Properties of Link Reversal Algorithms for Routing and Leader Election

by

Tsvetomira Radeva

B.S., Computer Science, B.S., Mathematics
State University of New York, College at Brockport, 2010

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2013

Signature of Author: ....
Department of Electrical Engineering and Computer Science
March 4, 2013

Accepted by: ....
Nancy Lynch, Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by: ....
Leslie A. Kolodziejski, Chair of the Committee on Graduate Students

Properties of Link Reversal Algorithms for Routing and Leader Election

by

Tsvetomira Radeva

Submitted to the Department of Electrical Engineering and Computer Science on March 4, 2013, in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science

Abstract

We present two link-reversal algorithms and some interesting properties that they satisfy. First, we describe the Partial Reversal (PR) algorithm [13], which ensures that the underlying graph structure is destination-oriented and acyclic. These properties of PR make it useful in routing protocols and algorithms for solving leader election and mutual exclusion. While proofs exist to establish the acyclicity property of PR, they rely on assigning labels to either the nodes or the edges in the graph. In this work we present simpler direct proof of the acyclicity property of partial reversal without using any external or dynamic labeling mechanisms.

Second, we describe the leader election (LE) algorithm of [16], which guarantees that a unique leader is elected in an asynchronous network with a dynamically-changing communication topology. The algorithm ensures that, no matter what pattern of topology changes occurs, if topology changes cease, then eventually every connected component contains a unique leader and all nodes have directed paths to that leader. Our contribution includes a complexity analysis of the algorithm showing that after topology changes stop, no more than $O(n)$ elections occur in the system. We also provide a discussion on certain situations in which a new leader is elected (unnecessarily) when there is already another leader in the same connected component. Finally, we show how the LE algorithm can be augmented in such a way that nodes also have the shortest path to the leader.

Thesis Supervisor: Nancy Lynch
Title: Professor of Electrical Engineering and Computer Science

Contents

1 Introduction **6**
- 1.1 Overview of Link Reversal Algorithms . . . 6
  - 1.1.1 Basic Link Reversal Algorithms . . . 6
  - 1.1.2 Algorithms for Routing and Leader Election . . . 8
  - 1.1.3 Other Link Reversal Algorithms . . . 11
- 1.2 Acyclicity Property of Partial Reversal . . . 12
  - 1.2.1 Existing Acyclicity Proofs . . . 12
  - 1.2.2 Our Results . . . 13
- 1.3 Leader Election Algorithm for Dynamic Networks . . . 14
  - 1.3.1 Leader Election Related Work . . . 14
  - 1.3.2 Leader Election Algorithm with Global Clocks . . . 15
  - 1.3.3 Leader Election Algorithm with Causal Clocks . . . 16
  - 1.3.4 Our Results . . . 16

2 Overview of I/O Automata **18**
- 2.1 I/O Automata . . . 18
- 2.2 Composition of I/O Automata . . . 19
- 2.3 Fairness . . . 20
- 2.4 Simulation Relations . . . 21

3 Partial Reversal Acyclicity **22**
- 3.1 System Model . . . 23
- 3.2 Original Algorithm . . . 23
  - 3.2.1 Algorithm Description . . . 23
  - 3.2.2 Properties . . . 25
- 3.3 New Algorithm . . . 27
  - 3.3.1 Algorithm Description . . . 27
  - 3.3.2 Acyclicity Property . . . 29
- 3.4 Simulation Relation . . . 32
  - 3.4.1 Description of *OneStepPR* . . . 33
  - 3.4.2 Relation between *PR* and *OneStepPR* . . . 33
  - 3.4.3 Relation between *OneStepPR* and *NewPR* . . . 36
- 3.5 Conclusion . . . 39

4 Leader Election Algorithm **40**

4.1 System Model 41

4.1.1 System Components 41

4.1.2 Logical Time 43

4.2 Problem Statement 45

4.3 Leader Election Algorithm 46

4.3.1 Informal Description of the LE Algorithm 46

4.3.2 The Height Structure 47

4.3.3 State Variables of the LE Algorithm 48

4.3.4 Description of the LE Algorithm 50

4.4 Proof of Correctness 55

4.4.1 Basic Invariants 55

4.4.2 Finite Number of Elections 56

4.4.3 Finite Number of New RL's 57

4.4.4 Accurate View of Neighbors' Heights 59

4.4.5 Leader-oriented DAG 60

4.5 Complexity Analysis 61

4.6 Stability Properties of the LE Algorithm 65

4.6.1 One-topology-change Stability Property 65

4.6.2 Other Stability Properties 67

4.6.3 Stability Properties of Other Algorithms 69

4.7 LE Algorithm and Shortest Paths 69

4.8 Conclusion and Future Work 71

5 Conclusion **73**

List of Figures

4.1 *Channel*$(\{u, v\})$ automaton 42
4.2 Automaton *LE* for node u 51
4.3 Code triggered by *channelDown* 52
4.4 Code triggered by the receipt of a message. 54
4.5 Subroutines of the LE Algorithm 54
4.6 Counterexample execution 67

Chapter 1

Introduction

1.1 Overview of Link Reversal Algorithms

Link reversal algorithms are a special class of distributed algorithms which have gained popularity in solving various problems such as routing, leader election, mutual exclusion and resource allocation in distributed systems [34]. A common feature of link reversal algorithms is a directed-graph structure in which the vertices represent computing nodes executing the algorithm, and the directions of the edges are reversed by the nodes under certain conditions in order to achieve some property required by the problem specification. In link reversal algorithms each computing node is responsible for reversing its incident edges only when a particular local property of the node is satisfied. The goal of link reversal algorithms is to eventually satisfy some global property in the entire system.

Link reversal algorithms were first introduced by Gafni and Bertsekas in [13] as a way of providing an efficient graph structure for routing. In [13], the authors presented a class of algorithms which are very simple and elegant, and at the same time have interesting and useful properties. Different modifications of these basic ideas are currently used to design efficient and elegant algorithms for solving various problems in distributed computing [34]. Our main goal in this thesis is to explore the properties of two particular link reversal algorithms, to prove such properties in a simple and clear way, and to study the behavior of these algorithms under different models of computation.

1.1.1 Basic Link Reversal Algorithms

The first two link reversal algorithms proposed in [13], are *Full Reversal (FR)* and *Partial Reversal (PR).* The input to each algorithm is a directed acyclic graph (DAG) with a fixed destination node, in which some node(s) may not have a directed path to the destination node. During the execution of the algorithms, each node reverses some (or all) of its incident edges in such a way that eventually each node in the system has a directed path to the destination. A node executes a step of either algorithm only if it is a sink (all its incident edges are incoming). The destination node never takes any steps. In FR when a node is a sink it reverses all of its incident edges. In PR, each node keeps a list of the edges reversed by its neighbors the previous time they took a step. When a node is a sink, it reverses only the edges that are *not* in the list, and then clears the list. In other words, while in FR nodes always reverse all their incident edges, in PR it is possible to reverse fewer edges.

Even though PR seems to be much more efficient than FR, the worst case performance for both algorithms is the same. We measure the efficiency of both algorithms by comparing the total number of reversals performed by all nodes. For FR, the authors of [34], following an approach from [3] and [4], prove a tight bound of $\Theta(n_b^2)$ worst case total number of reversals, where n_b is the number of nodes that have no path to the destination initially. In [34], they also show that the same tight bound applies to PR. Since such a conclusion is surprising and counter-intuitive, Charron-Bost et al. [7] apply a game theoretical approach in showing that PR is more efficient than FR. The authors conclude that the strategy of FR is a Nash equilibrium, but it has the largest social cost among all Nash equilibria, while the strategy of PR is not necessarily a Nash equilibrium, but if it is, it achieves a global optimum and has the minimum social cost.

In [13], the authors showed how to express both the FR and PR algorithms in such a way that the nodes in the graph are assigned unique labels from a totally ordered set, and reversals are based only on the values of these labels. Even though it has not been formally shown that these alternative algorithms are equivalent to PR and FR, we provide a brief description of these ideas because they are useful in understanding more complicated algorithms presented later in this thesis. The labels mentioned above are used to induce directions on the edges in the graph using the following simple rule: an edge is always directed from a node with a larger label to a node with a smaller label. For example, in the modification of FR, called the Pair Algorithm, each node's label (also called height) consists of two components: an integer and the node's unique id. When a node is a sink, it changes the first component of its height to one more than the largest height of its neighbors. The second component of the height is just a tie-breaker. It is easy to see that the simple rule used in the Pair Algorithm ensures that whenever a node is a sink it reverses all of its incident edges, just like in FR.

A similar modified algorithm is also available in the case of PR; it is called the Triple Algorithm. The Pair and Triple Algorithms are described in detail in [13]. Having such an assignment of heights as in the Pair and Triple Algorithms, it is easy to see that it is not possible for cycles to exist in the graph (since labels are unique and drawn from a totally-ordered set). Recall that the main application of FR and PR is for routing, so such an acyclicity property is crucial for the correctness and efficiency of any routing protocol. The authors of [13] show that such assignments of heights exist so that the properties of the Pair and Triple Algorithms are satisfied (including acyclicity); however, formal proofs of the equivalence of FR to the Pair Algorithm, and of PR to the Triple Algorithm, are not provided. The acyclicity property of PR is one of the two main topics of this thesis, so the relationship between PR and the Triple Algorithm is explored in more detail in Section 1.2 .

In [6], the authors present an even wider range of link reversal algorithms in which each edge in the graph is labeled (as opposed to labeling nodes as in [13]). Another difference in [6] is that the labels are binary, as opposed to unbounded in [13]. In addition to a DAG with a fixed destination node, the input to the Binary Link Labels (BLL) algorithm [6] also includes a labeling of 0 or 1 for each edge in the graph. The BLL algorithm itself is very simple; a node reverses some of its incident edges based on two simple rules about the labels of these edges:

- LR1: If at least one incident edge is labeled with 0, then reverse all the incident edges labeled with 0 and flip the labels on all incident edges.
- LR2: If no incident edge is labeled with 0, then reverse all the incident edges.

The authors of [6] show some very interesting properties of BLL summarized as follows:

1. No matter what the initial labeling of edges in the graph is, eventually the resulting graph is a destination-oriented DAG.
2. FR and PR are both special cases of BLL. FR is a special case of BLL in which the initial labeling is all-1's, and PR is a special case of BLL in which the initial labeling is all-0's.
3. Under a certain global property of the number and type of the edges in the graph, the BLL algorithm maintains the acyclicity of the graph. Both the all-0 and the all-1 initial labellings satisfy this global property, which implies that the FR and PR algorithms preserve the acyclicity of the graph.

As mentioned earlier, the acyclicity property of PR, in particular, is one of the two main topics of this thesis. In Section 1.2 we describe the two existing proofs (mentioned above) for the acyclicity property of PR, and we present a much simpler proof that does not use any labels on the nodes and edges and is a direct proof for the acyclicity property of PR.

1.1.2 Algorithms for Routing and Leader Election

The algorithms described so far use fairly simple rules to determine the edges to be reversed at each step. Now we look at some more complicated algorithms which are designed for networks in which the graph is dynamically changing. In such models, edges are allowed to go up and down, so the topology can change at each step of the algorithm execution. Such changes in the topology are usually assumed to be caused by the mobility of nodes. Therefore, algorithms designed for dynamic graphs can be used in a variety of practical networks, such as mobile ad-hoc networks, for example.

Temporally Ordered Routing Algorithm (TORA)

It is easy to see that the algorithms in [13] fail to produce routes to the destination in a dynamic-graph model. Consider an execution in which the destination is partitioned from the rest of the graph. In such an execution, the nodes in both FR and PR keep reversing edges infinitely often in search for the destination. Therefore, if the destination never reconnects to the rest of the graph, the nodes are not capable of detecting such a partition.

This problem is overcome in the Temporally Ordered Routing Algorithm (TORA) presented in [23].

Similarly to the original link reversal algorithms in [13], the input to TORA is a DAG with a fixed destination node, and the output is a destination-oriented DAG. In the dynamic-graph model, however, it is possible for the destination to be partitioned from the rest of the nodes. Therefore, another requirement that TORA satisfies is that if the destination is not in the same connected component as a given node, then eventually that node erases the incorrect routes it has to the destination.

Each node in TORA has a 5-tuple of integers, called a *height*, where the first three components of the height are called the *reference level*, the fourth component is called the *delta* value, and the last component is the node's unique id. We describe the purpose of each of these components in the next paragraph as we explain how the algorithm works. These heights are very similar to the heights of nodes used in the Pair and Triple Algorithms in [13]. Just like in [13], heights are compared lexicographically and are used to induce directions of the edges in the graph: an edge is always directed from a node with a larger height to a node with a smaller height. Since heights are unique and drawn from a totally-ordered set, it is easy to see that no cycles can occur in an execution of TORA.

Initially, all nodes start with an all-0 reference level component, and a delta value set to the shortest-path distance to the destination. The delta value here serves the purpose of orienting the edges in such a way that each node has a directed path to the leader. Since all of the first three components are 0, it is the delta value that determines the direction of the edges. When a node loses its last outgoing edge (it is a sink), due to an edge going down or a neighbor changing its height, it is clear that the node has no path to the destination any more. Therefore, any such node starts a "search" for the destination by increasing its height so that all of its edges now become outgoing. To be precise, the node changes its reference level part of the height to the following three values: (1) the current time, which denotes the time the search started, (2) the node's id, which denotes which node started the search and (3) 0, which denotes that the search is still in progress, for the three components of the reference level, respectively. Note that since nodes have access to a non-decreasing common global clock, changing the reference level in such a way guarantees that the new height is greater than all neighbors' heights.

Next, this new reference level propagates throughout the connected component. As some nodes increase their heights, their neighbors become sinks and need to increase their heights as well. Therefore, the reference level started by some node is adopted by other nodes in the same connected component, in search for an alternative path to the destination. The mechanism through which a reference level propagates throughout the connected component is different from the initial generation of the reference level. When a node becomes a sink and receives a reference level from a neighbor, it adopts the newly received reference level (by setting its first three components to the newly received reference level) and sets its delta value to one less than the delta value of the node from which the reference level was received. Therefore, by setting the delta value in such a way, the node ensures that the directions of the newly reversed edges coincide with the direction of the search for the destination, which continues forward and does not go back to the originator prematurely.

If some part of the search described above hits a dead end, nodes need to inform the originator of the reference level. To do so, the node that detects the dead end sets the third component (the reflection bit) of the current reference level from 0 to 1, which signifies that the reference level is now going inwards towards the originator of the reference level, as opposed to going outwards searching for the destination. When the originator of the reference level receives reflected reference levels from all its neighbors, it decides that the destination is partitioned from the current connected component. Therefore, all routes to the destination need to be erased.

The algorithm described above was designed to be used in practical networks, and it is relatively efficient compared to existing routing algorithms in dynamic networks [28]. However, most of the theoretical properties of TORA are left underexplored. While [35] provides a proof of the correctness of TORA in static networks, the complete proof of correctness of the algorithm is still not present. In the same paper, the authors also establish some properties of the behavior of TORA in dynamic networks. Some of the properties we prove in this thesis are also expected to apply to TORA too. We outline these properties in Section 1.3.

Leader Election Link Reversal Algorithm

A key insight in [21] is that the ideas of TORA can be easily modified to solve the problem of leader election (LE). In brief, an algorithm which solves leader election ensures that all nodes in the system elect a unique node to be the leader. The algorithm of [21] also satisfies an additional property: eventually, each node in the system has a directed path to the current leader. The system model of [21] is very similar to the one in TORA where a particular feature is the dynamically-changing graph.

The main idea of the algorithm in [21] is to use the information that TORA provides about the partitioning from the destination. In the case of LE, there is no fixed destination node; instead, the current leader is treated as the destination. Another difference is the addition of a sixth component to the height of [23] which contains the id of the current leader. Initially, there is a fixed leader in the graph and each node has a path to the leader. The execution of the LE algorithm proceeds very similarly to TORA. When a node determines that the current leader is not in the same connected component, it elects itself, and sends a wave of messages, encoded in the height, informing all nodes in the connected component that a new leader is elected.

The main disadvantage of the LE algorithm described above is that its correctness is guaranteed only in the case of a single topology change. The case in which the algorithm fails is when multiple topology changes cause multiple leaders to be elected. The algorithm is not capable of consistently determining which leader should win over the entire connected component.

To solve that problem a new algorithm was presented in [17] which adds a seventh component to the height of each node. The seventh component records the time at which a leader is elected. When multiple leaders exist in a single connected component, the oldest one (with the smallest timestamp) wins over the older leaders.

In the LE algorithm of [17], two of the components of a node's height are timestamps determined by the current value of a global clock. In [16] we asked the question of what happens to the correctness of the algorithm if a global clock is not accessible to the nodes. It turns out that the problem specification of LE is still satisfied even if nodes use local *causal* clocks instead.

In brief, a causal clock is a generic local clock at each node which guarantees that if there is "causal chain" of events between some event e_1 and some other event e_2, then we can determine the global order of events e_1 and e_2 by comparing the local causal clock times at which they occurred. Causal clocks do not have any information about the current time, but instead, they establish a causality relationship between events in the system. In [16] we show that this causality relationship is enough to ensure the correctness of the algorithm. However, other properties of the LE algorithm change in this new model.

This modified algorithm [16] and its properties are the second main topic of this thesis. In Section 1.3 we describe the algorithm in more detail, the properties of interest, and we analyze the run-time complexity of the algorithm.

1.1.3 Other Link Reversal Algorithms

A great resource for a summarized study of link reversal algorithms is provided by Welch and Walter in [34], where they present a tutorial about some of the most popular link reversal algorithms. Besides the algorithms we already discussed above, [34] also describes many more algorithms that solve problems such as mutual exclusion, distributed queuing, and scheduling in a distributed system. A few of these algorithms are briefly described below, summarizing their descriptions in [34].

The mutual exclusion algorithm in [29] ensures that all nodes in the system access a particular section of their code, called the *critical* section, in such a way that no two nodes access their critical sections at the same time. Also, all nodes which request to enter the critical sections should eventually be allowed to. The algorithm in [29] assumes that the communication graph is a tree and that there is a single token in circulation which grants access to the critical section. Directions of the edges in the graph ensure that the only sink in the graph is the node currently possessing the token. The topology being a tree ensures that if a node is not a sink then its only outgoing edge is in the direction of the node which currently has the token. Therefore, when a node requests to access its critical section, it send a request on that edge. When a node exits the critical section, it sends the token to one of its descendants who requested the token (in a FIFO way). When the token is sent over an edge, the direction of that edges is reversed. Further improvements to this algorithm are presented in [30] and [32]. Ideas from [27] and [33] are used in [34] to show the correctness of the algorithm.

A modification of the above algorithm is presented in [10] and [15] as the Arrow Protocol, which solves the problem of distributed queuing. The goal of the algorithm is to construct a total order of the nodes in the system so that their requests/actions can be ordered in some consistent way, in a distributed manner. In the Arrow Protocol, each node requests to join the total order and eventually learns its successor in the ordering, but not the entire ordering. A complete algorithm description and correctness proof are provided in great detail in [34].

Finally, an interesting modification of the Full Reversal (FR) algorithm, described earlier, can be shown to provide a solution to the problem of scheduling in a graph. The authors of [5] show that, if the destination node in FR is removed, then each node is a sink infinitely often. The main application of this property is in graph scheduling where each node is required to take some particular action infinitely often, while satisfying some properties (such as no two neighbors being scheduled at the same time).

1.2 Acyclicity Property of Partial Reversal

As mentioned in Section 1.1.1, Partial Reversal (PR) [13] is a link reversal algorithm designed to provide paths from each node in a directed acyclic graph (DAG) to a unique destination node. Since the main application of the algorithm is routing, it is crucial to ensure that the algorithm does not create any cycles in the graph. Currently, there are two distinct proofs, in [13] and [34], which show that the algorithm maintains the acyclicity of the graph. First, we describe the two proofs and then we propose a simpler proof which uses only properties of the PR algorithm and gives better insight into how the algorithm works.

1.2.1 Existing Acyclicity Proofs

In the original paper that introduces PR [13], the authors presented the Triple Algorithm which is claimed to be equivalent to the PR algorithm. In the Triple Algorithm, each node is assigned a triple of integers (a, b, id), which are compared lexicographically and used to induce directions on the edges of the graph. An edge is always directed from a node with a larger height to a node with a smaller height. The first two components of the height, a and b, are just integers used by the algorithm to reverse particular edges, while the third component, id, is the node's unique id, used as a tie-breaker.

During the execution of the Triple Algorithm, each node changes its height only if it is a sink. The label of a sink vertex u is changed to ensure that the new label is larger than those neighbors of u with the smallest first component, but smaller than the labels of all other neighbors. In more detail, if u is a sink with value of its triple (a_0, b_0, id), then u changes its height to (a_1, b_1, id) where:

- a_1 is one more than the minimum of all the a-values of all its neighbors.
- b_1 is one less than the minimum of all the b-values of only these neighbors of u whose a-value is a_1. If there are no such neighbors of u, b_1 is set to b_0.

These rules ensure that when a node is a sink it reverses the edges only to some of its neighbors (and sometimes all). Even though the authors in [13] show that such an assignment of heights exists, there is no formal proof that the above algorithm is equivalent to PR. Therefore, even though the Triple Algorithm ensures the acyclicity of the graph, this proof does not automatically apply to PR.

The second proof for the acyclicity of PR is presented in [6] and uses the fact that PR is a special case of BLL. As we mentioned earlier, in Section 1.1.1, PR is a special case of BLL in which the initial labeling is all 0's. In [6], it is also shown that for any circuit in the initial graph (and its labeling), if the following inequality is satisfied, then no cycles are created during the algorithm execution:

$$(w+s)(r+s) > 0, \text{ where:}$$

- r is the number of edges labeled with 1 which are right-way (directed in some particular direction, WLOG clockwise).
- w is the number of edges labeled with 1 which are wrong-way (directed in the opposite direction, WLOG counter-clockwise).
- s is the number of nodes such that both of its incident edges (pertaining to the given circuit) are incoming and labeled 0.

It is easy to see that in the case of PR the inequality above is satisfied because there are no edges labeled with 1 at all. Therefore, the value of $(w + s)(r + s)$ is the square of the number of sinks initially, which is guaranteed to be positive.

While the above property is very general and useful, its proof is rather involved. In Chapter 3, we present a much simpler proof for the acyclicity property of PR.

1.2.2 Our Results

In Chapter 3, we present a new simpler proof of the acyclicity property of PR, which does not use any mechanism of labeling nodes or edges. This part of the thesis has been presented in [24] and [25].

First, we introduce a new version of the original PR algorithm. In the original PR algorithm, each node keeps a dynamic list of neighbors which determines the set of edges to be reversed. However, if we observe the sets of edges reversed at each step, we notice that edges corresponding to the same sets of neighbors are reversed at every other step. Therefore, our new algorithm uses only the original sets of incoming and outgoing neighbors of each node, and reverses the corresponding set of edges, alternating between the two. Having such a simpler and more static algorithm, it is easier to prove that no cycles exist at any point of the execution. Our acyclicity proof relies on a few invariants based on the number of steps nodes have taken, and unlike existing proofs, does not use any labeling of the nodes or edges of the graph.

