LEADER ELECTION IN DISTRIBUTED NETWORKS USING AGENT BASED SELF-STABILIZING TECHNIQUE

A Thesis

by

RAGHAV TANDON

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

MASTER OF SCIENCE

August 2003

Major Subject: Computer Engineering

LEADER ELECTION IN DISTRIBUTED NETWORKS USING AGENT BASED SELF-STABILIZING TECHNIQUE

A Thesis

by

RAGHAV TANDON

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

MASTER OF SCIENCE

Approved as to style and content by:

Hoh In (Chair of Committee)	Jyh-Charn Steve Liu (Member)
A. L. Narasimha Reddy (Member)	Valerie Taylor (Head of Department)

August 2003

Major Subject: Computer Engineering

ABSTRACT

Leader Election in Distributed Networks Using Agent Based Self-stabilizing Technique.

(August 2003)

Raghav Tandon, B.E., Netaji Subhas Institute of Technology, New Delhi, India

Chair of Advisory Committee: Dr. Hoh In

There are many variants of leader election algorithm in distributed networks. In this research, an agent based approach to leader election in distributed networks is investigated. Agents have shown to be useful in several ways. In the theoretical perspective, agents sometime help in reducing the message complexity of the system and sometimes help in lowering time complexity. In a more practical sense, agents perform operations independent of the processors, thereby lending a more flexible algorithm supporting different types of networks.

To my beloved family and friends

ACKNOWLEDGMENTS

I wish to thank my advisor, Dr. Hoh In, for giving me the freedom to work in interesting research areas. I would like to thank you for all your advice and discussions. Through your patience and dedication, I was able to work on my thesis with a lot more enthusiasm. I would also like to thank Dr. Welch for the insightful discussion we had in her class on distributed systems. I would like to thank my supervisor, Dr. Shaw-Pin Miaou, who gave me the time and freedom to work on my thesis. Further, he constantly gave advice on the roadmap towards a successful research that I found very useful. I would also like to acknowledge my officemates, Jin and Seong, who helped me by giving lot of information related to the working of my thesis. I would like to thank my family and my friends for pushing me through my work. Thank you all.

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

In distributed networks, processors communicate with each other using shared memory or by exchanging messages with each other. For processors to perform any distributed task effectively the processors require coordination. In a pure distributed network, there is no central controlling processor that arbitrates decisions. Without a central authority or coordinator, any processor has to communicate with all processors in the network to make decision. Often during the decision process, not all processors make the same decision. Communication between processors takes time and further more, making the decision takes time. Coordination among processors becomes difficult when consistency is needed among all processors. Centralized controlling processor(s) can be selected among the group of available processors to reduce the complexity of decision-making. By having a centralized authority, decisions can be made in a more serialized fashion, which are simpler to execute. All decisions for processing a distributed task are decided by the controlling processor(s). Centralized control along with effective coordination can also be helpful in reducing the message complexity in the network by preventing flooding of messages by processors in the distributed network. At the same time, centralized control may have the disadvantage of higher time complexity as it weighs more on a serialized execution.

This thesis follows the style and format of *IEEE/ACM Transaction on Networking.*

Leader election is a technique that can be used to break the symmetry of a distributed network by determining a central controlling processor (leader) in the distributed network. A processor is elected as the leader among the group of processors in the distributed network. This processor acts as the centralized controller of this decentralized distributed network. Such a decentralized network can support highly centralized protocols. Some applications of leader election include finding a spanning tree with the elected leader as root, breaking a deadlock and reconstructing a lost token in a token ring network.

The purpose of leader election [1] is to choose a processor that will coordinate activities of the system. In any leader election algorithm, a leader is usually decided based on some criterion such as choosing the processor with the largest identifier as the leader. At the time when the leader is decided, the processors reach the terminated states. The terminated states, in a leader election algorithm, are partitioned into elected states and non-elected states. When a processor enters a non-elected state (or an elected state), the processors always remain in the non-elected state (or an elected state). Any leader election algorithm must be satisfied by the safety and liveness condition for an execution to be admissible.

- The liveness condition states that every processor will eventually enter an elected state or a non-elected state.
- The safety condition for leader election requires that only a single processor can enter the elected state. This processor becomes the leader of the distributed network.

Several leader election algorithms have been proposed over the years [2-17]. Of these proposals, some algorithms are found to be efficient but applicable to certain network topologies, timing constraints and sometimes, the size of the network (known as non-

uniform algorithms). Such leader election algorithms proposed until now require processors to be directly involved in leader election. Information is exchanged between processors by transmitting messages to each other. The processors exchange messages with each other and try to reach an agreement. Once an agreement is reached, a processor will be elected as leader and all other processors will acknowledge the presence of the leader. Distributed systems are continuously evolving and new architectures are being introduced every day. In this research, the following problems are examined to incorporate a more flexible leader election algorithm:

- Is it possible to have a single algorithm that can efficiently and correctly resolve a leader in any network topology?
- Can the leader election algorithm be isolated and executed without the processors intervention in decision-making?

Self-stabilizing mobile agents [18] have properties that help resolve the above issues. In this research, self-stabilizing mobile agents are used to execute the leader election among processors in a distributed network. Agents based self-stabilization [18] is a technique where agents are used to bring the system to the stable condition. Self-stabilization is a property that allows a system to recover to a stable state from an illegal/unstable state. Leader election in terms of self-stabilization methodology can be interpreted as a system that is unstable when it has no leader and attains stability when a processor is elected as leader among the processors in the distributed network and all processors are informed of the newly elected leader. Agent based self-stabilization is based on mobile agents which are active in nature unlike messages which are passive entities. Mobile Agents are messages that contain self-executable software code that can be executed on a processor. Agents can

perform execution based on certain rules and can make decision independent of the processor. Contrary to message passing technique, where processors decide for every incoming message, agent-based approach has several benefits.

- Agents are autonomous. Agents can independently decide which processor to migrate to next.
- When agents hop from one processor to another, agent can carry information like messages and leave information (trace, status etc.) at each processor they visit.
- Agents can sometime minimize number of message (hops) in the network.
- Having multiple agents can sometime help in solving a problem faster.

In this research, possible approaches for implementing agent based leader elections algorithm in distributed networks is investigated. A modular technique for leader election is investigated as part of the related work and the concept of agent-based self-stabilization is also introduced. Other works related to migration techniques are also explored in the related work [19], [20]. In this study, a solution for leader election using a single agent model is first proposed to give an understanding of the proposal. Then the leader election solution is proposed using multiple agents. The correctness of each solution is examined and an analysis of message and time complexity is deduced. An example of synchronous bi-directional ring is used to examine the working of leader election. Further, a case study is conducted to see the effect of a dynamic network on leader election. In the last section, the conclusions are drawn and possible future work on leader election using self-stabilizing agents is discussed.

