in turn propagates this information to their neighbors and so on. Eventually all the nodes in the component will become aware of the new leader. When two components merge because of a new link formation, then the merged component may end up having more than one leader. In that case the leader of one of the component that has the lower identification number becomes the leader of the new component.

When node i has no outgoing links due to a link reversal following reception of an Update message and the reference levels (λ_j, oid_j, r_j) are equal with $r_j = 1$ for all $j \in N_i$ and $oid_j = i$:

$lid_i = i$

$(\lambda_i, oid_i, r_i) = (-1, -1, -1)$

$\delta_i = 0$

When node i receives an Update message from neighboring node j such that $lid_j \neq lid_i$:

if lidi > lidj or (oidi = lidj and ri = 1) then

$lid_i = lidj$

$(\lambda_i, oid_i, r_i) = (0, 0, 0)$

$\delta_i = \delta_j + 1$

3.5 Self Stabilizing Leader election Algorithms:

There are several leader election algorithms for dynamic networks. However, the only self-stabilizing solutions we are aware of are presented in [49, 13]. The algorithm of [13] is simpler (both the pseudo-code and proof of correctness) and more efficient in terms of messages and message size than the solution in [49]. However, both solutions suffer from the same drawback, which is, the use of a global clock or the assumption of perfectly synchronized clocks. The following is quoted from [13]: "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." One goal of this paper is to solve the above open problem.

Furthermore, in the execution of the algorithm of [13], a process could change its choice of leader many times. Our third algorithm, DLEND, has the property that if a topological changes, however great, occurs, but if no variables are corrupted, no process changes its choice of leader more than once.

There are number of stabilizing leader election algorithms for static networks in the literature. Arora and Gouda [47] present a silent leader election algorithm in the shared memory model. Their algorithm requires O(N) rounds and O(logN) space, where N is a given upper bound on n, the size of the network. Dolev and Herman [32] give a non-silent leader election algorithm in the shared memory model. This algorithm takes O(Diam) rounds and uses O(N logN) space. Awerbuch et al. [48] solve the leader election problem in the message passing model. Their algorithm takes O(Diam) rounds and uses O(logDlogN) space, where D is a given upper bound on the diameter. Afek and Bremler [46] also give an algorithm for the leader election problem in the message passing model. Their algorithm takes O(n) rounds and uses O(log n) bits per process, where n is the size of the network. They do not claim that their algorithm works under the unfair daemon. In [32], we gave a uniform self-sabilizing leader election algorithm. This algorithm works under an arbitrary, i.e., unfair scheduler (daemon). The algorithm has an optimal space complexity of O(log n) bits per process. From an arbitrary initial configuration, the algorithm elects the leader and builds a BFS tree rooted at the leader within O(n) rounds, and is silent within O(Diam) additional rounds, where Diam is the diameter of the network. The algorithm does not require knowledge of any upper bound on either n or Diam.

CHAPTER 4

MODEL

Our algorithms have the following combination of features, they are asynchronous, self-stabilizing, and silent, converge in O(Diam) time, and use no global clock. They also use only O(1) variables per process; however, one of the variables is an unbounded integer, meaning that if the algorithm runs forever, the size of that integer grows without bound. However, as a practical matter, this is of little importance, since the size of that unbounded integer increases by at most one per step, and thus should not overflow a modest size memory, even if the algorithm runs for years.

All three of our algorithms elect a leader for each case, after a fault, each component of the network elects a leader within O(Diam) rounds, where Diam is the maximum diameter of any component, provided the variables are not corrupted.

We are given a connected undirected network, G = (V, E) of |V| = n processes, where n ≥ 2, and a distributed algorithm A on that network. Each process x has a unique ID, x.id. By an abuse of notation, we will identify each process with its ID.

A self-stabilizing [18, 21] system is guaranteed to converge to the intended behavior in finite time, regardless of the initial state of the system. In particular, a self-stabilizing distributed algorithm will eventually reach a legitimate state within finite time, regardless of its

initial configuration, and will remain in a legitimate state forever. An algorithm is called silent if eventually all execution halts.

We use the composite atomicity model of computation, where each process has variables. Each process can read the values of its own and its neighbors', but can only write to its own variables. Each transition from a configuration to another, called a step of the algorithm, is driven by a scheduler, also called a daemon.

The program of each process consists of a finite set of actions of the following form: < label >< informal name >< guard > → < statement >. We list the program of DLE, DLEP, and DLEND in Tables 1, 2, and 3, respectively. For each action, the label is listed in the first column, and an informal name is listed in the second column. The third column (guard) contains a list of clauses, all of which must hold for the action to execute, and the fourth column contains the statement of the action. The guard of an action in the program of a process x is a Boolean expression involving the variables of x and its neighbors. The statement of an action of x updates one or more variables of x. An action can be executed only if it is enabled, i.e., its guard evaluates to true.

In Tables 2, and 3, we assign a priority, a positive integer, to each action. The guard of each action is the conjunction of the clauses in the third column, together with the condition that no earlier (in terms of priority) action is enabled.

A process is said to be enabled if at least one of its actions is enabled. A step $\gamma_i \rightarrow \gamma_{i+1}$ consist of one or more enabled processes executing an action. The evaluations of all guards and executions of all statements of those actions are presumed to take place in one atomic step composite atomicity [21]. All three of our algorithms are uniform, i.e., every process has the same program.

When a process x executes the statement of an action, there could be neighbors of x that are executing statements during the same step. We specify that x uses the current values of its own variables (which could have just been changed during the current step), but old values of its neighbors' variables, i.e., values before the current step.

We use the distributed daemon. If one or more processes are enabled, the daemon selects at least one of these enabled processes to execute an action. We also assume that daemon is unfair, i.e., that it need never select a given enabled process unless it becomes the only enabled process.

We define a computation to be a sequence of configurations $\gamma_p \rightarrow \gamma_{p+1}.... \rightarrow \gamma_q$ such that each $\gamma_i \rightarrow \gamma_{i+1}$ is a step.

We measure the time complexity in rounds [21]. The notion of round [21], captures the speed of the slowest process in an execution. We say that a finite computation $б = \gamma_p \rightarrow \gamma_{p+1} \rightarrow .... \rightarrow \gamma_q$ is a round if the following two conditions hold:

1. Every process x that is enabled at γ_p either executes or becomes neutralize during some step of б. We say that a process x is neutralized at a step $\gamma \rightarrow \gamma'$ if x is enabled at γ and not enabled at γ', but x does not execute during that step.
2. The computation $\gamma_p \rightarrow .... \rightarrow \gamma_{q-1}$ does not satisfy condition 1.

We call a computation of positive length which fails to satisfy condition 1 an incomplete round.

We define the round complexity of a computation to be the number of disjoint rounds in the computation. More formally, we say that a computation $\gamma_p \rightarrow .... \rightarrow \gamma_q$ has round complexity m if there exist indices $p = i_0 < i_1 < .... < i_{m-1} < q$ such that

1. $\gamma_{ij+1} \rightarrow .... \rightarrow \gamma_{ij}$ is a round for all $1 \leq j < m$,
2. $\gamma_{im-1} \rightarrow .... \rightarrow \gamma_q$ is either a round or an incomplete round.

We remark that an incomplete round could have infinite length, since the unfair daemon might never select an enabled process. But this cannot happen for the algorithms given in this paper. We will show that every computation of each of our algorithms is finite, i.e., all the proposed algorithms in this paper “work” under the unfair daemon.

CHAPTER 5

DYNAMIC LEADER ELECTION

Our first algorithm, DLE, is somewhat similar to the algorithm of [9], although it is asynchronous and works under the unfair daemon. The basic idea is that every process that detects that it cannot possible be part of what will become a correct BFS tree declares itself to be a leader. When several processes in a component declare themselves to be leaders, one of them will capture the component.

5.1 Overview of DLE

Every process x has a leadership pair, (x.nlp, x.leader), indicating that x has chosen the process whose ID is x.leader as its leader. The number, x.nlp is called a negative leadership priority.

When a process l declares itself to e a leader, it chooses a priority number that is higher than the priority number of its previous leader; but it chooses a priority number of its previous leader; but it stores the negative number, because we want the smallest leadership pair to have priority.

In a legitimate (final) configuration, all processes in any one component C of the network have the same leadership pair, (nlp, l_c), where l_c is some process in the component, the leader of C. In addition, there is a BFS tree of the component rooted at l_c. Each process x has a

pointer to its parent in the BFS tree, as well as a level variable, whose value is the distance from x to l_c.

The basic technique of the algorithm is flooding. Under certain conditions, a process declares itself to be a leader by executing Action A1 (as listed in Table 1), creating a new leadership pair with declared leader then attempts to capture the entire component by flooding its leadership pair. The smallest (using lexical ordering), i.e., highest priority, leadership pair captures the entire component, and the algorithm halts.

The reason a new leader picks a higher priority than its old leader is that, because of deletion of links, it is possible that the old leader is no longer in the same component. Giving priority to the "youngest" leader guarantees that the leader of highest priority is in the component, provided at least one round has elapsed since any link was deleted.

5.2 Definition of DLE

5.2.1 Variables of DLE

For any process x, we have variables:

1. x.id, the ID of x. We assume that IDs are unique, and that they form an ordered set. That is, if x, y, and z are distinct process, then either x.id < y.id or y.id < x.id; and if x.id < y.id and y.id < z.id, then x.id < z.id.

   By an abuse of notation, we will use the same notation to refer to a both a process and its ID.

2. x.leader, the process that x has selected to be its leader, which we call the leader of x.
3. x.level, a non-negative integer which, in a legitimate configuration, must be the distance from x to x.leader.
4. x.nlp, a non-positive integer called the negative leader priority of x. The value of x.nlp is the negative of the priority that x.leader assigned to itself when declared itself to be a leader. The value of x.nlp is not bounded; however, in practice, it will not overflow the memory of a process, even if the algorithm runs for years.
5. x.vector = (x.nlp, x.leader, x.level), the vector of x. Vectors are ordered lexically.

   Although we list x.vector as a variable, it is actually an ordered tuple of other variables, and hence requires no extra space.
6. x.parent, the parent of x. In a legitimate configuration, if x is not the leader of its component, x.parent is that neighbor of x which is in the BFS tree rooted at the leader.

   If x is the leader, then x.parent = x.

   Because of a fault, x.parent might not be the ID of x or of any neighbor of x. In this case, we say that x.parent is unlawful.

5.2.2 Functions of DLE

Let N(x) be the set of neighbors of x, and U(x) = N(x) U {x}.

1. If v = (nlp, l, d) is a vector, we define successor (v) = (nlp, l, d+1), the smallest vector that is larger than v.

2. Min_Nbr_Vector(x) = min {y.vector: y ϵ U(x)}, the minimum neighborhood vector of x.
3. Local_Minimum(x), Boolean, meaning that x is local minimum and its own leader, and also a local root, i.e., x.leader = x.id, x.level = 0, and x.leader = x.parent = x.
4. Good_Root(x), Boolean, meaning that x is a local minimum and its own leader, and also a local root, i.e., x.leader = x.id, x.level = 0, and x.leader = x.parent = x.
5. Good_Child(x), x is a good child, meaning that x.parent.vector = Min_Nbr_Vector(x) and x.vector = successor(Min_Nbr_Vector(x)).
6. Parent(x) = p ϵ N(x) such that x.vector = successor(p.vector). If there is no such neighbor of x, define Parent(x) = x.