Finally, since the new algorithm is very similar to the original one, we provide a simulation relation from the original algorithm to the new one, to formally show a mapping between the two. The simulation relation establishes a correspondence between the different lists in the two algorithms, and concludes that for every step of the original algorithm, there exists a sequence of steps in the new algorithm, so that both algorithms result in the same directions of the edges in the graph. Because of general properties of simulation relations, such a relation shows that our new acyclicity proof carries over to the original PR algorithm.

1.3 Leader Election Algorithm for Dynamic Networks

In Section 1.1.2 we described a leader election (LE) algorithm [16] that uses a link reversal approach to elect a unique leader in a dynamically-changing graph. In this section, we provide some background about LE algorithms and their applications and consider two particular versions of the LE algorithm described in Section 1.1.2: one using a global clock [17] and one using causal clocks [16]. We briefly describe both versions of the LE algorithm, specify some of their interesting properties, and state our results.

1.3.1 Leader Election Related Work

The leader election problem has been studied extensively in various models in distributed computing, including static and dynamic networks. In this section we focus on LE algorithms for dynamic networks so that we can try to compare them to the LE algorithms studied in this thesis (those in [17] and [16]) in terms of efficiency, simplicity, and stability properties. A more detailed related work section is available in [16] describing multiple LE algorithms for dynamic networks; here, we mention just a few of the most popular algorithms.

Even among algorithms for dynamic networks, there are various algorithms designed for very different models. For example, in [14], Hatzis et al. present a LE algorithm for dynamic networks in which the mobility of nodes is limited in particular ways so that the communication between them is not affected. In [22] and [31] respectively, Masum et al. and Vasudevan et al. describe two LE algorithms for the broadcast model (as opposed to point-to-point communication). Finally, in [2] Brunekreef et al. describe an algorithm in which nodes are allowed to crash and subsequently recover.

The LE algorithm we consider in this thesis, similarly to the ones in [17] and [16], assumes point-to-point communication, no restrictions on topology changes, and no fault tolerance. Several other algorithms exist which solve the LE problem in similar models. In [11], Derhab and Badache present a leader election algorithm for mobile ad hoc networks which is based on [21] (similarly to the LE algorithms discussed in this thesis). Unfortunately, the proof of correctness in [11] is only for the synchronous case and assuming only one topology change. Two other LE algorithms are proposed in [8] and [26] but their correctness has not been proved, but established only through simulations.

Recently, a LE algorithm for dynamic networks has been presented in [9] which includes a complete proof of correctness, complexity analysis and description of interesting stability properties. Unlike the algorithms we consider in this thesis, the algorithm in [9] is self-stabilizing and works in a shared-memory model. These differences in the system model make it difficult to compare the two algorithms in terms of efficiency, but it is be interesting to see how the stability properties differ in the two models. A discussion about these properties appears in Chapter 4. Moreover, the algorithm in [9] is completely asynchronous, unlike [17] where nodes have access to a global clock. The two algorithms in [17] and [16], however, are simpler and it is much easier to reason about their correctness and stability properties.

1.3.2 Leader Election Algorithm with Global Clocks

As we mentioned in Section 1.1.2, the LE algorithm with global clocks [17] is a modification of the Temporally Ordered Routing Algorithm (TORA) [23]. First, we provide a very high-level description of the algorithm.

The input to the algorithm is a DAG with a unique leader node, where each node in the system has a directed path to the leader. The goal of the algorithm is to maintain a leader-oriented graph despite edges in the graph going up and down.

Each node in the system has a 7-tuple of integers called a height. Five of these components serve the same purpose as the components in TORA. As mentioned earlier in Section 1.1.3, two extra components are added in order to record the identity of the current leader and the time it was elected. Whenever a node is a sink (has no outgoing edges), it has no path to the current leader any more, so it reverses all of its incident edges. Reversing all incident edges acts as the start of a search mechanism for the current leader. Each node that becomes a sink (as a result of a neighbor reversing the common edge) reverses some edges to its neighbors and in effect propagates the search throughout the connected component. Once a node becomes a sink but all its neighbors already participate in the same search, it means that the search has hit a dead end and the current leader is not present in that part of the connected component. Such dead-end information is then propagated back towards the originator of the search. When a node that started a search receives such dead-end messages from all of its neighbors, it concludes that the current leader is not present in the connected component, and so the originator of the search elects itself as the new leader. Note that this mechanism is the same as the search for the destination in TORA (described in Section 1.1.2), the only difference being that if the originator of the search receives dead-end information from all its neighbors, it elects itself instead of just deciding that the destination has been partitioned. Finally, the information about the new leader propagates throughout the network via an extra "wave" of messages. In other words, the leader floods the network with messages containing the new leader ID and time of election.

In the algorithm described above, two of the components of a node's height are timestamps recording the time when a new "search" for the leader is started, and the time when a leader is elected. In the algorithm [17], these timestamps are obtained from a global clock accessible to all nodes in the system. A complete proof of correctness of the above algorithm is provided in [17].

Next, we describe an interesting property of the LE algorithm which establishes how often the leader changes. Due to the dynamicity of the network in which the algorithm executes, it is natural that if links go up and down often the leader also changes often. It is clear that when a connected component gets partitioned from the current leader, the algorithm is required to elect a new leader in the partitioned component in order to satisfy the problem specifications. In other cases, however, it is possible that a leader is present in a given connected component, but a new one is elected (unnecessarily) and replaces the old leader. A stability property of LE is defined in [17] to explain under what circumstances such behavior is guaranteed not to occur. The stability property outlines a particular case in which only a single topology change occurs. In this specific case it is shown that no new leader is elected in the connected component of the current leader. In this thesis we also discuss how other types and sequences of topology changes can affect the algorithm execution.

1.3.3 Leader Election Algorithm with Causal Clocks

Next, we present a modification of the LE algorithm in [17], proposed in [16], which uses a more general type of clocks, called *causal clocks*, instead of a global clock, for the two timestamp components of the height of each node. In brief, the main differences between the two types of clocks can be summarized as follows:

1. Instead of having access to a single global clock as in [17], each node has access to a local clock in [16].

2. While the global clock in [17] provides each node with the same time base, the only property that causal clocks satisfy is the causal relation between events. A formal definition of causal clocks is available in [16] but we do not use that definition in this thesis. Intuitively, causal clocks guarantee that we can determine the order in which two events occurred only if there is a "causal chain" of events linking one to the other.

A complete proof of correctness of the LE algorithm with causal clocks is provided in [16]. In this thesis, we include the statements of the main results in the proof and build upon them in Section 4.5.

Recall the stability property we described above, which applies to the global-clock version of the algorithm. In [16], we provided a counterexample showing that this stability property does not hold in the case of causal clocks. In particular, we showed that it does not hold when nodes have access to Lamport clocks [19].

1.3.4 Our Results

In this section, we provide a summary of the results related to the leader election algorithm included in the thesis.

We consider a slight modification of the timing model. Instead of working with a global clock as in [17] or causal clocks as in [16], we focus on a particular type of causal clocks that we call logical clocks. We present the LE algorithm and summarize the proof of correctness from [16]. Since causal clocks are a generalization of logical clocks, the proof of correctness in [16] carries over to the LE algorithm in our timing model.

One contribution of this thesis is the complexity analysis of the LE algorithm with respect to the number of elections that the algorithm performs before converging. In Chapter 4, Theorem 4.5.4, we show that after topology changes stop, at most $O(n)$ elections occur in any arbitrary execution of the LE algorithm, before stabilizing to a connected component with a unique leader.

Another contribution of the thesis is a discussion of various stability properties satisfied by the LE algorithm. In Chapter 4, Section 4.6, we show a particular counterexample execution in which a single topology change causes a new leader to be elected and consequently, messages to be sent throughout the entire connected component. We also describe additional stability properties which are, unfortunately, not satisfied by the LE algorithm. We also compare these properties to the stability properties satisfied by the algorithm in [9].

Finally, in Chapter 4, Section 4.7, we provide an additional improvement of the LE algorithm. Namely, we show how to combine the LE algorithm with a shortest-path component such that the augmented algorithm guarantees that (1) a unique leader is elected in the system, and (2) each node in the system has the shortest directed path to the elected leader.

Chapter 2

Overview of I/O Automata

In this chapter, we provide the mathematical background needed in understanding the system model we use throughout the thesis – I/O Automata. We summarize some of the relevant the definitions of I/O automata provided in [20]. We begin by providing the main definitions in Section 2.1. Next, in Section 2.2, we define an operation on I/O automata, called composition, which helps us compose different I/O automata components together to form a more complicated system. In Section 2.3 we describe the notion of *fairness* which describes how different components in the system all "get turns" to execute their actions. Finally, in Section 2.4 we present a particular proof technique, called a simulation relation, which "runs automata side by side" to provide a way to compare their behaviors.

2.1 I/O Automata

The I/O Automata model is a very general model for asynchronous computation, which is useful in modeling not only the algorithms in this thesis, but also a wide range of distributed algorithms for both message-passing and shared-memory systems. An I/O automaton is a distributed component which interacts with other system components through *actions*. Actions can be of three types: *input*, *output*, and *internal*. The input and output actions are used for communication between automata and with the environment. Internal actions are only visible to the automaton. While the automaton decides when to execute internal and output actions, input actions are not under its control; they just arrive from another automaton or from the environment.

Formally, the *signature* of an I/O automaton is a description of its input, output and internal actions. A signature S is a triple consisting of three disjoint sets of actions: the input actions, $in(S)$, the output actions, $out(S)$, and the internal actions, $int(S)$. We define the *external actions*, $ext(S)$, to be $in(S) \cup out(S)$; the *locally controlled actions*, $local(S)$, to be $out(S) \cup int(S)$; and $acts(S)$ to be all the actions of S.

An I/O automaton A consists of five components:

- $sig(A)$, a signature

- *states*(A), a (not necessarily finite) set of states
- *start*(A), a nonempty subset of *states*(A) known as the start states or initial states
- *trans*(A), a state-transition relation, where *trans*$(A) \subseteq$ *states*$(A) \times$ *acts*$($*sig*$(A)) \times$ *states*(A); this relation must have the property that for every state s and every input action π, there is a transition $(s, \pi, s') \in$ *trans*(A)
- *tasks*(A), a task partition, which is an equivalence relation on *local*$($*sig*$(A))$ having at most countably many equivalence classes

We call an element (s, π, s') of *trans*(A) a *transition*, or a *step*, of A. If for a particular state s and action π, A has some transition of the form (s, π, s'), then we say that π is enabled in s.

The fifth component of the I/O automaton definition, the task partition *tasks*(A), is used to define fairness conditions of an execution of an automaton. These conditions ensure that, during its execution, the automaton gives fair turn to each one of its tasks.

Next, we define an *execution* of an I/O automaton. An *execution fragment* of A is either a finite sequence $s_0, \pi_1, s_1, \pi_2, \cdots, \pi_r, s_r$ or an infinite sequence $s_0, \pi_1, s_1, \pi_2, \cdots, \pi_r, s_r, \cdots$ of alternating states and actions of A such that $(s_k, \pi_{k+1}, s_{k+1})$ is a transition of A for every $k \geq 0$. If the execution is finite, it must end with a state. An execution fragment beginning with a start state is called an *execution*. We denote the set of executions of A by *execs*(A). A *trace* of an execution α of A, denoted by *trace*(α), is the subsequence of α consisting of all external actions. We denote the set of all traces of all executions of A by *traces*(A).

2.2 Composition of I/O Automata

The composition operation allows for a complex system to be constructed by composing automata representing simpler building blocks of the system. The composition identifies actions with the same name in different component autmata. When any component automaton performs an action π, then all other components that have π in their signatures also perform it. Formally, we define a countable collection $\{S_i\}_{i \in I}$ of signatures to be *compatible* if for all $i, j \in I$, $i \neq j$, all of the following hold:

1. *int*$(S_i) \cap$ *acts*$(S_j) = \emptyset$
2. *out*$(S_i) \cap$ *out*$(S_j) = \emptyset$
3. No action is contained in infinitely many sets *acts*(S_i)

We say that a collection of automata is compatible if their signatures are compatible. When we compose a collection of automata, output actions of the components become output actions of the composition, internal actions of the components become internal actions of the composition, and actions that are inputs to some components but outputs of none become input actions of the composition. Formally, the *composition* $S = \Pi_{i \in I} S_i$ of a countable compatible collection of signatures $\{S_i\}_{i \in I}$ is defined to be the signature with

- $out(S) = \cup_{i \in I} out(S_i)$
- $int(S) = \cup_{i \in I} int(S_i)$
- $in(S) = \cup_{i \in I} in(S_i) - \cup_{i \in I} out(S_i)$

Now the *composition* $A = \Pi_{i \in I} A_i$ of a countable, compatible collection of I/O automata $\{A_i\}_{i \in I}$ can be defined. It is the automaton defined as follows:

- $sig(A) = \Pi_{i \in I} sig(A_i)$
- $states(A) = \Pi_{i \in I} states(A_i)$
- $start(A) = \Pi_{i \in I} start(A_i)$
- $trans(A)$ is the set of triples (s, π, s') such that, for all $i \in I$, if $\pi \in acts(A_i)$, then $(s_i, \pi, s'_i) \in trans(A_i)$; otherwise, $s_i = s'_i$
- $tasks(A) = \cup_{i \in I} tasks(A_i)$

One example of using the composition operation on automata is to model a message-passing system where each channel and each process is a separate I/O automaton. By naming the send and receive actions properly, we can just use the composition operation to build the entire system. For example, if the receive action of a process i is named $rcv_i(m)$ and the action through which a channel delivers a message m to process i is also called $rcv_i(m)$, they will always get executed simultaneously. This way, we do not have to worry about making sure that when the channel delivers a message to the process the process actually receives it.

2.3 Fairness

Recall that the fifth component of the definition of an I/O automaton is a partition of locally-controlled actions where each equivalence class in the partition represents some task that the automaton is supposed to perform. The notion of fairness is that each task gets infinitely many opportunities to perform one of its actions.

Formally, an execution fragment α of an I/O automaton A is said to be *fair* if the following conditions hold for each class C of $tasks(A)$:

1. If α is finite, then C is not enabled in the final state of α.
2. If α is infinite, then α contains either infinitely many events (occurrences of actions) from C or infinitely many occurrences of states in which C is not enabled.

In other words, each task C is given turn infinitely often. When that happens, either an action of C gets performed or no action of C can be performed because none is enabled. We denote the set of fair executions of A by $fairexecs(A)$. We say β is a *fair trace* of A if β is the trace of a fair execution of A, and we denote the set of fair traces by $fairtraces(A)$.

2.4 Simulation Relations

In certain cases, it is useful to be able to show that a property "carries over" from one automaton to another. For example, if a simple automaton B is shown to satisfy a given property, we may want to extend B to a more complicated automaton A and still need that property to hold. One way to do this is to prove the property all over again, hoping that a similar proof to the one for B would work for A. Another approach is to run automata A and B "side by side" and observe their behaviors. If the external behaviors with respect to the property of interest follow some expected pattern, then we can determine whether the property that is true for B is also true for A. Such a proof technique is called a *simulation relation.*

Formally, let A and B be two I/O automata with the same external interface; we think of A as the lower-level automaton and B as the higher-level automaton. Suppose f is a binary relation over $states(A)$ and $states(B)$, that is, $f \subseteq states(A) \times states(B)$; we use the notation $u \in f(s)$ as an alternative way of writing $(s, u) \in f$. Then f is a simulation relation from A to B, provided that both of the following are true

1. If $s \in start(A)$, then $f(s) \cap start(B) \neq \emptyset$.
2. If s is a reachable state of A, $u \in f(s)$ is a reachable state of B, and $(s, \pi, s') \in trans(A)$, then there is an execution fragment α of B starting with u and ending with some $u' \in f(s')$, such that $trace(\alpha) = trace(\pi)$.

The first condition, or start condition, asserts that any start state of A has some corresponding start state of B. The second condition, or step condition, asserts that any step of A, and any state of B corresponding to the initial state of the step, have a corresponding sequence of steps of B. This corresponding sequence can consist of one step, many steps, or even no steps, as long as the correspondence between the states is preserved and the external behavior is the same.

The definition above is used in the next theorem which states the main property of simulation relations.

Theorem 2.4.1. *If there is a simulation relation from A to B, then $traces(A) \subseteq traces(B)$.*

In Chapter 3, we use the notion of a simulation relation in a slightly different way. Instead of comparing the resulting traces after applying the function f in part (2) of the definition, we compare the resulting topology of the communication graph.

Chapter 3

Partial Reversal Acyclicity

In this chapter, we present the Partial Reversal (PR) algorithm and a new simple and direct proof of its acyclicity property. As mentioned in Chapter 1, PR is a link-reversal algorithm that works in a directed acyclic graph and guarantees that the system reaches a state where each node has a directed path to some fixed destination node, by only reversing its incident edges. We are particularly interested in a property of PR that states that no cycles are created at any point during the execution of the algorithm. In Chapter, 1 we described two particular proofs of the acyclicity property of PR, one of which [13] proves the property for a different algorithm (the Triple Algorithm) and the authors of [13] claim that it carries over to PR, and the other one [6] proves the property for a more general algorithm, of which PR is a special case. In this chapter, we present a direct and simpler proof of the acyclicity property of PR.

First, we introduce a new version of the original PR algorithm. In the original PR algorithm, each node keeps a dynamic list of neighbors which determines the set of edges to be reversed. However, if we observe the sets of edges reversed at each step, we notice that edges corresponding to the same sets of neighbors are reversed at every other step. Therefore, our new algorithm uses only the original sets of incoming and outgoing neighbors of each node, and reverses the corresponding set of edges, alternating between the two. Having such a simpler and more static algorithm, it is easier to prove that no cycles exist at any point of the execution. Our acyclicity proof relies on a few invariants based on the number of steps nodes have taken, and unlike existing proofs, does not use any labeling of the nodes or edges of the graph.

Finally, since the new algorithm seems to be very similar to the original one, we provide a simulation relation from the original algorithm to the new one, to formally show a mapping between the two. The simulation relation establishes a correspondence between the different lists in the two algorithms, and concludes that for every step of the original algorithm, there exists a sequence of steps in the new algorithm, so that both algorithms result in the same directions of the edges in the graph. The existence of such a relation shows that our new acyclicity proof carries over to the original PR algorithm.

The rest of the chapter is organized as follows: Section 3.1 describes how we model the system; Section 3.2 presents the original PR algorithm in more detail, and shows some useful properties of the algorithm; Section 3.3 describes our new algorithm and some of its properties, including the acyclicity proof; Section 3.4 provides a simulation relation from PR to the new algorithm, and presents the main conclusion, in Theorem 3.4.5, that PR maintains acyclicity using our new proof; Section 3.5 summarizes our results.

The results of this chapter appeared in [25] and [24].

3.1 System Model

We model the system as an undirected graph $G = (V, E)$ where V is the set of nodes and E is the set of edges. The graph has a single predetermined destination node $D \in V$. The set of neighbors of a particular node u in G is defined as $nbrs_u$. Since no nodes and edges are added or removed from the graph, G is constant throughout the execution of the algorithm. Let a directed version of G be denoted as $G' = (V, E')$, such that for a given edge $\{u, v\} \in E$ either $(u, v) \in E'$ or $(v, u) \in E'$, but not both. We also define an initial graph G'_{init} which represents the initial directed graph. Assuming G'_{init} is fixed, let $in\text{-}nbrs_u$ and $out\text{-}nbrs_u$ be the sets of nodes corresponding to incoming and outgoing edges of any node u in G'_{init}. Note that $nbrs_u$ is defined as the set of neighbors of u in G (the undirected graph), while $in\text{-}nbrs_u$ and $out\text{-}nbrs_u$ are defined with respect to G'_{init} (the initial directed graph). None of the sets $nbrs_u$, $in\text{-}\ nbrs_u$, and $out\text{-}nbrs_u$ changes throughout the execution of the algorithm, and so $nbrs_u = in\text{-}nbrs_u \cup out\text{-}nbrs_u$ at any state of the system.

3.2 Original Algorithm

3.2.1 Algorithm Description

In this section we present the original PR algorithm [13] and express it as an I/O automaton (*PR*). Refer to Chapter 2 for an overview of I/O Automata.

The entire system is modeled as a single I/O Automaton (as described in [20]) with a single set of actions – *reverse*(S). In other words, the only action a node can do is to reverse a set of incident edges. The set S represents all nodes that are taking a step together, where each one of these nodes reverses a set of edges to its neighbors. The destination node D does not reverse its incident edges, and so it is never in S. For each node u, *PR* has a state variable *list*$[u]$ which contains all the neighbors of u which took a step since the last time u took a step. Initially *list*$[u]$ is empty. Additionally, the *PR* automaton has a state variable, *dir*$[u, v]$, one for each ordered pair (u, v), which represents the direction of the edge between u and v from u's perspective.

The only precondition for the *reverse*(S) action is that all nodes in S are sinks. The effect of the reversal is that the edge between u and each neighbor of u *not* in *list*$[u]$ is reversed (from *in* to *out*). However, if *list*$[u]$ contains all neighbors of u, then the edges to all neighbors are reversed. Also, each neighbor v of u that has its edge to u reversed, adds u to *list*$[v]$. Finally, after reversing the particular edges, u empties *list*$[u]$.

Algorithm 1 *PR* automaton

```
Signature:
  $reverse(S)$, $S \subseteq V$, $S \neq \emptyset$, $D \notin S$

States:
  for each $u$, $v$ where $\{u, v\} \in E$:
    $dir[u, v] \in \{in, out\}$, initially $in$ if $v \in in\text{-}nbrs_u$ or
                                          $out$ if $v \in out\text{-}nbrs_u$
    $dir[v, u] \in \{in, out\}$, initially $in$ if $u \in in\text{-}nbrs_v$ or
                                          $out$ if $u \in out\text{-}nbrs_v$
  for each $u$, $list[u]$, a set of nodes $W \subseteq nbrs_u$, initially empty

Transitions:
$reverse(S)$
   Precondition:
     for each $u \in S$
       for each $v \in nbrs_u$, $dir[u, v] = in$
   Effect:
     for each $u \in S$
       if $list[u] \neq nbrs_u$ then
         for each $v \in nbrs_u \setminus list[u]$
           $dir[u, v] := out$
           $dir[v, u] := in$
           $list[v] := list[v] \cup \{u\}$
       else
         for each $v \in nbrs_u$
           $dir[u, v] := out$
           $dir[v, u] := in$
           $list[v] := list[v] \cup \{u\}$
       $list[u] := \emptyset$
Tasks:
  $\{reverse(S), S \subseteq V, S \neq \emptyset, D \notin S\}$
```

3.2.2 Properties

The following invariants establish some basic properties of the algorithm above. Invariant 3.2.1 ensures the consistency of edge directions with respect to both endpoints of the edge. Invariant 3.2.2 shows the possible contents of *list*$[u]$ for any node u. Corollary 3.2.3 follows directly from Invariant 3.2.2 concluding that if u is not a sink, then *list*$[u]$ is a subset of either *in-nbrs*$_u$ or *out-nbrs*$_u$. Corollary 3.2.4 states that *list*$[u]$ must be equal to either the set of *in-nbrs*$_u$ or the set of *out-nbrs*$_u$, whenever u is a sink.