CHAPTER II

RELATED WORK

2.1 A Modular Technique for Designing Leader Finding Algorithms [21]

In this thesis, a modular technique to solve the leader election problem is proposed for distributed, asynchronous networks. The problem of efficient leader finding is reduced to the problem of efficient serial traversal of the corresponding network. This technique solves the problem in two stages:

1) The first stage involves the traversal of the entire network,
2) The second stage involves the leader finding algorithm, which uses the algorithm of stage 1 as a distributed subroutine.

The modular technique uses message-passing model where tokens are the entities used to communicate information among the processors. Initially all the nodes are inactive and the phase of each node is set to –1. When a node N becomes active and starts the algorithm, the nodes phase changes to 0 and token is created of the form (0, N) where 0 represents the phase of N. A token (p, a) can be in one of three modes:

1) Annexing mode. A token in the annexing mode attempts to annex all the nodes in the network to its domain. For the traversal of the network, the token uses a traversal algorithm, and the token annexes the nodes it passes during the traversal.
2) Chasing mode. A token in this mode is chasing another token (p, b) in the annexing mode, in an attempt to reach it and to create a higher phase.
3) Candidate mode. A token in the candidate mode is waiting for a token in the chasing or annexing token with the same phase.

Token may be created by one or more processors and not necessarily all the processors. When token are created they are initialized in the annexing mode. A token moves into chasing mode when it finds that a node is annexed by a token with a higher identity value. The annexing tokens with higher phases shall combine with the chasing tokens to form new phases and continue to annex the network. This process continues until a token has covered the entire network and only a single token exists in the network. The single token having traversed the entire network has identified the leader. The message complexity for this approach is bounded by [(f(n) + n)(log2k + 1) (or (f(m) + n)(log2k + 1)], where n is the number of nodes in the network, m the number of edges in the network, k is the number of nodes that start the algorithm, and f (n) [f(m)] is the message complexity of traversing the nodes [edges] of the network. This approach has a time complexity comparable to the message complexity. Having a lower time complexity as compared to the message complexity is left as an open problem.

2.2 Agent Based Self-Stabilization

Dijkstra first introduced the concept of self-stabilization in distributed networks [22]. In the paper, a system is defined as self-stabilizing when "regardless of its initial state it is guaranteed to arrive at a legitimate state in a finite number of steps." The self-stabilization algorithm can be extended to the leader election problem. Initially the system has no leader. For having a leader in the system, the local state of one processor should be changed to the elected state. The local state of all the other processors should be the non-elected stated. This will be defined as a legitimate state for leader election. One solution that extends self-stabilization for leader election in anonymous ring is described in the paper [23].

2.2.1 Mobile Agent Based Systems

Most techniques approach the problem of leader election using message passing techniques. Processors communicate with each other using messages and try to resolve a leader. This approach requires processors to directly participate in the leader election process. Mobile agent is a self-executable software code that is transmitted as a message between processors. Mobile agents are autonomous entities and can hop from one processor to another. Agents provide an independent processing from normal system functions except for certain task such as agent creation. Contrary to message passing technique, where processors decide for every incoming messages, in agent-based approach, agent themselves decide which processor they wish to migrate to next. Agent can carry information like messages and leave information at each processor they visit. The information variables carried by them are termed as briefcase variables [18]. Agent can perform changes in local variable of a process. Agents perform atomic operations that are executed at the processor locally. This property is explained in the paper [18].

2.2.2 Multiple Agent Model

By having multiple agents, each agent can perform the task in parallel and help in converging to desired result faster. Having multiple agents sometime can also be a problem. Agents may be required to meet to exchange information with each other either during or at the end of the task to decide on a result. As the number of agents grows, coordination between agents may become more complicated. Multiple agents in a system are possible when one or more processors create agents. Multiple agents are also possible with agent

replication. Agent replication is a property that allows agents to create copies of them selves.

2.2.3 Self-Stabilization [24]

Agent based self-stabilization is an approach for stabilizing system by using agents. As agents can behave autonomously, processors are not involved in the stabilization process. Agents can traverse the entire network and monitor processors that have illegitimate state and correct the state of the processor. Since agents act as autonomous, entities they need to decide on their route so that coordination can be achieved between the different agents to achieve the desired task. Some migration techniques are described in the next section.

2.3 Migration Techniques

Agent Migration is essentially a neighbor selection or routing problem. Agents acting as autonomous entities need to decide on the path to take and the nodes to cover. In case of multiple agents, the agents may need to decide on how to maximize the search. Agent may be explicitly required to meet each other and decide on rendezvous points. Having decided on a meeting place, the agents should know how to choose a path to get there. In a system, it is possible that each agent will employ same or different migration technique based on the purpose they serve. There are several approaches for neighbor selection. Most of these techniques are borrowed from token-based schemes.

2.3.1 Random Walks [25], [26], [27], [28]

Random walk is an approach where a decision for migration is governed by the probability of neighbor selection. If d is the degree of a vertex v in a graph, then the probability of choosing any neighbor of vertex v is given by 1/d. It is interesting to note a Markov Chain can represent the random walk by simply extending to the fact that the probability of choosing a vertex from v that is not its neighbor is zero. A simple random walk means that either of two directions in a bi-directional system is given a probability of half. Random walks are useful when the agent has no information of where the other agents are. Random walks also ensure that eventually the agent shall cover the entire network as long as the network is a connected graph. With random walk, the agents are guaranteed to meet but the time that it will take may be very larger or sometimes very small. The hitting time, as it is known, has an upper bound of Ω (n^3). Random walks can also be applied to unidirectional networks such as unidirectional rings. In such a case, the agent decides whether to move or stay at a particular vertex in time.

2.3.2 Biased Random Walks [20]

In order to favor a certain decision, a bias is introduced in the randomness. Neighbor selection is defined by two components – random choice, biased choice. If the probability of making a bias choice is p, then the probability of making a random choice is 1-p. The introduction of the bias is for reasons such as restricting the direction of movement to a certain area of the network to maximize the expected benefits over the long-term behavior of the walk. Biased Random Walks can be implemented for systems that allow only unidirectional movements. Here again, the same two components of random choice and

bias choice are used in deciding whether to stop or to move to neighbor. Such a bias may be useful for making faster rendezvous among agents. For examples, in a unidirectional circular ring, bias may be introduced that makes agents with larger identifiers to move slowly and agents with smaller identifiers to move faster.

CHAPTER III

LEADER ELECTION USING SELF-STABILIZING AGENTS

In distributed networks, leader election is needed to elect a leader or a central controller among the processors of the network. This study focuses on the use of self-stabilizing mobile agents for the election of a leader. Processors do not directly participate in the process of leader election. Processors are only involved in the initiation of the leader election process. At the point of initiation of the leader election algorithm, processors create agents. Once agents are created, the agents work together to find the processor with the largest identifier. Using agent based architecture the following solution in proposed. In the first subsection, the definitions and assumptions of the model are stated. In the next subsection, the model is proposed for a single agent system. In the last subsection, the model using multiple agents is proposed and the advantage of having multiple agents is examined.