5.2.3 Legitimate state of the Algorithm DLE

A configuration of the network is legitimate if

1. For any component C of the network, there is exactly one process, $l_c \epsilon$ C which is a good root, and every other process in C is a good child.
2. For any component C of the network, x.vector = (l_c.nlp, l_c, d(x, l_c)) for all x ϵ C, where d is (hop) distance between processes. That is, all processes in C have the same leadership pair and level equal to the distance between processes. That is, all processes in C have the same leadership pair, and level equal to the distance to the leader of component.

5.2.4 Actions of DLE

The program of DLE is given in Table 1 as a list of actions.

Table 1: Program of DLE for Process x

A1	Reset	Local_Minimum(x)	→	x.nlp ← x.nlp – 1
		⌐Good_Root(x)		x.leader ← x.id
				x.level ← 0
				x.parent ← x
A2	Attach	⌐Local_Minimum(x)	→	Vector(x) ←
		⌐Good_Child(x)		successor (Min_Nbr_Vector(x))
				x.parent ← Parent(x)

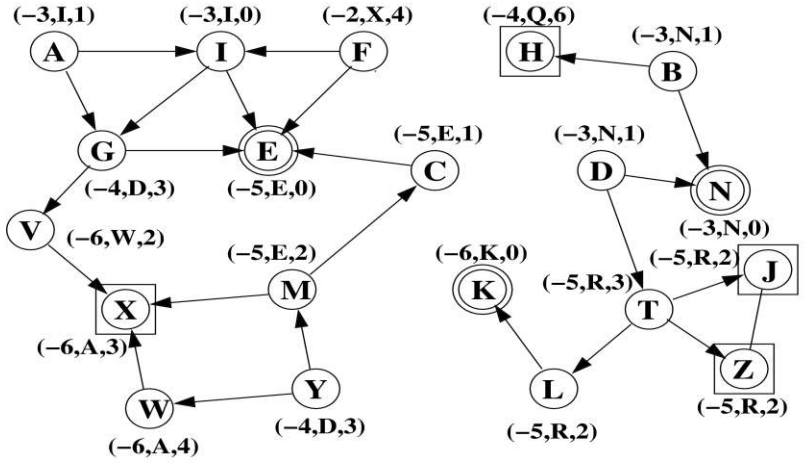
Table 1

5.3 Example Execution

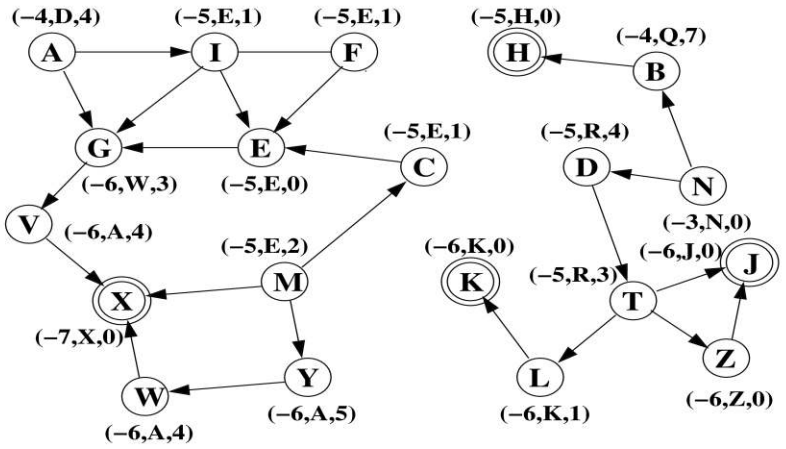
Figure 5.1 shows an example calculation. For simplicity in this example, we let the adversary select every enabled process at each step.

Each process x is labeled with its vector, x.vector = (x.nlp, x.leader, x.level). The start configuration, shown in (a), has three good roots, enclosed by squares. The bad minima execute Action A1 of Table 1. and the smallest new leadership pair floods the component, by all other

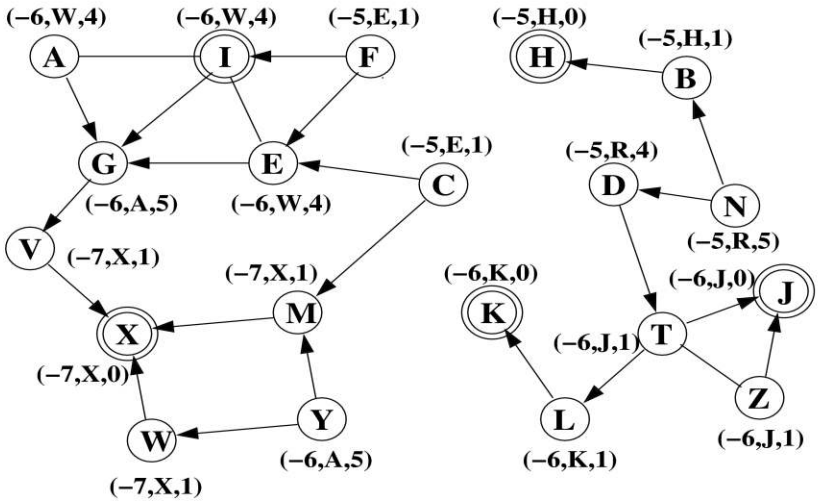
processes executing Action A2. The resulting configuration, shown in (g), is legitimate.



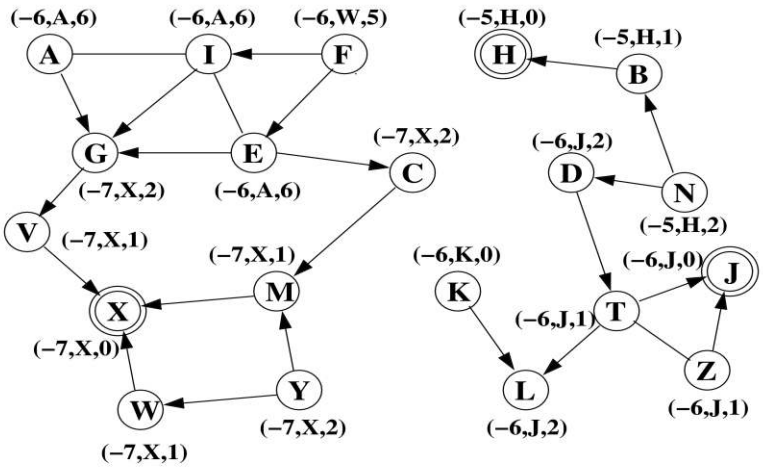
(a)



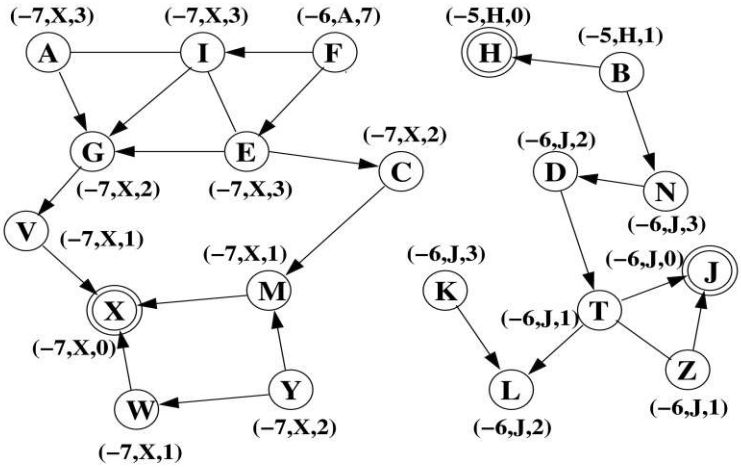
(b)


(−6,W,4)
(−6,W,4)
(−5,E,1)
A
I
F
(−5,H,0)
H
(−5,H,1)
B
G
E
(−5,E,1)
C
(−5,R,4)
D
N
(−6,A,5)
(−6,W,4)
(−5,R,5)
V
(−7,X,1)
(−7,X,1)
(−6,K,0)
(−6,J,0)
J
M
K
X
(−6,J,1)
T
(−7,X,0)
Z
L
Y
(−6,J,1)
W
(−6,A,5)
(−6,K,1)
(−7,X,1)

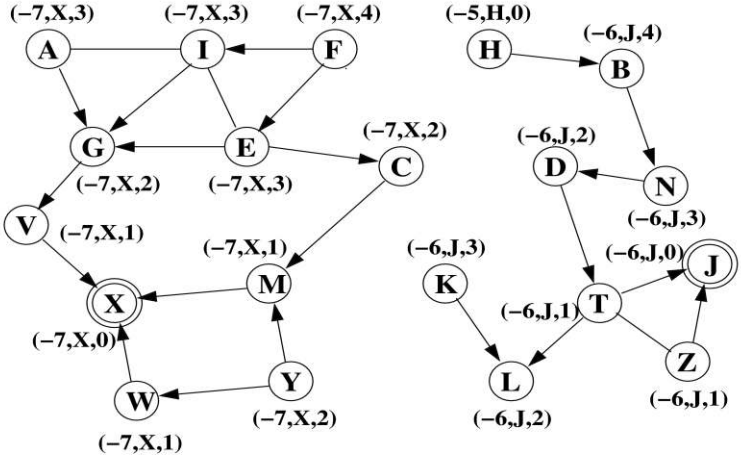
(c)


(−6,A,6)
(−6,A,6)
(−6,W,5)
(−5,H,0)
A
I
F
H
(−5,H,1)
B
G
E
(−7,X,2)
C
(−6,J,2)
D
N
(−7,X,2)
(−6,A,6)
(−5,H,2)
V
(−7,X,1)
(−7,X,1)
(−6,K,0)
(−6,J,0)
J
M
K
X
(−6,J,1)
T
(−7,X,0)
Z
L
Y
(−6,J,1)
W
(−7,X,2)
(−6,J,2)
(−7,X,1)

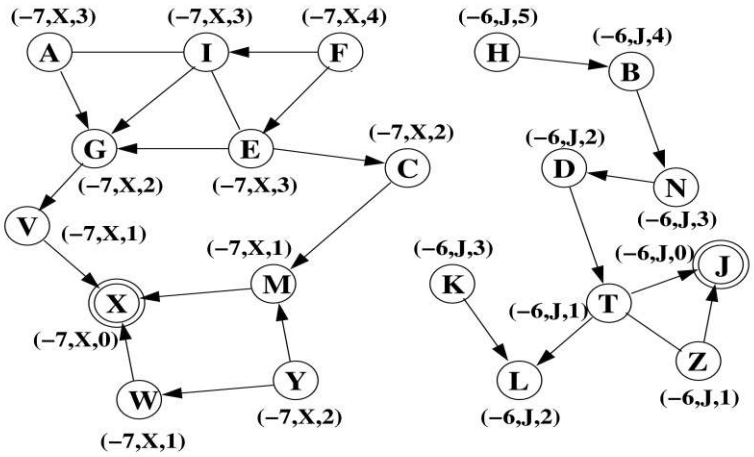
(d)


(−7,X,3)
(−7,X,3)
(−6,A,7)
(−5,H,0)
(−5,H,1)
A
I
F
H
B
(−7,X,2)
(−6,J,2)
G
E
C
D
N
(−7,X,2)
(−7,X,3)
(−6,J,3)
V
(−7,X,1)
(−7,X,1)
(−6,J,3)
(−6,J,0)
J
M
K
X
(−6,J,1)
T
(−7,X,0)
Z
L
Y
(−6,J,1)
W
(−7,X,2)
(−6,J,2)
(−7,X,1)

(e)


(−7,X,3)
(−7,X,3)
(−7,X,4)
(−5,H,0)
(−6,J,4)
A
I
F
H
B
(−7,X,2)
(−6,J,2)
G
E
C
D
N
(−7,X,2)
(−7,X,3)
(−6,J,3)
V
(−7,X,1)
(−7,X,1)
(−6,J,3)
(−6,J,0)
J
M
K
X
(−6,J,1)
T
(−7,X,0)
Z
L
Y
(−6,J,1)
W
(−7,X,2)
(−6,J,2)
(−7,X,1)

(f)



(g)

Figure 5.1:

Example execution starting from an arbitrary state. (a) shows a configuration with two components and ten different leadership pairs. In (b), the four bad minima of (a), namely H, X, J, and Z, have executed Action A1 of Table 1, creating four new leadership pairs. After flooding, the configuration is legitimate, with surviving leadership pairs (−6, J) and (−7, X), as shown in (g).

