Hierarchical Leader Election Algorithm With Remoteness Constraint

Mohamed Tbarka

ENSIAS

University of Oxford

A thesis submitted for the degree of

Software Engineering

2019

This thesis is dedicated to
my grand father
for inspiring me.

Acknowledgements

I would like to thank the all library media specialists for their participation in the survey who supported my work in this way and helped me get results of better quality. I am also grateful to the members of my committee for their patience and support in overcoming numerous obstacles I have been facing through my research

I would like to thank my fellow doctoral students for their feedback, co-operation and of course friendship. In addition I would like to express my gratitude to the staff of INFRES - Telecom ParisTech for the last minute favors.

Nevertheless, I am also grateful to the Mr Petr Kuznetsov for sharing his dissertation woes, and a glimmer of hope for post-dissertation normalcy and Mrs Sanaa ElFkihi for his efficient supervising.

I would like to thank my friends for accepting nothing less than excellence from me. Last but not the least, I would like to thank my family: my parents and to my brothers and sister for supporting me spiritually throughout writing this thesis and my my life in general.

Abstract

A hierarchical algorithm for electing a leaders' hierarchy in an asynchronous network with dynamically changing communication topology is presented including a remoteness's constraint towards each leader. The algorithm ensures that, no matter what pattern of topology changes occur, if topology changes cease, then eventually every connected component contains a unique leaders' hierarchy. The algorithm combines ideas from the Temporally Ordered Routing Algorithm (TORA) for mobile ad hoc networks with a wave algorithm, all within the framework of a height-based mechanism for reversing the logical direction of communication links. Moreover, an improvement from the algorithm in is the introduction of logical clocks as the nodes' measure of time, instead of requiring them to have access to a common global time. This new feature makes the algorithm much more flexible and applicable to real situations, while still providing a correctness proof. It is also proved that in certain well behaved situations, a new leader is not elected unnecessarily.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Leader election is an important primitive for distributed computing, useful as a sub-routine for any application that requires the selection of a unique processor among multiple candidate processors. Applications that need a leader range from the primary-backup approach for replication-based fault-tolerance to group communication systems [26], and from video conferencing to multi-player games [11].

In a dynamic network, communication channels go up and down frequently. Causes for such communication volatility range from the changing position of nodes in mobile networks to failure and repair of point-to-point links in wired networks. Recent research has focused on porting some of the applications mentioned above to dynamic networks, including wireless and sensor networks. For instance, Wang and Wu propose a replication-based scheme for data delivery in mobile and fault-prone sensor networks [29]. Thus there is a need for leader election algorithms that work in dynamic networks.

We consider the problem of ensuring that, if changes to the communication topology cease, then eventually each connected component of the network has a unique leader (introduced as the "local leader election problem" in [7]). Our algorithm is an extension of the leader election algorithm in [18], which in turn is an extension of the MANET routing algorithm TORA in [22]. TORA itself is based on ideas from [9].

Gafni and Bertsekas [9] present two routing algorithms based on the notion of link reversal. The goal of each algorithm is to create directed paths in the communication topology graph from each node to a distinguished destination node. In these algorithms, each node maintains a height variable, drawn from a totally-ordered set; the (bidirectional) communication link between two nodes is considered to be directed from the endpoint with larger height to that with smaller height. Whenever a node becomes a sink, i.e., has no outgoing links, due to a link going down or due to notification of a neighbor's changed height, the node increases its height so that at least one of its incoming links becomes outgoing. In one of the algorithms of [9], the height is a pair consisting of a counter and the node's unique id, while in the other algorithm the height is a triple consisting of two counters and the node id. In both algorithms, heights are compared lexicographically with the least significant component being the node id. In the first algorithm, a sink increases its counter to be larger than the counter of all its neighbors, while in the second algorithm, a more complicated rule is employed for changing the counters.

The algorithms in [9] cause an infinite number of messages to be sent if a portion of the communication graph is disconnected from the destination. This drawback is overcome in TORA [22], through the addition of a clever mechanism by which nodes can identify that they have been partitioned from the destination. In this case, the nodes go into a quiescent state.

In TORA, each node maintains a 5-tuple of integers for its height, consisting of a 3-tuple called the reference level, a delta component, and the node's unique id. The height tuple of each node is lexicographically compared to the tuple of each neighbor to impose a logical direction on links (higher tuple toward lower.)