3.1 Definitions and Assumptions

In the model, distributed networks are considered as a generally connected undirected graph G with n vertices. Each processor in the network is represented as a vertex of graph G. A connection between two processors is represented as an edge between two vertices that connect the two processors. The graph being undirected implies that the edges (links) of the graph are bi-directional. The processor where an agent is created is designated as the 'home processor' of the agent. In the distributed network, initially processors have no knowledge of the processors that are its neighbors [21]. In this model, processors can

assume any one of two states – uncovered and covered. A processor is in uncovered state, when the processor has knowledge of zero or more but not all its neighbors. In the covered state, the processor has knowledge of the identities of all neighboring processors.

When a processor initiates leader election, an agent is created that runs the algorithms for finding leader in the network. In the synchronous network, the agents are created simultaneously. However, in the asynchronous case, it is possible that only a single agent is created as processors initiate the leader election at different times. In an asynchronous network, each processor initiates the leader election task independently. The processors do not have knowledge of when other processors initiate the leader election task as there is no central controlling authority and the time is not synchronized among the processors.

The following proposal looks into two models, single agent based model and multiple agent based model. The purpose of a single agent model is to introduce the notion of agent traversal in the distributed network. The multi-agent model is modified to provide a more efficient approach to solve the leader election. Agents are assumed reliable and tolerant to any failures in the network. The time taken for agents to perform local computations is assumed negligible. Each hop by an agent is assumed equivalent to the transfer of message between the two processors. The size of the message that is transferred with every agent hop is assumed negligible.

3.2 Leader Election with Single Agent

In a single agent model, a single agent in the network alone will perform leader election. This task of leader election is split into two simpler tasks. The first task of the agent is to locate the processor with the largest identifier for leader election. The second task of the

agent is to announce the processor as leader to the rest of the processors. The agent needs to cover the entire network while storing the identifiers of each processor. The agent is said to have covered the entire network, when the state of each processor is changed from uncover to covered and the agent returns to its home processor. At this stage, all the processors have the identity of all neighboring processors.

3.2.1 Network Traversal Algorithm

The first algorithm is the network traversal algorithm. The goal of the algorithm is to ensure that the agent covers the entire network and the agent terminates the algorithm only when all processors are in covered state and the agent returns to the home processor. The algorithm is designed to make the agent traverse the network in a mix of breadth and depth first search. At each processor, the agent will first identify all the neighbor of the current processor. This step is performed in a breadth first search fashion. An unidentified neighbor is chosen and visited by the agent to obtain it identity. During the visit to the neighbor, the agent leaves the identifier of the current processor for which it is performing the neighbor identification. Once all the neighbors of the processor are identified, the processor's state is changed from uncovered to covered. There after the agent chooses one of the neighbors that has an uncovered state and migrates to that processor. Upon reaching the new processor, the agent runs the same procedure as above. When no neighbor of a current processor is in the uncovered state, the agent will migrate back towards the home processor. This fashion of choosing neighbors to traverse the entire network is similar to the depth first search. The procedure of neighbor identification and choosing a random neighbor after identification that is similar to the breath first search is illustrated in figure 1.

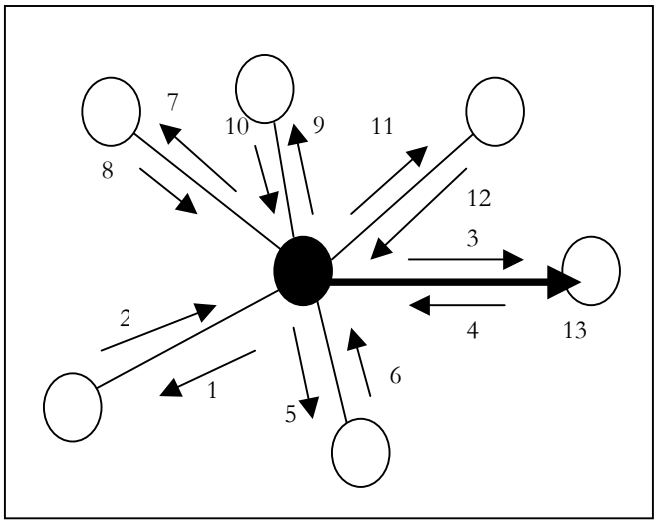

Fig. 1. Neighbor Identification.

The dark arrow in figure 1 indicates the migration of the agent to another processor as step 13, once all the neighbors of the current processor have been identified. The procedure of network traversal that is similar to the depth first search is illustrated in figure 2. In figure 2, the dotted lines represent migration of agent to the source node that is directed towards the home processor. Neighbors are chosen randomly based on a probability.

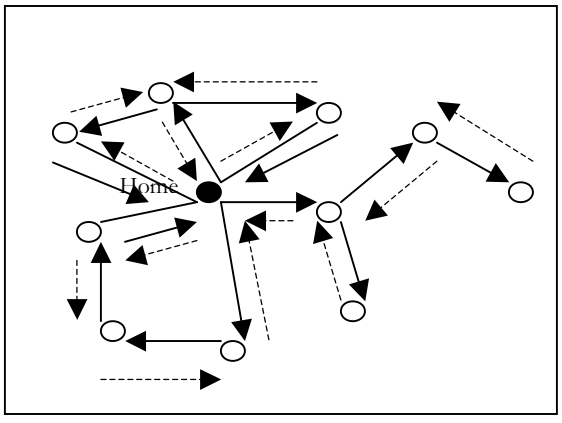

Fig. 2. Network Traversal.

Each uncovered neighbor is chosen with equal probability. If d is the number of uncovered neighbors, then the probability that a neighbor is chosen is given by 1/d. At any processor, if the agent finds no neighbor in uncovered state then it shall proceed back towards its home agent. The neighbor selected in this case is the Source_Node, i.e. the neighbor from where the agent first migrated to the current processor.

The agent shall store some information to allow it to traverse the entire network and ensure network covering. These variables are as follows:

- List of [Processor ID, Processor Status]

This is a list of the identifiers of processors visited and their corresponding state recorded at the last visit

- Parent_Processor

This is the processor identifier of the processor that created the agent.

- Current_Uncovered_Processor

This is the identifier of the processor for which the agent is current busy identifying its neighbors

- Previous_Processor_ID

This variable is used to store the identifier of the previous processor from which the agent just visited.

- Covered_Previous = False

This variable is used to ensure that the agent comes back to the processor having the identifier Current_Uncovered_Processor if it not covered.

The agent at each processor will record the following variables so that it can keep track of the state of the processor and trace back to the home processor once it has covered the entire network.

- Processor_ID

This identifier is fixed even before an agent visits the processor.

- Source_Node = null

This variable helps to keep track of the path that leads back to the home processor.

- List of ProessorID[Neighbor ID]

This is a list of processor identifiers for each corresponding neighbor of the processor.