5.4 Proofs for DLE

We define a configuration to be clean if it has no bad minima. Within one round after the cessation of faults, the configuration will be clean, since every bad minimum will declare itself to be a leader, and there is no action of any process which can create a bad minimum.

If the configuration is clean, each component can contain any number of leadership pair in a component C. Then (nlp, l) will flood the component, and eventually the leadership pair of every process in C will be (nlp, l).

When all components have done this, the configuration is legitimate.

Lemma 5.1 If at least one round has elapsed since the most recent link change, the configuration is clean.

Proof: No execution by any process can cause any process become a bad minimum, or to acquire an unlawful parent. Within one round, all bad minima will execute Action A1, declaring themselves to leaders and configuration will be clean.

Lemma 5.2 If the configuration is not legitimate, some process is enabled.

Proof: We break the proof into three cases, either the configuration is not clean, or there is some component which has more than one leadership pair, or there is some process x such that x.level is not equal to the distance from x to x.leader.

If a process x is a bad minimum or has an unlawful parent, then x is enabled to execute Action A1 or A2.

If there is more than one leadership pair in a component, then choose neighboring processes x and y which have different leadership pairs. Without loss of generality, x.vector > y.vector. If x.nlp > y.nlp, or if x.nlp = y.nlp and x.leader > y.leader, then x.vector > successor (y.vector), and x is enabled to execute Action A2.

If there is only one leadership pair in each component, then there is some process such that x.level is not the distance to its leader. We break into cases. Suppose there is some process x such that x.level is

larger than the distance from x to x.leader. Choose such an x which is closest to its leader. Then x must have a neighbor y which is closer to the leader has the correct value of y.level. Since x.level > y.level + 1, x is enabled to execute Action A2. The other case is that there is some process x such that x.level is smaller than the distance from x to x.leader. Pick such an x whose value of x.level is minimum. Then y.level ≥ x.level for all y ϵ N(x), and thus x is a bad minimum, and can execute Action A1.

Let Diam be the maximum diameter of any component.

Lemma 5.3 If at least Diam rounds have elapsed since the configuration was clean, and no further faults have occurred, then the configuration is legitimate.

Proof: No new leadership pairs will be created, since no process can execute Action A1.

For any component C, let (nlp, l) be the minimum leadership pair present in component C when the configuration is first clean. We claim that l must be a process of C. Suppose l is not in C. Let x be the process in C whose vector is minimum. Then x.leader = l. If x ≠ l, then x is a bad minimum, contradiction.

We also claim that x.vector ≥ (nlp, l, d(x, l)) for any x ϵ C, since otherwise, the component would contain a bad minimum.

For any 0 ≤ t ≤ Diam, within t rounds after the configuration is first clean, all processes with distance t of l will have their final vectors; i.e., if

d(x, l) ≤ t, then x.vector = (nlp, l, d(x,l)). Thus within Diam rounds after the configuration is clean, i.e., after at most Diam + 1 rounds altogether, the configuration will be legitimate.

Theorem 5.4 From an arbitrary configuration, DLE converges to a legitimate configuration and is silent within Diam + 1 rounds, where Diam is the largest diameter of any component of the network.

Proof: Convergence follows from Lemma 5.1 and 5.3. Silence follows from that fact that if the configuration is legitimate, no process is enabled.

5.4.1 The Unfair Daemon

A distributed algorithm might be proved to converge in a finite number of rounds, but still possibly never converge under the unfair daemon, since that unfair daemon by proving that every computation of DLE is finite.

Lemma 5.5 Every computation of DLE is finite.

Proof: Our proof is by contradiction. Suppose that $\gamma_0 \rightarrow \gamma_1 \rightarrow \ldots \rightarrow \gamma_t \rightarrow \gamma_{t+1} \rightarrow \ldots$ is an infinite computation of DLE. Since there are only finitely many processes, some process must execute infinitely often.

If x.parent is unlawful, then x.parent becomes lawful the first time x executes, and never again becomes unlawful. Thus, there can be only finitely many steps of the computation to begin at a configuration after that step, and thus, without loss of generality, there is no step at which x.parent changes from unlawful to lawful for any x. Thus, at any step where any process x executes, x.vector decreases.

A process x can execute Action A1 only if it is a bad minimum. There is no action of any process that can cause x to become a bad minimum; thus, each process can execute A1 at most once. Thus, without loss of generality, our computation does not contain any instance of Action A1, and thus no new leadership pairs are created.

There can be at most n leadership pairs at γ_0. Let x be a process that executes an action infinitely many times. Since x.vector decreases at each such execution, there must be some configuration γ_m after which x does not change its leadership pair. Let d be the value of x.level at γ_m. Since x.level must decrease every time x executes after γ_m, there can be at most d remaining steps at which x executes an action, contradiction.

Theorem 5.6 DLE converges to a legitimate state under the unfair daemon, and is silent.

Proof: By lemma 5.5, every computation of DLE must eventually reach a configuration at which no process is enabled. By Lemma 5.2, this configuration is legitimate.

CHAPTER 6

DYNAMIC LEADER ELECTION WITH PRIORITY

Algorithm DLE, given in Chapter 5, selects an arbitrary member of each component to be a leader of that component. In this section, we introduce the requirement that the elected leader of each component be the best process in the component, where "best" can be defined any number of ways, depending on the application. Our method is to define some kind of priority measure on all processes, and then make sure that the elected leader is the Process which has the highest priority in the component. For example, the process of highest priority could be the process of least ID, or of greatest ID, or of greatest degree, i.e., number of neighbors.

If no fault occurs for O(Diam) rounds, DLE always chooses a leader for each component, but this leader could be any process of the component. This leader is likely not to have been a leader before the fault; it is even possible that the loss of one link of a component could cause the component to elect a new leader, even if no processes of the component were lost and no new processes were added. We show an example of this in Figure 6.1. This behavior could be undesirable in practice. If we define priority of processes in such a way that it is largely unaffected by small faults, we will decrease the frequency of leadership changes in practice.

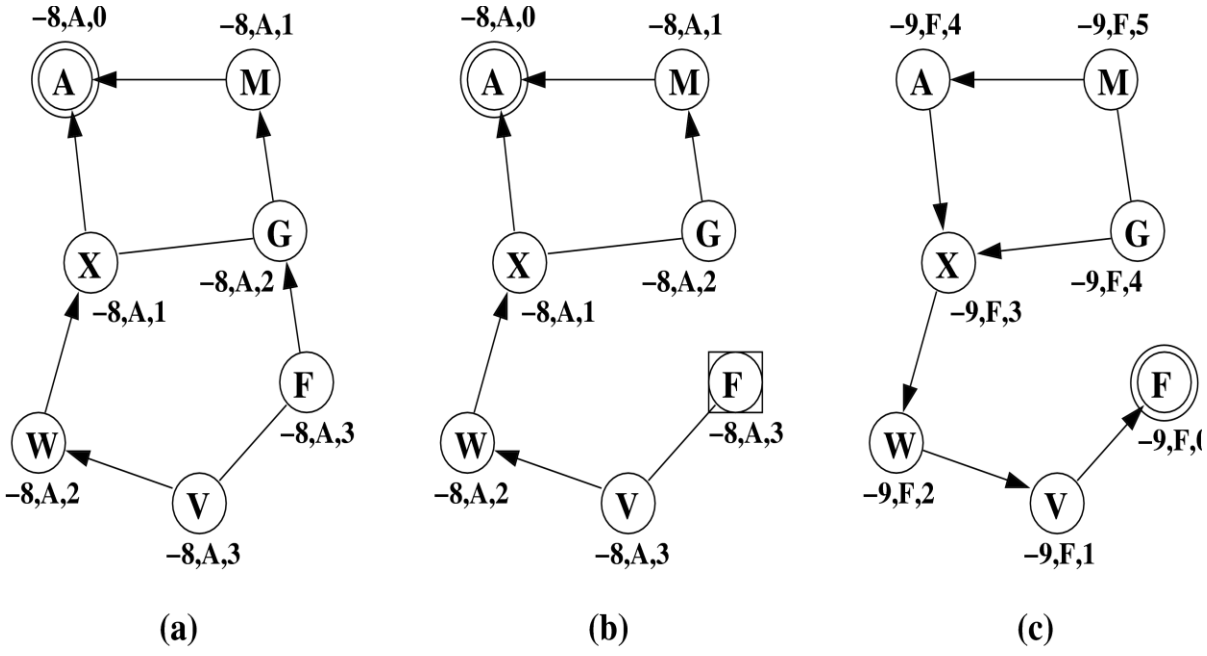


Figure 6.1:

In DLE, the loss of just one link can cause a change of leader. A legitimate configuration is shown in (a). The link between F and G is lost in (b), and F is the sole bad minimum. F declares itself to be a leader, and within six rounds, the configuration is once again legitimate, and F is the leader, as shown in (c).

6.1 Overview of DLEP

We assume an abstract function Priority of a process x, which may depend only on the topology of the network, the ID of x, and data obtained by x from the application layer. In other words, Priority(x) is not affected by any change of the variables of our algorithm. We also assume that priority(x) can be computed by x in O(1) time.

If S is any non-empty set of processes, define Max_Priority(S) = max {Priority(x): x ϵ S}, and let Best(S) is unique, since we can use ID as a tie-breaker. The output condition of DLEP is that, for any component C of the network, Best(C) will be elected leader of C.

DLEP consists of four phases. The first phase, which builds elects a preliminary leader l_c for each component C, and builds a preliminary BFS tree of C rooted at l_c, is exactly an emulation of the algorithm DLE given in Section 6.2. The second and third phases of DLEP make use of the preliminary BFS tree in each component to compute the final leader and the final BFS tree of that component.

The second phase of DLEP consists of a convergecast wave in the preliminary BFS tree. Let Tx be the subtree of the preliminary BFS tree rooted at x. During the second phase, the intermediate leader of each x is computed to be Best(T_x). Thus, the intermediate leader of l_c is Best(C), which will also be the final leader of C.

The third phase of DLEP consists of a broadcast wave, during which every process is told the identity of Best(C), and selects that process to be its final leader.

The fourth phase of DLEP consists of a flooding wave from Best(C) which builds the final BFS tree in C.

6.2 Definition of DLEP

We now give the formal definition of Algorithm DLEP.

6.2.1 Variables of DLEP

For any process x, we have variables, as listed below. The definitions of the variables given here refer to the values those variables will have when the algorithm stabilizes.