The purpose of the reference level is to indicate when nodes have lost their directed path to the destination. Initially, the reference level is all zeroes. When a node loses its last outgoing link due to a link going down the node starts a new reference level by changing the first component of the triple to the current time, the second to its own id, and the third to 0, indicating that a search for the destination is started. Reference levels are propagated throughout a connected component, as nodes lose outgoing links due to height changes, in a search for an alternate directed path to the destination. Propagation of reference levels is done using a mechanism by which a node increases its reference level when it becomes a sink; the delta value of the height is manipulated to ensure that links are oriented appropriately. If the search in one part of the graph is determined to have reached a dead end, then the third component of the reference level triple is set to 1. When this happens, the reference level is said to have been reflected, since it is subsequently propagated back toward the originator. If the originator receives reflected reference levels back from all its neighbors, then it has identified a partitioning from the destination.The purpose of the reference level is to indicate when nodes have lost their directed path to the destination. Initially, the reference level is all zeroes. When a node loses its last outgoing link due to a link going down the node starts a new reference level by changing the first component of the triple to the current time, the second to its own id, and the third to 0, indicating that a search for the destination is started. Reference levels are propagated throughout a connected component, as nodes lose outgoing links due to height changes, in a search for an alternate directed path to the destination. Propagation of reference levels is done using a mechanism by which a node increases its reference level when it becomes a sink; the delta value of the height is manipulated to ensure that links are oriented appropriately. If the search in one part of the graph is determined to have reached a dead end, then the third component of the reference level triple is set to 1. When this happens, the reference level is said to have been reflected, since it is subsequently propagated back toward the originator. If the originator receives reflected reference levels back from all its neighbors, then it has identified a partitioning from the destination.

The key observation in [18] is that TORA can be adapted for leader election: when a node detects that it has been partitioned from the old leader (the destination), then, instead of becoming quiescent, it elects itself. The information about the new leader is then propagated through the connected component. A sixth component was added to the height tuple of TORA to record the leader's id. The algorithm presented and analyzed in [18] makes several strong assumptions. First, it is assumed that only one topology change occurs at a time, and no change occurs until the system has finished reacting to the previous change. In fact, a scenario involving multiple topology changes can be constructed in which the algorithm is incorrect. Second, the system is assumed to be synchronous; in addition to nodes having perfect clocks, all messages have a fixed delay. Third, it is assumed that the two endpoints of a link going up or down are notified simultaneously of the change.

We present a modification to the algorithm that works in an asynchronous system with arbitrary topology changes that are not necessarily reported instantaneously to both endpoins of a link. One new feature of this algorithm is to add a seventh component to the height tuple of [18]: a timestamp associated with the leader id that records the time that the leader was elected. Also, a new rule by which nodes can choose new leaders is included. A newly elected leader initiates a "wave" algorithm [27]:

when different leader ids collide at a node, the one with the most recent timestamp is chosen as the winner and the newly adopted height is further propagated. This strategy for breaking ties between competing leaders makes the algorithm compact and elegant, as messages sent between nodes carry only the height information of the sending node, every message is identical in structure, and only one message type is used.

In this paper, we relax the requirement in [18] (and in [15]) that nodes have perfect clocks. Instead we use a more generic notion of time, a causal clock $\mathcal{T}$, to represent any type of clock whose values are non-negative real numbers and that preserves the causal relation between events. Both logical clocks [16] and perfect clocks are possible implementations of $\mathcal{T}$. We also relax the requirement in [18] (and in [15]) that the underlying neighbor-detection layer synchronize its notifications to the two endpoints of a (bidirectional) communication link throughout the execution; in the current paper, these notifications are only required to satisfy an eventual agreement property.

Finally, we provide a relatively brief, yet complete, proof of algorithm correctness. In addition to showing that each connected component eventually has a unique leader, it is shown that in certain well-behaved situations, a new leader is not elected unnecessarily; we identify a set of conditions under which the algorithm is "stable" in this sense. We also compare the difference in the stability guarantees provided by the perfect-clocks version of the algorithm and the causal-clocks version of the algorithm. The proofs handle arbitrary asynchrony in the message delays, while the proof in [18] was for the special case of synchronous communication rounds only and did not address the issue of stability.