- Leader_ID

This identifier is the value of the identifier of the processor, which is elected leader. The algorithm for network covering that the agent executes when it visits a processor is shown in figure 3.

3.2.1.1 Correctness of network traversal algorithm

Lemma 1: The network-covering algorithm ensures that the agent covers the entire network before initiating the leader announcement algorithm.

Network covering means changing the state of each processor in the network from uncovered to the covered state. This algorithm will ensure network covering as the algorithm ensure that the agent performs a depth first search of the entire network while changing the state of each processor from uncovered to covered state.

1. Update ProcessorID[NeighborID] = Agent.Previous_Processor_ID
2. Agent.Previous_Processor_ID = Processor_ID
3. IF Processor_ID != Agent.Current_Uncovered_Processor AND Agent.Covered_Previous = False THEN
    a. IF processor does not exist in the agent list THEN
        i. Add [Processor ID, Uncovered] to the agent list
        ii. Source_Node = Agent.Current_Uncovered_Processor
    b. END IF
    c. IF Current Processor is now covered THEN
        i. Update [Processor ID, Covered]
    d. END IF
    e. Go back to Agent.Current_Uncovered_Processor
4. ELSE
    a. IF Processor_ID != Agent.Current_Uncovered_Processor AND Agent.Covered_Previous = True THEN
        i. Agent.Current_Uncovered_Processor = Processor_ID
        ii. Agent.Covered_Previous = False
    b. END IF
    c. IF Current Processor is now covered THEN
        i. Update [Processor ID, Covered]
        ii. Agent.Covered_Previous = True
        iii. IF all neighbors are covered THEN
            1. IF Processor is home processor THEN
                a. Initiate the 'NETWORK ANNOUNCEMENT ALGORITHM'
            2. ELSE
                a. Go back to Source_Node
            3. END IF
        iv. ELSE
            1. Choose a neighbor having uncovered state randomly and migrate
        v. END IF
    d. ELSE
        i. Choose an unidentified neighbor randomly and migrate
    e. END IF
5. END IF

Fig. 3. Network Covering Algorithm.

It is known fact that a depth first search will cover the entire network. Hence this algorithm will cover the entire network.

Lemma 2. The algorithm is deadlock free.

As per the algorithm, an agent shall visit a processor at most d + 1 times, d is the number of neighbors of the processor. This will happen when either the processor or its neighbors are being discovered. So the agent shall not visit a processor after that. Thus, an agent will never get trapped in a cycle and loop for ever. However, it may visit the processor while the agent traces back to the home processor. Since the Source_Node constructs a spanning tree, rooted at the home processor, the agent shall always be able to reach the home processor.

3.2.1.2 Analysis of network traversal algorithm

- Time Complexity (T): Time taken by the agent is no more than 3*m + n - 1 where m is the number of edges in the network. In the worst-case $m = n^2$.

Termination time of the algorithm is considered in terms of the number of hops taken before the agent call the leader announcement algorithm. This then becomes applicable to both asynchronous and synchronous networks. It is assumed that the time taken to execute the above algorithm is negligible compared to the hop time. The hop time may vary with each hop in an asynchronous network but is a constant time in synchronous system. To identify a processor's neighbor, the agent shall first visit the neighbor and inform the neighbor of the processor id and then come back and inform the processor of the neighbor's processor identifier. This requires the agent to cross an edge both ways. Once the agent changes the processor's state to covered it shall choose a neighboring processor with uncovered state. So the processor will cross the edge once more. This way, the agent shall traverse each edge 3 times. When an agent finds that has all neighbors of the current

processor are covered, then the agent chooses the Source_Node. The Source_Node variable is basically a spanning tree of the network with the home processor of the agents serving as root of the spanning tree. Since a spanning tree has no more than n-1 edges, hence the number of edges the agent will traverse shall be n -1. Hence, the time take for the agent to traverse the network is 3*m+n-1 which is equivalent to O(m+n).

$$T \in O(m+n) \quad \ldots(1)$$

- Message Complexity (Ψ)

The number of messages is equivalent to the number of hops taken by the client. The maximum number of hops that an agent takes in order to cover the network is 3*m+n-1.

$$\Psi \in O(m+n) \quad \ldots(2)$$

3.2.2 Leader Announcement Algorithm

Once the agent has covered the network entirely, then each processor is in the covered state. This means that each processor has knowledge of the processors that are its neighbors. The agent now need to announce the leader to each processor and thus initiates the leader announcement algorithm. At the point of initiation of the leader announcement algorithm, the agent resets the processor state to uncover for all processor in the briefcase. The agent computes the leader based on the largest processor identifier seen in the list of processor identifiers. When the agent visits a processor it performs the execution shown in figure 4.

```
1. Source_Node = Agent.Previous_Processor_ID
2. Change the state of processor to covered in the briefcase
3. Inform the processor if it is the leader or not.
4. Let d = number of neighbors of processor that have state uncovered in briefcase
5. IF d = 0 THEN
        a. IF processor is the home processor THEN
                i. Terminate
        b. ELSE
                i. Go back to Source Neighbor
        c. END IF
6. ELSE
        a. Agent.Previous_Processor_ID = Processor_ID
        b. Choose a neighbor that has uncovered state in the briefcase with a probability of 1/d
7. END IF
```

Fig. 4. Leader Announcement Algorithm.

3.2.2.1 Correctness of leader announcement algorithm

Lemma 1. The leader announcement algorithm ensures that all processors know the elected leader.

Here the agent performs a modified depth first search. The agent visits a processor's neighbor only if it is not covered. The agent shall trace back to home only when all the neighbors of a processor are covered. This is similar to the depth first search except that the agent visits fewer edges, as the neighbors are already known. As the DFS algorithm is known to be deadlock free, the agent shall successfully announce the leader to the entire network.

Lemma 2. The algorithm is deadlock free

This algorithm is a modification of depth first search algorithm that works even for cycles, and is known to be deadlock free.

3.2.2.2 Analysis of leader announcement algorithm

- Time Complexity: Time taken by the agent is no more than 2*n.

The time complexity of the algorithm is the number of units needed for the agent to announce the leader of the network. Here, it is assumed that the time taken to execute the above algorithm is negligible compared to the hop time. The maximum time taken by the agent is 2*n as the agent hops only to processors that it never visited.

$$T \in O(n) \qquad \ldots(3)$$

- Message Complexity: The number of messages is equivalent to the number of hops taken by the client which is 2*n.

Over all, the two algorithm together have a time complexity in worst case of 3*(m + n) and the same message complexity.

$$\Psi \in O(m+n) \qquad \ldots(4)$$

3.3 Leader Election with Multiple Agents

A purely distributed system has no central coordinator. A processor would need to communicate with all the processors to carry out a task. When processors in the network realize there is no leader in the network then the processors will initiate the leader election. As initiation takes place at multiple processors, one or more agents (one on each processor) are created that will search for the leader. The algorithms proposed for the previous single agent model can work with slight modification but the algorithm will not be always efficient. In order to have a better performance (with respect to time) by using many agents, agents

need to work in parallel with more knowledge of what other agents are doing. The task for leader election is split into two simpler tasks – neighbor identification and leader announcement.