1. x.id , the ID of x. (This is not actually a variable of DLEP, since the algorithm cannot change it.)
2. The following variables are used for the first phase of DLEP, which emulates DLE.
3. x.p_leader, the preliminary leader of x, which corresponds to x.leader in DLE.
4. x.nplp, a non-positive integer called the negative preliminary leader priority of x, which corresponds to x.nlp in DLE. The value of x.nplp is the negative of the priority that x.p_leader assigned to itself when it declared itself to be a preliminary leader,
5. x.p_level, the preliminary level of x, the distance from x to x.p_leader, which corresponds to x.level in DLE.
6. x.p_vector = (x.nplp, x.p_leader, x.p_level), the preliminary vector of x, which corresponds to x.vector in DLE. Preliminary vectors are ordered lexically.
7. x.p_parent, the parent of x in the preliminary BFS tree, which corresponds to x.parent in DLE.
8. x.i_leader, the intermediate leader of x, whose value in a stable configuration is Best(Tx).
9. x.ilp, the intermediate leader priority of x, whose value in a stable configuration is Priority(Best(Tx)).
10. x.i_vector = (x.ilp, x.i_leader), the intermediate vector of x. Intermediate vectors are ordered lexically.

11. x.f_leader, the final leader of x, whose value in a stable configuration is Best(C), the elected leader of the component C that x belongs to.
12. x.f_level , the final level of x, whose value in a stable configuration is the distance from x to x.f_leader.
13. x.f_parent, whose value in a stable configuration is the parent of x in the final BFS tree.

Although we list x.p_vector and x.i_vector as variables, they are actually ordered tuples of other variables, and hence require no extra space.

6.2.2 Functions of DLEP

As in DLE, some of the functions of DLEP are given names which are capitalized versions of the names of variables. In such cases, the value of the function is what x believes the value of the variable should be.

1. If (nplp, l, d) is a preliminary vector, we define successor (nplp, l, d) = (nplp, l, d + 1), the smallest vector that is larger than (nplp, l, d).
2. Min_Nbr_P_Vector(x) = min {y.p_vector : y ϵ N(x) U {x}}, the minimum neighbor preliminary vector of x.
3. Local_Minimum(x) $\equiv$ Min_Nbr_P_Vector(x) $\geq$ x.p_vector, Boolean.
4. Good_Root(x) $\equiv$ Local_Minimum(x) ^ (x.p_leader = x) ^ (x.p_level = 0), Boolean.
5. Good_Child(x) $\equiv$ x.p_vector = successor (Min_Nbr_P_Vector(x)), Boolean.

6. P_Parent(x) = y ϵ N(x) such that y.p_vector = Min_Nbr_P_ Vector(x). If there is more than one such neighbor of x, choose the one with the smallest ID. If there is no such neighbor of x, define P_Parent(x) = x.
7. P_Chldrn(x) = {y : Good_Child(y) and y.p_parent = x}
8. We define the Boolean function Local_P_Tree_Ok(x) on a process x to mean that, as far as x can tell by looking at its variables and those of its neighbors, the preliminary leader and the preliminary BFS tree have been constructed. More formally, Local_P_Tree_Ok(x) is true if the following conditions hold for x:
    - x is either a good root or a good child.
    - x.p_level = 0 if and only if x is a good root.
    - x.p_leader = x if and only if x is a good root.
    - y.p_leader = x.p_leader for all y ϵ N(x).
    - $|y.p\_level - x.p\_level| \leq 1$ for all y ϵ N(x).
9. $$I\_Vector(x) = \max \begin{cases} (Priority(x), x) \\ \max \{y.i\_vector : y \epsilon P\_Chldrn(x)\} \end{cases}$$
10. $$F\_Leader(x) = \begin{cases} x.i\_leader & \text{if } Good\_Root(x) \\ x.p\_parent.f\_leader & \text{otherwise} \end{cases}$$
11. $$F\_Level(x) = \begin{cases} 0 & \text{if } x.f\_leader = x \\ 1 + \min \{y.f\_level: y \epsilon N(x)\} & \text{otherwise} \end{cases}$$

12. F_Parent(x) = p ϵ N(x) such that 1+ f_level(p) = f_level(x). If there is more than one such neighbor of x, choose the one with the smallest ID. If there is no such neighbor of x, define F_Parent(x) = x.

6.2.3 Legitimate Configurations for DLEP

We define a configuration of the network to be pre-legitimate if

1. For any component C of the network, there is exactly one process, $PL_C \epsilon C$, which is a good root; and all other processes of C are good children.
2. For any component C of the network, x.p_vector = (PL_C.nplp, PL_C, d(x, PL_C)) for all x ϵ C, where d(x, PL_C) is the distance from PL_C to x. That is, all processes in C have the same preliminary leadership pair, and preliminary level equal to the distance to the preliminary leader of the component.
3. If x.p_leader $\neq$ x, then x.p_vector = successor(x.p_parent.p_vector), i.e., x is a good child.

A configuration of the network is legitimate if it is pre-legitimate, and if, for each component C and for all x ϵ C:

1. x.i_vector = (Priority(y), y), where y = Best(Tx), where Tx is the subtree of the preliminary BFS tree of C rooted at x.
2. x.f_leader = Best(C).
3. x.f_level = d(x, Best(C)), the distance from x to Best(C).
4. x.f_parent = F_Parent(x).

6.2.4 Actions of DLEP

Table 2: Program of DLEP

A1	Reset	Local_Minimum(x)	→ x.nplp ← x.nplp − 1
priority 1		¬Good_Root(x)	x.p_leader ← x.id
			x.p_level ← 0
			x.p_parent ← x
A2	Preliminary	¬Local_Minimum(x)	→ p_vector(x) ←
priority 1	BFS Tree	¬Good_Child (x)	successor
			(Min_Nbr_Vector(x))
			x.p_parent ←
			P_Parent(x)
A3	Intermediate	x.i_vector ≠ I_Vector(x)	→ x.i_vector ←
priority 2	Vector	Local_P_Tree_Ok(x)	I_Vector(x)
A4	Final	x.f_leader ≠ F_Leader(x)	→ x.f_leader ←
priority 3	Leader	Local_P_Tree_Ok(x)	F_Leader(x)
A5	Final	x.f_level ≠ F_Level(x)	→ x.f_level ←
priority 4	Level	Local_P_Tree_Ok(x)	F_Level(x)
		∀y ϵ N(x): y.f_leader=x.f_leader	
A6	Final	x.f_parent ≠ F_Parent(x)	→ x.f_parent ←
priority 5	Parent	Local_P_Tree_Ok(x)	F_Parent(x)
		∀y ϵ N(x): y.f_leader = x.f_leader	

Table 2

6.2.5 Explanation of Actions

Action A1 corresponds to Action A1 of DLE, while Action A2 corresponds to Action A2 of DLE. Together, these two actions cause the preliminary leader, l_C of each component C to be chosen, and the preliminary BFS tree to be constructed.

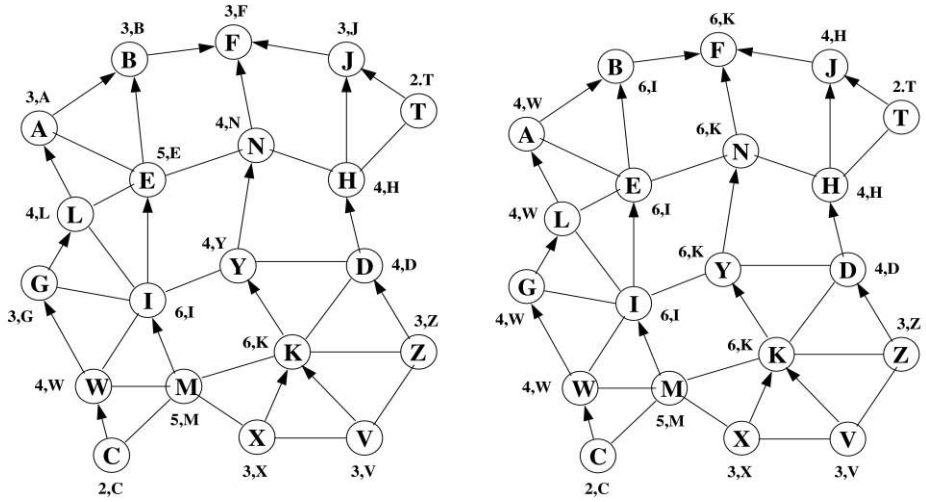
Action A3 is the action of the convergecast wave that chooses the intermediate vector for each process after the preliminary BFS tree has been constructed. It is possible for some processes to execute A3 prematurely because they believe, based on local information, that the preliminary BFS tree is finished; in these cases, these processes will recompute their intermediate vectors later.

The final leader of the component C, namely, FL_C, will be the intermediate leader of l_C. Action A4 is the action of the broadcast wave, starting at l_C, that informs every process of the choice of final leader.

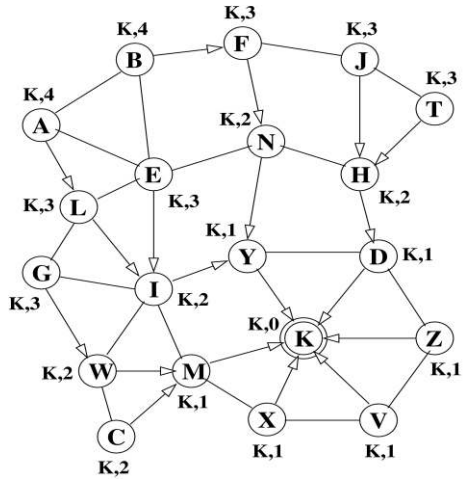
After every process knows the final leader, Actions A5 and A6 construct the BFS tree, assigning to each process its final level and final parent, in a broadcast wave starting at FL_C.

6.3 Example Computation

In Figure 6.2, we show some steps of a computation of DLEP.



(a) (b)



(c)

Figure 6.2:

Some configurations of a computation of DLEP on a component of a network. In this example, we define Priority(x) = δx, the degree of x. (a) shows the component after the preliminary BFS tree has been constructed. The preliminary leader is F, and the preliminary parent pointers are shown as arrows. Each process x is labeled with the ordered pair (Priority(x), x). Other variables are not shown. (b) shows the values of x.i_vector for each x, and the arrows still show preliminary parent pointers. Other variables are not shown. (c) shows the values of x.f_leader and x.f_level for each x, and the pointers of the final BFS tree are indicated by arrows. The final leader is K, indicated by a double circle. Other variables are not shown. At this step in the computation, the configuration is legitimate.

6.4 Proofs of DLEP

Our proof of correctness of DLEP uses the convergence stair method. We define a sequence of benchmarks, each of which is a closed predicate, i.e., if any benchmark holds, it will hold until the next fault. The sequence is also logically nested, meaning that each benchmark is a condition of the next benchmark.

- Benchmark B1: The preliminary BFS tree is complete, i.e., for each component C, there is a unique process

  $PL_C \in C$ such that x.p_vector = (l.nlp, PL_C, d(x, PL_C)) for any process $x \in C$, where d is distance, and if x.p_parent = P_Parent(x) for all x.

  Benchmark B1 holds when the first phase of DLEP is complete, i.e., the emulation of DLE is done.

- Benchmark B2: Benchmark B1 holds, and, for any process $x \in C$, x.ilp = Max_Priority(Tx) and x.i_leader = Best(Tx), where Tx is the subtree of the preliminary BFS tree rooted at x.

  Benchmark B2 holds when the second phase of DLEP is complete. At that point, for each component C, PL_C knows Best(C), since PL_C.i_leader = Best(C).