Invariant 3.2.1. *In every reachable state of PR, for each u and v where $\{u, v\} \in E$, $dir[u, v] = in$ iff $dir[v, u] = out$.*

Proof. Initially, each $dir[u, v]$ variable is set according to *in-nbrs*$_u$, *out-nbrs*$_u$, *in-nbrs*$_v$, and *out-nbrs*$_v$, so if the edge $\{u, v\}$ is directed from v to u, then $dir[u, v] = in$, and $dir[v, u] = out$.

Assuming this property is true in some state s, we now show that it remains true in any state s' that is reachable from s in a single step of the algorithm. If neither u nor v reverses the edge between them, then $dir[u, v]$ and $dir[v, u]$ remain the same, so the invariant remains correct in s'. If u takes a step and reverses its edge to v, then $s.dir[u, v] = in$ because u is a sink in s. Therefore, $s.dir[v, u] = out$. When u executes a step of the algorithm, it sets $s'.dir[u, v] = out$ and $s'.dir[v, u] = in$. Therefore, the property remains true in s'. If v takes a step in s, then $s.dir[v, u] = in$. When v reverses the edge, it sets $s'.dir[v, u] = out$ and $s'.dir[u, v] = in$, and the property remains true. □

The following invariant states that at any state of the system *list*$[u]$ consists of either only *in-nbrs*$_u$ or *out-nbrs*$_u$. Also, since all nodes in the list already reversed their edges back to u, all edges corresponding to nodes in the list are incoming. We also show that if the list consists of *in-nbrs*$_u$ (*out-nbrs*$_u$), then all *out-nbrs*$_u$ (*in-nbrs*$_u$) have incoming edges to u.

Invariant 3.2.2. *In every reachable state of PR, for each node u, exactly one of the following is true:*

1. *For each $w \in$ out-nbrs$_u$, $dir[u, w] = in$ and*
   $list[u] = \{v | v \in$ in-nbrs$_u$ and $dir[u, v] = in\}$.

2. *For each $w \in$ in-nbrs$_u$, $dir[u, w] = in$ and*
   $list[u] = \{v | v \in$ out-nbrs$_u$ and $dir[u, v] = in\}$.

Proof. (by induction on the number r of completed steps)

Initially, the list is empty. Part 2 is true because all *in-nbrs*$_u$ initially have incoming edges to u, and also because no *out-nbrs*$_u$ initially have incoming edges to u. We also need to show that part 1 is false. If u is a source, part 1 does not hold because the direction of the edges to all *out-nbrs*$_u$ is *out*. If u is not a source, part 1 is false because *list*$[u]$ is empty initially.

Assuming the property is true after r steps, we now show that it is true after $r + 1$ steps. Let the state of the system after r steps be s, and the state of the system after $r + 1$ steps be s'.

Case 1: The $r+1$'st step of the execution includes a step of u.

Case 1.1: $s.list[u] \neq nbrs_u$ and part 1 is true in s.

We show that part 2 is true in s', and part 1 is false in s'.

Since part 1 is true in s, by the inductive hypothesis $s.list[u] = \{v | v \in \textit{in-nbrs}_u$ and $dir[u,v] = in\}$. Also, u is a sink in s, so all edges to nodes in $\textit{in-nbrs}_u$ are incoming. Therefore, $s.list[u] = \textit{in-nbrs}_u$. Because $s.list[u] \neq nbrs_u$, when u takes a step, it reverses $nbrs_u \setminus \textit{in-nbrs}_u = \textit{out-nbrs}_u$, and so all nodes in $\textit{in-nbrs}_u$ still have incoming edges to u in s'. Also, $s'.list[u] = \emptyset$, and part 2 is true because no $\textit{out-nbrs}_u$ have incoming edges to u in s'. Moreover, part 1 is not true in s' because u has outgoing edges to all nodes in $\textit{out-nbrs}_u$, and since $s.list[u] \neq nbrs_u$ and $s.list[u] = \textit{in-nbrs}_u$, it follows that that $\textit{out-nbrs}_u \neq \emptyset$.

Case 1.2: $s.list[u] \neq nbrs_u$ and part 2 is true in s.

We show that part 1 is true in s', and part 2 is false in s'.

Since part 2 is true in s, by the inductive hypothesis $s.list[u] = \{v | v \in \textit{out-nbrs}_u$ and $dir[u,v] = in\}$. Also, u is a sink in s, so all edges to nodes in $\textit{out-nbrs}_u$ are incoming. Therefore, $s.list[u] = \textit{out-nbrs}_u$. Because $s.list[u] \neq nbrs_u$, when u takes a step, it reverses $nbrs_u \setminus \textit{out-nbrs}_u = \textit{in-nbrs}_u$, and so all nodes in $\textit{out-nbrs}_u$ still have incoming edges to u in s'. Also, $s'.list[u] = \emptyset$, and part 1 is true because no $\textit{in-nbrs}_u$ have incoming edges to u in s'. Moreover, part 2 is not true in s' because u has outgoing edges to all nodes in $\textit{in-nbrs}_u$, and since $s.list[u] \neq nbrs_u$ and $s.list[u] = \textit{out-nbrs}_u$, it follows that that $\textit{in-nbrs}_u \neq \emptyset$.

Case 1.3 $s.list[u] = nbrs_u$ and part 1 is true in s.

We show that part 1 true in s', and part 2 is false in s'.

Since part 1 is true in s, by the inductive hypothesis $s.list[u] = \{v | v \in \textit{in-nbrs}_u$ and $dir[u,v] = in\}$. Also, u is a sink in s, so all edges to nodes in $\textit{in-nbrs}_u$ are incoming Therefore, $s.list[u] = \textit{in-nbrs}_u$. Because $s.list[u] = nbrs_u$, when u takes a step, it reverses $\textit{in-nbrs}_u$, so that all nodes in $\textit{in-nbrs}_u$ have outgoing edges from u in s'. Therefore, part 2 is false because its first condition is false. Moreover, the first condition of part 1 is satisfied because $\textit{out-nbrs}_u = \emptyset$. Additionally, no $\textit{in-nbrs}_u$ have incoming edges to u, so $s'.list[u] = \emptyset$, and thus part 1 is true.

Case 1.4 $s.list[u] = nbrs_u$ and part 2 is true in s.

We show that part 2 is true in s', and part 1 is false in s'.

Since part 2 is true in s, by the inductive hypothesis $s.list[u] = \{v | v \in \textit{out-nbrs}_u$ and $dir[u,v] = in\}$. Also, u is a sink in s, so all edges to nodes in $\textit{out-nbrs}_u$ are incoming. Therefore, $s.list[u] = \textit{out-nbrs}_u$. Because $s.list[u] = nbrs_u$, when u takes a step, it reverses $\textit{out-nbrs}_u$, so that all nodes in $\textit{out-nbrs}_u$ have outgoing edges from u in s'. Therefore, part 1 is false because its first condition is false. Moreover, the first condition of part 2 is satisfied because $\textit{in-nbrs}_u = \emptyset$. Additionally, no $\textit{out-nbrs}_u$ have incoming edges to u, so $s'.list[u] = \emptyset$, and thus part 2 is true.

Case 2: The $r+1$'st step of the execution includes a step of some node $v \in nbrs_u$.

Note that Case 2 is disjoint from Case 1 because no two neighboring nodes can be sinks at the same time. Let $T = nbrs_u \cap S$, that is, T is the set of neighbors v of u such that the $r+1$'st step of the execution includes a step of v. By the definition of the case, $T \neq \emptyset$.

All neighbors of u that take a step in s (all nodes in T) are added to $s'.list[u]$. Let v be an arbitrary neighbor of u in T. In s, the edge between u and v must be from u to v, while in s' the direction of the edge must be from v to u.

Case 2.1: Part 1 is true in s.

We show that part 1 is true in s', and part 2 is false in s'.

By part 1, $s.list[u] \subseteq in\text{-}nbrs_u$ and all nodes in $out\text{-}nbrs_u$ have incoming edges to u. Therefore, $v \notin out\text{-}nbrs_u$, and so $v \in in\text{-}nbrs_u$. When v is added to $list[u]$, it is true that $s'.list[u] = \{v | v \in in\text{-}nbrs_u \text{ and } dir[u, v] = in\}$. No edges to $out\text{-}nbrs_u$ are reversed in this step, so part 1 is satisfied. Part 2 is not true in s' because $s'.list[u]$ contains at least one node, $v \in in\text{-}nbrs_u$, which was just added to $s'.list[u]$ in step $r + 1$.

Case 2.2: Part 2 is true in s.

We show that part 2 is true in s', and part 1 is false in s'.

By part 2, $s.list[u] \subseteq out\text{-}nbrs_u$ and all nodes in $in\text{-}nbrs_u$ have incoming edges to u. Therefore, $v \notin in\text{-}nbrs_u$, and so $v \in out\text{-}nbrs_u$. When v is added to $list[u]$, it is true that $s'.list[u] = \{v | v \in out\text{-}nbrs_u \text{ and } dir[u, v] = in\}$. No edges to $in\text{-}nbrs_u$ are reversed in this step, so part 2 is satisfied. Part 1 is not true in s' because $s'.list[u]$ contains at least one node, $v \in out\text{-}nbrs_u$, which was just added to $s'.list[u]$ in step $r + 1$.

Case 3: Neither u nor any $v \in nbrs_u$ takes a step during the $r+1$'st step of the execution.

Since only u or its neighbors can change $list[u]$, in this case $s.list[u] = s'.list[u]$. Moreover, none of u's incident edges are reversed during this step, so the property remains true.

□

Corollary 3.2.3. *In any reachable state of PR, for any node u,* $list[u] \subseteq in\text{-}nbrs_u$ *or* $list[u] \subseteq out\text{-}nbrs_u$ *(or both if* $list[u] = \emptyset$*).*

Corollary 3.2.4. *In any reachable state of PR, if u is a sink, then* $list[u] = in\text{-}nbrs_u$ *or* $list[u] = out\text{-}nbrs_u$.

3.3 New Algorithm

3.3.1 Algorithm Description

In this algorithm, nodes use only the initial *in-nbrs* and *out-nbrs* sets to determine which edges to reverse in each step. Whenever a node is a sink, it reverses the edges corresponding to either its *in-nbrs* or *out-nbrs* set, alternating between the two. In order to determine which set is about to be reversed, each node keeps track of the parity of the number of steps taken so far. If the node has taken an even number of steps, then it reverses the set of *in-nbrs*; if it has taken an odd number of steps then the set of *out-nbrs* is reversed. Initially, nodes have taken zero steps, so they reverse their *in-nbrs* the first time they take a step. Note that unlike *PR*, this algorithm does not need to maintain any dynamic sets of neighbors, but only the parity of the number of steps taken so far.

The entire system is modeled as a single I/O Automaton with a single set of actions – *reverse*(u) – where u is any node in V, which is currently a sink. The destination node never reverses any of its incident edges, so $u \neq D$. Moreover, associated with each node are

Algorithm 2 *NewPR* automaton

```
Signature:
   $reverse(u)$, $u \in V$, $u \neq D$
States:
  for each $u$, $v$ where $\{u, v\} \in E$:
    $dir[u,v] \in \{in, out\}$, initially $in$ if $v \in in\text{-}nbrs_u$ or
                                          $out$ if $v \in out\text{-}nbrs_u$
    $dir[v,u] \in \{in, out\}$, initially $in$ if $u \in in\text{-}nbrs_v$ or
                                          $out$ if $u \in out\text{-}nbrs_v$
  for each node $u$, $count[u]$, integer, initially 0

Derived State:
  for each node $u$, $parity[u] \in \{even, odd\}$, $even$ if $count[u]$ is even
                                                $odd$ if $count[u]$ is odd
Transitions:
$reverse(u)$
   Precondition:
     for each $v \in nbrs_u$, $dir[u,v] = in$
   Effect:
     if $parity[u] = even$ then
       for each $v \in in\text{-}nbrs_u$
         $dir[u,v] := out$
         $dir[v,u] := in$
     else
       for each $v \in out\text{-}nbrs_u$
         $dir[u,v] := out$
         $dir[v,u] := in$
     $count[u] := count[u] + 1$
Tasks:
  $\{reverse(u), u \in V, u \neq D\}$
```

two variables: $dir[u, v]$ which represents the direction of the edge between nodes u and v, and history variable $count[u]$ which keeps track of the number of steps u has taken so far. There is also has a derived variable $parity[u]$, which is a function of $count[u]$ that represents its parity; it is used to keep track of which set of neighbors is to be reversed next.

The precondition for a node u to perform a $reverse(u)$ action is that it is a sink. The effect of the reversal is that depending on the value of $parity[u]$, either the edges corresponding to nodes in $in\text{-}nbrs_u$ or $out\text{-}nbrs_u$ are reversed. Also, $count[u]$ is incremented, which results in flipping the parity bit.

Note that it is possible that in the $reverse(u)$ action u does not reverse any edges because either $in\text{-}nbrs_u = \emptyset$ or $out\text{-}nbrs_u = \emptyset$. This case occurs only when nodes are initially sinks or sources. When such an action is performed, all u does is increment the step counter (flip the parity bit) without reversing any edges. In this case u remains a sink but now the parity has the correct value, so u can perform a regular $reverse(u)$ action the next time it takes a step.

It is important to notice the main differences between PR and $NewPR$:

- In PR, each node keeps one list of neighbors which changes as edges are reversed, while in $NewPR$ nodes have two constant lists, *in-nbrs* and *out-nbrs*, and a *parity* bit to alternate between the lists.

- In PR, there are two possible ways nodes reverse their edges (depending on whether all neighbors are in the list or not), and so whenever a node is a sink it reverses some edges and empties the list. In $NewPR$, however, it is possible that a node is a sink but the parity does not have the right value to reverse the corresponding set of edges. This happens to nodes that are originally sinks or sources. During this "dummy" step, a node does not reverse any edges but only increments its step count, so the next time it takes a step, the parity corresponds to the list of edges to be reversed. This extra step in $NewPR$ causes it to incur a greater cost in certain situations, compared to PR.

- In PR, a set of nodes takes a step at once, while in $NewPR$ only one node at a time can take a step.

It is important to note that PR keeps a dynamic list of nodes in order to determine which edges to reverse, while $NewPR$ is a lot more static because it always reverses one of two constant sets. We believe that describing the algorithm in such a way simplifies it and makes it easier to understand. Moreover, the dummy step in $NewPR$ helps treat all nodes equivalently and thus makes it possible to state nice invariants based on the number of steps nodes have taken. On the other hand, the increased number of steps, and the restriction of only one node taking a step at a time, affect the complexity of the algorithm, but we are not concerned with this issue in this paper.

3.3.2 Acyclicity Property

The proof of the acyclicity property of $NewPR$ consists of Invariant 3.3.1 and Invariant 3.3.2, which are then combined into Theorem 3.3.3 concluding that PR maintains acyclicity.

Since the input to the PR algorithm is a DAG, we can embed it in a plane, ensuring all edges are initially directed from left to right. Therefore, for each node u all edges associated with nodes in $\textit{in-nbrs}_u$ are to the left of u, and all nodes associated with edges in $\textit{out-nbrs}_u$ are to the right of u.

Invariant 3.3.1 states that if the *parity* of two neighboring nodes is the same, then we can determine whether the edge between them is directed from left to right, or right to left.

Invariant 3.3.1. *In any reachable state, if u and v are neighbors, then:*

(a) If $parity[u] = parity[v] = even$, then the edge $\{u, v\}$ is directed from left to right.

(b) If $parity[u] = parity[v] = odd$, then the edge $\{u, v\}$ is directed from right to left.

Proof. (by induction on the number r of total number of steps taken by all nodes)

In the initial state $parity[u] = parity[v] = even$. Part (b) is vacuously true, and part (a) is true because initially all edges are directed from left to right.

Assume both properties are true after r steps. Let the state of the system after r steps be s. We need to show that the properties are true after $r + 1$ steps. Let the state of the system after $r + 1$ steps be s'.

Note that an arbitrary node can take the $r + 1$'st step. If neither u nor v takes a step, then both properties remain true. Therefore, we are concerned only with cases in which either u or v takes a step. Since the two properties are symmetric with respect to u and v, without loss of generality, assume u is taking the $r + 1$'st step.

If $s'.parity[u] = s'.parity[v] = even$, part (b) is vacuously true, so we show part (a). Since u takes a step, then it must be a sink in s, so the edge $\{u, v\}$ is directed from v to u in s. Since u takes the $r + 1$'st step, then $s.parity[u] = odd$.

Since $s.parity[u] = odd$, by the second case of the code of the $reverse(u)$ action, the edges corresponding to $\textit{out-nbrs}_u$ (to the right of u) are reversed. If v is to the right of u, then the edge $\{u, v\}$ is reversed and is now directed from left to right in s'. If v is to the left of u, the edge $\{u, v\}$ is not reversed and remains directed from left to right.

Similarly, for the proof of part (b), we assume $s'.parity[u] = s'.parity[v] = odd$, which implies that part (a) is vacuous, and we use the same arguments to show that part (b) is satisfied. $\square$

Invariant 3.3.2 has four parts, establishing different properties of the number of steps that nodes have taken. Part (a) gives a range of the possible number of steps of a node v, given the number of steps its neighbor, node u, has taken. Parts (b) and (c) show two possible cases in which it can be concluded that two neighboring nodes have taken the same number of steps. Part (d) states that if one node has taken strictly more steps that its neighbor, then the edge between them is directed from the node which has taken more steps to the node which has taken fewer steps. Combined together the invariants 3.3.1 and 3.3.2 give us a way of using the number of steps and directions of edges to show that it is not possible to create a cycle in the graph.

Invariant 3.3.2. *In any reachable state, if u and v are neighbors, then:*

(a) If count$[u] = n$, then count$[v] \in \{n-1, n, n+1\}$.

(b) If count$[u] = n$, where n is odd, and v is to the right of u, then count$[v] = n$.

(c) If count$[u] = n$, where n is even, and v is to the left of u, then count$[v] = n$.

(d) If count$[u] >$ count$[v]$, then the edge $\{u, v\}$ is directed from u to v.

Proof. (by induction on the number r of total number of steps taken by all nodes)

In the initial configuration no node has taken any steps yet, so *count*$[u]$ = *count*$[v] = 0$. Therefore, all four parts are true initially.

Suppose all properties are true after r steps. Let the state of the system after r steps be s. We need to show that all properties are true after $r+1$ steps. Let the state of the system after $r+1$ steps be s'.

Note that an arbitrary node can take the $r+1$'st step. If neither u nor v takes a step, then all properties remain true. Therefore, we are concerned only with cases in which either u or v takes a step.

Assume $s'.count[u] = k$.

Case 1: u takes the $r+1$'st step. Therefore, u is a sink in s and the edge $\{u, v\}$ is directed from v to u. Also, $s.count[u] = k-1$, and by the inductive hypothesis part (a), $s'.count[v] = s.count[v] \in \{k-2, k-1, k\}$. By the inductive hypothesis part (d), $s.count[v] \geq s.count[u]$, and therefore $s'.count[v] = s.count[v] \in \{k-1, k\}$.

Part (a): $s'.count[u] = k$, and so it is true that $s'.count[v] \in \{k-1, k, k+1\}$.

Part (b): Assume k is odd, and v is to the right of u in s'. If $s.count[v] = k-1$, then $s.count[u] = s.count[v] = k-1$, which is even, so by Invariant 3.3.1 (a), the edge $\{u, v\}$ is directed from u to v, a contradiction. So $s.count[v] = s'.count[v] = k$.

Part (c): The proof for part (c) is analogous to that of part (b). By Invariant 3.3.1 (b), $s.count[v] \neq k-1$. Therefore, $s'.count[v] = k$.

Part (d): Assume $s'.count[u] > s'.count[v]$, so $s'.count[v] \neq s'.count[u]$. If k is odd, by part (b) applied to s', v must be to the left of u. Also, since k is odd, $k-1$ is even, so when u takes a step, it reverses its left edges. Thus, the edge $\{u, v\}$ is reversed and is now directed from u to v. Similarly, if k is even, part (c) applied to s' implies that v must be to the right of u. Since $k-1$ is odd when u takes a step, it reverses all the edges to its right, and so the edge $\{u, v\}$ is now directed from u to v.

Case 2: v takes the $r+1$'st step. Therefore, v is a sink in s, so the edge $\{u, v\}$ is directed from u to v. Also, $s.count[u] = s'.count[u] = k$, and by the inductive hypothesis of part (a), $s.count[v] \in \{k-1, k, k+1\}$. If $s.count[v] = k+1$, then $s.count[v] > s.count[u]$, and by the inductive hypothesis part (d) the edge $\{u, v\}$ is directed from v to u, a contradiction. Therefore, $s.count[v] \in \{k-1, k\}$, and so $s'.count[v] \in \{k, k+1\}$.

Part (a): From the facts above it follows that $s'.count[v] \in \{k, k+1\}$.

Part (b): Assume k is odd, and v is to the right of u in s'. If $s.count[v] = k$, then $s.count[u] = s.count[v] = k$, which is odd, so by Invariant 3.3.1 (b), the edge $\{u, v\}$ is directed from v to u, a contradiction. So, $s.count[v] = k-1$, and therefore $s'.count[v] = k$.

Part (c): The proof for part (c) is analogous to part (b). By Invariant 3.3.1 (a), $s.count[v] \neq k$. Therefore, $s.count[v] = k - 1$, and $s'.count[v] = k$.

Part (d): Assume $s'.count[u] > s'.count[v]$. By part (a) applied to s', $s'.count[v] \in \{k-1, k, k+1\}$, so $s'.count[v] = k - 1$. Since v takes a step in r, then $s.count[v] = k - 2$. This is a contradiction to the inductive hypothesis of part (a), and therefore it is not possible for v to take the $r + 1$'st step in this case.

□

The next theorem uses Invariant 3.3.1 and part (d) of Invariant 3.3.2 to show that nodes in a circuit can never form a cycle because of the relation between the edge directions and the number of steps the nodes have taken.

Let $s.G' = (V, E')$ be the directed graph in state s, where V is the same set of nodes as in the undirected graph G, and E' is the set of directed edges determined using the *dir* variables as follows. The edge between any pair of nodes u and v is directed from u to v if and only if $dir[u, v] = out$.

Theorem 3.3.3. *In any reachable state s of the execution of NewPR the underlying directed graph $s.G'$ is acyclic.*

Proof. Suppose in contradiction that there exists a cycle in some reachable state s of the system. Let $s.G'$ be the directed graph in state s. Therefore, there is a sequence of nodes: $u, v_1, v_2, \ldots, v_n, u$ such that the edges between these nodes are directed from u to v_1, from v_n to u, and from v_i to v_{i+1}, for all $1 \leq i < n$. By Invariant 3.3.2 (d) the number of steps of the nodes in the sequence is non-increasing: $s.count[u] \geq s.count[v_1] \geq s.count[v_2] \geq \ldots \geq s.count[v_n] \geq s.count[u]$. Since node $s.count[u]$ is both in the beginning and the end of the sequence, it follows that $s.count[u] = s.count[v_1] = s.count[v_2] = \ldots = s.count[v_n] = s.count[u]$.