3.3.1 Neighbor Identification Algorithm

Unlike a single agent model where network traversal was the first subtask, the multi-agent model requires only processors to be familiar with their neighbors. The task of each agent is only to identify the neighbors of its home processors. While the agents migrate to other processors, the agents will check if that processor has spawned an agent for leader election. If the processor has not yet created an agent, the agent will spawn a new agent with this processor designated as its home processor. The new agent will then identify the neighbors of the current processor and the parent agent will migrate back to its home processor. The algorithm is shown below in figure 5.

3.3.1.1 Correctness of neighbor identification algorithm

Lemma 1. The algorithm is deadlock free.

In this algorithm, the agent in any step of execution is not waiting on any variable. The read operation is for the processor's identifier, which is a read-only variable. The other read operation is on the variable Agent_Create that is set by the processor or another agent. The agent performs a write on the Leader_ID only to reset in the case that no agent has been created on the processors. As the agent does not wait for any value and continues to identify the neighbors of its home processor, the agent will finish the algorithm without any deadlock.

```
1.  IF Processor_ID != Agent.Parent_Processor THEN
    1.1.  ProcessorID[NeighborID] = Agent.Parent_Processor
    1.2.  Agent.Previous_Processor_ID = Processor_ID
    1.3.  IF Agent_Create = False THEN
        1.3.1.    Agent_Create = True
        1.3.2.    Spawn (Agent_Child)
        1.3.3.    Agent_Child.Parent_Processor = Processor_ID
        1.3.4.    IF Processor_ID < Agent.Parent_Processor THEN
            1.3.4.1.      Agent_Child.SelfDestruct = TRUE
        1.3.5.    END IF
    1.4.  END IF
    1.5.  IF Leader_ID < Agent.Parent_Processor THEN
        1.5.1.    Agent_Child.SelfDestruct = TRUE
    1.6.  ELSE
        1.6.1.    Agent.Leader_ID = Processor_ID
    1.7.  END IF
    1.8.  Migrate to Home Processor
2.  ELSE
    2.1.  ProcessorID[NeighborID] = Agent.Previous_Processor_ID
    2.2.  IF for all identified neighbors there exists one neighbor with processor id >
          Agent.Parent_Processor THEN
        2.2.1.    Agent.SelfDestruct = True
    2.3.  END IF
    2.4.  IF there are still some unidentified neighbors THEN
        2.4.1.    Migrate to neighbor
    2.5.  ELSE IF Agent.SelfDestruct = True THEN
        2.5.1.    Leader_ID = Agent.Leader_ID
        2.5.2.    Agent_Create = False
        2.5.3.    Exit and Cleanup
    2.6.  ELSE
        2.6.1.    Leader_ID = Processor_ID
        2.6.2.    Initialized List and add neighbor to the list and set state as uncovered
        2.6.3.    Add parent processor to list and mark its state as covered
        2.6.4.    Start Leader Announcement Algorithm
    2.7.  END IF
3.  END IF
```

Fig. 5. Neighbor Identification Algorithm for Multiple Agents.

Lemma 2. The neighbor identification algorithm ensures that each processor has knowledge of every neighbor's identity.

To prove the correctness of the algorithm, it is sufficient to show that every agent identifies

all the neighbors of its home processor and an agent is created at every processor.

- In this algorithm, the main role of an agent is to ensure that the neighbors of its home processors are identified. The agent migrates to a neighbor (Step 2.2.1) and returns to the parent with the neighbor's identifier (Step 1.8). The only operation that the agent performs at the home processors is updating the previous processors identifier (Step 2.1) and migrating to another neighbor (Step 2.2.1). The only operation that the agent performs at neighboring processors is reading their identifier (Step 1.2) and updating their neighbor identifier of the processor with its home processors identifier (Step 1.1). Further as there is no deadlock, the agent will be able to identify all the neighbors of its home processor.
- It is not sufficient to allow a processor to create an agent. In the case of asynchronous systems, each processor may not initiate an agent. To ensure that an agent is created at every processor, when an agent visits a processor other than its home processor, it will spawn an agent when no agent was created at the processor prior to its visit (Step 1.3.1). As an agent visits all its neighbors to ensure that an agent is created, agents created in the neighboring processors will ensure that their home processor's neighboring processors have agents and so on. This propagation will spread to the entire network as long as the network is connected and no processors can be isolated. Since this is already an assumption, the algorithm will ensure that agents are created at every processor in the network.

From the above two points, it can be concluded that all the neighbors of every processor

will be identified by the algorithm.

3.3.1.2 Analysis of neighbor identification algorithm

- Time complexity: Time taken by the agent is no more than O(n) in synchronous case.

Synchronous Case:

In the synchronous case, the agents are created at each processor simultaneously. A processor in a network can have no more than n-1 neighbors. The task of the agent is to identify the neighbors of its home processor. This requires n-1 hops to the neighbors and n-1 hops back to the home processors leading to a value of 2*(n-1) which is of the order O(n).

Asynchronous Case:

In the asynchronous case, there is no concept of time. Like the synchronous case, an agent will terminate the algorithm after 2*(n-1) hops.

$$T \in O(n) \qquad \ldots(5)$$

- Message Complexity

The number of messages is equivalent to the number of hops taken by the agent which is approximately 2*(n-1) per agent in the worst case. All agents will traverse each edge of their home processor twice. Further, two agents can cross over simultaneously. Hence the number of hops that agents will make is 4*m in the worst case. So the number of message exchanged is in the order O(m).

$$\Psi \in O(m) \qquad \ldots(6)$$

3.3.2 Leader Announcement

As part of the next subtask, agents will traverse the network to check if their processor has the largest identifier. At the end of this search there will be only one agent that should exist. The agent of the leader will have traversed the entire network and created a spanning tree rooted at the leader processor.

Initially, the agent on creation will initialize a list for storing the processor id and state of each processor with value as uncovered. These processor identifiers are obtained as the agent traverses the network and adds the list of the processors that are the neighbors of a processor. When the agent is starts the leader announcement algorithm, it adds the home processor and its neighbors to this list. The agents start the leader announcement algorithm from the home processors. The agents assume that their home processor has the largest identifier. At the home processors the agent chooses a neighbor that it has not yet covered. The agent hops to a neighbor and updates its state to visited. At the new processors, the agent adds all the neighbors that are not in its list.

Here again the agent chooses a neighbor that is not yet visited and hops to that neighbor. If an agent finds that all neighbors are already visited, then the agent will migrate back to the processors where it came from. The variable Source_Node specifies the source processor (directed towards that home processor).