- Benchmark B3: Benchmark B2 holds, and for any process $x \in C$, x.f_leader = Best(C).

  Benchmark B3 holds when the third phase of DLEP is complete. At that point, every process x has correctly identified its final leader.

- Benchmark B4: Benchmark B3 holds, and for any process x ϵ C, x.f_level = d(x, Best(C)), and x.f_parent = F_Parent(x).

  Benchmark B4 holds when the fourth phase of DLEP is complete, namely the BFS tree rooted at the final leader of each component is finished.

The actions of DLEP are prioritized according to the benchmarks. Actions A1 and A2 of Table 2 are the actions of first phase of DLEP, i.e., the actions of the emulated DLE. Because of the hierarchical nature of the code, these actions take precedence over the others, and thus the values of the variables x.i_leader, x.f_leader, x.f_level, and x.f_parent do not retard progress toward Benchmark B1.

Action A3 of Table 2 is the only action of the second phase. In a convergecast wave in each component C, each process x sets x.ilp to Max_Priority(Tx) and x.i_leader to Best(Tx). When this wave reaches PL_C for each C, Benchmark B2 holds.

Action A4 of Table 2 is the only action of the third phase. Initially, PL_C, for each C, sets its value of f_leader to its value of i_leader; since Benchmark B2 holds, this value is Best(C). That value is then broadcast to all processes in the component, and then Benchmark B3 holds.

Action A6 of Table 2 is the only action of the fourth phase. After Benchmark B3 holds, there is exactly one process in each component C which knows it is the final leader of C. In a flooding wave starting from

that final leader, each process computes its distance to the final leader of its component; Benchmark B4 then holds, and DLEP is silent.

We prove convergence of DLEP in a sequence of lemmas. Throughout the remainder of this section, we assume that we are given a computation of DLEP, which starts at an arbitrary configuration. Of course, this start configuration could have resulted from a legitimate configuration, followed by any number of faults. By the definition given in Section 2, a computation has no faults; i.e., when a fault occurs, the next configuration is the start of a new computation.

Lemma 6.1 Benchmark B1 holds within Diam + 1 rounds.

Proof: This follows from Lemma 5.3, since the first phase of DLEP precisely emulates DLE.

Lemma 6.2 If at least t rounds have elapsed after Benchmark B1 holds, and if x is a process such that $d(x, PL_C) \geq Diam - t - 1$, then

$x.sub\_flp = max \{y.flp: y \in Tx\}$

Proof: By induction on t. If t = 0, the statement is vacuous. Otherwise, by the inductive hypothesis, the statement of the lemma holds for all $y \in$ P_Chldrn(x). Within one more round, x will execute Line 8 of the code, and we are done.

Lemma 6.3 Benchmark B2 holds within Diam + 1 rounds after Benchmark B1 holds.

Proof: Apply Lemma 6.2 for t = Diam + 1.

Lemma 6.4 If at least $t \geq 1$ rounds have elapsed after Benchmark B2 holds, and if x is a process such that $d(x, PL_C) \leq t - 1$, then x.f_leader = Best(C).

Proof: By induction on $d(x, PL_C)$. We first show that the result holds for $x = PL_C$. When Benchmark B2 holds, PL_C.sub_flp = Best(C), and after at least one additional round has elapsed, PL_C.f _leader = Best(C). Suppose $d(x, PL_C) = d > 0$, and $t \geq d+1$. After $t-1$ rounds have elapsed, Parent(x).f_leader = Best(C), by the inductive hypothesis. Within one more round, x.f_leader = Best(C).

Lemma 6.5 Benchmark B3 holds within Diam + 1 rounds after Benchmark B2 holds.

Proof: Apply Lemma 6.4. If at least $t \geq 1$ rounds have elapsed after for t = Diam + 1.

Lemma 6.6 Let x be a process in a component C, let d = d(x, Best(C)), and let $t \geq 0$, and suppose that at least t rounds have elapsed after Benchmark B3 holds. Then

(a) x.f_level $\geq \min \{t, d\}$.

(b) If $t > d$ then x.f_level = d.

Proof: If d = 0, then (a) is trivial, and (b) follows from that fact that

F_Level (Best(C)) =0.

We prove the case $d > 0$ by induction on t. If t = 0, then (a) is trivial, and (b) is vacuous. Suppose $t > 0$. Note that NL(x) = N(x), since Benchmark

B3 holds. By the triangle inequality, d(y, Best(C)) ≥ d−1 for all y ∈ N(x). Since d > 0, there exists z ∈ N(x) such that d(z, Best(C)) = d − 1.

After t − 1 rounds y.f_level ≥ min t − 1, d − 1 by the inductive hypothesis for all y ∈ N(x), and thus, F_Level (x) ≥ 1 + min {t − 1, d − 1} = min {t, d}. After one more round, x.f_level ≥ min {t, d}, and (a) is proved.

We now prove (b). Assume t > d. By (a), x.f_level ≥ d. After t − 1 rounds have elapsed, by the inductive hypothesis, z.f_level = d − 1, and thus F_Level(x) ≤ d. Within one more round, we have x.f_level ≤ d, and we are done.

Lemma 6.7 Benchmark B4 holds within Diam + 1 rounds after Benchmark B3 holds.

Proof: Apply Lemma 6.6(b) for t = Diam + 1.

Corollary 6.8 Within 4Diam + 4 rounds after the initial configuration, DLEP is silent, and the network is in a legitimate configuration.

6.4.1 The Unfair Daemon

We now prove that DLEP works under the unfair daemon.

Lemma 6.9 Every computation of DLEP contains only finitely many instances of a structural action.

Proof: The forgetful function is a morphism from DLEP to DLE. Any structural action of DLEND maps to a structural action of DLE. By Lemma 6.5, we are done.

Lemma 6.10 If a computation of DLEP contains no structural action, then it contains only finitely many instances of Action A3.

Proof: By contradiction. Suppose a computation of DLEP contains no structural action and also contains infinitely many instance of Action A3. Pick a process x which executes Action A3 infinitely many times. If there is more than one choice, pick x to maximize x.p_level .

Since y.p_level > x.p_level for all y ϵ P_Chldrn(x), there is some configuration γ_m in the computation after which no member of P_Chldrn(x) executes Action A3. Thus, after γ_m, the value of I_Vector(x) does not change, and therefore x can execute Action A3 at most once after γ_m, contradiction.

Lemma 6.11 If a computation of DLEP contains no instance of Action A1, A2, or A3, then it contains only finitely many instances of Action A4.

Proof: By contradiction. Pick a process x which executes Action A4 infinitely many times. If there is more than one choice, pick x to minimize x.p_level.

If x.p_level = 0, then F_Leader(x) = x.i_leader, which does not change. Otherwise, since x.p_parent.p_level < x.p_level , F_Leader(x) = x.parent.f_leader, which does not change. Thus, x can execute Action A4 at most once, contradiction.

Lemma 6.12 If a computation of DLEP contains no instance of Action A4, then it contains only finitely many instances of Action A5.

Proof: For any process x, x.f_leader does not change, and thus x.f_level cannot increase. Each time x executes Action A5, the value of x.f_level

decreases, and it cannot be less than zero. Thus, no process can execute Action A5 infinitely many times.

Lemma 6.13 If a computation of DLEP contains no instance of Action A4 or A5, then it contains only finitely many instances of Action A6.

Proof: For each process x, the value of P_Parent(x) does not change; thus, x can execute Action A6 at most once.

Theorem 6.14 Every computation of DLEP is finite, and ends at a legitimate state.

Proof: By Lemma 6.9, every computation contains only finitely many instances of a structural action. By Lemma 6.10, after the last execution of a structure action, there are only finitely many instances of Action A3. By Lemma 6.11, after the last execution of Action A3, there are only finitely many instances of Action A4. By Lemma 6.12, after the last execution of Action A4, there are only finitely many instances of Action A5. By Lemma 6.13, after the last execution of Action A5, there is only finitely many instances of Action A6, and then there are no more actions, i.e., DLEND is silent.

By Corollary 6.8, the last configuration of the computation is legitimate.

CHAPTER 7

DYNAMIC LEADER ELECTION NO DITHERING

Post–legitimate Configurations. Suppose that γ is a legitimate configuration for a distributed algorithm A on a given network G. Suppose G′ is a new network that is obtained from G by an arbitrary topological change; i.e., the processes of G′ are the same as those of G, and no variables of any process have been changed, but the links may be different. This change defines a configuration γ′ on G′, where each process has the same values of its variables as at γ. If a process x contains a variable which is a pointer to a process y which is a neighbor of x in G, and if y is no longer a neighbor of x in γ′, the pointer does not change, but it has nothing to point to. In this case, we say that that pointer is unlawful at γ′. We say that γ′ is post–legitimate configuration.

We now present Algorithm DLEND for the dynamic leadership election problem. DLEND has the following properties:

1. **Self Stabilization and Silence:** Starting from an arbitrary configuration, within O(Diam) rounds, a legitimate configuration is reached and there are no further actions.
2. **Incumbent Priority:** Starting from a post–legitimate configuration, if a component C contains at least one process which was a leader at the previous legitimate configuration, one of those processes will be elected leader of that component.

3. **No Dithering:** Starting from a post–legitimate configuration, no process will change its choice of leader more than once. DLEND shares the first property with Algorithms DLE and DLEP. The second property is an extension of the priority property of DLEP. To achieve the incumbent and no dithering properties, we introduce colors to guide the order of computation.

7.1 Overview of DLEND

DLEND is very much like DLEP, except that, to achieve the no dithering property, each process is given a color, which is an integer in the range [0 . . . 5]. The color of a process is related to its current role in the computation. The purpose of the colors is to ensure that final leader of a process is not computed too early. In a computation that starts from a post–legitimate configuration, the processes pass through the following sequence of colors.

1. In a legitimate configuration, x.color = 0 for each process x.
2. Each color changes to 1 when the preliminary BFS tree is being constructed.
3. All processes change color to 2 in a convergecast wave when the preliminary BFS tree is completed.
4. All processes change color to 3 in a broadcast wave after the preliminary leader has color 2.

5. All processes change color to 4 in a convergecast wave that computes the intermediate vector of each process. Each process x chooses as its intermediate leader a process in the subtree Tx which was a leader in the previous legitimate configuration, if any.

   It is possible for a process to change color to 2, 3, or 4 prematurely, and then go back to color 1. This can occur when some of the preliminary calculations of the preliminary BFS

   tree are incorrect, and need to be redone. However, when a good root changes its color to 4, the preliminary BFS tree has been correctly calculated.
6. All processes change color to 5 in a broadcast wave, during which each process chooses a new value of the final leader. All processes in a component choose the same final leader, which is the intermediate leader of the preliminary leader.

Finally, in a flooding wave starting from the final leader, all processes change their color to 0. They then construct the final BFS tree, and eventually the configuration is legitimate and silent.

7.2 Definition of DLEND

7.2.1 Variables of DLEND

DLEND uses all the variables of DLEP, plus some additional variables listed below.

- x.former_leader_in_subtree, Boolean, meaning that Tx contains a process which was a final leader in the last legitimate configuration.
- x.color ϵ {0, 1, 2, 3, 4, 5}.

We also redefine one variable.

- x.i_vector = (x.former_leader_in_subtree, Priority(x.i_leader), x.i_leader)

7.2.2 Functions of DLEND

DLEND uses the following functions defined for DLEP in Section 4.