Leader election has been extensively studied, both for static and dynamic networks, the latter category including mobile networks. Here we mention some representative papers on leader election in dynamic networks. Hatzis et al. [12] presented algorithms for leader election in mobile networks in which nodes are expected to control their movement in order to facilitate communication. This type of algorithm is not suitable for networks in which nodes can move arbitrarily. Vasudevan et al. [28] and Masum et al. [20] developed leader election algorithms for mobile networks with the goal of electing as leader the node with the highest priority according to some criterion. Both these algorithms are designed for the broadcast model. In contrast, our algorithm can elect any node as the leader, involves fewer types of messages than either of these two algorithms, and uses point-to-point communication rather than broadcasting. Brunekreef et al. [2] devised a leader election algorithm for a 1-hop wireless environment in which nodes can crash and recover. Our algorithm is suited to an arbitrary communication topology.

Several other leader election algorithms have been developed based on MANET routing algorithms. The algorithm in [23] is based on the Zone Routing Protocol [10]. A correctness proof is given, but only for the synchronous case assuming only one topology change. In [5], Derhab and Badache present a leader election algorithm for ad hoc wireless networks that, like ours, is based on the algorithms presented by Malpani et al. [18]. Unlike Derhab and Badache, we prove our algorithm is correct even when communication is asynchronous and multiple topology changes, including network partitions, occur during the leader election process.

Dagdeviren et al. [3] and Rahman et al. [24] have recently proposed leader election algorithms for mobile ad hoc networks; these algorithms have been evaluated solely through simulation, and lack correctness proofs. A different direction is randomized leader election algorithms for wireless networks (e.g., [1]); our algorithm is deterministic.

Fault-tolerant leader election algorithms have been proposed for wired networks. Representative examples are Mans and Santoro's algorithm for loop graphs subject to permanent communication failures [19], Singh's algorithm for complete graphs subject to intermittent communication failures [25], and Pan and Singh's algorithm [21] and Stoller's algorithm [26] that tolerate node crashes.

Recently, Datta et al. [4] presented a self-stabilizing leader election algorithm for the shared memory model with composite atomicity that satisfies stronger stability properties than our causal-clocks algorithm. In particular, their algorithm ensures that, if multiple topology changes occur simultaneously after the algorithm has stabilized, and then no further changes occur, (1) each node that ends up in a connected component with at least one pre-existing leader ultimately chooses a pre-existing leader, and (2) no node changes its leader more than once. The self-stabilizing nature of the algorithm suggests that it can be used in a dynamic network: once the last topology change has occurred, the algorithm starts to stabilize. Existing techniques (see, for instance, Section 4.2 in [6]) can be used to transform a self-stabilizing algorithm for the shared-memory composite-atomicity model into an equivalent algorithm for a (static) message-passing model, perhaps with some timing information. Such a sequence of transformations, though, produces a complicated algorithm and incurs time and space overhead (cf. [6,13]). One issue to be overcome in transforming an algorithm for the static message-passing model to the model in our paper is handling the synchrony that is relied upon in some component transformations to message passing (e.g., [14]).

Chapter 2

Preliminaries

2.1 System Model

We assume a system consisting of a set P of computing nodes and a set χ of directed communication channels from one node to another node. χ consists of one channel for each ordered pair of nodes, i.e., every possible channel is represented. The nodes are assumed to be completely reliable. The channels between nodes go up and down, due to the movement of the nodes. While a channel is up, the communication across it is in first-in-first-out order and is reliable but asynchronous (see below for more details).

We model the whole system as a set of (infinite) state machines that interact through shared events (a specialization of the IOA model [17]). Each node and each channel is modeled as a separate state machine. The events shared by a node and one of its outgoing channels are notifications that the channel is going up or going down and the sending of a message by the node over the channel; the channel up/down notifications are initiated by the channel and responded to by the node, while the message sends are initiated by the node and responded to by the channel. The events shared by a node and one of its incoming channels are notifications that a message is being delivered to the node from the channel; these events are initiated by the channel and responded to by the node.

2.2 Modeling Asynchronous Dynamic Links

We now specify in more detail how communication is assumed to occur over the dynamic links. The state of $Channel(u, v)$, which models the communication channel from node u to node v, consists of a statusuv variable and a queue *mqueueuv* of messages.