At every processor that the agent visits, the agent sets the Leader_ID to its home processors identifier. However, if the Leader_ID variable is already set to a larger identifier, the agent will terminate as there exist an agent whose home processor has a larger identifier. The algorithm is shown below in figure 6.

```
1.  IF Agent.Parent_Processor = Processor_ID THEN
    1.1.  Source_Node = HOME
    1.2.  IF all neighbors are visited THEN
          1.2.1.  Terminate
    1.3.  ELSE
          1.3.1.  Agent.Previous_Processor_ID = Processor_ID
          1.3.2.  Migrate to a neighbor which the agent has not yet visited
    1.4.  END IF
2.  ELSE
    2.1.  IF for all neighbors, neighbor does not exist in the list THEN
          2.1.1.  Add neighbor to the list and set state as unvisited
    2.2.  END IF
    2.3.  IF Processor_ID > Agent.Parent_Processor THEN
          2.3.1.  IF Agent_Create = False THEN
                2.3.1.1.  Spawn (Agent_Child)
                2.3.1.2.  Agent_Child.Parent_Processor = Processor_ID
                2.3.1.3.  Leader_ID = Processor_ID
          2.3.2.  ENDIF
          2.3.3.  Exit and Cleanup
    2.4.  ENDIF
    2.5.  IF for all neighbors, there are still some unidentified neighbors THEN
          2.5.1.  IF Agent_Create = False THEN
                2.5.1.1.  Spawn (Agent_Child)
                2.5.1.2.  Agent_Child.Parent_Processor = Processor_ID
                2.5.1.3.  Leader_ID = Processor_ID
          2.5.2.  ENDIF
          2.5.3.  Wait until neighbors are identified
          2.5.4.  Agent_Create = False
    2.6.  END IF
    2.7.  IF Leader_ID > Agent.Parent_Processor THEN
          2.7.1.  Exit and Cleanup
    2.8.  ELSE IF Leader_ID < Agent.Parent_Processor THEN
          2.8.1.  Leader_ID = Agent.Parent_Processor
          2.8.2.  Agent_Create = False
          2.8.3.  For all neighbors
                2.8.3.1.  IF neighbor does not exist in the list THEN
                      2.8.3.1.1.  Add the neighbor to the list
                2.8.3.2.  END IF
          2.8.4.  Source_Node = Agent.Previous_Processor_ID
          2.8.5.  Agent.Previous_Processor_ID = Processor_ID
    2.9.  END IF
    2.10. IF there is a neighbor that is unvisited THEN
          2.10.1.  Migrate to a neighbor whose state is unvisited (by choosing neighbor randomly)
    2.11. ELSE
          2.11.1.  Migrate to Source_Node
    2.12. END IF
3.  END IF
```

Fig. 6. Leader Announcement Algorithm.

3.3.2.1 Correctness of leader announcement algorithm

Lemma 1: The algorithm is deadlock free

The algorithm that the agent executes, allows the agent to cover the entire network as long as its home processors has the largest identifier. There are several ways in which deadlock can occur

1) Leader_ID was not reset and is set to a very large identifier of a processor that does not exist

The Leader_ID is reset at that point when leader election commences at each processor or the agent cleans up at the end of the neighbor identification algorithm. Further, processors are assumed to be non faulty. Hence, for a processor, the Leader_ID variable can only be assigned a value of a processor's identifier that exists.

2) Waiting for a processor to finish Neighbor Identification Algorithm

In the case of neighbor identification algorithm, there arises no deadlock. So an agent that is commences the neighbor identification algorithm is bound to terminate the algorithm. An agent waiting for the algorithm to terminate will wait only for a finite number of steps.

3) No cycles

When an agent migrates to a processors two cases arise. First, the agent covers not all but some neighbors of the processor. In this case, the agent will choose from one of the unvisited neighbors and migrate there. So the agent will never revisit a processor in this case. Second, all neighbors have been covered by the agent. In this case, the agent will

migrate back to the source processors. Here an agent will revisit a processor as long as its neighbors are not covered. But at every visit it will follow a different path, as it will choose from a one of the uncovered neighbors. Since the path will be different every time the agent returns to the source processor, there can be no cycle. So there is no way that the agent can be stuck in a cycle.

Lemma 2. At the end of the leader announcement algorithm, there will be only one agent in the network and all processors will be informed of the leader.

At the beginning of leader announcement algorithm, the agent will try to choose uncovered neighbors of every processor it migrates to in order to cover the network. As there are no cycles in the network, the agent will choose a new path for traversal in an attempt to cover all processors. On every processor that the agent migrates, it will update the variable Leader_ID equal to its home processor identifier, as long as the current Leader_ID value is smaller. However, if the value is larger, the agent terminates. As it is assumed that all processors have distinct identifiers, all agents except one will terminate when each agent attempts to visit all the processors of the network.

3.3.2.2 Analysis of leader announcement algorithm

- Time complexity

Synchronous Case:

The worst-case scenario occurs when an agent start the leader announcement algorithm while all processors have not finished the neighbor identification algorithm. The agent will have to wait at each processor until the neighbor identification algorithm is terminated. The

maximum time any agent will take to complete the neighbor identification algorithm is 2*(n-1). Since in synchronous system, agents are created simultaneously, the maximum waiting time will be at most n-1 which is the number of edges a processor can have. So the time that an agent can take to terminate leader announcement is the sum of waiting time for neighbor identification algorithm and the time to walk on a spanning tree which is 2*(n-1) + 2*(n-1) which is equivalent to O(n).

The total time for a leader election will then be O(n)

Asynchronous Case:

In the asynchronous case, time taken for an agent to hop each link may vary. Further, the processors are not synchronized meaning that agents may be created at anytime not necessarily together. The number of hops that the agent will take for leader announcement algorithm will be 2*(n-1). It is possible that agents are not created at all processors. When an agent visits a processor it will replicate an agent if an agent is not created and the neighbors of the processor have not been identified. In the worst case, replication will occur at every processor but one where the agent was created and the replicated agent shall perform neighbor identification before the former agent can proceed. So the time taken by the agent to come back will be a function of the degree of all vertices of the graph. Let δ be the maximum degree of any vertex in the graph. At each processor, the agent will wait for the neighbor identification to terminate which is equivalent to the degree of the vertex (processor). Since there are n-1 such processors, the equivalent time will be $2*(n-1)*\delta$. The agent having traversed all the processors may find the last processor with the largest identifier. In that case, the agent created at the last processor will start leader announcement

and take another 2*(n-1). The overall time will be 4*(n-1) + 2*(n-1)* δ which is equivalent to O(n*δ).

$$T \in O(n*\delta) \quad \ldots(7)$$

- Message complexity

The number of messages is equivalent to the total number of hops that all the agents make. Each agent can make at most 2*(n-1) hops. Since there are n agents, the total number of hops is 2*n*(n-1). This means a message complexity of $O(n^2)$.

$$\Psi \in O(n^2) \quad \ldots(8)$$

After the termination of the network-covering algorithm only a single agent exists in the network. Hence only a single agent runs the leader announcement algorithm. The result for leader announcement algorithm remains the same.