- successor(nplp, l, d) = (nplp, l, d + 1), where (nplp, l, d) is a preliminary vector.
- Min_Nbr_P_Vector(x).
- Local_Minimum(x).
- Good_Root(x).
- Good_Child (x).
- P_Parent(x).
- P_Chldrn(x).
- Priority(x).
- Local_P_Tree_Ok(x)

One function of DLEP is redefined for DLEND.

- $\text{I\_Vector}(x) = \max \begin{cases} (\text{Is\_Leader}(x), \text{Priority}(x), x) \\ \max \{y.\text{i\_vector} : y \epsilon \text{P\_Chldrn}(x)\} \end{cases}$

DLEND uses a number of additional functions, as well.

- Local_I_Vector_Ok(x), Boolean, which is true if x.i_vector = I_Vector(x).
- Local_I_Leader_Ok(x), Boolean, which is true if either x.i_leader = x, or x.i_leader = y.i_leader for some y ϵ P_Chldrn(x).

  Note that Local_I_Vector_Ok(x) => Local_I_Leader_Ok(x), but that the converse does not hold.
- Local_F_Leader_Ok(x), Boolean, which is true if either x is a good root and x.f_leader = x.i_leader, or x is a good child and x.f_leader = x.p_parent.f_leader.
- Is_Leader(x), Boolean, meaning that x.f_leader = x.

If x is a good child, we say that x is color compatible with its parent if x.color, x.parent.color ϵ {1, 2, 3, 4, 5} and x.color = x.parent.color is odd, x.color is even and |x.parent.color − x.color | ≤ 1, x.color, x.parent.color ϵ {0, 5}, or x.color, x.parent.color ϵ {0, 1}.

The allowable combinations of colors of a process and its preliminary parent are illustrated in Figure 7.1.

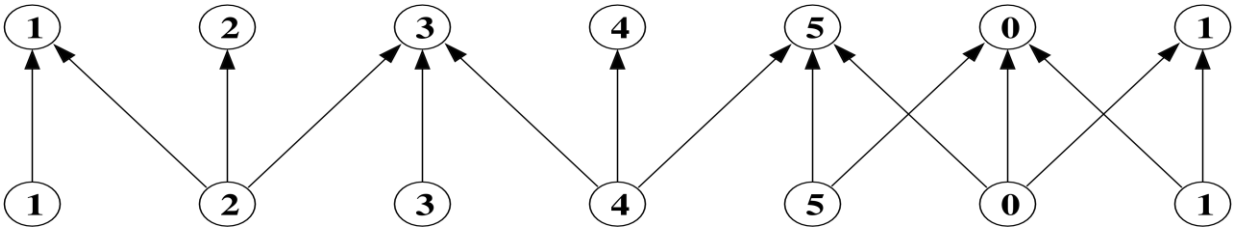


Figure 7.1:
Compatible combinations of colors for a true child x and its preliminary parent. All other combinations are incompatible.

Define the Boolean function Color_Error(x) to be true if either x is a true root which is color incompatible with some y ϵ P_Chldrn(x), or x is a true child which is color incompatible with x.p_parent.

Define the Boolean function P_Error(x) to be true if x.color ∉ {0, 1} and there is some y ϵ N(x) such that y.p_vector > successor(x.p_vector), i.e., x perceives that y is enabled to execute Action A3. Define the Boolean function I_Error(x) to be true if either x.color = 4 and ¬Local_I_Vector_Ok(x) or x.color = 5 and ¬Local_I_Leader_Ok(x).

Let Error(x) be the Boolean function which is true if Color_Error(x), P_Error(x), or I_Error(x).

Let Normal_Start(x) be the Boolean function which is true if x.color = 0 and any one of the following conditions holds.

1. ¬Local_I_Leader_Ok(x).
2. ¬Local_F_Leader_Ok(x).
3. x is a good root and y.color = 1 for some y ϵ P_Chldrn(x).
4. x is a good child and x.p_parent.color = 1.

Let Can_Start(x) be the Boolean function which is true if any one of the following conditions holds.

1. x.color ≠ 1 and Error(x).
2. Normal_Start(x).

We define additional Boolean functions for DLEND:

- Is_Leader(x), which is true if x.f_leader = x.

Additional non-Boolean functions of DLEND are defined below.

- $F\_Level(x) = \begin{cases} 0 & \text{if } x.f\_leader = x \\ 1 + \min \{y.f\_level: y \in N(x)\} & \text{otherwise} \end{cases}$
- F_Parent(x) = p ϵ N(x) such that p.color = 0 and 1 + f_level(p) = f_level(x). If there is more than one such neighbor of x, choose the one with the smallest ID. If there is no such neighbor of x, define F_ Parent(x) = x.

7.2.3 Actions of DLEND

Table 3: Program of DLEND

A1	Start	Can_Start(x)	→ x.color ← 1
priority 1			
A2	Declare	Local_Minimum(x)	→ x.nplp ← x.nplp − 1
priority 2	P–Leader	¬Good_Root(x)	x.p_leader ← x.id
			x.p_level ← 0
			x.color ← 1
			x.p_parent ← x
A3	P–Attach	¬Local_Minimum(x)	→ p_vector(x) ←
priority 3		¬Good Child(x)	successor(
			Min_Nbr_Vector(x))
			x.color ← 1
			x.p_parent ←
			P_Parent(x)
A4	Wave 2	x.color = 1	→ x.color ← 2
priority 4		∀y ϵ N(x):y.color ϵ {1, 2}	
		∀y ϵ P_Chldrn(x):y.color=2	
		Local_P_Tree_Ok(x)	
		¬Error(x)	
A5	Wave 3	x.color = 2	→ x.f_leader ←
priority 5		Good_ Root(x) V	F_Leader(x)

		(x.p_parent.color = 3)	x.color ← 3
		∀y ∈ N(x): y.color ∈ {2, 3}	
		Local_P_Tree_Ok(x)	
		¬Error(x)	
A6	Convergecast	x.color = 3	→ x.i_vector ←
priority 6	Intermediate	∀y ∈ N(x):y.color ∈ {3, 4}	I_Vector(x)
	Leader	Local_P_Tree_Ok(x)	x.color ← 4
		¬Error(x)	
A7	Broadcast	x.color = 4	→ x.f_leader ←
Priority 7	Final	Good_Root(x) V	F_Leader(x)
	Leader	(x.p_parent.color = 5)	x.color ← 5
		∀y ∈ N(x):y.color ∈ {4, 5}	
		Local_P_Tree_Ok(x)	
		¬Error(x)	
A8	Become	x.color = 5	→ x.f_level ← 0
priority 8	Final	∀y ∈ N(x):y.color = 5	x.f_parent ← x
	Leader	x.f_leader = x	x.color ← 0
		Local_P_Tree_Ok(x)	
A9	F–Attach	x.color = 5	→ x.f_level ← 0
priority 8		∀y ∈ N(x):y.color ∈ {5, 0}	x.f_parent ←
		∃y ∈ N(x):y.color = 0	F_Parent(x)
		Local_P_Tree_Ok(x)	x.color ← 0
		¬Error(x)	

A10	F–Level	x.color = 0	→ x.f_level ← F_Level(x)
Priority 8		∀y ∈ N(x):y.color = 0	x.f_parent ←
		x.f_level ≠ F_Level(x)	F_Parent(x)
		Local_P_Tree_Ok(x)	
		¬Error(x)	
A11	F–Parent	x.color = 0	→ x.f_parent ←
F_Parent(x)			
priority 9		∀y ∈ N(x):y.color = 0	
		x.f_parent ≠ F_Parent(x)	
		Local_P_Tree_Ok(x)	
		¬Error(x)	

Table 3

7.2.4 Explanation of Actions

Action A1 changes the color of process to 1, indicating that computation of the preliminary BFS tree is to start, or restart. A process x is enabled to execute A1 when it decides, based on the values of its neighbors, that it must start the computation of the preliminary BFS tree, or that there has been a fault that cannot be corrected without restarting that computation. Color 1 is "contagious," i.e., if x.color = 0 and a neighbor process has color 1, x can execute A1.

Action A2 corresponds to Action A1 of DLE, while Action A3 corresponds to Action A2 of DLE. Together, these two actions cause the preliminary leader, l_C of each component C to be chosen, and the preliminary BFS tree to be constructed. While a process is executing those actions, its color remains 1.

Actions A4 and A5 have no analog in Algorithms DLE and DLEP. All processes execute A4 from the bottom of the preliminary BFS tree, changing their colors to 2, and then top-down in the same tree, changing their colors to 3. No other variables are changed, so these actions do not contribute to computation of the preliminary, intermediate, or final leaders. This apparently pointless "waste" of 2Diam rounds is needed to ensure the no dithering property of DLEND, as we explain in our discussion of Figure 7.3.

Action A6 is the action of the convergecast wave that chooses the intermediate vector for each process after the preliminary BFS tree has been constructed. As each process executes A6, its color changes to 4.

It is possible for some processes to execute A6 prematurely because they believe, based on local information, that the preliminary BFS tree is finished; in these cases, these processes will recompute their intermediate vectors later. However, the no dithering property is guaranteed by the fact that, if the computation started from a post–legitimate state, no good root will ever execute Action A6 unless it is the

actual preliminary leader and the preliminary BFS tree has been correctly constructed.

The final leader of the component C, namely, FL_C, will be the intermediate leader of l_C. Action A7 is the action of the broadcast wave, starting at l_C, that informs every process of the choice of final leader. Each process changes its color to 5 when it executes A7.

When FL_C executes A7, it then executes Action A8, changing its color to 0, starting construction of the final BFS tree. All processes change their color to 0 in a flooding wave starting from FL_C, as they execute Action A9.

Actions A10 and A11 complete the construction of the final BFS tree, assigning to each process its final level and final parent.

7.3 Example Computations

In Figure 7.2, we show some steps of a computation of DLEND starting from a configuration where the preliminary BFS tree has been correctly constructed.