Let v_i be the rightmost node of the cycle. Then there must be some subsequence of nodes v_{i-1}, v_i, v_{i+1}, such that the edge $\{v_{i-1}, v_i\}$ is directed from left to right, and the edge $\{v_i, v_{i+1}\}$ is directed from right to left. We also know that $s.count[v_{i-1}] = s.count[v_i] = s.count[v_{i+1}]$. By the definition of $parity[u]$, $s.parity[v_{i-1}] = s.parity[v_i] = s.parity[v_{i+1}] = p$. By Invariant 3.3.1 (b) applied to v_{i-1} and v_i, it follows that $p = even$. By Invariant 3.3.1 (a) applied to v_i and v_{i+1}, it follows that $p = odd$, a contradiction. □

3.4 Simulation Relation

In this section we show that *PR* simulates *NewPR* which allows us to conclude that the acyclicity property of *NewPR* carries over to *PR*. First, we introduce a slight modification of the *PR* algorithm – instead of allowing a set of nodes to take a step at the same time, we now require only one node to take a step at a time. Let this modified version of *PR* be *OneStepPR*. We use *OneStepPR* as an intermediate step in showing that *PR* simulates *NewPR*. To do so, first, we provide a binary relation from *PR* to *OneStepPR*, and then another binary relation from *OneStepPR* to *NewPR*. For these two relations, we show, in

Theorem 3.4.2 and Theorem 3.4.4, respectively, that for each reachable state of one algorithm there exists a reachable state of the other algorithm such that these two states are related by the given relation. The main guarantee of both relations is to preserve the same directed version G' of the graph. Finally, in Theorem 3.4.5 we show the main result of the chapter which states that PR does not create any cycles in the graph.

3.4.1 Description of *OneStepPR*

OneStepPR is very similar to PR. It has the same state variables (*dir* and *list*), and a similar set of actions. Instead of allowing a set of nodes S to take a step together, in *OneStepPR*, only a single node u performs a $reverse(u)$ action. The precondition for this action is that u is a sink, and the effect of the action is that, similarly to PR, u reverses the edges to its neighbors which are not in $list[u]$. However, if $list[u] = nbrs_u$, then all edges incident to u are reversed.

3.4.2 Relation between PR and *OneStepPR*

We now define a binary relation R' from reachable states of PR to reachable states of *OneStepPR*, in order to show that both algorithms preserve the same directed version G' of the graph. Let s be a reachable state of PR and t be a reachable state of *OneStepPR*. We define $(s, t) \in R'$ if:

1. $s.G' = t.G'$
2. For each node u, $s.list[u] = t.list[u]$.

Lemma 3.4.1. *(a) For each initial state s of PR, there exists an initial state t of OneStepPR such that $(s, t) \in R'$.*

(b) For each pair of reachable states s of PR, and t of OneStepPR, with $(s, t) \in R'$, and for every step (s, s') of PR, there exists a finite sequence of steps of OneStepPR starting with t and ending with some t' such that $(s', t') \in R'$.

Proof. Initially, both directed graphs are the same and all nodes' lists are empty, so part (a) of the lemma is true.

To show that part (b) is true, assume $(s, t) \in R$ where s is a reachable state of PR, and t is a reachable state of *OneStepPR*. We need to show that for each step $(s, reverse(S), s') \in trans(PR)$, there exists a finite sequence of steps of *OneStepPR* starting with t and ending with some t' such that $(s', t') \in R$. Let the corresponding sequence of steps of *OneStepPR* consist of a $reverse(u)$ action for each $u \in S$. Let $S = \{u_1, u_2, \cdots, u_n\}$; then the sequence of steps in *OneStepPR* is $(reverse(u_1), reverse(u_2), \cdots, reverse(u_n))$, and $(t = t_0, t_1, t_2, \cdots, t_{n-1}, t_n = t')$ is the corresponding sequence of states.

Consider an arbitrary node $u_i \in S$. In PR, u_i is a sink and reverses a particular set of incident edges determined by the contents of $s.list[u_i]$. First, we show that the $reverse(u_i)$ action is enabled in state t_{i-1} by proving that u_i is a sink in t_{i-1}. We know u_i is a sink in

Algorithm 3 *OneStepPR* automaton

```
Signature:
  $reverse(u)$, $u \in V$, $u \neq D$

States:
  for each $u$, $v$ where $\{u, v\} \in E$:
    $dir[u, v] \in \{in, out\}$, initially $in$ if $v \in in\text{-}nbrs_u$ or
                                          $out$ if $v \in out\text{-}nbrs_u$
    $dir[v, u] \in \{in, out\}$, initially $in$ if $u \in in\text{-}nbrs_v$ or
                                          $out$ if $u \in out\text{-}nbrs_v$
  for each $u$, $list[u]$, a set of nodes $W \subseteq nbrs_u$, initially empty

Transitions:
$reverse(u)$
   Precondition:
     for each $v \in nbrs_u$, $dir[u, v] = in$
   Effect:
     if $list[u] \neq nbrs_u$ then
       for each $v \in nbrs_u \setminus list[u]$
         $dir[u, v] := out$
         $dir[v, u] := in$
         $list[v] := list[v] \cup \{u\}$
     else
       for each $v \in nbrs_u$
         $dir[u, v] := out$
         $dir[v, u] := in$
         $list[v] := list[v] \cup \{u\}$
     $list[u] := \emptyset$
Tasks:
  $\{reverse(u), u \in V, u \neq D\}$
```

t, and u_i does not take a step until t_{i-1}. No other node could have reversed u_i's edges from incoming to outgoing in the interval $[t, t_{i-1}]$, and so u_i is a sink in each state in $[t, t_{i-1}]$.

Part 1: Here we show that $s'.G' = t'.G'$. To show this, we argue that the same sets of edges are reversed in both algorithms. The sets of edges to be reversed depend only on the contents of the list, so we need to show that $s.list[u_i] = t_{i-1}.list[u_i]$. By part (2) of the relation we know that $s.list[u_i] = t.list[u_i]$, and we also showed that u_i is a sink in each state in $[t, t_{i-1}]$. Therefore, no neighbor of u_i is a sink in this interval, because no two neighboring nodes can be sinks at the same time. Since no neighbor of u_i is a sink, then no neighbor of u_i takes a step in $[t, t_{i-1}]$. Therefore, u_i's list remains the same, and so $s.list[u_i] = t.list[u_i] = t_{i-1}.list[u_i]$. Because the sets of edges reversed in any state depend only on the contents of the lists in that state, it follows that the same sets of edges are reversed in both algorithms. By part (1), $s.G' = t.G'$, so after the same sets of edges are reversed in both graphs, it follows that $s'.G' = t'.G'$. Therefore, part (1) is satisfied.

Part 2: Here we show that $s'.list[u] = t'.list[u]$ for all u. Fix an arbitrary node u. Depending on which nodes take steps in s, there are three possible cases:

Case 1: If $u \in S$, then we know that in both algorithms the lists are emptied after each reversal, so $s'.list[u] = t'.list[u] = \emptyset$.

Case 2: $u \notin S$ but some of u's neighbors are in S. Let $T = nbrs_u \cap S$, $T \neq \emptyset$, that is, T is the set of neighbors of u which take a step together. In *PR*, all nodes in T are added to $s'.list[u]$. Therefore, $s'.list[u] = s.list[u] \cup T$. In *NewPR*, all nodes in T take a step one at a time, and are added to $list[u]$ one at a time. Therefore, for some arbitrary $u_i \in T$, $t_i.list[u] = t_{i-1}.list[u] \cup \{u_i\}$. Consequently, $t'.list[u] = t.list[u] \cup T$. By part (2) we know that $s.list[u] = t.list[u]$, and therefore, $s'.list[u] = t'.list[u]$.

Case 3: $u \notin S$ and none of u's neighbors are in S. Since $list[u]$ can be modified only by u and its neighbors, and neither u nor any of its neighbors take a step, it follows that $s'.list[u] = s.list[u]$. Similarly, $t'.list[u] = t.list[u]$. Therefore, by part 2 of the relation, it follows that $s'.list[u] = s.list[u] = t.list[u] = t'.list[u]$.

Both parts of the relation are satisfied for s' and t', so $(s', t') \in R'$.

□

Now we show by induction, using the previous theorem as a building block, that for each reachable state of *PR* there exists a reachable state of *OneStepPR* such that the two states are related by the simulation relation R.

Theorem 3.4.2. *For any reachable state s of PR there exists a reachable state t of OneStepPR such that $(s, t) \in R'$.*

Proof. We prove the following statement, which immediately implies the theorem: For any non-negative integer k, and for any state s that is the final state of a k-step execution of *PR*, there exists a reachable state t of *OneStepPR* such that $(s, t) \in R'$. The proof is by induction on k.

Base Case: In the base case where $k = 0$, the final state of a k-step execution is the unique initial state of the *PR* algorithm. By Lemma 3.4.1 (a), for each initial state s of *PR*, there exists an initial state t of *OneStepPR* such that $(s, t) \in R'$. Since t is an initial state of *OneStepPR*, it is a reachable state.

Inductive Step: Assume that for any state s that is the final state of a k-step execution of PR, there exists a reachable state t of *OneStepPR* such that $(s, t) \in R'$. We need to show that for any state s' that is the final state of a $k + 1$-step execution of PR, there exists a reachable state t' of *OneStepPR* such that $(s', t') \in R'$.

Fix a state s' which is the final state of a k+1-step execution of PR. Let $(s'', reverse(u), s')$ be the final step of this execution. Then s'' is the final state of a k-step execution of PR. By the inductive hypothesis, it follows that there exists a reachable state t'' of *OneStepPR*, such that $(s'', t'') \in R'$. Now we apply Lemma 3.4.1 (b) to $(s'', t'') \in R'$ and (s'', s') being a step of PR. It follows that there exists a sequence of steps of *OneStepPR* starting with t'' and ending in some state t' such that $(s', t') \in R'$. We append this sequence of steps to some execution of *OneStepPR* which ends in t''. The resulting execution of *OneStepPR* ends in state t', and therefore, t' is a reachable state in *OneStepPR*. We have shown that for state s', which is the final state of a $k + 1$-step execution of PR, there exists a reachable state t' of *OneStepPR* such that $(s', t') \in R'$. □

3.4.3 Relation between *OneStepPR* and *NewPR*

We now define a binary relation from states of *OneStepPR* to states of *NewPR*, which satisfies specific properties outlined in Lemma 3.4.3. The main guarantee of the relation is to preserve the equivalence of the directed graphs in both algorithms. Let s be a reachable state of *OneStepPR* and t be a reachable state of *NewPR*. We define $(s, t) \in R$ if all of the following conditions hold:

1. $s.G' = t.G'$
2. For each node u, if $t.parity[u] = even$ then $s.list[u] \subseteq out\text{-}nbrs_u$.
3. For each node u, if $t.parity[u] = odd$ then $s.list[u] \subseteq in\text{-}nbrs_u$.

Lemma 3.4.3. *(a) For each initial state s of OneStepPR, there exists an initial state t of NewPR such that $(s, t) \in R$.*

(b) For each pair of reachable states s of OneStepPR, and t of NewPR, with $(s, t) \in R$, and for every step (s, s') of OneStepPR, there exists a finite sequence of steps of NewPR starting with t and ending with some t' such that $(s', t') \in R$.

Proof. Initially, both graphs are the same, so part 1 of the relation is satisfied. Also initially, $list[u] = \emptyset$, which implies that parts 2 and 3 are true. This proves part (a) of the lemma.

To show that part (b) of the lemma is true, assume $(s, t) \in R$ where s is a state of *OneStepPR*, and t is a state of *NewPR*. We need to show that for each step $(s, reverse(w), s')$ $\in trans(OneStepPR)$, there exists a finite sequence of steps of *NewPR* starting with t and ending with some t' such that $(s', t') \in R$. This sequence consists of either one or two consecutive $reverse(w)$ steps. If $s.list[w] \neq nbrs_w$, the corresponding sequence of steps of *NewPR* is a single $reverse(w)$ step. Otherwise, $NewPR$ executes two consecutive $reverse(w)$ steps. The first $reverse(w)$ action is enabled because by part 1 $s.G' = t.G'$, and since w is a sink in s, it is also a sink in t. The second $reverse(w)$ action is enabled because w did not reverse any edges in the previous step, so it is still a sink.

We now show that $(s', t') \in R$, which involves proving that the three parts of R hold for s' and t'.

Part 1: We prove that $t.G' = s.G'$.

Case 1: $t.parity[w] = even$. Since part 2 is true with respect to s and t, $s.list[w] \subseteq out\text{-}nbrs_w$. By Corollary 3.2.4, because w is a sink, $s.list[w] = out\text{-}nbrs_w$.

Case 1.1: $s.list[w] \neq nbrs_w$. The corresponding step in $NewPR$ is a $reverse(w)$ action.

When w takes a step in *OneStepPR* it reverses the edges to all nodes in $nbrs_w \backslash s.list[w] = in\text{-}nbrs_w$. Node w reverses the same set of edges in $NewPR$ because $t.parity[w] = even$. Therefore, since $s.G' = t.G'$, and the set of edges reversed in going from s to s' is the same as the set of edges reversed in going from t to t', it follows that $s'.G' = t'.G'$.

Case 1.2: $s.list[w] = nbrs_w$. The corresponding steps in $NewPR$ are two consecutive $reverse(w)$ actions.

In *OneStepPR*, w reverses all edges corresponding to nodes in $out\text{-}nbrs_w$. In $NewPR$, when w executes the first $reverse(w)$ action, since $t.parity[w] = even$ and $in\text{-}nbrs_w = \emptyset$, w does not reverse any edges to neighbors, but only increments its step counter. The result of that action is that $t.parity[w]$ is flipped from *even* to *odd*. Next, w performs the second $reverse(w)$ action. Since $parity[w]$ is *odd*, w reverses all edges corresponding to nodes in $out\text{-}nbrs_w$. Therefore, since $s.G' = t.G'$, and the set of edges reversed in going from s to s' is the same as the set of edges reversed in going from t to t', it follows that $s'.G' = t'.G'$.

Case 2: $t.parity[w] = odd$. Since part 3 is true with respect to s and t, $s.list[w] \subseteq in\text{-}nbrs_w$. By Corollary 3.2.4, because w is a sink, $s.list[w] = in\text{-}nbrs_w$.

Case 2.1: $s.list[w] \neq nbrs_w$. The corresponding step in $NewPR$ is a $reverse(w)$ action.

When w takes a step in *OneStepPR* it reverses the edges to all nodes in $nbrs_w \backslash s.list[w] = out\text{-}nbrs_w$. Node w reverses the same set of edges in $NewPR$ because $t.parity[w] = odd$. Therefore, since $s.G' = t.G'$, and the set of edges reversed in going from s to s' is the same as the set of edges reversed in going from t to t', it follows that $s'.G' = t'.G'$.

Case 2.2: $s.list[w] = nbrs_w$. The corresponding steps in $NewPR$ are two consecutive $reverse(w)$ actions.

In *OneStepPR*, w reverses all edges corresponding to nodes in $in\text{-}nbrs_w$. In $NewPR$, when w executes the first $reverse(w)$ action, since $t.parity[w] = odd$ and $out\text{-}nbrs_w = \emptyset$, w does not reverse any edges to neighbors, but only increments its step counter. The result of that action is that $t.parity[w]$ is flipped from *odd* to *even*. Next, w performs the second $reverse(w)$ action. Since $parity[w]$ is *even*, w reverses all edges corresponding to nodes in $in\text{-}nbrs_w$. Therefore, since $s.G' = t.G'$, and the set of edges reversed in going from s to s' is the same as the set of edges reversed in going from t to t', it follows that $s'.G' = t'.G'$.

Part 2: Here we show that for each node u, if $t.parity[u] = even$ then $s.list[u] \subseteq out\text{-}nbrs_u$.

Fix an arbitrary node u. Assume $t'.parity[u] = even$ because otherwise part 2 is vacuously true.

Case 1: $u = w$. Then u is the node that takes the step, so $s'.list[u] = \emptyset$, which implies part 2 for s' and t'.

Case 2: $u \neq w$ and $w \in nbrs_u$. Since u does not take a step, it follows that $t.parity[u] = t'.parity[u] = even$.

Claim: $w \in out\text{-}nbrs_u$

Case 2.1: $t.parity[w] = even$

By Invariant 3.3.1 (a), the edge between u and w is directed from left to right. Also, because w is a sink in s and t, the edge is directed from u to w. Therefore, w is to the right of u, and so $w \in out\text{-}nbrs_u$.

Case 2.2: $t.parity[w] = odd$

Since $t.parity[u] \neq t.parity[w]$, then $t.count[u] \neq t.count[w]$. By Invariant 3.3.2 (c), w is to the right of u, and so $w \in out\text{-}nbrs_u$.

So far, in the claim above, we established that $w \in out\text{-}nbrs_u$. Since w is added to $s'.list[u]$ in the step of *OneStepPR*, $s'.list[u] = s.list[u] \cup \{w\}$. By Corollary 3.2.3, $list[u]$ is always a subset of either $in\text{-}nbrs_u$ or $out\text{-}nbrs_u$. Since $w \in out\text{-}nbrs_u$ and $w \in s'.list[u]$, it has to be the case that $s'.list[u] \subseteq out\text{-}nbrs_u$. Part 2 remains true with respect to s' and t' because we have assumed that $t'.parity[u] = even$ and we just showed that $s'.list[u] \subseteq out\text{-}nbrs_u$.

Case 3: $u \neq w$ and $w \notin nbrs_u$

Since only u and its neighbors can change the contents of the list, and neither u nor any of its neighbors take a step, $s.list[u] = s'.list[u]$. Also because u does not take a step, $t.parity[u] = t'.parity[u]$, and so part 2 remains true for s' and t'.

Part 3: We show that for each node u, if $t.parity[u] = odd$ then $s.list[u] \subseteq in\text{-}nbrs_u$. The proof is symmetric to the proof of part 2. □

Now we show by induction, using the previous theorem as a building block, that for each reachable state of *OneStepPR* there exists a reachable state of *NewPR* such that the two states are related by the simulation relation R.

Theorem 3.4.4. *For any reachable state s of OneStepPR there exists a reachable state t of NewPR such that $(s,t) \in R$.*

Proof. We prove the following statement, which immediately implies the theorem: For any non-negative integer k, and for any state s that is the final state of a k-step execution of *OneStepPR*, there exists a reachable state t of *NewPR* such that $(s,t) \in R$. The proof is by induction on k.

Base Case: In the base case where $k = 0$, the final state of a k-step execution is the unique initial state of the *OneStepPR* algorithm. By Lemma 3.4.3 (a), for each initial state s of *OneStepPR*, there exists an initial state t of *NewPR* such that $(s,t) \in R$. Since t is an initial state of *NewPR*, it is a reachable state.

Inductive Step: Assume that for any state s that is the final state of a k-step execution of *OneStepPR*, there exists a reachable state t of *NewPR* such that $(s,t) \in R$. We need to show that for any state s' that is the final state of a $k+1$-step execution of *OneStepPR*, there exists a reachable state t' of *NewPR* such that $(s', t') \in R$.

Fix a state s' which is the final state of a $k+1$-step execution of *OneStepPR*. Let $(s'', reverse(u), s')$ be the final step of this execution. Then s'' is the final state of a k-step execution of *OneStepPR*. By the inductive hypothesis, it follows that there exists a reachable state t'' of *NewPR*, such that $(s'', t'') \in R$. Now we apply Lemma 3.4.3 (b) to $(s'', t'') \in R$ and (s'', s') being a step of *OneStepPR*. It follows that there exists a sequence of steps of *NewPR* starting with t'' and ending in some state t' such that $(s', t') \in R$. We append this sequence of steps to some execution of *NewPR* which ends in t''. The resulting execution of *NewPR* ends in state t', and therefore, t' is a reachable state in *NewPR*. We have shown that for state s', which is the final state of a $k+1$-step execution of *OneStepPR*, there exists a reachable state t' of *NewPR* such that $(s', t') \in R$. □

Finally, we present the main result of the chapter, which shows that the acyclicity property of *NewPR* carries over to *PR*.

Theorem 3.4.5. *In any reachable state s of the execution of PR the underlying directed graph $s.G'$ is acyclic.*

Proof. Let s be any reachable state of *PR*. By Theorem 3.4.2, there exists a reachable state r of *OneStepPR* such that $(s, r) \in R'$. By Theorem 3.4.4, there exists a reachable state t of *NewPR* such that $(r, t) \in R$. By the definition of R', $s.G' = r.G'$, and by the definition of R, $r.G' = t.G'$. It follows that $s.G' = t.G'$. By Theorem 3.3.3, $t.G'$ is acyclic, and therefore, $s.G'$ is acyclic too. □

3.5 Conclusion

We have presented a modification of the Partial Reversal algorithm, *NewPR*, which preserves the behavior of the original *PR* algorithm. By rewriting the algorithm in this way, we are able to prove useful properties of *NewPR*, including the fact that no cycles are created in any execution of the *NewPR* algorithm. Unlike the existing acyclicity proofs, ours does not assume any labels on either the nodes or the edges of the graph, and uses only properties of the PR algorithm to establish the acyclicity property. We have also defined a binary relation from the original *PR* algorithm to the modified version *NewPR*, which implies that the acyclicity property applies to the original *PR* algorithm as well.

A straightforward extension of the results above would be to show a binary relation in the reverse direction too (from *NewPR* to *PR*). Such a relation would imply that *PR* and *NewPR* are equivalent to each other with respect to the direction of the edges in the graph, which is a much stronger result than just showing that neither algorithm creates cycles in the graph.

Chapter 4

Leader Election Algorithm

In this chapter, we present the leader election (LE) algorithm of [17] and [16] and reason about some of its properties. We begin by briefly describing the leader election problem and the context in which the LE algorithm mentioned above solves the problem.

First, we describe the type of systems which we consider for solving the LE problem. We consider mobile ad-hoc networks (MANETs) consisting of nodes which communicate with each other though point-to-point communication channels. While nodes are guaranteed not to fail, channels are allowed to go up and down in order to model the mobility of nodes in a MANET. We call such changes in the communication graph topology changes. We also assume that communication between the nodes is FIFO and reliable.

Informally, the goal of a leader election (LE) algorithm is to specify a unique node in the system as the leader, and also for all nodes in the system to know the identity of the leader. Since frequent topology changes may be disruptive to the execution of any LE algorithm, we require the above properties to hold only after topology changes have ceased. Since all nodes have a consistent view of the leader, such a LE algorithm can be used to collect information from all the nodes at the leader. Therefore, it is helpful to assign logical directions to the edges of the communication graph so that each node has a directed path to the leader and can easily forward information to the leader on any outgoing edge.

More precisely, an algorithm which solves the LE problem guarantees that after the last topology change in the system, the following properties are satisfied in every connected component:

- Each node has the same leader id, say ℓ, where ℓ is also in the same connected component.
- The connected component is configured as a directed acyclic graph (DAG) with ℓ as the unique sink.
- No messages from the LE algorithm are in transit.