3.4 Example: Synchronous Bi-directional Rings

A synchronous bi-directional ring can be represented by an undirected connected graph G with n vertices where every vertex on the graph is connected to exactly two vertices as shown in figure 7. At the time of leader election initiation each processor shall create an agent and the neighbor identification algorithm will commence. Each agent will identify one of its neighbors and then the other incase it has not been identified. At the end of round two, there will be a maximum of n/3 agents. The reason being that an agent does 2 comparisons, so only one agent can be a winner among the 3 agents. Then these agents will commence the leader announcement algorithm.

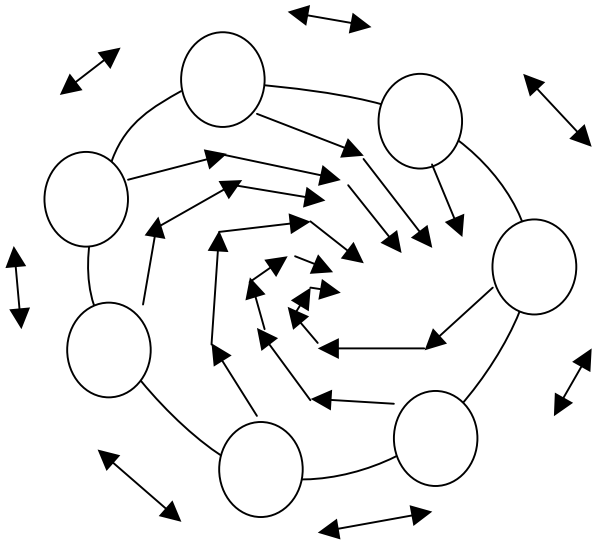

Fig. 7. Leader Election in Synchronous Bi-directional Ring.

For the leader announcement algorithm, the agents will choose a random direction (either clock wise or anticlockwise) and will migrate to processors in the same direction until it encounters a processor with a larger identifier then its home processor. Since all processors have unique identifiers, there will be only one agent at the end of n+2 rounds. The number of messages exchanged at the end of the leader election will be 2*n + n + n*(n-1)/2 which is the order $O(n^2)$. The first n comes as each agent makes a hop to its neighbor and back. The second term is a hop for only those processors that did not have one of their neighbors identified. This can be maximum half the total number of processors. The third term comes as each agent makes 1 less hop than the previous agent. Starting from n-1 hops down to 1 the sum of n consecutive number is of the order $O(n^2)$. The time taken by the algorithm is 4 + n which is in the order of O(n).

CHAPTER IV

CASE STUDY: EFFECT OF NETWORK DYNAMICS ON LEADER ELECTION

The algorithm proposed for leader election works with fixed networks with the assumption that processors never fail while the leader election is taking place. In this subsection, the various cases related to network dynamics are considered. There are several scenarios possible. What will happen in such scenarios? The answer is considered in the following cases.

4.1. Processor Join

In a network, it is possible that a processor may join the network at different stages of leader election.

4.1.1. Before Leader Election

This case is elementary, as leader election process will commence in this processor as the processor will call the leader election algorithm or an agent will migrate to the processor and upon finding that no agent was created, create an agent using replication.

4.1.2. During Neighbor Identification

While the neighbor identification algorithm is running, if a processor joins the network, then once again, the processor will call the leader election algorithm or an agent will migrate to the processor and create an agent, which will start the neighbor identification algorithm.

4.1.3. During Leader Announcement

While agents are running the leader announcement algorithm in the network, there are two possibilities.

Case 1: There is a possibility that the agents have not visited at least one processor that is the neighbor of the new processors.

In this case, the agent will migrate to the new processor as well. When an agent migrates to the new processors, it will find that the agent is not created and will replicate an agent, which will start the neighbor identification algorithm. The agent will wait for the processor's agent to finish the neighbor identification algorithm only if the agent's home processor identifier is larger than the new processors identifier. In the other case, when the agent's home processor identifier is smaller than the new processor's identifier, the agent will terminate and the processor's agent will finish the neighbor identification algorithm and then commence the leader announcement algorithm.

Case 2: The agents have already visited the all the processors that are the neighbors of the new processors.

This case is treated similar to termination of leader election algorithm that is discussed as the next case.

4.1.4. After Leader Election Algorithm Terminates

When the leader election algorithm terminates and a processor joins the network, it will depend on the processor when it initiates the leader election algorithm. Once the processor initiates the leader election, an agent will be created that will execute the neighbor

identification algorithm. If the identifier of processor is smaller than the leader processor's identifier, then the agent will terminate after the termination of the neighbor identification algorithm. Before the termination of the algorithm, the agent will set the leader identifier for the processor and the processor and its neighbors will have knowledge of each other. If the identifier of the processor is larger than the leader processor's identifier, then the agent will initiate the leader announcement algorithm after the neighbor identification algorithm and set the leader identifier for all the processors equal to the new processor identifier.

4.2. Processor Leaves

There is a possibility that processors terminate due to reasons such as crashes or communication link failure may cause a processor to get disconnected from the network. For the leader election, processors with identifier smaller than leader are not important. Only when a processor with the largest identifier disconnects from the network, is there a need to find another leader. The processor can disconnect during several stages of the leader election process. When the processor disconnects the processor's agent may cause another agent of the processor with the next largest identifier to terminate. If the exists an agent that has not yet covered this processor (with the next largest identifier), then it will create the agent again and this agent will run the leader announcement algorithm and terminate the leader election algorithm successfully. If however there is no agent that will visit this processor with the next largest identifier, then the leader election algorithm with terminate abnormally without a leader elected. In this case, after a certain timeout the processors will detect that there is no leader and restart the leader election algorithm.

CHAPTER V

CONCLUSION AND FUTURE WORK

A solution for leader election problem in distributed network was presented. The algorithms for leader election was built on agent based self-stabilization framework. The solution is generalized of any type of topology of the network. The termination time in case of synchronous networks is found to be of the order O(n). In asynchronous networks, the number of hops before termination of leader election is bounded by O(m+n). The message complexity for both synchronous and asynchronous networks was found to be approximately $O(n^2)$. An advantage of this technique for solving leader election is that an agent records the processor identifiers for each neighbor of all processors. So in case of subsequent runs of leader election algorithms, these can be reused and save the time for network covering. From the example, it was observed, that when the leader announcement was included in the network covering algorithm then the leader announcement message and time complexity is eliminated.

In the paper [21], the time complexity obtained was very high and comparable to the message complexity of the algorithm. In this study, a much small order of time complexity is obtained though the algorithm is limited to use in only bi-directional networks. Further, in the study, the effect of implementing leader election when processors leave or join the network was examined. Several cases were drawn to see the effectiveness of the algorithm. As part of the future work, the study of leader election for the case of network partitioning and network merging needs to be examined.