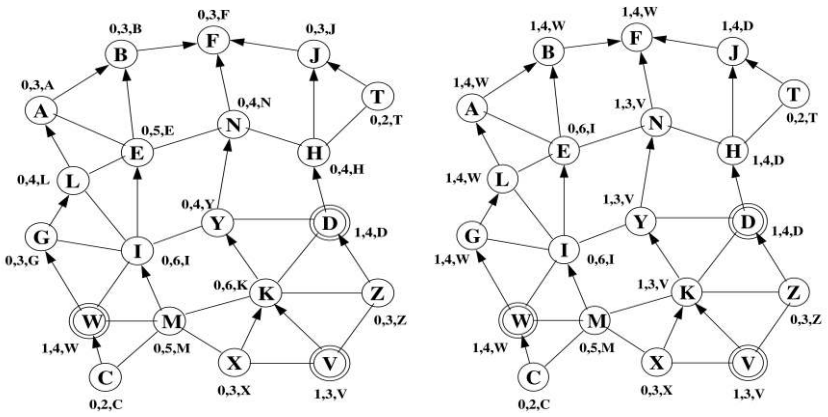
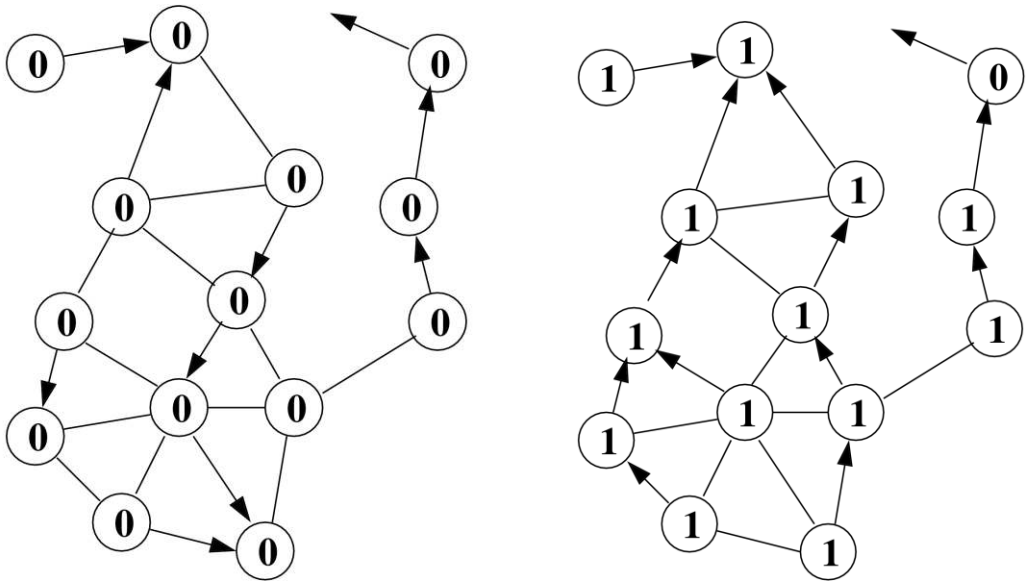


Figure 7.2:

We show some steps of DLEND for a component of a network, where we take Priority(x) = δx, the degree of x. (a) shows a network after the preliminary BFS tree has been constructed, and all colors are 2. Is_Leader holds for D, W, and V. The preliminary leader is F, and the preliminary parent pointers are shown as arrows. Each process x is labeled with its leadership triple (Is_Leader(x), Priority(x), x). The maximum (using lexical ordering) leadership triple is that of W, which will thus eventually be chosen to be the final leader. Other variables are not shown. (b) shows the values of x.i_vector for each x, and the arrows still show preliminary parent pointers, at a configuration where all processes have color 4. For each process x, the intermediate vector x.i_vector is shown. The value of x.i_vector is the maximum leadership triple in Tx. Other variables are not shown. (c) shows the final values of x.f_leader and x.f_level for each x, at a step when the configuration is legitimate. All processes have color 0. The pointers of the final BFS tree, rooted at the final leader W, are indicated by arrows. Other variables are not shown.

In Figure 7.3, we illustrate the role that the "extra" colors, 2 and 3, play in preventing premature execution of Action A7, and thus ensuring the no dithering property. We show a "worst case" scenario, where a tree has been constructed which is not the correct preliminary BFS tree, but no process in that tree can detect that. The chain of three processes on the right side of each figure is not part of the tree, but that fact cannot be detected by its neighboring process in the tree.

The situation will be resolved if the daemon selects the upper right process in the figure, but if the unfair daemon never selects that process, ultimately the convergecast Action A6 wave, where all colors change to 4, becomes unable to proceed. The tree becomes deadlocked, and eventually the daemon is forced to select the upper right process.



(a) (b)

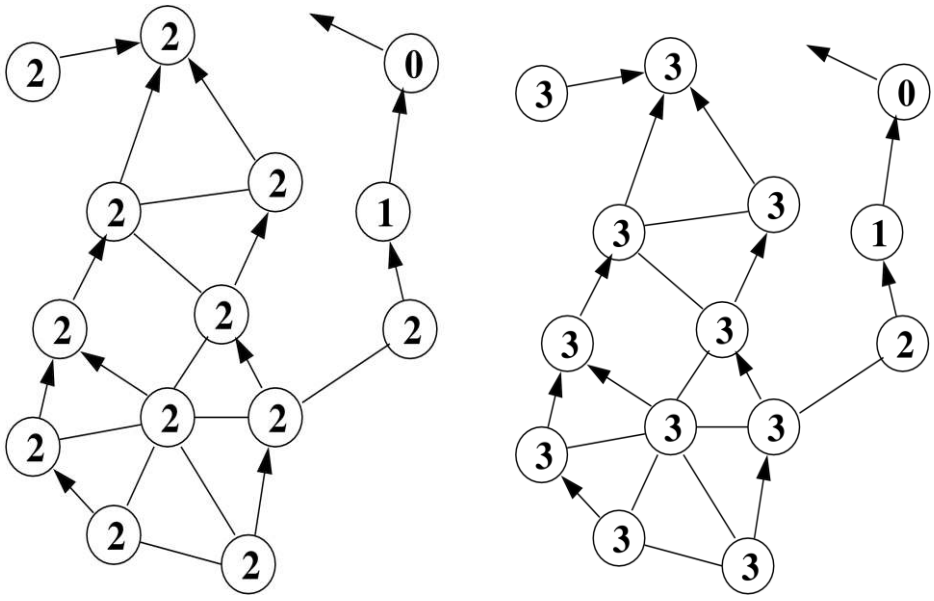


Figure 7.3:

Example computation starting from a post–legitimate state, showing how colors prevent processes from executing Action A7 more than once.

7.4 Proofs of DLEND

Throughout this subsection, let $\Gamma = \gamma_0 \rightarrow \gamma_1 \rightarrow \ldots \gamma_{t-1} \rightarrow \gamma_t \rightarrow \ldots$ be a computation of DLEND.

Lemma 7.1 Γ contains only finitely many structural actions.

Proof: The forgetful function is a morphism from DLEND to DLE. Any structural action of DLEND maps to a structural action of DLE. By Lemma 5.5, we are done.

Let S be the set of processes which execute infinitely many color actions during Γ.

Lemma 7.2 Γ contains only finitely many configurations where x.color = 4 and ¬Local_I_Vector_Ok(x) for some $x \in S$.

Proof: By Lemma 7.1, we can assume, without loss of generality, that contains no structural action. Pick $x \in S$. If there are only finitely many configurations of Γ where x.color = 4, we are done. Otherwise, since $x \in S$, x must execute Action A6 infinitely many times during Γ. Thus, x.color = 4 during each of infinitely many intervals.

Let $\gamma_s \cdots \rightarrow \cdots \rightarrow \gamma_t$ be one of those intervals, where $s > 0$. By definition of A6, Local_I_Vector_Ok(x) holds at γ_s.

The only action that can change the value of i_vector are A6. Thus, I_Vector(x) remains unchanged during the interval. If $y \in$ P_Chldrn(x), then y cannot execute either A6 or A7 during the interval, and thus I_Vector(y) remain unchanged during the interval. It follows that Local_I_Vector_Ok(x) holds throughout the interval.

Lemma 7.3 Γ contains only finitely many configurations where x.color =5 and ¬Local_I_Leader_Ok(x) for some x ϵ S.

Proof: By Lemma 7.1, we can assume, without loss of generality, that Γ contains no structural action. By Lemma 7.2, we can assume, without loss of generality, that Γ contains no configurations where x.color = 4 and ¬Local_I_Vector_Ok(x) for some x ϵ S.

Suppose x ϵ S and x executes Action A7 infinitely often in Γ. By definition of that action, Local_I_Leader_Ok(x) holds immediately x executes A7 and will remain true until some y ϵ P_Chldrn(x) executes Action A6. However, as long as x.color = 5, y cannot execute A6, and thus Local_I_Leader _Ok(x) will remain true.

Lemma 7.4 Γ contains only finitely many configurations where x.color = 5 and ¬Local_F_Leader_Ok(x) for some x ϵ S.

Proof: By Lemma 7.1, we can assume, without loss of generality, that Γ contains no structural action. Pick x ϵ S. If there are only finitely many configurations of Γ where x.color = 5, we are done. Otherwise, since x ϵ S, x must execute Action A8 infinitely many times during Γ. Thus, x.color = 5 during each of infinitely many intervals.

Let $\gamma_s \cdots \rightarrow \cdots \rightarrow \gamma_t$ be one of those intervals, where s > 0. By definition of A7, Local_F_Leader_Ok(x) holds at γ_s. Either x is a good root, or x.parent.color = 5 at γ_{s-1}.

The only action that can change the value of f_leader is A7. Thus,

x.p_leader does not change during the interval. x.p_parent cannot execute A7 during the interval, and thus Local_F_Leader_Ok(y) remain true during the interval.

Lemma 7.5 If $x \notin S$ and $y \in Tx \cap S$, then eventually y.color $\in \{1, 2\}$.

Proof: By induction on y.p_level.

Lemma 7.6 If $x \notin S$ and $x \in Ty \cap S$, then eventually y.color $\in \{1, 2\}$.

Proof: By backwards induction on y.p_level .

Lemma 7.7 If $x \notin S$, then $Tx \cap S = \emptyset$.

Lemma 7.8 If $x \notin S$ and $x \in Ty$, then $y \notin S$.

Proof: Use the other lemmas.

Lemma 7.9 Any computation of DLEND is finite.

Lemma 7.10 If no action of DLEND is enabled, then the configuration is legitimate.

Lemma 7.11 Let γ_0 be a post–legitimate configuration, and let L be the set of all processes for which x.f_leader = x at γ_0.

(a): For any component C, Any computation of DLEND that begins at γ_0 elects some member of $C \cap L$ to be the leader of C, provided $C \cap L \neq \emptyset$.

(b): For any process x, during any computation of DLEND which begins at γ_0, the value of x.f_leader changes at most once.

CHAPTER 8

SKETCHES OF PROOFS

DLE. The proof of correctness of DLE is fairly simple.

After one round of a computation Γ of DLE has elapsed, every process is either a true child or a true root, and, in each component C, the leadership pair of some true root l_C is the minimum leadership pair in that component. Within Diam additional rounds, a BFS tree is constructed in C rooted at l_C. Thus, DLE converges within O(Diam) rounds, and is silent upon reaching a legitimate configuration.

A distributed algorithm might be proved to converge in a finite number of rounds, but still possibly never converge under the unfair daemon, since that daemon might never select a specific enabled process. We prove that DLE works under the unfair daemon by proving that every computation of DLE is finite.

For each process x, the value of x.vector cannot decrease, and in fact, at every step of the computation, x.vector increases for some process x. Furthermore, from the initial configuration, we can compute an upper bound on the number of possible future values of x.vector. Thus, x can execute only finitely many times during the computation, and the computation thus cannot be infinite.

DLEP. The proof of DLEP uses the convergence stair method. We define a nested sequence of benchmarks, each of which is a closed predicate, and the last one of which is legitimacy.

There is a morphism from DLEP to DLE, meaning that every configuration of DLEP maps to a configuration of DLE, and that this mapping is consistent with the actions of the algorithms. From the fact that DLE is correct and silent under the unfair daemon, we can thus conclude that every computation of DLEP eventually reaches a configuration where first benchmark holds. i.e., the preliminary BFS tree is complete.

We define the next benchmark to mean that all values of i vector are correct, the next benchmark to mean that all values of f_leader are correct, and the fourth and last benchmark to mean that the configuration is legitimate.

We can show that it takes O(Diam) rounds to achieve the first benchmark, and O(Diam) additional rounds to achieve each subsequent benchmark. Thus, DLEP converges in O(Diam) rounds.