As we mentioned in Chapter 1, we consider a particular LE algorithm, presented in [17] and [16], and prove useful properties about its efficiency and its behavior in different timing models. In Section 4.1 we present the system model; in Section 4.2 we formally define the leader election problem; in Section 4.1.2 we define the timing model we consider for the algorithm; in Section 4.3 we present the algorithm overview and details, similarly to [16]. In Section 4.4 we present a summarized version of the correctness proof of the LE algorithm, the full version of which is available in [16].

The next sections include our main contributions. In Section 4.5 we present a few results on the complexity of the LE algorithm. In particular, we show that the number of elections that occur before the algorithm terminates is $O(n)$. In Section 4.6 we present a particular property of the LE algorithm which characterizes the maximum number of elections that may occur under particular topology-change patterns. In particular, we show that even under very carefully designed patterns of topology changes, it is still possible to have "unnecessary" elections (electing a new leader when there is an existing one already in the same connected component). Finally, in Section 4.7 we discuss how to combine the LE algorithm with a shortest path algorithm in order to obtain shortest paths to the leader.

4.1 System Model

4.1.1 System Components

We assume a system consisting of a set $\mathcal{P}$ of computing nodes and a set $\mathcal{X}$ of undirected communication channels. We assume each node has a positive integer as its unique id drawn from a set of ids I. $\mathcal{X}$ consists of one channel for each ordered pair of nodes, i.e., every possible channel is represented. This assumption is useful to model the fact that, as nodes move, any pair of nodes can get connected if they get sufficiently close to each other. The nodes are assumed to be completely reliable. The channels between nodes go up and down, due to the movement of the nodes or any other changes in the communication topology.

The system is modeled as a set of I/O automata [20]. For an overview of I/O automata, refer to Chapter 2. Each node and each channel is modeled as a separate I/O automaton. First, we specify how communication is assumed to occur over the dynamic channels. The state of $Channel(\{u, v\})$, which models the communication channel between nodes u and v, includes a $status_{\{u,v\}}$ variable with possible values *up* and *down*. The initial value of $status_{\{u,v\}}$ is determined by the initial communication topology. The channel transitions between the two values of its $status_{\{u,v\}}$ variable through inputs from the environment $channelUp_{\{u,v\}}$ and $channelDown_{\{u,v\}}$, called "topology changes". The environment is also modeled as an automaton with no input actions and two sets of output actions: $channelUp_{\{u,v\}}$ and $channelDown_{\{u,v\}}$. We assume that for all traces of the environment automaton it is true that the *channelUp* and *channelDown* events for each channel alternate and the first topology change event for each channel, if any, is *channelUp*. We also assume that the $channelUp_{\{u,v\}}$ and $channelDown_{\{u,v\}}$ actions are input actions to the nodes at the endpoints of the channel (nodes u and v).

The state of $Channel(\{u, v\})$ also includes variables $mqueue_{u,v}$ and $mqueue_{v,u}$ which hold messages in transit from u to v and from v to u, respectively. Both queues are initially

Signature:
Input:
$send(m)_{u,v}$, $m \in M$
$send(m)_{v,u}$, $m \in M$
$channelUp_{\{u,v\}}$
$channelDown_{\{u,v\}}$
Output:
$receive(m)_{u,v}$, $m \in M$
$receive(m)_{v,u}$, $m \in M$

States:
$status_{\{u,v\}}$, a boolean with values from the set $\{up, down\}$, initially *down*
$mqueue_{u,v}$, a FIFO queue of elements of M, initially empty
$mqueue_{v,u}$, a FIFO queue of elements of M, initially empty

Transitions:

```
$send(m)_{u,v}$
    Effect:
        add $m$ to $mqueue_{u,v}$

$send(m)_{v,u}$
    Effect:
        add $m$ to $mqueue_{v,u}$

$channelUp_{\{u,v\}}$
    Effect:
        $status_{\{u,v\}} = up$

$channelDown_{\{u,v\}}$
    Effect:
        $status_{\{u,v\}} = down$
        $mqueue_{u,v} = \emptyset$
        $mqueue_{v,u} = \emptyset$

$receive(m)_{u,v}$
    Precondition:
        $status_{\{u,v\}} = up$
        $m$ is first on $mqueue_{u,v}$
    Effect:
        remove first element of $mqueue_{u,v}$

$receive(m)_{v,u}$
    Precondition:
        $status_{\{u,v\}} = up$
        $m$ is first on $mqueue_{v,u}$
    Effect:
        remove first element of $mqueue_{v,u}$
```

Tasks:
$\{receive(m)_{u,v} : m \in M\}$
$\{receive(m)_{v,u} : m \in M\}$

Figure 4.1: $Channel(\{u, v\})$ automaton

empty. An attempt by node u to send a message to node v results in the message being appended to $mqueue_{u,v}$ if the status of the channel is *up*; otherwise, there is no effect. When the channel is *up*, the message at the head of $mqueue_{u,v}$ can be delivered to node v; when a message is delivered, it is removed from $mqueue_{u,v}$. Thus, messages are delivered in FIFO order. When a $channelDown_{\{u,v\}}$ event occurs, both queues are emptied and neither u nor v is alerted to which messages in transit have been lost.

We also consider a liveness property of the channel, which guarantees that if a channel remains *up* for infinitely long, then every message sent over the channel during this *up* interval is eventually delivered. The complete I/O automaton code for the channel between nodes u and v is available in Figure 4.1. We assume all messages come from a message alphabet M.

Now, we describe in detail the structure of a node automaton. Each node u has input actions of the form $receive(m)_{v,u}$, for every node v, through which u receives messages from node v. Also, each node u has output actions of the form $send(m)_{u,v}$, for every node v, through which u sends messages to v. Each node u also keeps track of the nodes with which it can communicate at any given time by maintaining an array of *neighbors* N. When a $channelUp_{\{u,v\}}$ event occurs at node u, it adds node v to N, and when a $channelDown_{\{u,v\}}$ event occurs at node u, it removes v from N. In addition to these actions, u also has internal and/or external actions implementing the particular algorithm at hand.

Next, we specify the composition of the entire system. We assume that the system is modeled as a *send/receive system*, as introduced in Chapter 14 of [20], where the system is composed of the node automata and channel automata defined above. In such a system, the different types of events are *send* and *receive* events through which node automata interact with channel automata, internal events of node automata, and possibly other input and output events depending on the particular problem at hand. The composition properties of such a system guarantee that nodes and channels interact correctly with each other. For example, when node u performs a $send(m)_{u,v}$ output action, a simultaneous input action $send(m)_{u,v}$ is performed by $Channel(\{u,v\})$. Let the resulting send/receive system be S. Next, we compose S with an environment automaton E defined above to obtain the *LE* system that we consider throughout this chapter.

4.1.2 Logical Time

We would also like to provide nodes in the system with some notion of time, which is necessary for the correct execution of the LE algorithm. Since it is difficult to guarantee that nodes have access to real time, we assume that the system is augmented with logical time which provides nodes with the ability to infer useful information from the order of events in the execution, as opposed to a real-time clock. For example, if some event can potentially cause or affect some other event, logical time ensures that the first event is assigned a smaller timestamp than the second event.

First, we introduce some notation we use in the definition of logical time and also in other definitions in the thesis. Given a partially-ordered set A, we define the the set $A_\perp = A \cup \{\perp\}$ to denote the union of all elements of A and the element $\perp$. Furthermore, we assume that

$\perp$ is strictly smaller than all elements of A. Intuitively, we think of $\perp$ as an undefined value.

Similarly to the definition of logical time in [20] (Chapter 18), we assume our system LE is augmented with a function L that assigns logical times to all events in all executions of the send/receive system S that is a component of LE. Note that in [20], logical time is defined with respect to a send/receive system not composed with an environment. We assume that the function L, described next, assigns logical times only to events of the send/receive system component of LE. Let α be an arbitrary execution of LE, and let $T_\perp$ be a totally-ordered logical-time domain with $\succeq$ being its comparison operator. The function L maps every event e of α to a logical time value $t \in T$, subject to the following constraints:

1. For each pair of events e_1 and e_2 in α, $L(e_1) \neq L(e_2)$.

2. For each pair of events e_1 and e_2 in α occurring at the same node u, $L(e_1) \prec L(e_2)$. This means that the logical times assigned to events at each node are increasing.

3. For all nodes u and v, such that $u \neq v$, and all $m \in M$, if $send(m)_{u,v}$ is a send event and $receive(m)_{u,v}$ is the corresponding receive event, then $L(send(m)_{u,v}) \prec L(receive(m)_{u,v})$.

4. For any value $t \in T$, there are only finitely many events e such that $L(e) \preceq t$.

Moreover, we assume that each node u has a local variable *clock* in its state, which provides u with access to logical time. The value of *clock* is initialized to $\perp$ and maintained in such a way that for each step (s, e, s') of u, $s.clock \prec L(e) = s'.clock$. In other words, the clock value at a node u is always equal to the logical time of the most recent event at u; if there is no such event, then the value of the logical clock is $\perp$.

Throughout this thesis, we assume that the system used to implement the LE algorithm is the LE system augmented with logical time.

Next, we give an example of a specific function to produce logical times, called *Lamport time* [19].

Example 4.1.1. ***Lamport Time:*** *The domain T for Lamport times is the set of ordered pairs (t, u) where t is a non-negative integer and u is a node id. Pairs are compared lexicographically.*

In order to determine the logical time assigned to some event e at some node u, we consider two possible cases.

Case 1: *Event e is not a receive event. If e is the first event at node u, then the logical time assigned to e is $(0, u)$. Otherwise, the logical time assigned to e is $(t+1, u)$, where (t, u) is the logical time of the latest preceding event at node u.*

Case 2: *Event e is a receive event. If e is the first event at node u, then the logical time assigned to e is $(t + 1, u)$ where t is the first component of the logical time of the corresponding $send(m)_{u,v}$ event. Otherwise, the logical time assigned to e is $(t+1, u)$, where t is the maximum value of the first component of the logical times of (1) the corresponding $send(m)_{u,v}$ event, and (2) the latest event preceding e at node u.*

Next, we justify why Lamport time is an example of logical time. Parts (1) and (2) of the definition of logical clocks are satisfied because the logical times of subsequent events at node u are increasing, and because the second component is a unique id. Part (3) is satisfied because of the way logical clock values are assigned to receive events. Finally, Part (4) is satisfied because the first component of the logical times assigned to subsequent events at some node u are increasing by at least 1; *therefore, since there are a finite number of nodes in the system, there cannot be an infinite number of logical times less than some $t \in T$ assigned to events.*

Note that given the definition of Lamport Time, we can assign a local clock value clock at each node u as follows. The initial value of clock is $(0, u)$. *The value of clock at any point in the execution is equal to the logical time of the last event at u.*

4.2 Problem Statement

In this section we formally define the leader election problem. First, we provide an intuitive description of the problem statement. In a dynamic setting where channels go up and down, an algorithm is said to solve the leader election problem if eventually each connected component of the communication topology has a unique node elected as the leader and all other nodes know the id of the leader. Since we also plan to use the resulting topology for routing information from the nodes to the leader, an extra requirement for the LE algorithm is to provide a direction for each edge in the communication graph such that each node has a directed path to the leader and there are no cycles in the entire graph. Next, we provide a formal definition.

Let the *LE* system be composed of a send/receive system S and an environment automaton E, as defined in Section 4.1. We assume there exists a function f which maps each state s_u of a node u to a node id, the "candidate leader node". We also assume there exists a function g which represents node u's view of the direction of the link between u and v. For each state s_u of some node u and each node v, let the possible values of $g(s_u, v)$ be $\{(u, v), (v, u)\}$ if $v \in N_u$ in state s_u. Otherwise, if $v \notin N_u$ in state s_u, then $g(s_u, v) = \perp$.

Let α be an arbitrary fair execution of *LE* that contains only a finite number of topology change events. Also, let β be a suffix of α that does not contain any topology change events. Since there are no topology changes in β, the graph induced by the communication topology is fixed; we define that graph $G_\beta = (V_\beta, E_\beta)$ as follows. For each node v in *LE*, the set of vertices V_β contains a vertex v. There exists an edge $\{u, v\} \in E_\beta$ between two vertices u and v of G_β if and only if the status of *Channel*$(\{u, v\})$ between nodes u and v in *LE* is *up*.

We say that system *LE solves the LE problem* if for every execution α of *LE* that contain only a finite number of topology change events, there exists a suffix β with no topology changes, and some unique node ℓ in each connected component of G_β, such that the following conditions are satisfied:

1. If some nodes u and v are in the same connected component of G_β, then for all states s_u and s_v of u and v in β it is true that $f(s_u) = f(s_v) = \ell$. This part ensures that eventually the connected component has a unique stable leader.

2. For each edge $\{u, v\} \in E_\beta$ and for all states s_u and s'_u of node u in β, and all states s_v and s'_v of node v in β, it is true that $g(s_u, v) = g(s'_u, v) = g(s_v, u) = g(s'_v, u)$. In other words, eventually both endpoints of an edge agree on the direction of the edge, and that direction is stable.

3. If $u \neq \ell$, then there exists a sequence of nodes $(u = v_1, v_2, \cdots, v_k = \ell)$, such that for all $1 \leq i < k$, it is true that $g(s_{v_i}, v_{i+1}) = (v_i, v_{i+1})$, where s_{v_i} is a state of node v_i in β. In other words, there exists a directed path from each node in the connected component to the unique leader.

4. There does not exist a sequence of nodes $(u = v_1, v_2, \cdots, v_k = u)$, such that for all $1 \leq i < k$, it is true that $g(s_{v_i}, v_{i+1}) = (v_i, v_{i+1})$, where s_{v_i} is a state of node v_i in β. In other words, there are no cycles formed by the directions imposed on the channels.

4.3 Leader Election Algorithm

In this section we describe our LE algorithm in more detail. Note that, as mentioned in Section 1.1.2, the LE algorithm is derived from the Temporally Ordered Routing Algorithm (TORA) [23] (as described in Section 1) by adding two more components to the main structure used in the algorithm, called the height. We mentioned that the extra two components are used to record the time when a particular search for the leader is started and the time when a new leader is elected, respectively. In an earlier version of the LE algorithm [17], these timestamps are derived from a real-time clock. In this thesis, however, similarly to [16], we use the notion of logical time, defined in Section 4.1.2, to assign values to these timestamps. In Section 4.3.4, we explain in detail how the algorithm uses these timestamps together with the main idea from TORA to guarantee that eventually a unique leader is elected. Moreover, our LE algorithm, similarly to TORA, uses only edge reversals throughout its execution and does not rely on complicated mechanisms to direct the edges in the direction of the leader.

The rest of this section is organized as follows. In Section 4.3.1, we give a very high-level intuition for how the LE algorithm works; in Section 4.3.2, we define a structure called a *height* which helps encode algorithm information; in Section 4.3.3, we present the state variables of each node automaton in the LE algorithm; in Section 4.3.4 we present the LE algorithm and describe all of the actions of the algorithm in detail.

4.3.1 Informal Description of the LE Algorithm

Each channel in the system has a logical direction imposed on it, forming a *link* (we describe how these directions are determined later). Due to topology changes, nodes may lose some of their incident links, or get new ones throughout the execution. Whenever a node u loses its last outgoing link because of a topology change, it has no path to any other node, including the node that u thought to be the current leader; therefore, u reverses the directions of all of its incident edges. Reversing all incident edges acts as the start of a search mechanism for the current leader. Each node that receives information about the newly started search reverses the edges to some of its neighbors and in effect propagates the search throughout the connected component. Once a node becomes a sink and all of its neighbors are already participating in the same search, it means that the search has hit a dead end and the node that u thought to be the leader is not present in this part of the connected component. Such dead end information is then propagated back towards the originator of the search. When a node u that started a search receives such dead end messages from all of its neighbors, it concludes that the node that u thought to be the leader is not present in the connected component, and so node u elects itself as the new leader. Finally, this new leader information propagates throughout the network via an extra "wave" of messages.

One difficulty that arises in solving LE in dynamic networks is dealing with the partitioning and merging of connected components. For example, when a connected component is partitioned from the current leader due to links going down, the above algorithm ensures that a new leader is elected using the mechanism of waves searching for the leader and convergecasting back to the originator. On the other hand, it is also possible that two connected components merge together resulting in two leaders in the new connected component. When the information about two different leaders is being propagated in the new connected component, eventually, some node needs to compare both and decide which one to continue propagating. In the LE algorithm that we consider, such a choice is made based on the logical times at which the two leaders are elected such that a "newer" leader has priority over an "older" one. Therefore, even though conflicting information about two different leaders may be propagating in the same connected component, the algorithm ensures that, if topology changes stop, eventually each connected component has a unique leader.

In the next sections we describe in more detail how all of this information about leaders, elections, and reference levels is encoded in a structure called a *height*, and how the algorithm uses this information to solve the LE problem.

4.3.2 The Height Structure

Here, we introduce the *height* structure which represents some necessary information in a node's current state in the algorithm execution, including the node's view of the id of the current leader, the ongoing search for a leader, etc.

A height is a 7-tuple $(\tau, oid, r, \delta, nlts, lid, id)$, where the type of each component is described below:

- τ is a member of $T_\perp$.
- oid is a member of $I_\perp$.
- r is a boolean with values 0 and 1.
- δ is an integer.
- $nlts$ is a member of $T_\perp$.

- *lid* is a member of I.
- *id* is a member of I.

Next, we define the rules we use to compare two heights to each other. We denote the components of the height of a node v as (τ^v, oid^v, r^v, δ^v, $nlts^v$, lid^v, v). Let $height_u$ and $height_v$ be the heights of node u and v, respectively.

- τ and *oid* are compared as elements of $T_\perp$ and $I_\perp$, respectively.
- $r^u \geq r^v$ iff $r^u = 1$ or $r^v = 0$.
- $nlts^u \geq nlts^v$ iff one of the following is true:
  1. $nlts^v \succeq nlts^u$, $nlts^u \neq \perp$ and $nlts^v \neq \perp$
  2. $nlts^v = \perp$ and $nlts^u \neq \perp$
  3. $nlts^u = nlts^v = \perp$
- δ, *lid* and *id* are integers, so we can compare them as such.

Given these rules, we can compare two heights $height_u$ and $height_v$ using lexicographic ordering. We assume that each message in the system consists of just one such height object.

Note that the first part of the rule for comparing the *nlts* component states that the *nlts* component of a node u is greater than the *nlts* component of a node v iff the logical time value stored in $nlts^u$ is *smaller* than the logical time value stored in $nlts^v$. In Section 4.3.3 we define the direction of an edge between two nodes to be from the node with a larger height to the node with a smaller height. Since we are comparing heights lexicographically, the rule for comparing *nlts* components is useful in directing an edge from a node with height containing an "older" (smaller logical timestamp) leader to a node with a height containing a "newer" (larger logical timestamp) leader. Moreover, in the second part of the same rule we want to ensure that $nlts^u \geq nlts^v$ when $nlts^u$ is not $\perp$ and $nlts^v$ is $\perp$ because we use $\perp$ specifically for undefined values. Finally, in part 3 of the rule, we allow for two $\perp$ values to be compared.

4.3.3 State Variables of the LE Algorithm

Each node u has the following state variables, which we describe next in detail: *clock*, a logical clock value, initialized and maintained as described in Section 4.1.2; N, a set of node ids; *height*, an array of heights indexed by node ids from the set I; $sendBuffer(v)$ for all $v \in N$. Refer to the pseudocode in Figure 4.2 for a list of these variables.

Node u's local variable N is a set of node ids representing the current set of neighbors of u. When a *channelUp* event occurs at node u for the channel from u to v, node u puts the id of node v in N. When a *channelDown* event occurs at node u for the channel from u to v, node u removes the id of v from the neighbor set N. For the purposes of the algorithm,

u considers as its neighbors only those nodes in N. Initially, the set N is empty. In other words, each node is in a connected component of its own, and no edges are present in the communication graph.

The height array of each node u contains u's own height and the heights of its neighbors, where *height*$[v]$ denotes u's view of v's height, and *height*$[u]$ denotes u's own height. Initially, *height*$[u]$ is initialized to $(\perp, \perp, 0, 0, \perp, u, u)$. Therefore, initially, each node is its own leader.

Finally, node u stores the messages it needs to send to its neighbors in a collection of $sendBuffer(v)$ variables, one for each node v. Initially, $sendBuffer(v)$ is empty.

Each node u assign virtual directions to its incident links using the array *height*. For each link $\{u, v\}$, u considers the link as incoming (directed from v to u) if *height*$[v] >$ *height*$[u]$; otherwise u considers the link as outgoing (directed from u to v).

Finally, we provide some intuition on the purpose of each component in the height. Note that a height token can be present either at a node's state or in a message from one node to another. Throughout the following descriptions in this section and the sections to follow, we refer to the first three components if a node's height as the *reference level* (RL), and the fifth and sixth components as the *leader pair* (LP).

- τ, a value from $T_\perp$; τ is $\perp$ if no search for an alternate path to the leader is in progress; otherwise, τ is the value of the clock of the originator of some search for the leader at the time when the search was initiated. Initially, τ is set to $\perp$.

- *oid*, a value from $I_\perp$; *oid* is $\perp$ if no search for an alternate path to the leader is in progress; otherwise, *oid* is the id of the node that started the current search. Initially, it is set to $\perp$.

- r, a bit that is set to 0 (implying an unreflected RL) when the search is initiated and set to 1 (implying a reflected RL) when the search hits a dead end.

- δ, an integer that is set to ensure that links are directed appropriately between neighbors with the same first three components. During the execution of the algorithm δ serves two different purposes. When the algorithm is in the stage of searching for the leader, the δ value ensures that as a node u receives information about a new search (a new RL) from a node v, the direction of the edge between them is from v to u; in other words it is the same as the direction of the search propagation. Therefore, u sets its RL to be the same as the RL of v and sets its δ to one less than v's. When a leader is already elected, the δ value helps orient the edges of each node towards the leader. Therefore, when node u receives information about a new leader (a new LP) from node v, it sets its height to the height of v and sets the δ value to one more than v's.

- *nlts*, a value from $T_\perp$ which represents the logical time when the current leader was elected.

- *lid*, a value from $I_\perp$ which represents the id of the current leader.

- *id*, a value from $I_\perp$ which represents the node's unique ID.

4.3.4 Description of the LE Algorithm

Each process automaton in the algorithm consists of four different kinds of transitions, one for each of the possible input and output actions in the system: a channel going up, a channel going down, the receipt of a message, and the sending of a message to another node. Each of these transitions is assumed to be atomic. The automaton has no internal actions. Next, we describe each of the input and output transitions in more detail. The pseudocode for the automaton is presented in Figure 4.2. In order to ensure consistency with the code presented in [17] and [16], parts of the pseudocode (the effect clauses of $receive(m)_{v,u}$ and $channelDown_{\{u,v\}}$) are shown in Figures 4.3 and 4.4 in the style of [17] and [16].