REFERENCES

[1] J. Welch and H. Attiya, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. London, UK: McGraw-Hill Publishing Company, 2001.

[2] Y. Afek and A. Gafni, "Time and message bounds for election in synchronous and asynchronous complete networks," in *Proc. 4th Annu. ACM Symp. on Principles of Distributed Computing*, Minaki, Canada, Aug. 1985, pp. 186-195.

[3] J. E. Burns, "A formal model for message passing systems," *Tech. Rep. TR-91*, Indiana University, Sep. 1980.

[4] D. Dolev, M. Klawe, and M. Rodeh, "An O(nlogn) unidirectional distributed algorithm for extrema finding in a circle," *Journal of Algorithms*, vol. 3, no. 3, pp. 245-260, Sep. 1982.

[5] G. Fredrickson and N. Lynch, "The impact of synchronous communication on the problem of electing a leader in a ring," in *Proc. 16th Annu. ACM Symp. on Theory of Computing*, Washington, D.C., 1984, pp. 493-503.

[6] E. Gafni and Y. Afek, "Election and traversal in unidirectional networks," in *Proc. 3rd Annu. ACM Symp. on Principles of Distributed Computing*, Vancouver, B.C., Canada, Aug. 1984, pp. 190-198.

[7] E. Gafni and Y. Afek, "Simple and efficient distributed algorithms for election in complete networks," in *Proc.. 22nd Annu. Allerton Conference on Communication, Control, and Computing*, Monticello, Ill., Oct. 1984, pp. 689-698.

[8] E. Gafni and W. Korfhage, "Distributed election in unidirectional Eulerian networks," in *Proc. 22nd Annual Allerton Conference on Communication, Control, and Computing*, Monticello, Ill., Oct. 1984, pp. 699-700.

[9] R. G. Gallager, "Choosing a leader in a network," *Unpublished memorandum*, M.I.T., Cambridge, Mass., 1977.

[10] R. G. Gallager, P. M. Humblet, and P. M. Spira, "A distributed algorithm for minimum-weight spanning trees," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 1, pp. 66-77, Jan. 1983.

[11] D. S. Hirshberg, and J. B. Sinclair, "Decentralized extrema-finding in circular configurations of processors," *Commun. ACM*, vol. 23, no. 11, pp. 627-628, Nov. 1980.

[12] P. Humblet, "Selecting a leader in a clique in O(n log n) messages," in *Intern. Memo., Laboratory for Information and Decision Systems*, M.I.T., Cambridge, Mass., 1984.

[13] E. Korach, S. Moran, and S. Zaks, "Tight lower and upper bounds for some distributed algorithms for a complete network of processors," in *Proc. 3rd Annu. ACM Symp. on Principles of Distributed Computing*, Vancouver, B.C., Canada, Aug. 1984, pp. 199-207.

[14] E. Korach, D. Rotem, and N. Santoro, "A probabilistic algorithm for decentralized extrema-finding in a circular configuration of processors," *Technical Report*, University of Waterloo, Ontario, Canada, 1981.

[15] E. Korach, D. Rotem, and N. Santoro, "Distributed algorithms for finding centers and medians in networks," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 3, pp. 380-401, July 1984.

[16] I. Lavalléé and G. Roucairol, "A fully distributed (minimal) spanning tree algorithm," *Information Processing Letters*, 23, pp. 55-62, Aug. 1986.

[17] P. M. B. Vitanyi, "Distributed election in an Archimedean ring of processors," *in Proc. 16th Annu. ACM Symp. on Theory of Computing*, Washington, D.C., 1984, pp. 542-547.

[18] S. Ghosh, "Cooperating mobile agents and stabilization," in *Proc. Workshop on Self-stabilization*, pp. 1-18, Springer, 2001.

[19] A. Ital and M. Rodeh, "Symmetry breaking in distributed networks," *Information and Computation*, vol. 88, no. 1, pp. 60-87, Sep. 1990.

[20] Y. Azar, A. Z. Broder, A. R. Karlin, N. Linial, and S. Phillips, "Biased random walks," in *Proc. 24th Annu. ACM Symp. on the Theory of Computing*, Victoria, British Columbia, Canada, May 1992, pp. 1-9.

[21] E. Korach, S. Kutten and S. Moran, "A modular technique for the design of efficient distributed leader finding algorithms," *ACM Trans. on Programming Languages and Systems*, vol. 12, no. 1, pp. 84-101, Jan. 1990

[22] E. W. Dijkstra, "Self-stabilization systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, pp. 643-644, Nov. 1974.

[23] A. Mayer,Y. Ofek, R. Ostrovsky and M. Yung, "Self-stabilizing symmetry breaking in constant-space," in *Proc. 24th ACM Symp. on Theory of Computing*, Victoria, British Columbia, Canada, May 1992, pp. 667-678.

[24] T. Herman, "Self-stabilization at WSS'01 and DISC'01," *ACM SIGACT News*, vol. 33, no. 1, pp. 54-57, Mar. 2002.

[25] B. Krishnamachari, X. Xie, B. Selman, and S. Wicker, "Analysis of random noise and random walk algorithms for satisfiability testing," in *Proc. 6th International Conference on Principles and Practice of Constraint Programming*, Singapore, Sep. 2000, pp. 278-290.

[26] S. Dolev, E. Schillert, and J. Welch, "Random walk for self-stabilizing group communication in ad hoc networks," in *Proc. 21st IEEE Symp. on Reliable Distributed Systems*, Osaka University, Suita, Japan, Oct. 2002, pp. 70-79.

[27] A. Israeli and M. Jalfon, "Token management schemes and random walks yield self-stabilizing mutual exclusion," in *Proc 9th ACM Symp. on Principles of Distributed Computing*, Quebec City, Quebec, Canada, Aug. 1990, pp. 119-131.

[28] P. Tetali and P. Winkler, "On a random walk problem arising in self-stabilizing token management," in *Proc. 10th ACM Symp. on Principles of Distributed Computing*, Montreal, Quebec, Canada, Aug. 1991, pp. 273-280.

VITA

Raghav Tandon was born in Lucknow, India, on July 10, 1977. He received a Bachelor of Engineering in computer engineering, in June 1999 from the Netaji Subhas Institute of Technology (formerly known as Delhi Institute of Technology), New Delhi, India. From July of 1999, he worked in the position of software engineer and senior software engineer at Lucent Technologies, Bangalore, India. When Lucent Technology received the Bell Labs certification, he was assigned the position of Member of Technical Staff. He entered graduate school at Texas A&M University in pursuit of a Master of Science in computer engineering in July 2001. While pursuing his masters, he also worked as a research assistant in the area of Web GIS for Transportation at TTI. His permanent address is: c/o Adesh Tandon, 24 South Park Apartments, Kalkaji, New Delhi, India-110019.

The typist of this thesis was Raghav Tandon.