To prove that DLEP works under the unfair daemon, we prove that every computation of DLEP is finite. Given a computation Γ of DLEP, we have, from the properties of DLE and the morphism of DLEP to DLE, that Γ contains only finitely many instances of Actions A1 and A2. We then prove that, after the last instance of one of those actions, Γ contains only finitely many instances of Action A3. We then prove that, after the last instance Action A3, there are only finitely many instances of Action A4. Proceeding in this fashion, we eventually prove that Γ has only finitely many actions, and thus ends at a configuration where no process is

enabled. We then prove that if no process is enabled, the configuration is legitimate.

DLEND. DLEND elects both a preliminary leader and a final leader in each component, but uses colors to control the order of actions, in order to enforce the incumbent and no dithering properties.

Consider an action Γ which begins at post–legitimate configuration, which implies that every process has color 0. If the configuration of a component C differs only slightly from legitimate, it could happen that DLEND converges without changing the color of any process, and where every process has the same final leader as initially. Otherwise, we can prove that no process has color 5 until after the preliminary BFS tree is constructed. After that, each process changes its color to 5 exactly once, at which time, and no other, it can change its choice of final leader. All processes in C will choose the same final leader. Each process of C will then, just once, change its color to 0, after which the final BFS tree is constructed rooted at the elected final leader.

We also need to prove that, starting at any configuration, DLEND is eventually silent. Suppose Γ is a computation of DLEND. By the result we proved for DLE, Γ can contain only finitely many instances of a structural action. We then prove that Error(x) can be true for some process x only finitely many times, after which there can be only finitely many instances of a color action. After that, all colors are 0, and the only

actions that can be enabled are A10 and A11. After finitely many steps, there are no more actions, and the configuration is legitimate.

CHAPTER 9

CONCLUSION AND FUTURE WORK

We present three silent self-stabilizing asynchronous distributed algorithms for the leader election problem in a dynamic network with unique IDs, using the composite model of computation. A leader is elected for each connected component of the network. A BFS tree is also constructed in each component, rooted at the leader. This election takes O(Diam) rounds, where Diam is the maximum diameter of any component. All three algorithms work under the unfair daemon.

The three algorithms differ in their leadership stability. The first algorithm, which is the fastest in the worst case, chooses an arbitrary process as the leader. The second algorithm chooses the process of highest priority in each component, where priority can be defined in a variety of ways. The third algorithm has the strictest leadership stability. If the configuration is legitimate, and then any number of topological faults occur at the same time but no variables are corrupted, the third algorithm will converge to a new legitimate state in such a manner that no process changes its choice of leader more than once, and each component will elect a process which was a leader before the fault, provided there is at least one in that component.

This work can be extended in various ways. It would be useful to relax the conditions for "no dithering" further, meaning, one can explore if it is possible to maintain this property even in worse scenarios.

Another area to investigate could be the application of the proposed dynamic leader election algorithms in the K-Clustering algorithms [8] or Group Membership problem. Can we use the proposed solutions to make those protocols more suitable in dynamic environment?

BIBLIOGRAPHY

1. N. Malpani, J.L. Welch, and N. Vaidya, “Leader Election Algorithms for Mobile Ad Hoc Networks,” Proc. Fourth Int’l Workshop Discrete Algorithms and Methods for Mobile Computing and Comm., pp. 96-103, 2000.

2. D Peleg. Distributed Computing A Locality-Sensitive Approach. Society for Industrial and Applied Mathematics, 2000.

3. J Sun. Mobile ad hoc networking: An essential technology for pervasive computing. In Proceedings International Conferences on Info-Tech & Info-Net, pages 316-321, 2001.

4. Carla-Fabiana Chiasserini and Ramesh R. Rao. Pulsed battery discharge in communication devices. In MOBICOM, pages 88-95, 1999.

5. M.Scott Corson, Joseph P. Macker, and Gregory H. Cirincione. Internet-based mobile ad hoc networking. IEEE Internet Computing, 3(4):63-70, 1999.

6. Charles E Perkins. Ad hoc networking. Addison Wesley Professional, 2001.

7. A. Lerner I. Chlamtac. Fair algorithms for maximal link activation in multi-hop radio networks. IEEE Transactions on Communications, 35(7), 1987.

8. A. K. Datta, L. L. Larmore, and P. Vemula. Self-stabilizing leader election in optimal space. In 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), pages 109–123, Detroit, MI, November 2008. Also to appear in Theoretical Computer Science.

9. M Weiser. Hot topics: Ubiquitous computing. IEEE Computer, 26(10):71-72, Oct 1993.

10. Deborah Estrin, Ramesh Govindan, John S. Heidemann, and Satish Kumar. Next century challenges: Scalable coordination in sensor networks. In MOBICOM, pages 263-270, 1999.

11. Joseph M. Kahn, Randy H. Katz, and Kristofer S. J. Pister. Next century challenges: Mobile networking for “smart dust”. In MOBICOM, pages 271-278, 1999.

12. Emmanuelle Anceaume, Maria Gradinariu, and Matthieu Roy. Self-organizing systems case study: peer-to-peer systems.

13. Rebecca Ingram, Patrick Shields, Jennifer E. Walter, and Jennifer L. Welch. An asynchronous leader election algorithm for dynamic networks. In IPDPS, pages 1–12, 2009.

14. G.Bosilca T. Angskun, G. Fagg, J. Pjesivac-Grbovic, and J. Dongara. Self-healing network for scalable fault tolerant runtime environments. DAPSYS 2006, 6th Australian-Hungarian Workshop on Distributed and Parallel Systems, pages 21-23, September 2006.

15. Satoshi Asami, Nisha Talagala, and David A. Pattersin. Designing a self-maintaining storage system. In IEEE Symposium on Mass Storage Systems, pages 222-233, 1999.

16. JD Strunk and GR Ganger. A human organization analogy for self-*systems. Technical report, FCRC Proceedings of the first Workshop on Algorithms and Architectures for Self-Managing Systems In conjuction with Federated Computing Research Conference, Jun 2003.

17. EW Dijkstra. Ewd386 the solution to a cyclic relaxation problem. In Selected writings on Computing: A personal Perspective, pages 34-35. Springer-Verlag, 1982. EWD386's original date is 1973.

18. EW Dijkstra. Self stabilizing systems in spite of distributed control. Communications of the Association of the Computing Machinery, 17:643-644, 1974.

19. A Arora. A foundation of fault-tolerant computing. Ph.D. dissertation, The University of Texas at Austin, Dec 1992.

20. Anish Arora and Mohamed G. Gouda. Closure and Convergence: A foundation of fault-tolerant computing. IEEE Trans. Software Eng., 19(11):1015-1027, 1993.

21. S Dolev. Self-Stabilization. The MIT Press, 2000.

22. J. Heidemann and R. Govindan. An overview of embedded sensor networks, November 2004. http://www.isi.edu/johnh/PAPERS/Heidemann04a.html.

23. Josep L. W. Kesels. An exercise in proving self-stabilzation with a variant function. Inf. Process. Lett., 29(1):39-42, 1988.

24. Anish Arora and Mohamed G. Gouda. Distributed reset. IEEE Trans. Computers, 43(9):1026-1038, 1994.

25. Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization (extended abstract). In PODC, pages 27-34, 1996.

26. Yehuda Afek and Shlomi Dolev. Local stabilizer. In ISTCS, pages 74-84, 1997.

27. Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilzation by local checking and correction (extended abstract). In FOCS, pages 268-277, 1991.

28. G Varghese. Self-stabilization by local checking and correction. Ph.D. dissertation, MIT, 1993.

29. G Varghese. Self-stabilization by counter flushing. In PODC, pages 244-253, 1994.

30. Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing algorithms. In PODC, pages 45-54, 1996.

31. Alain Cournier, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Enabling snap-stabilizatio. In ICDCS, pages 12-19, 2003.

32. Shlomi Dolev and Ted Herman. Superstabilzing protocols for dynamic distributed systems. Chicago J. Theor. Comput. Sci., 1997.

33. Joffroy Beauquier, Sylvie Delaet, Shlomi Dolev, and Sebastien Tixeuil. Transient fault detectors. In DISC, pages 62-74, 1998.

34. A. G. Ganek T. A. Corbi. The dawning of the autonomic computing era. http://www.cs.drexel.edu/~jsalvage/Winter2010/CS576/ autonomic.pdf.

35. G.-C. Roman, Q. Huang, and A. Hazemi, “Consistent Group Membership in Ad Hoc Networks,” Proc. 23rd Int’l Conf. Software Eng. (ICSE ’01), pp. 381-388, 2001.

36. S. Han and Y. Xia. Optimal leader election scheme for peer-to-peer applications. In Proc. 6th Int’l. Conf. on Networking, page 29, 2007.

37. B. Lehane, L. Dolye, and D. O’Mahony, “Ad Hoc Key Management Infrastructure,” Proc. Int’l Conf. Information Technology:

Coding and Computing (ITCC '05), vol. 2, pp. 540-545, 2005.

38. A.D. Amis, R. Prakash, T.H.P. Vuong, and D.T. Huynh, "Maxmin D-Cluster Formation in Wireless Ad Hoc Networks," Proc. IEEE INFOCOM '00, pp. 32-41, 2000.

39. D.J. Baker and A. Ephremides, "The Architectural Organization of a Mobile Radio Network via a Distributed Algorithm," IEEE Trans. Comm., vol. 29, no. 11, pp. 1694-1701, 1981.

40. M. Gerla and J.T.-C. Tsai, "Multicluster, Mobile, Multimedia Radio Network," ACM/Baltzer Wireless Networks, vol. 1, no. 3, pp. 255-265, 1995.

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

42. V.D. Park and M.S. Corson, "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks," Proc. IEEE INFOCOM '97, pp. 1405-1413, Apr. 1997.

43. S. Vasudevan, J. Kurose, and D. Towsley, "Design and Analysis of a Leader Election Algorithm for Mobile Ad Hoc Networks," Proc. 12th IEEE Int'l Conf. Network Protocols (ICNP '04), pp. 350-360, Oct. 2004.

44. E. Gafni and D. Bertsekas. Distributed Algorithms for Generating Loop-free Routes in Networks with Frequently Changing Topology. IEEE Transactions on Communications 29(1):11-15, January 1981.

45. M.S. Corson and A.Ephremides. A Distributed Routing Algorithm for Mobile Wireless Networks. ACM Wireless Networks Journal 1(1):61-82, Febraury 1995.

46. Yehuda Afek and Anat Bremler. Self-stabilizing unidirectional network algorithms by power-supply (extended abstract). In SODA, pages 111–120, 1997.

47. A Arora and MG Gouda. Distributed reset. IEEE Transactions on Computers, 43:1026–1038, 1994.

48. B Awerbuch, S Kutten, Y Mansour, B Patt-Shamir, and G Varghese. Time optimal self-stabilizing synchronization. In Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC' 93), pages 652–661, 1993.

49. Abdelouahid Derhab and Nadjib Badache. A self-stabilizing leader election algorithm in highly dynamic ad hoc mobile networks. IEEE Transactions on Parallel and Distributed Systems, 19(7):926–939, 2008.

VITA

Graduate College
University of Nevada, Las Vegas

Hema Piniganti

Degrees:
Bachelor of Engineering in Computer Science, 2008
Osmania University, India

Master of Science in computer science, 2010
University of Nevada Las Vegas

Thesis Title: Self-Stabilizing Leader Election in Dynamic Networks

Examination Committee:
Chair Person, Dr. Ajoy K Datta, Ph.D.
Committee Member, Dr. Lawrence L. Larmore, Ph.D.
Committee Member, Dr. Yoohwan Kim, Ph.D.
Graduate Faculty Representative, Dr. Emma E. Regentova, Ph.D.