In the descriptions below, we consider a notion similar to that of a sink, from the traditional definition in graph theory. We consider a node u to be a *local sink* if all of u's neighbors have the same LP as u, node u has no outgoing links and it is not its own leader. Formally, a node u is a local sink if the following predicate on u's state is true: $(\forall v \in N_u, (nlts^u, lid^u) = (nlts^v, lid^v)$ and $height[u] < height[v])$ and $lid^u \neq u$. Note that in [17] and [16], instead of the new term, "local sink", that we use here, the term "sink" is redefined to correspond to the definition we just gave for a local sink.

Before we describe each of the algorithm transitions, we present five subroutines, shown in Figure 4.5, which are used as building blocks in the algorithm transitions and are invoked by the code in Figures 4.3 and 4.4.

- ADOPTLPIFPRIORITY: To perform this subroutine, node u sets $height[u]$ to $(\tau^v, oid^v, r^v, \delta^v + 1, nlts^v, lid^v, u)$.

  In other words, node u copies the height of node v and then sets the δ value to one more than v's so that the edge between u and v is directed from u to v. The actual name of the subroutine, involving "priority", will become clear later in this section, when we describe how the subroutine is invoked.

- PROPAGATELARGESTREFLEVEL: By performing this subroutine node u does the following: (1) it sets (τ^u, oid^u, r^u) to the maximum value (τ^w, oid^w, r^w) among the heights of all of u's neighbors, and (2) it sets δ^u to $\min\{\delta^w | w \in N$ and $(\tau^u, oid^u, r^u) = (\tau^w, oid^w, r^w)\} - 1$.

  In other words, node u adopts the largest RL among its neighbors and sets its δ value to one less than the smallest δ value among the neighbors who have the same new RL as u. Such manipulation of the δ value ensures that u's new height is larger than all of its neighbors' height except the one from which it got the new RL. Therefore, the direction of the edge between u and the node from which u received the new RL is towards u, and all other incident edges to u are outgoing in u's view.

- REFLECTREFLEVEL: To perform this subroutine, node u sets the r bit in its height to 1. That is, node u sets its height to $(\tau^u, oid^u, 1, 0, nlts^u, lid^u, u)$. This changes the RL from being unreflected to being reflected.

```
**Signature:**
Input:
  $receive(m)_{v,u}$, $m \in M$
  $channelUp_{\{u,v\}}$
  $channelDown_{\{u,v\}}$
Output:
  $send(m)_{u,v}$, $m \in M$

**States:**
  $clock$, a logical clock value, initially $\perp$
  $N$, a set of node identifiers, initially $\emptyset$
  $height$, an array of height tokens, indexed by node ids from $I$, initialized as follows:
    $height[u] = (\perp, \perp, 0, 0, \perp, u, u)$
    for every $v \in I$ where $v \neq u$, $height[v] = null$
  for every $v \in N$:
    $sendBuffer(v)$, a FIFO queue of messages from $M$, initially empty

**Transitions:**
$receive(m)_{v,u}$
   Effect:
     see Figure 4.4
$channelUp_{\{u,v\}}$
   Effect:
     $N = N \cup \{v\}$
     append $height[u]$ to $sendBuffer(v)$
$channelDown_{\{u,v\}}$
   Effect:
     see Figure 4.3
$send(m)_{u,v}$
   Precondition:
     $m$ is at the head of $sendBuffer(v)$
   Effect:
     remove $m$ from the head of $sendBuffer(v)$

**Tasks:**
  $\{send(m)_{u,v} : m \in M\}$
```

Figure 4.2: Automaton LE for node u

- STARTNEWREFLEVEL: By executing this subroutine, node u sets the value of its height to $(clock, u, 0, 0, nlts^u, lid^u, u)$. In other words, it records the value of its logical clock and its own ID in the first two components of the height, and leaves the LP the same. The LP denotes which leader the RL is searching for.

- ELECTSELF: Node u performs this subroutine by setting the value of its height to $(\perp, \perp, 0, 0, clock, u, u)$. By doing this, it resets the first four components to their default values and updates the leader ID and the election timestamp to its own ID and its current logical time.

Now that we have described what each one of the subroutines does, we are ready to explain how the code in Figures 4.3 and 4.4 invokes these subroutines. We describe three of the transitions of the algorithm; the fourth one, the sending of a message, is straightforward.

channelDown event: The pseudocode for the *channelDown* action appears in Figure 4.3. When a node u receives a notification that one of its incident links has gone down, it may no longer have a path to the leader. Therefore, it does one of the following two actions: (1) if it has no neighbors at all, then it elects itself by executing subroutine ELECTSELF in Figure 4.5, or (2) if node u has at least one neighbor and u is a local sink, then it starts a new reference level (a search for the leader). In the second case, node u executes subroutine STARTNEWREFLEVEL in Figure 4.5 and then appends its new height to the *sendBuffer* of each of its neighbors. Note that when a new RL is started, the *nlts* and *lid* components remain the same as in the old height. This way node u ensures that the new search is looking for the current leader in u's view. If neither of these two conditions above is satisfied, the node takes no further action.

channelUp event: When a node u receives a notification of a channel going up to another node, say v, then u sends its current height to v and includes v in its neighbor set N.

```
When channelDown_{u,v} event occurs:
1.  $N := N \setminus \{v\}$
2.  if ($N = \emptyset$) then:
3.      ELECTSELF
4.  else if ($\forall v \in N, (nlts^u, lid^u) = (nlts^v, lid^v)$ and
            $height[u] < height[v]$) and $lid^u \neq u$ then:
5.      STARTNEWREFLEVEL
6.      append $height[u]$ to $sendBuffer(v)$ for all $v \in N$
7.  end if
```

Figure 4.3: Code triggered by *channelDown*

Receipt of a message: When a node u receives a message h_v from another node v, containing v's height, node u performs one of many possible sequences of actions based on its own height and the heights of its neighbors, as shown in Figure 4.4.

Whenever a node u receives a message from node v, it first records node v's height in its own height array, as shown in line 1 of the code. Node u also makes a copy of its current height in the variable *myOldHeight*.

Next, in line 3, node u checks whether its leader pair is the same as node v's. If this is not the case and if node v's LP has priority over node u's LP $((nlts^v, lid^v) < (nlts^u, lid^u))$, then node u executes subroutine ADOPTLPIFPRIORITY in line 17. By executing this subroutine node u "adopts" the LP of node v. Recall that because of the way the *nlts* components are compared, if $nlts^v \succ nlts^u$, then $(nlts^v, lid^v) < (nlts^u, lid^u)$. In other words, the leader pair of a "newer" leader is adopted to replace the leader pair of an "older" leader which helps information about new leaders to spread more quickly throughout the connected component.

Throughout the rest of this description, in lines 4-16, we assume the LP's of nodes u and v are the same.

Next, in line 4, node u checks whether it is a local sink. If this is not the case, node u does not perform any further action. Throughout the rest of the description, in lines 5-15, we assume node u is a local sink.

Next, in line 5, node u checks whether all of its neighbors have the same RL. If this is not the case, node u executes subroutine PROPAGATELARGESTREFLEVEL in line 14. This situation occurs when one or more RL's, started by some other nodes, are being propagated in the connected component and reach node u.

Throughout the rest of the description, in lines 6-12, we assume all of node u's neighbors have the same RL. There are three cases to consider:

Case 1: In line 6, node u checks whether the τ component is $\perp$ and the r bit is 0. This indicates that the common RL is not the default RL $(\perp, \perp, 0)$ and it is unreflected. In this case, node u executes subroutine REFLECTREFLEVEL in line 7. This case indicates that a particular branch of the search for the leader has hit a dead end.

Case 2: In line 8, node u checks whether the τ component is $\perp$, the r bit is 1, and *oid* is the same as u's ID. If this is the case, it means that the common RL is not the default RL $(\perp, \perp, 0)$, it is reflected and node u is the originator of the RL. In this case node u executes subroutine ELECTSELF in line 9. This implies that all the branches of the RL that u started reached dead ends and were reflected. Therefore, the search for a leader failed, and u elects itself as the new leader.

Case 3: Neither of the conditions in Cases 1 and 2 are satisfied. Therefore, the RL is either the default one $(\perp, \perp, 0)$, or it is reflected but node u is not the originator of the RL. In this case node u executes subroutine STARTNEWREFLEVEL in line 11. This case can occur when due to particular patterns of topology changes u is "surrounded" by neighbors with such heights and does not have any other neighbors to propagate the RL to.

Finally, in lines 20-22 node u checks whenever its height has changed from the value in *myOldHeight*, and if this is the case it sends a message with its new height to all of its neighbors.

When node u receives a message h from node $v \in N$:

```
 1.  $height[v] := h$
 2.  $myOldHeight := height[u]$
 3.  if $((nlts^u, lid^u) = (nlts^v, lid^v))$ then:  // leader pairs are the same
 4.     if $(\forall p \in N, (nlts^u, lid^u) = (nlts^p, lid^p)$ and
              $height[u] < height[p])$ and $lid^u \neq u$ then:  // u is a local sink
 5.        if $\exists(\tau, oid, r)$, $\forall w \in N$, $(\tau^w, oid^w, r^w) = (\tau, oid, r)$ then:
 6.           if $((\tau \neq \perp)$ and $(r = 0))$ then:
 7.              REFLECTREFLEVEL
 8.           else if $((\tau \neq \perp)$ and $(r = 1)$ and $(oid = u))$ then:
 9.              ELECTSELF
10.           else // $(\tau = \perp)$ or $(\tau \neq \perp$ and $r = 1$ and $oid \neq u)$
11.              STARTNEWREFLEVEL
12.           end if
13.        else // neighbors have different ref levels
14.           PROPAGATELARGESTREFLEVEL
15.        end if
        // else not local sink, do nothing
16.     end if
17.  else if $((nlts^v, lid^v) < (nlts^u, lid^u))$ then:  // new LP has priority
18.     ADOPTLPIFPRIORITY($v$)
19.  end if
20.  if ($myOldHeight \neq height[u]$) then:
21.      append $height[u]$ to $sendBuffer(v)$ for all $v \in N$
22.  end if
```

Figure 4.4: Code triggered by the receipt of a message.

```
ADOPTLPIFPRIORITY($v$)
1.  $height[u] := (\tau^v, oid^v, r^v, \delta^v + 1, nlts^v, lid^v, u)$
PROPAGATELARGESTREFLEVEL
1.  $(\tau^u, oid^u, r^u) := max\{(\tau^w, oid^w, r^w) | \; w \in N\}$
2.  $\delta^u := min\{ \; \delta^w \; | \; w \in N$ and $(\tau^u, oid^u, r^u) = (\tau^w, oid^w, r^w)\} - 1$
REFLECTREFLEVEL
1.  $height[u] := (\tau, oid, 1, 0, nlts^u, lid^u, u)$
STARTNEWREFLEVEL
1.  $height[u] := (clock, u, 0, 0, nlts^u, lid^u, u)$
ELECTSELF
1.  $height[u] := (\perp, \perp, 0, 0, clock, u, u)$
```

Figure 4.5: Subroutines of the LE Algorithm

4.4 Proof of Correctness

The composed system described above is proved to be correct in [16] in the case of causal clocks, which can be shown to include the logical clocks that we consider in this thesis. We can claim this because the system model we use in this thesis is a special case of the one in [16], based on the fact that a logical clock can be expressed as a particular type of causal clock. The other major difference between the two models, the synchrony of the notifications for a *channelUp* or a *channelDown* at the endpoints of an edge in our model versus the asynchrony of these notifications in [16], allows for simplifying some of the arguments in the original proof in [16].

4.4.1 Basic Invariants

First, we present a few useful invariants which holds at any point in the execution of the algorithm, even before topology changes stop. Invariants 4.4.1 and 4.4.2 show the relationship between the timestamps created by the algorithm for some node u and the logical-time values of the clocks at node u and any neighbor v of u.

We define a *height token* to be a height object as defined in Section 4.3.2 which can be present either at a node's state or in a channel. We also say *a height token is for some node* u when the last component of the height token is node u's ID. In the following invariant, we denote by $nlts(h)$ and $\tau(h)$ the *nlts* and τ components, respectively, of some height token h.

Given a state in which $Channel(\{u, v\})$ has status *up*, we define the (u, v) *height sequence* as the sequence of height tokens $(h_0, h_1, \cdots, h_m)$, where $h_0 = height_u[u]$, $h_m = height_v[u]$, and $h_1, \ldots, h_{m-1}$ is the sequence of height tokens in the message queue $queue_{u,v}$ of $Channel(\{u, v\})$ (the messages in transit from u to v), where h_m is at the head of the queue.

Invariant 4.4.1. *If h is a height token for a node u in the (u, v) height sequence, then $nlts(h) \preceq clock_u$ and $\tau(h) \preceq clock_u$.*

Invariant 4.4.2. *If h is a height token for a node u in the height array of node v then $nlts(h) \preceq clock_v$ and $\tau(h) \preceq clock_v$.*

Invariant 4.4.1 ensures that the timestamps of any height token for node u are not greater than the clock value of node u. Invariant 4.4.2 ensures that the timestamps of any height token for some node u are not greater than the local clock of some v whose height array contains u's height. The proofs of both invariants follow by induction on the number of steps in any arbitrary execution of the algorithm and by the definition of logical time.

Next, we show that in the (u, v) height sequence for some neighbors u and v, the LP component of each subsequent height is smaller than or equal to the previous one. Also, the RL component of each subsequent height is greater than or equal to the previous one, assuming the LP components of the heights are the same.

Invariant 4.4.3. *Let $(h_0, h_1, \cdots, h_m)$ be the (u, v) height sequence of any $Channel(\{u, v\})$, whose status is up. Then, the following are true if $m > 0$:*

1. $h_0 = h_1$

2. For all l*,* $0 \leq l < m$*,* $LP(h_l) \leq LP(h_{l+1})$*.*

3. For all l*,* $0 \leq l < m$*, if* $LP(h_l) = LP(h_{l+1})$*, then* $RL(h_l) \geq RL(h_{l+1})$*.*

Intuitively, the second part of the invariant follows from the fact that in the pseudocode, a node adopts a new LP only if the new value is smaller than its current one. Similarly, the third part of the invariant follows from the fact that a node propagates or reflects a RL only when it receives the new RL from a node with the same LP and only if the new RL is larger than the current RL. Also, when a node starts a new RL, it keeps the same LP as in its old height, and sets its RL to be larger than its old RL. The proof of this invariant follows by induction on number of steps in the execution and then case analysis on all the possible subroutines executed by nodes u and v.

4.4.2 Finite Number of Elections

Next, we present a few lemmas to show that only a finite number of elections can occur after the last topology change.

Before we proceed, we require some additional notation and assumptions. First, we fix an arbitrary execution α of the LE system. We assume there is at least one topology change event in α, denoted e_{LTC}; otherwise, it is trivial to show that nodes do not perform any actions and the execution satisfies the LE problem requirements. This is true because initially, each node is its own leader in its own connected component and so condition (1) of the LE problem definition is satisfied. Also, since initially there are no edges in the communication graph, conditions (2), (3), and (4) are satisfied vacuously.

Next, we define a subtree $LT(s, \ell)$ of the component graph that connects all nodes that have ever had the LP (s, ℓ).

Formally, we define the following with respect to any state in α. Let (s, ℓ) be a LP such that there are no topology change events after the event that creates leader pair (s, ℓ). Let $LT(s, \ell)$ be the subtree of the connected component whose vertices consist of all nodes that have ever taken on LP (s, ℓ) in α (even if they later took on a different LP). Moreover, the directed edges in $LT(s, \ell)$ are all ordered pairs (u, v) such that v adopts LP (s, ℓ) due to the receipt of a message from u. Since a node can take on a particular LP only once by Invariant 4.4.3, $LT(s, \ell)$ is a tree rooted at ℓ.

Note that the definition of $LT(s, \ell)$ is with respect to the connected component after the last topology change. Otherwise, the connected component is not well-defined.

Lemma 4.4.4. *Let* (s, ℓ) *be a LP that is created in* α *and such that no topology changes occur in* α *after the event that creates leader pair* (s, ℓ)*. Let* h *be a height token for some node* u *that appears in the state of some node or in some channel at some point in* α*. Then* $RL(h) = (\bot, \bot, 0)$ *and* $\delta(h)$ *is the distance in* $LT(s, \ell)$ *from* ℓ *to* u*.*

The proof of this lemma is by induction on the number of steps in the execution and a case analysis on all the possible ways node u can change its height in any step of the execution.

Lemma 4.4.5. *Suppose that, at some point in α, some node u adopts leader pair (s, ℓ) such that there are no topology change events in α after the event that creates leader pair (s, ℓ). Then, node u never subsequently becomes a local sink in α.*

Suppose in contradiction that u becomes a local sink during some event e later in the execution. It must be the case that immediately before e, in u's view, all of its neighbors have a common LP (s', ℓ'). Let event e' be the event in which u gets LP (s', ℓ'). We consider the finite execution fragment β beginning from the state after event e' and ending with the state after event e. Note that node u's LP does not change in β.

Let node v be the parent of u in $LT(s', \ell')$. Therefore, immediately after event e', by the pseudocode, the link between u and v is from u to v, in u's view. Since u becomes a local sink during event e, it must be true that in β either (1) u decreases its height, or (2) u learns about an increased height of v. The first case is not possible because u's LP does not change in β and, by Lemma 4.4.4, its RL and δ value do not change either. In the second case, if according to node u's state, the change in v's height is due to a different LP, then node u adopts the new LP, which is a contradiction to the fact that u's LP does not change in β. If v's LP does not change in u's view, by Lemma 4.4.4, we know that v's RL and δ value do not change either. Therefore, in both cases node u has an outgoing link towards node v immediately before event e, and therefore, it does not become a local sink during event e.

Lemma 4.4.6. *No node elects itself an infinite number of times in α.*

The proof of this lemma follows from the observations that a node needs to be a local sink in order to elect itself, and once a node elects itself, it can only become a local sink again if it adopts a new LP. At the time of the last topology change, however, there are only a finite number of different LP's. Therefore, after e_{LTC}, u can adopt each one of these LP's, and thus elect itself a finite number of times. Any other LP's that node u adopts are started after the last topology change and, by Lemma 4.4.5, node u does not become a local sink after adopting them. Therefore, it does not elect itself subsequently.

4.4.3 Finite Number of New RL's

Next, we show that there is only a finite number of new reference levels started after the last topology change. The proof of this result relies on several additional properties related to the DAG structure induced by the adoption of new RL's by nodes. Therefore, first we present the definition of an *RD* DAG. Intuitively, $RD(t,p)$ is a DAG that contains all nodes that have the reflected RL $(t, p, 1)$ or the unreflected RL $(t, p, 0)$ at some point in the execution. For brevity, we define the *prefix* of a RL to be the first two components of the RL. Therefore, both the reflected RL $(t, p, 1)$ and the unreflected RL $(t, p, 0)$ have RL prefix (t, p). A node u is the parent of a node v in $RD(t,p)$ if v got RL prefix (t, p) before u. Next, we provide a formal definition of $RD(t,p)$.

Let event e denote the event in α by which some node p starts a new RL $(t,p,0)$, such that there are no topology change events in α after e. Let $RD(t,p)$ be the subgraph of the connected component of G (the graph induced by the communication topology) whose vertices consist of p and all nodes that have taken on RL prefix (t,p) by executing either PROPAGATELARGESTREFLEVEL or REFLECTREFLEVEL in α (even if these nodes got a different RL prefix after having RL prefix (t,p)). In $RD(t,p)$, a directed edge exists from u to v if u and v are connected in G and u has RL prefix (t,p) prior to the event in which v first takes on RL prefix (t,p). We say that node u is a *predecessor* of node v in $RD(t,p)$ and v is a *successor* of u in $RD(t,p)$.

The following three lemmas show some of the properties of RD.

Lemma 4.4.7. *Let h be a height token for node u with RL prefix (t,p), such that there are no topology change events in α after the event that creates RL $(t,p,0)$. Then, u is in $RD(t,p)$.*

This lemma shows that, assuming RL $(t,p,0)$ is started after the last topology change, if a node has a height with RL prefix (t,p) at some point of the execution, then it must have gotten that RL as a result of either PROPAGATELARGESTREFLEVEL or REFLECTREFLEVEL, and not ADOPTLPIFPRIORITY. Recall that by the definition of $RD(t,p)$, a node is in $RD(t,p)$ if it got RL (t,p) as a result of either PROPAGATELARGESTREFLEVEL or REFLECTREFLEVEL.

Lemma 4.4.8. *If at some point in α there is a height token for node u with RL $(t,p,1)$, such that there are no topology change events in α after the event that creates RL $(t,p,0)$, then all neighbors of u are in $RD(t,p)$.*

This lemma ensures that if a node has a reflected RL started after the last topology change then all of its neighbors are in $RD(t,p)$. The proof of this lemma is by induction on the number of steps in α starting from the event that creates RL $(t,p,0)$. In the base case, when RL $(t,p,0)$ is started, the property is true vacuously because there is no height token with RL $(t,p,1)$ yet. The argument in the inductive step is that if some node u has RL $(t,p,1)$, then by the definition of $RD(t,p)$ and Lemma 4.4.7, node u could have gotten this RL by executing either PROPAGATELARGESTREFLEVEL or REFLECTREFLEVEL. Next, we do case analysis on these two possibilities to show that all of u's neighbors are also in $RD(t,p)$.

Lemma 4.4.9. *Consider two height tokens, h_u for a node u with $RL(h_u) = (t,p,r_u)$ and $\delta(h_u) = d_u$, and h_v for a neighboring node v with $RL(h_v) = (t,p,r_v)$ and $\delta(h_v) = d_v$, where RL $(t,p,0)$ is started in event e in α after which no topology changes occur. Then the following are true:*

1. *If $r_u < r_v$, then u is a predecessor of v in $RD(t,p)$. If u is a predecessor of v in $RD(t,p)$ then $r_u \leq r_v$.*

2. *If $r_u = r_v = 0$, then $d_u > d_v$ if and only if u is a predecessor of v.*

3. If $r_u = r_v = 1$, then $d_v > d_u$ if and only if u is a predecessor of v.

This lemma shows the relationship between two nodes' r bits and their positions in the RD DAG. For example, if two neighboring nodes have different r bit values, then the lemma states that the node with r bit 0 is the predecessor of the node with r bit 1. If two neighboring nodes have the same r bit, then we need to compare the δ values of their heights in order to determine which one is the successor and which one is the predecessor. The proof of Lemma 4.4.9 makes use of Lemma 4.4.7 and the definition of RD DAG in order to limit the ways in which the two nodes u and v have gotten RL prefix (t, p). From Lemma 4.4.8 we know that if a node is not in $RD(t, p)$, then none of its neighbors have a RL $(t, p, 1)$; this fact is used in the proof of part (3) of Lemma 4.4.9. The rest of the proof of Lemma 4.4.9 is by induction on number of steps in the execution. The main idea is to perform case analysis on all the possible ways in which node u can change (increase or decrease) its height.

Finally, we use these three main lemmas, together with some helper results to show that no node starts an infinite number of reference levels.

Lemma 4.4.10. *No node starts an infinite number of new reference levels in α.*

The proof of this lemma uses a very similar technique to the proof of Lemma 4.4.6. That is, there are only a finite number of different LP's that a node can adopt present in the system at the time of e_{LTC}. Moreover, by Lemma 4.4.6, we know that there is a finite number of new LP's created after the last topology change. Next, we perform case analysis on the different ways in which u can start a new RL to show that a node cannot start an infinite number of new RL's with the same LP. In the case analysis of the different ways to start a new RL we use Lemmas 4.4.7 and 4.4.9 in order to compare u's RL to the RL's of its neighbors and determine their relationship in $RD(t, p)$.

4.4.4 Accurate View of Neighbors' Heights

Next, we present some of the main results that show that eventually no messages are in transit and each node has an accurate view of its neighbors' heights.

Lemma 4.4.11. *In α, eventually the following is true. In each connected component, all nodes have the same leader pair.*

The proof follows from Lemma 4.4.6 and the fact that eventually each node in the system receives a message containing the lowest LP and adopts that LP.

Lemma 4.4.12. *In α, eventually there are no messages in transit.*

This result follows from Lemma 4.4.11 and the fact that eventually all nodes in the system have the same RL (in addition to having the same LP). Therefore, we can show that nodes do not change their height, and consequently do not send new messages.

Corollary 4.4.13. *In α, eventually every node has an accurate view of its neighbors' heights.*

4.4.5 Leader-oriented DAG

Finally, we present two more useful lemmas and then we combine all the lemmas from Sections 4.4.1 – 4.4.4 to prove the main result that eventually the properties required for a solution to the LE problem are satisfied.

Lemma 4.4.14. *In every state of α, a node is not a local sink.*

This lemma implies that a node is never a local sink immediately before and immediately after executing an action. Recall that each transition of the algorithm involves executing the entire block of code of Figure 4.4 and the corresponding subroutines (Figure 4.5) invoked from it. The proof of this lemma performs case analysis on all the possible events that can occur at each node to show that after an action has been executed by any node u, u is no longer a local sink.

The next lemma establishes the relationship between the RL and δ values of a node's height and also states what the RL and δ values of a leader's height are.

Lemma 4.4.15. *Consider any height token h for node u. In every state of α, if $RL(h) = (\perp, \perp, 0)$, then $\delta(h) \geq 0$. Furthermore, $RL(h) = (\perp, \perp, 0)$ and $\delta(h) = 0$ if and only if u is a leader.*

The proof of this lemma also follows by induction on the number of steps in α and case analysis on all the possible events that can occur at a node.

Finally, we present the main theorem stating that the the properties required for a solution to the LE problem are satisfied.

Theorem 4.4.16. *In α, eventually each connected component satisfies the properties required for a solution to the LE problem.*

Proof. First, we fix a connected component CC. By Lemma 4.4.11, in α, eventually all nodes in the component have the same LP, say (s, ℓ). By Lemma 4.4.12, in α, eventually there are no messages in transit. By Corollary 4.4.13, in α, every node eventually has an accurate view of its neighbors' heights. Moreover, eventually, there are no more topology changes in α. We fix a prefix of α in which these properties have stabilized. Let s be the final global state of that prefix. At this point we have shown that part (2) of the LE problem definition is satisfied because no more topology changes occur and nodes have an accurate view of their neighbors' heights. Moreover, by the definition of a node's height, it follows that it is not possible to have cycles in the graph, and so part (4) of the LE problem definition is satisfied.

First, we show that in state s node ℓ must be in connected component CC. Suppose in contradiction that node ℓ is not in the component. Since cycles are not possible, there is some node u in the component that has no outgoing edges in state s. But this node is not ℓ, since we are assuming ℓ is not in the component. However, by Lemma 4.4.14, no node is a local sink. Therefore, since each node in CC has the same LP and is not its own leader, the only reason it is not a local sink is that it has some outgoing edge, which is a contradiction. So far, we have shown that part (1) of the LE problem definition is satisfied.

Now that we know that node ℓ is in connected component CC, we can proceed to show that we have an ℓ-oriented DAG in state s. Lemma 4.4.15 states that node ℓ, and only node ℓ, has RL $(\perp, \perp, 0)$ and zero δ. Since no node has a RL smaller than $(\perp, \perp, 0)$, Lemma 4.4.15 implies that each node, except ℓ, in the component has either a $(\perp, \perp, 0)$ RL and $\delta(h) \geq 0$ or a RL greater than $(\perp, \perp, 0)$. In either case, ℓ has the smallest height in the entire component in state s and therefore has no outgoing links. By Lemma 4.4.14, no node is a local sink. Therefore, there are no local sinks in the component in state s, and since all nodes have the same LP and no node, except ℓ, is a leader, it follows that each node, except ℓ, has an outgoing link. Also, since there are no cycles in the graph and ℓ is the unique node with no outgoing links, it follows that in state s the graph is a leader-oriented DAG, where ℓ is the leader. Therefore, part (3) of the LE problem definition is satisfied.

□

4.5 Complexity Analysis

In this subsection, we analyze the complexity of the LE algorithm. We show that the LE algorithm described above performs no more than $O(n)$ elections after the last topology change before satisfying the properties required for solving the LE problem. We begin by giving a quick overview of the structure of the proof. First, in Lemma 4.5.2, we show that if some node u ever gets a reflected RL that was started after the last topology change, then u also had the same unreflected RL earlier. Next, in Lemma 4.5.3, we show that if after the last topology change some node elects itself, then it must be the case that all nodes in the connected component had the RL which led to the election at some point in the execution. In other words, all nodes participated in the unsuccessful search for the leader. Finally, in Theorem 4.5.4, we show that after topology changes stop, no node elects itself more than twice. We end the section with a discussion on translating the bound on the number of elections into a bound on the total number of reversals performed by the nodes.

Lemma 4.5.1. *Suppose some node v gets RL $(t, u, 0)$ in event e, and then gets RL $(t, u, 1)$ in event e'. Then, between events e and e', v's RL does not change.*

Proof. By the properties of logical clocks we know that the same RL $(t, u, 0)$ cannot be started twice, because both such events occur at the same node, so they cannot have the same logical clock timestamp. Also, by the pseudocode, when an unreflected RL is reflected, the LP remains the same, so all nodes that have RL $(t, u, 0)$ or $(t, u, 1)$ have the same LP, say LP (s, w). Since node v has RL $(t, u, 0)$ immediately after event e and has RL $(t, u, 1)$ immediately after event e', both with LP (s, w), then, by Invariant 4.4.3, v's LP does not change between events e and e'.

Suppose in contradiction, node v gets some RL (t', u', r), different from $(t, u, 0)$, between events e and e'. Since v's LP does not change in that interval, by the pseudocode, v can change its RL only by propagating, reflecting or starting a RL. Therefore, by Invariant 4.4.3, RL (t', u', r) is larger than $(t, u, 0)$, and also RL $(t, u, 1)$ is larger than RL (t', u', r). This is a contradiction because there is no RL, generated by the LE algorithm, which is larger than $(t, u, 0)$ and smaller than $(t, u, 1)$. □

Lemma 4.5.2. *Let RL $(t, u, 0)$ be started in event e after the last topology change. Also, suppose some node v gets RL $(t, u, 1)$ in event e' by executing either* PROPAGATELARGESTREFLEVEL *or* ADOPTLPIFPRIORITY. *At some point between events e and e' node v's RL is $(t, u, 0)$.*

Proof. Suppose node v has RL $(t, u, 1)$ immediately after event e' and it does not get it by executing REFLECTREFLEVEL. Since RL $(t, u, 0)$ is started during event e, after the last topology change, and the reflected RL $(t, u, 1)$ is present in the connected component during event e', then it must be the case that some node $y \neq v$ executed REFLECTREFLEVEL between e and e', and thus created RL $(t, u, 1)$. Let the sequence of nodes $(y = w_0, w_1, w_2, \cdots, w_{k-1}, w_k = v)$ represent the "path through which RL $(t, u, 1)$ reached node v". More formally, for each $1 < i \leq k$, node w_i first got RL $(t, u, 1)$ by executing either PROPAGATELARGESTREFLEVEL or ADOPTLPIFPRIORITY after receiving a message from node w_{i-1}. Next, we show by induction on the indices, i, of the sequence defined above, starting from index 1, that node v has RL $(t, u, 0)$ at some point between events e and e'.

Base Case: $i = 1$. By the pseudocode, the precondition for node y to reflect RL $(t, u, 1)$ is that all of its neighbors have RL $(t, u, 0)$, in node y's view. This implies that each of u's neighbors has RL $(t, u, 0)$ at some point after event e, which creates RL $(t, u, 0)$, and before node y creates RL $(t, u, 1)$. Since, event e', in which node v gets RL $(t, u, 1)$, occurs after the event in which that RL is created, it follows that node w_1 has RL $(t, u, 0)$ at some point between events e and e'.

Inductive Hypothesis: Assume the property is true for i. Therefore, node w_i has RL $(t, u, 0)$ at some point between events e and e'.

Inductive Step: We need to show that the property is true for $i + 1$. We need to show that node w_{i+1} has RL $(t, u, 0)$ at some point between events e and e'. Suppose in contradiction that node w_{i+1} does not have RL $(t, u, 0)$ at any point between events e and e'.

Let event e_i be the event in which node w_i gets RL $(t, u, 1)$. By the pseudocode and the assumption in the lemma statement, there are two ways in which node w_i gets RL $(t, u, 1)$.

Case 1: Node w_i executes ADOPTLPIFPRIORITY. By the properties of logical clocks we know that the same RL $(t, u, 0)$ cannot be started twice, because both such events occur at the same node, so they cannot have the same logical clock timestamp. Also, by the pseudocode, when an unreflected RL is reflected, the LP remains the same, so all nodes that have RL $(t, u, 0)$ or $(t, u, 1)$ have the same LP. Therefore, it is not possible for node w_i to execute ADOPTLPIFPRIORITY while having RL $(t, u, 0)$ and result in having RL $(t, u, 1)$.

Case 2: Node w_i executes PROPAGATELARGESTREFLEVEL. Therefore, by the pseudocode, immediately before event e_i, node w_i must be a local sink and the maximum RL among its neighbors, in its view, must be $(t, u, 1)$. Since we know that node w_i gets RL $(t, u, 1)$ before node w_{i+1} does, it follows that immediately before event e_i node w_{i+1}'s RL, in w_i's view, is different from $(t, u, 1)$. Also, since node w_i is a local sink immediately before event e_i, it must be the case that, in w_i's view, node w_{i+1} has the same LP as node w_i and a larger height. By Lemma 4.5.1, the RL of w_i immediately before event e_i is $(t, u, 0)$, and so the only possible value for the height of node w_{i+1}, in w_i's view, is RL $(t, u, 0)$ and a δ value higher than the δ value of node w_i. Therefore, node w_{i+1} must have RL $(t, u, 0)$ at some point between e and e'. This is a contradiction to the initial assumption.

□

Lemma 4.5.3. *Let RL $(t, u, 0)$ be started in event e after the last topology change. If node u elects itself, in event e' as a result of receiving RL $(t, u, 1)$ from all of its neighbors, then all nodes in u's connected component have RL $(t, u, 1)$ at some point between events e and e'.*

Proof. Suppose in contradiction that there exists a node that does not have RL $(t, u, 1)$ at any point between events e and e'. By the pseudocode and the preconditions for a node to elect itself, we know that just before event e' at least one node, which is a neighbor of u, has RL $(t, u, 1)$. Since the topology is connected there must exist two neighboring nodes y and z such that y has RL $(t, u, 1)$ at some point between events e and e' and z does not. Let e_y be the event in which node y gets RL $(t, u, 1)$.

Claim A: Node z does not have RL $(t, u, 0)$ at any point prior to event e'.

Suppose in contradiction node z has RL $(t, u, 0)$ at some point prior to event e'. By Lemma 4.4.7, node z is in $RD(t, p)$. There exists a sequence of nodes $(z = v_1, v_2, \cdots v_{k-1}, v_k = u)$ such that for $1 < i \leq v$ it is true that v_{i-1} is the successor of v_i in RD (t, u). By assumption, node u elects itself in event e' by receiving reflected RL's from all its neighbors. Therefore, immediately before event e', node v_{k-1} has RL $(t, u, 1)$, in node u's view. Therefore, at some previous point, node v_{k-1}'s RL must be $(t, u, 1)$. Next, we do induction on the indices in the sequence of nodes, starting from index $k - 1$, to show that node z must have RL $(t, u, 1)$. In the base case, we already established that node v_{k-1} has RL $(t, u, 1)$ at some point before event e'. Suppose node v_j has RL $(t, u, 1)$ at some point before event e'. By construction, node v_j is a predecessor of node v_{j-1} in RD (t, u). By applying Lemma 4.4.9, part (1), it follows that node v_{j-1} has r bit 1 at some point before event e'. Therefore, it follows that node z must also have r bit equal to 1 before event e', which is a contradiction to the initial assumption. (end of claim)

By Lemma 4.5.2, we know that before having RL $(t, u, 1)$, node y must have RL $(t, u, 0)$ at some earlier point, unless y got RL $(t, u, 1)$ by executing REFLECTREFLEVEL. However, by the pseudocode, we know that in order for node y to reflect a RL, all of its neighbors, including z, must have the same unreflected RL $(t, u, 0)$ in node y's view. Therefore, at some earlier point node z must have RL $(t, u, 0)$. This is a contradiction to Claim A.

Let event e'_y be the event in which node y first gets RL $(t, u, 0)$. Next, we reason about how node y got RL $(t, u, 1)$ during event e_y. Note that, by Lemma 4.5.1, between events e'_y and e_y node y does not change its height. There are two possible cases:

Case 1: Node y executes ADOPTLPIFPRIORITY. By the properties of logical clocks we know that the same RL $(t, u, 0)$ cannot be started twice, because both such events occur at the same node, so they cannot have the same logical clock timestamp. Also, by the pseudocode, when an unreflected RL is reflected, the LP remains the same, so any node that has RL $(t, u, 0)$ or $(t, u, 1)$ has the same LP. Therefore, it is not possible for node y to execute ADOPTLPIFPRIORITY while having RL $(t, u, 0)$ and result in having RL $(t, u, 1)$.

Case 2: Node y executes PROPAGATELARGESTREFLEVEL. In order for node y to propagate RL $(t, u, 1)$ during event e_y, it needs to be a local sink immediately before event e_y and, in its view, the largest RL among its neighbors must be $(t, u, 1)$. Since node y is a local sink immediately before event e_y and its RL is $(t, u, 0)$, it must be the case that, in y's view, node z has the same LP as y and a larger height. The only possible value for the height of node z, in y's view, is RL $(t, u, 0)$ and a δ value higher than the δ value of node y. This is a contradiction to Claim A. □

Theorem 4.5.4. *No node elects itself more than twice after e_{LTC}.*

Proof. Suppose in contradiction that some node u elects itself three times after e_{LTC}. Let the first time u elects itself be in event e_1, the second time – in event e_2, and the third time – in event e_3. In order for u to elect itself in event e_3, u must become a local sink after event e_2. Since no topology changes occur after event e_2, the only way for u to become a local sink is to adopt a different LP, say (t_2, v). By Lemma 4.4.5 if (t_2, v) was created after e_{LTC}, then after u adopts (t_2, v), it will not become a local sink subsequently. Therefore, it has to be the case that (t_2, v) is created before e_{LTC}. We also know that, by the properties of logical clocks, if u is to adopt (t_2, v), it must be the case that $t_2 \geq L(e_2)$. Recall that for each event e, $L(e)$ is the logical time value assigned to that event.

Now, we reason about the second time node u elects itself (in event e_2). From the pseudocode we know that after e_{LTC}, node u can elect itself only after it receives reflected RL's from all its neighbors. Let that reflected RL be $(s, u, 1)$, and the corresponding unreflected RL be $(s, u, 0)$ such that $s < L(e_2)$. This inequality follows from the fact that both these events occur at the same node, and the properties of logical clocks ensure that $s < L(e_2)$. Also, assume that the LP of u at the time RL $(s, u, 0)$ is started is (t_1, x) where, by Invariant 4.4.2, $t_1 < L(e_2)$.

Let w be some node in u's connected component such that w has a height token with LP (t_2, v) at some point in the execution before event e_{LTC} occurs. We know such a node exists because, otherwise, LP (t_2, v) would not be present in any node's state or in any channel in u's connected component at the time of the last topology change. Therefore, it would not be possible for node u to adopt LP (t_2, v) later in the execution. By Lemma 4.5.3, node w must have RL $(s, u, 1)$ at some point before event e_2. Now, we consider how w first got RL prefix (s, u). By the pseudocode, node w can get RL prefix (s, u) by either adopting a LP, propagating a RL, or reflecting a RL.

Case 1: If w adopts LP (t_1, x) (the LP with which the RL $(s, u, 0)$ is started by u), then it follows that $t_2 \leq t_1$. However, we know that $t_1 < L(e_2)$, so it follows that $t_2 < L(e_2)$, which is a contradiction.

Case 2: Now, suppose that w gets RL prefix (s, u) by propagating a larger RL or reflecting a RL. Therefore, by the pseudocode it must be the case that $(t_1, x) = (t_2, v)$ in order for node w to execute PROPAGATELARGESTREFLEVEL. However, we know that $t_1 < L(e_2)$ and $t_2 \geq L(e_2)$, so it is not possible for $t_1 = t_2$.

□

Corollary 4.5.5. *After the last topology change, no more than $O(n)$ new LP's are created.*

We would like to be able to translate this bound on the number of elections to a bound on the total number of link reversals performed by the algorithm. However, the number of reversals performed by the algorithm depends on the state of the system at the time of the last topology change because of the possibility of pileups of messages (containing potentially distinct LP's and RL's) in the channels immediately before the last topology change. Each such distinct LP or RL can cause multiple reversals after the last topology change. Depending on the particular patterns of topology changes, it is possible that some channels contain an unbounded number of messages containing distinct LP's and RL's. Therefore, this would result in an unbounded number of reversals performed by the algorithm after the last topology change.

4.6 Stability Properties of the LE Algorithm

In this section, we describe certain conditions under which even though a leader is elected in the connected component, some topology changes trigger the election of a new leader. We call such properties *stability properties* because they give us insight into how "stable" an election is; the more leaders are elected unnecessarily (when there is already an existing leader in the connected component), the more "unstable" the algorithm is.

Naturally, such unnecessary elections are not desirable because after a new leader is elected, messages about that new leader need to propagate through the entire connected component. The more unnecessary elections, the worse the general efficiency of the algorithm is, in terms of the total messages sent in an arbitrary execution.

4.6.1 One-topology-change Stability Property

First, we describe a stability property satisfied by the LE algorithm of [17], which uses a global clock. Then, by giving a particular counterexample execution, we show that the LE algorithm with logical clocks, which we consider in this thesis, does not satisfy this property. Finally, we state a similar property that holds in the case of logical clocks.

First, we need to define the notion on a global clock. The LE algorithm in [17] assumes that there exists a positive real-valued global time gt assigned to each event e_i, such that $gt(e_i) < gt(e_{i+1})$ and, if the execution is infinite, the global times increase without bound. The global time of the first event in the execution is defined to be 0.

The following theorem appears in [17] and states that when the algorithm is in a particular "nice state", with a unique leader being elected and no messages in transit, then no single *channelDown* event can cause a new leader to be elected.

Theorem 4.6.1. *Suppose at global time $t = gt(e)$, for some event e in some arbitrary execution α, a connected component G' is a leader-oriented DAG with no messages in transit and leader ℓ. Further, suppose a link in G' goes down in event e. Let the resulting connected* *component containing ℓ be G. Then, as long as there are no further topology changes involving links incident to nodes in G, no node in G elects itself after event e.*

The proof of this theorem relies on the fact that at time t, any old reference levels have a timestamp smaller than t. Therefore, any reference levels started after time t have a greater timestamp than any reference levels started before time t. This makes it impossible for old reference levels to get propagated and, thus, result in an election. Finally, we use a property of the LE algorithm in [17], similar to Lemma 4.4.5, to show that no new reference level started after the last topology change (the *channelDown* event at time t) can result in a node becoming a local sink, and consequently electing itself.

Next we show that the result above does not apply to the LE algorithm with logical clocks. Formally, we show that the following conjecture is not satisfied by the LE algorithm with logical clocks.

Conjecture 4.6.2. *Suppose at some state s in some arbitrary execution α, a connected component G' is a leader-oriented DAG with no messages in transit and leader ℓ. Further, suppose a channelDown event occurs in G'. Let G be the connected component containing ℓ after the channelDown event. Then, as long as there are no further topology changes involving links incident to nodes in G, no node in G elects itself after the channelDown event.*

We provide a particular counterexample execution fragment which starts with the network topology in Figure 4.6 (a). We assume no messages are in transit in this configuration. We show that after some point the system is quiescent and satisfies the properties required for a solution to the LE problem. Then, we introduce a single *channelDown* event and show that eventually a new leader is elected as a result. Figure 4.6 illustrates the entire execution fragment.

In the initial configuration (part (a)) all nodes have a path to the leader L. All reference levels are set to the default values. The δ values correspond to the shortest paths to the leader. Next, in part (b), two new links (depicted in gray) go up and one link (depicted with a dashed line) goes down. As a result, node a is a local sink and starts a new reference level with timestamp 15. In part (c), two more links go down. As a result, node d is a local sink and starts a new reference level with timestamp 20. Next, in part (d), node b reflects the reference level it received from node d. Also, two links go up before the message from node b, containing the reflected RL, is delivered at node d. After the two new links go up, all messages in the system are delivered. At this point no messages are in transit and all nodes have a path to the leader L. Then, in part (e), a *channelDown* event occurs at node c, which is also the last topology change. As a result, node c is a local sink and, thus, starts a new reference level. Since nodes have logical clocks, the value of node c's clock may be lower than that of node d because there may not be a causal chain of events to ensure that the logical clock value of c during the last topology change is higher than the logical clock value of d when d started RL $(20, d, 0)$. Finally, node a receives the new reference level from node c, and since a is a local sink, it propagates the largest reference level of its neighbors, which

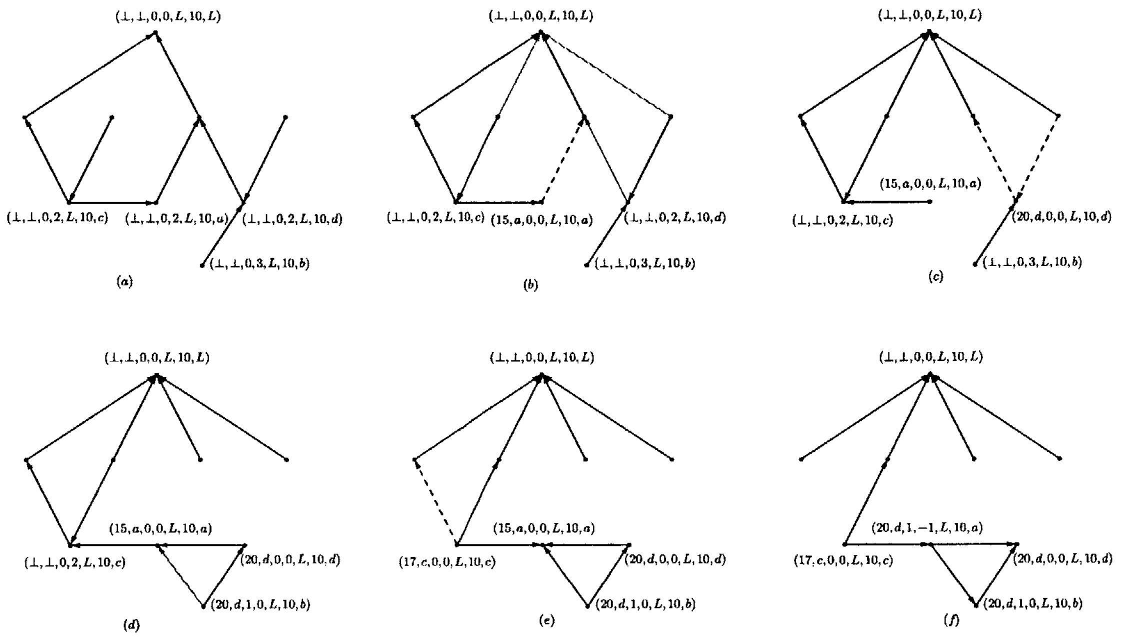

Figure 4.6: Counterexample execution

is b's reference level. Node d receives a reflected reference level from both of its neighbors and, therefore, elects itself.

Even though the stability property of Theorem 4.6.1 does not hold in the case of logical clocks, we can state a very similar property that does.

Theorem 4.6.3. *Suppose at some state s in some arbitrary execution α, a connected component G' is a leader-oriented DAG with no messages in transit and leader ℓ. Let the largest logical clock value of all nodes in state s be t. Further, suppose a link in G' goes down in event e. Let the resulting connected component containing ℓ be G. Then, at any point of α after which there are no topology changes involving links incident to nodes in G, and the logical clock of each node is greater than t, no node in G elects itself.*

The proof of this theorem is almost identical to the proof of Theorem 4.6.1. We have circumvented the problem of old RL's propagating and leading to an election by requiring all nodes to reach high enough logical clock values before guaranteeing that no more elections occur.

4.6.2 Other Stability Properties

It is possible that the LE algorithm with a global clock satisfies other, stronger stability properties like, for example, no leader is elected unnecessarily when topology changes are limited to a particular part of the connected component, or no leader is elected unnecessarily when the only topology changes are *channelDown* events. However, in the case of logical clocks, we can show that none of these properties hold because of Conjecture 4.6.2.

More precisely, we state the following conjectures and then explain why they are not satisfied by the LE algorithm with logical clocks.

Conjecture 4.6.4. *Suppose at some state s in some arbitrary execution α, a connected component G' is a leader-oriented DAG with no messages in transit and leader ℓ. Further, suppose only a finite number of topology changes occur in α and let C be the sequence of topology changes that occur in α after state s, subject to the following constraints. There exists a set of nodes $S \subset V'$, where V' is the set of nodes in G', such that no topology change events occur involving links incident to any node $u \notin S$. Let G be the connected component containing ℓ after the last topology change in C. Then, as long as there are no further topology changes involving links incident to nodes in G, no node in G elects itself after the last topology change of C.*

This conjecture states that an unnecessary election does not occur if topology changes are limited to some set of nodes, strictly smaller than the set of all nodes in the connected component. In other words, if topology changes are localized, then no new leader is elected in the connected component.

Conjecture 4.6.5. *Suppose at some state s in some arbitrary execution α, a connected component G' is a leader-oriented DAG with no messages in transit and leader ℓ. Further, suppose only a finite number of topology change occur in α and let C be the sequence of channelDown events that occur in α after state s. Let G be the connected component containing ℓ after the last topology change in C. Then, as long as there are no further topology changes involving links incident to nodes in G, no node in G elects itself after the last channelDown event in C.*

This conjecture states that an unnecessary election does not occur if topology changes are limited to *channelDown* events. In other words, if no new links go up after some point in the execution, then no new leader is elected in the connected component.

Note that a special case of both of the conjectures above is that only one *channelDown* event occurs in G'. That is, after state s only one topology change occurs and so the sequence C contains only one element. Then, if either of Conjecture 4.6.4 and Conjecture 4.6.5 is true, then that would imply that Conjecture 4.6.2 is true. However, we know this is not the case because of the counterexample in Section 4.6.1. Similarly to Theorem 4.6.3 , the modified statements guarantee the corresponding properties hold after some time in which clock values are sufficiently large, as opposed to after the last topology change.

Next, we give some intuition for this lack of nice stability properties for logical clocks. Suppose we fix a particular connected component and assume there are currently no messages in transit and all nodes have a path to the current leader. Therefore, all nodes in the component have the same LP, but it is possible for different nodes in the component to have different (reflected or unreflected) reference levels. Such RL's may correspond to successful searches for the current leader. Now suppose some pattern of topology changes triggers a new RL to be started. Since the algorithm uses logical clocks, this new RL may have a timestamp smaller than other existing timestamps in the component. Therefore, if some node receives the new RL, and one of its neighbors has a larger old RL, it is the old one that gets propagated. So, it is possible that an old RL propagates and eventually returns to the originator of the RL. This results in a new election even though there is currently another leader in the component.

4.6.3 Stability Properties of Other Algorithms

Finally, it is worth mentioning some stability properties satisfied by other LE algorithms in the literature. We focus on one particular LE algorithm from [9], which is very similar to the LE algorithm we consider in this thesis. As mentioned in Chapter 1, the algorithm in [9] is a self-stabilizing LE algorithm for dynamic networks and works in a shared-memory model. It is also completely asynchronous and does not use any notion of clocks. Moreover, the algorithm in [9] satisfies two main stability properties that the authors call the *incumbency* and *no dithering* properties. Here, we summarize these two stability properties.

The *incumbency* property states that in a given connected component in a post-legitimate state, if at least one node was the leader in the previous legitimate state, then it is one of those past leaders that gets elected as the final leader. Here a legitimate state refers to a state in which there is a unique leader and each node knows the ID of the leader; a post-legitimate state is derived from a legitimate state after some number of topology changes occur. The *no dithering* property states that throughout the execution of the algorithm, each node (of a particular connected component) may change its choice of leader at most once.

It is clear that the algorithm in [9] satisfies a much wider range of stability properties compared to the LE algorithm we consider in this thesis. However, due to the differences in the system models, it is very difficult to determine whether these differences are due to the algorithm in [9] being more efficient, or due to the algorithm in [9] having stronger system-model assumptions. For example, one difference between the two system models is that the algorithm in [9] assumes a shared memory model, which implies that a node can read its neighbors' state variables instantaneously. This is not possible in a message-passing model, which we consider for our LE algorithm.

4.7 LE Algorithm and Shortest Paths

In this section we show that we can augment our LE algorithm in such a way that it guarantees that each node eventually obtains a shortest path, with respect to the number of hops, to the leader. Such a guarantee can be especially useful in routing applications where one of the main efficiency metrics is the length of the paths.

There are various shortest-path distributed algorithms in the literature. One of the first and simplest algorithms to find the shortest paths in a graph is the Bellman-Ford algorithm [1], which can easily be adapted to work in a distributed setting. The simple strategy in Bellman-Ford is to keep adjusting each node's estimate of a shortest path to the destination until it stabilizes to the minimum such value. Each node u does so by looking at the estimates of each of its neighbors v and checking if a path through v is shorter than u's current estimate. Meanwhile the destination node keeps its estimate at 0.

Next, we show how to augment our LE algorithm in order to ensure that eventually each node has a shortest path to the leader. However, unlike the Bellman-Ford algorithm, each node in our shortest-path algorithm ignores its current distance estimate and always chooses the minimum of its neighbors' estimates. We need this modification in order to ensure that the algorithm works correctly in the presence of topology changes.

We introduce the following additions to the LE algorithm. First, we list the new variables added to the state of each node u:

- Each node u keeps track of its parent in the connected component through which u has a shortest path to the current leader. For this purpose, node u has a variable $parent_u$ in its state, initially set to $\perp$.

- Each node u keeps track of its distance estimate to the current leader and its neighbors' distance estimates to the current leader. Node u does so by maintaining an array $dist_u$ of integers, indexed by node ID's. Initially, $dist_u[u] = 0$ and $dist_u[v]$ of $v \neq u$ is set to $\perp$ because, in the initial state, each node is its own leader in its own connected component.

- Each node u maintains a boolean flag *updateDist* that indicates whether the node needs to update its distance estimate. Initially, *updateDist* is set to $false$.

Next, we describe the new actions for each node u, and the modifications to existing actions:

- Whenever a node changes its height, learns about a new height of some of its neighbors, receives a new estimate $dist_u[v]$ for one of its neighbors $v \in N_u$, or receives a notification for a *channelDown* event, the flag *updateDist* is set to *true*.

- A new action *update* is added such that its precondition is that the *updateDist* flag is *true*. The *update* action performs an update of the parent and distance estimates of node u. More precisely, let $minDist = \min\{dist_u[v] | v \in N_u \text{ and } lid^u = lid^v\}$. If $u \neq lid^u$, then node u sets $dist_u[u] := 1 + minDist$, and it also sets $parent_u = v$, where $dist_u[v] = minDist$. If $u = lid^u$, then $dist_u[u] := 0$ and $parent_u = \perp$. After the update is complete, the *updateDist* flag is set back to $false$.

- Whenever a node changes its $dist_u[u]$ value or receives a notification for a *channelUp* event, it sends its $dist_u[u]$ value to all of its neighbors.

Next, we show that the augmented LE algorithm described above guarantees that the properties required for a solution to the LE problem are satisfied and also, eventually, each node has a shortest path to the leader in that connected component. Since the additional actions of the algorithm do not modify any of the original variables of the LE algorithm, the result of Theorem 4.4.16 still holds, and guarantees that eventually the properties required for a solution to the LE problem are satisfied. Therefore, eventually, each node u in each connected component knows the ID of the current unique leader in that connected component. We fix a connected component CC; let the unique leader in CC be ℓ.

Theorem 4.7.1. *In each execution α that contains only a finite number of topology changes, there exists a state s such that for any state after s in α it is true that each node u in CC is at distance d from the leader ℓ iff $dist_u[u] = d$.*

Proof Sketch. Next, we provide a proof sketch of the correctess of the augmented algorithm. Similar correctness proofs of self-stabilizing algorithms also appear in [12] and [18].

We prove this theorem by strong induction on d. In the inductive hypothesis we assume that there exists a state s such that for any state after s in α it is true that each node u in CC is at distance $i < d$ from the leader ℓ iff $dist_u[u] = i$. In the inductive step we need to show that there exists another state s' such that for any state after s' in α it is true that each node v in CC is at distance d from the leader ℓ iff $dist_v[v] = d$.

In one direction, we need to show that if node v is at distance d from ℓ, then there exists a state in which $dist_v[v] = d$ and also $dist_v[v] = d$ in all subsequent states. Therefore, we need to first show that node v sets $dist_v[v] := d$ at some point in α and then subsequently never changes that value. The proof of this direction is by contradiction and case analysis on the possible values of $dist_v[v]$.

In the other direction, we need to show that if there exists a state such that $dist_v[v] = d$ in all subsequent state, then node v is at distance d from ℓ. Again, we assume for contradiction that v is not at distance d from ℓ and do case analysis on the possible values of that distance. □

4.8 Conclusion and Future Work

In this chapter, we introduced the LE algorithm of [17] and [16], summarized its proof of correctness, and presented some additional analysis of its properties. More precisely, we analyzed the complexity of the algorithm in terms of the number of elections that occur before a unique leader is elected and all nodes in the graph have a directed path to the leader. We showed that, in the worst case, no more than $O(n)$ elections occur after the last topology change. We also described a stability property guaranteed by the algorithm in [17] (using a global clock). We showed that this property no longer holds for the system model of [16] and this thesis, which uses logical clocks instead of a global clock. We provided discussion and intuition on additional stability properties of both our LE algorithm and other LE algorithms in literature. Finally, we showed how to combine the LE algorithm with a very simple self-stabilizing algorithm in order to ensure that not only does each node have a directed path to the leader, but also that this path is among the shortest ones.

It still remains to examine the behavior of the algorithm under different clock models. In this thesis and in [16], we have used logical clocks as the timing model, which is weakening the strong global-clock model of [17]. We would like to understand exactly what timing guarantees are necessary for the correctness of the algorithm, and whether different timing models make a difference in the efficiency of the algorithm. Also, while we have good intuition on why various stability properties do not hold for the LE algorithm with logical clocks, it is interesting to formalize the exact conditions under which certain stability properties hold, or to find weaker stability properties that can be guaranteed by the algorithm.

Finally, while Corollary 4.5.5 bounds the number of elections in the simplified system model of this thesis, we conjecture that the same result holds in the more general model of [16], which includes causal clocks and asynchronous topology change notifications at both endpoints of a channel.

Chapter 5

Conclusion

In this thesis, we have presented two link-reversal algorithms and some of the interesting properties they satisfy. In Chapter 3, we showed that in any execution of the Partial Reversal (PR) algorithm [13], no cycles are created in the graph induced by the connectivity of the nodes. The proof of this result relies only on simple invariants of the PR algorithm, and unlike existing proofs, does not assume a global assignment of heights to the nodes, or a more complicated algorithm of which PR is a special case.

While the proof of the acyclicity property of PR itself is quite simple, it provides a key insight into the working of the PR algorithm. Reasoning about the actual edge directions and local reversals performed by the algorithm, as opposed to imposing higher-level structures on the global topology, exposes simple invariants that describe fully and concisely the properties of PR. Similar techniques can potentially be useful in getting an in-depth understanding of other link-reversal algorithms, like for example the acyclicity and termination properties of the BLL algorithm [6], mentioned in Chapter 1.

In Chapter 4, we presented a leader election (LE) link-reversal algorithm [16] designed for dynamic networks and proved a bound on the number of total elections that occur after topology changes stop. We also discussed various patterns of topology changes that cause unnecessary elections in the sense that a new leader is elected when there is already an existing one in the same connected component. Finally, we showed that the LE algorithm can be extended in such a way that it provides all nodes in the system with the shortest paths to the elected leader. This is a property that is usually not satisfied by link-reversal algorithms, and at the same time it is very desirable given that the main application of many link-reversal algorithms is routing.

Several other properties of the LE algorithm turned out to be more difficult to prove or required additional assumptions. For example, the bounds we were able to provide for the complexity proof of the LE algorithm are in terms of the total number of elections that occur in the system after topology changes cease. Ideally, we would also like to provide bounds in terms of the total number of reversals performed by the algorithm, the total number of messages sent, or even a time bound on the total time it takes for the algorithm to stabilize, assuming some bound on message delays. The main difficulty we faced in trying to prove these bounds is the fact that particular patterns of topology changes can result in pile-ups of an unbounded number of messages in the channels.

Moreover, while we presented multiple cases in which the LE algorithm elects leaders unnecessarily when there is already an existing leader in the same connected component, it is still not clear whether there are weaker such stability properties that are guaranteed by the algorithm. Also, it is evident that such properties depend on the timing model used in the algorithm. Therefore, it is worthwhile classifying different timing models with respect to the the correctness, stability and efficiency of the LE algorithm implemented in these models. For example, instead of considering logical clocks, as we do in this thesis, or a global clock, as in [17], we can experiment with different clocks like, for instance, approximately-synchronized clocks. In that timing model, we are guaranteed that at any point in the execution of the algorithm, the clocks of any two nodes are no more than ϵ time units apart from each other.

It is also possible to modify the LE algorithm itself, instead of changing the type of clocks nodes use, in order to make it less volatile to topology changes and less dependent on the timing model. For example, it is possible to use a modification to the algorithm, similar to the concept of $WelchTime$ [20]. In the modified algorithm, nodes have (possibly unsynchronized) hardware clocks. Whenever a node receives information about a clock value "from the future", the node waits until its own clock reaches that value. Such a technique may successfully ensure that the causal relation between events in the LE algorithm is preserved.

As we mentioned in Chapters 1 and 4, some LE algorithms, like the self-stabilizing algorithms in [9], guarantee a much wider range of stability properties and provide better bounds on the number of leaders elected in an arbitrary execution. Since our LE algorithm also has a self-stabilizing flavor, it would be an interesting challenge to convert it to a fully self-stabilizing algorithm. One challenge in doing that would be to determine what the effect of corrupted clock values is on the correctness and complexity guarantees of the algorithm. Our conjecture is that when the modified LE algorithm starts from an arbitrary state, eventually, the logical clock values are restored to satisfying some well-behaved properties that are sufficient for the correctness of the LE algorithm.

Some advantages of our algorithm, compared to similar algorithms in [9], is that the simplicity of link-reversal algorithms make them easy to understand, implement, and analyze their safety and liveness properties. Moreover, our algorithm closely resembles the Temporally Ordered Routing Algorithm (TORA) [23], whose efficiency has been evaluated in practical mobile ad-hoc networks, and it has been shown to perform reasonably-well compared to other routing algorithms for such systems. This leads us to believe that our LE algorithm can also be implemented successfully in real mobile networks; together with the shortest-paths component we described in Section 4.7 of Chapter 4, it may provide even better paths to the leader/destination than TORA.

Bibliography

[1] Richard Bellman. On a routing problem. Technical report, DTIC Document, 1956.

[2] Jacob Brunekreef, Joost-Pieter Katoen, Ron Koymans, and Sjouke Mauw. Design and analysis of dynamic leader election protocols in broadcast networks. *Distributed Computing*, 9:157–171, 1996.

[3] Costas Busch, Srikanth Surapaneni, and Srikanta Tirthapura. Analysis of link reversal routing algorithms for mobile ad hoc networks. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 210–219, New York, NY, USA, 2003. ACM.

[4] Costas Busch and Srikanta Tirthapura. Analysis of link reversal routing algorithms. *SIAM J. Comput.*, 35(2):305–326, August 2005.

[5] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, October 1984.

[6] Bernadette Charron-Bost, Antoine Gaillard, Jennifer Welch, and Josef Widder. Routing without ordering. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 145–153, New York, NY, USA, 2009. ACM.

[7] Bernadette Charron-Bost, Jennifer Welch, and Josef Widder. Link reversal: How to play better to work less. In *Algorithmic Aspects of Wireless Sensor Networks*, volume 5804 of *Lecture Notes in Computer Science*, pages 88–101. Springer Berlin Heidelberg, 2009.

[8] Orhan Dagdeviren and Kayhan Erciyes. A hierarchical leader election protocol for mobile ad hoc networks. In *Computational Science ICCS 2008*, volume 5101 of *Lecture Notes in Computer Science*, pages 509–518. Springer Berlin Heidelberg, 2008.

[9] Ajoy K. Datta, Lawrence L. Larmore, and Hema Piniganti. Self-stabilizing leader election in dynamic networks. In *Stabilization, Safety, and Security of Distributed Systems*, volume 6366 of *Lecture Notes in Computer Science*, pages 35–49. Springer Berlin Heidelberg, 2010.

[10] Michael J. Demmer and Maurice P. Herlihy. The arrow distributed directory protocol. In *Distributed Computing*, volume 1499 of *Lecture Notes in Computer Science*, pages 119–133. Springer Berlin Heidelberg, 1998.

[11] Abdelouahid Derhab and Nadjib Badache. A self-stabilizing leader election algorithm in highly dynamic ad hoc mobile networks. *IEEE Trans. Parallel Distrib. Syst.*, 19(7):926–939, July 2008.

[12] Shlomi Dolev. *Self-stabilization.* MIT press, 2000.

[13] Eli M. Gafni and Dimitri P. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *Communications, IEEE Transactions on*, 29(1):11 - 18, January 1981.

[14] Kostas P. Hatzis, George P. Pentaris, Paul G. Spirakis, Vasilis T. Tampakas, and Richard B. Tan. Fundamental control algorithms in mobile networks. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pages 251–260, New York, NY, USA, 1999. ACM.

[15] Maurice Herlihy, Srikanta Tirthapura, and Roger Wattenhofer. Competitive concurrent distributed queuing. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 127–133, New York, NY, USA, 2001. ACM.

[16] Rebecca Ingram, Tsvetomira Radeva, Patrick Shields, Saira Viqar, Jennifer E. Walter, and Jennifer L. Welch. A leader election algorithm for dynamic networks with causal clocks. *Distributed Computing*, pages 1–23, 2013.

[17] Rebecca Ingram, Patrick Shields, Jennifer E. Walter, and Jennifer L. Welch. An asynchronous leader election algorithm for dynamic networks. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[18] Xavier Koegler. Around the tempo toolset userguide. Available at `http://groups.csail.mit.edu/tds/papers/Koegler/koegler-tempo.pdf`, 2007.

[19] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[20] Nancy Ann Lynch. *Distributed algorithms.* Morgan Kaufmann, 1996.

[21] Navneet Malpani, Jennifer L. Welch, and Nitin Vaidya. Leader election algorithms for mobile ad hoc networks. In *Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications*, pages 96–103, New York, NY, USA, 2000. ACM.

[22] Salahuddin Mohammad Masum, Amin Ahsan Ali, and Mohammad Touhid-youl Islam Bhuiyan. Asynchronous leader election in mobile ad hoc networks. In *Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, volume 2, pages 827–831, Washington, DC, USA, 2006. IEEE Computer Society.

[23] Vincent D. Park and M. Scott Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *Proceedings of the INFOCOM 1997. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution*, pages 1405–1413, Washington, DC, USA, 1997. IEEE Computer Society.

[24] Tsvetomira Radeva and Nancy Lynch. Partial reversal acyclicity. Technical report, Massachusetts Institute of Technology, CSAIL, 2011.

[25] Tsvetomira Radeva and Nancy A. Lynch. Partial reversal acyclicity. In *PODC*, pages 353–354, 2011.

[26] Muhammad Rahman, M. Abdullah-Al-Wadud, and Oksam Chae. Performance analysis of leader election algorithms in mobile ad hoc networks. *Intl J. of Computer Science and Network Security*, 8(2):257–263, 2008.

[27] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7(1):61–77, January 1989.

[28] Elizabeth M. Royer and Chai-Keong Toh. A review of current routing protocols for ad hoc mobile wireless networks. *Personal Communications, IEEE*, 6(2):46 –55, apr 1999.

[29] Jan L. A. Snepscheut. Fair mutual exclusion on a graph of processes. *Distributed Computing*, 2:113–115, 1987.

[30] Gerard Tel. *Introduction to distributed algorithms*. Cambridge university press, 2000.

[31] Sudarshan Vasudevan, Jim Kurose, and Don Towsley. Design and analysis of a leader election algorithm for mobile ad hoc networks. In *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*, pages 350–360. IEEE, 2004.

[32] Jennifer E. Walter, Guangtong Cao, and Mitrabhanu Mohanty. A k-mutual exclusion algorithm for wireless ad hoc networks. In *Proceedings of the First Annual Workshop on Principles of Mobile Computing.*, 2001.

[33] Jennifer E. Walter, Jennifer L. Welch, and Nitin H. Vaidya. A mutual exclusion algorithm for ad hoc mobile networks. *Wireless Networks*, 7(6):585–600, 2001.

[34] Jennifer L. Welch and Jennifer E. Walter. Link reversal algorithms. *Synthesis Lectures on Distributed Computing Theory*, 2(3):1–103, 2011.

[35] Shah-An Yang and John S. Baras. Tora, verification, proofs and model checking. In *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks*, 2003.