The possible values of the $status_{uv}$ variable are *Up* and *Down*. The channel transitions between the two values of its $status_{uv}$ variable through $ChannelUp_{uv}$ and $ChannelDown_{uv}$ events, called the "topology change" events. We assume that the *ChannelUp* and *ChannelDown* events for the channel alternate. The *ChannelUp* and *ChannelDown* events for the channel from u to v occur simultaneously at node u and the channel, but do not occur at node v.

The variable $mqueue_{uv}$ holds messages in transit from u to v. An attempt by node u to send a message to node v results in the message being appended to $mqueue_{uv}$ if the channel's status is *Up*; otherwise there is no effect. When the channel is *Up*, the message at the head of $mqueue_{uv}$ can be delivered to node v; when a message is delivered, it is removed from $mqueue_{uv}$. Thus, messages are delivered in FIFO order.

When a $ChannelDown_{uv}$ event occurs, $mqueue_{uv}$ is emptied. Neither u nor v is alerted to which messages in transit have been lost. Thus, the messages delivered to node v from node u during a (maximal-length) interval when the channel is *Up* form a prefix of the messages sent by node u to node v during that interval.

2.3 Configurations and Executions

The notion of configuration is used to capture an instantaneous snapshot of the state of the entire system. A configuration is a vector of node states, one for each node in $\mathcal{P}$, and a vector of channel states, one for each channel in χ . In an initial configuration: spacing

– each node is in an initial state (according to its algorithm),

– for each channel $Channel(u, v)$, $mqueue_{uv}$ is empty, and

– for all nodes u and v, $status_{uv} = status_{vu}$ (i.e., either both channels between u and v are up, or both are down).

Define an execution as an infinite sequence $C_0, e_1, C_1, e_2, C_2, ...$ of alternating configurations and events, starting with an initial configuration and, if finite, ending with a configuration such that the sequence satisfies the following conditions: spacing

- C_0 is an initial configuration.
- The preconditions for event ei are true in C_{i-1} for all $i \geq 1$.
- C_i is the result of executing event e_i on configuration C_{i-1} , for all $i \geq 1$ (only the node and channel involved in an event change state, and they change according to their state machine transitions).
- If a channel remains Up for infinitely long, then every message sent over the channel during this Up interval is eventually delivered.
- For all nodes u and v, Channel(u, v) experiences infinitely many topology change events if and only if Channel(v, u) experiences infinitely many topology change events; if they both experience finitely many, then after the last one, $status_{uv} = status_{vu}$.

Given a configuration of an execution, define an undirected graph G_{chan} as follows: the vertices are the nodes, and there is an (undirected) edge between vertices u and v if and only if at least one of $Channel_{uv}$ and $Channel_{vu}$ is Up. Thus G_{chan} indicates all pairs of nodes u and v such that either u can send messages to v or v can send messages to u. If the execution has a finite number of topology change events, then G_{chan} never changes after the last such event, and we denote the final version of final final G_{chan} as G_{chan} . By the last bullet point above, an edge in G_{chan} indicates bidirectional communication ability between the two endpoints.

We also assign a positive real-valued global time gt to each event e_i , $i \geq 1$, such that $gt(ei) < gt(e_{i+1})$ and, if the execution is infinite, the global times increase without bound. Each configuration inherits the global time of its preceding event, so $gt(C_i) = gt(e_i)$ for $i \geq 1$; we define $gt(C_0)$ to be 0. We assume that the nodes do not have access to gt.

Instead, each node u has a causal clock $\mathcal{T}_u$, which provides it with a non-negative real number at each event in an execution. $\mathcal{T}_u$ is a function from global time (i.e., positive reals) to causal clock times; given an execution, for convenience we sometimes use the notation $\mathcal{T}_u(e_i)$ or $\mathcal{T}_u(C_i)$ as shorthand for $\mathcal{T}_u(gt(e_i))$ or $\mathcal{T}_u(gt(C_i))$. The key idea of causal clocks is that if one event potentially can cause another event, then the clock value assigned to the first event is less than the clock value assigned to the second event. We use the notion of happens-before to capture the concept of potential causality. Recall that an event e_1 is defined to happen before [16] another event e_2 if one of the following conditions is true:

1. Both events happen at the same node, and e_1 occurs before e_2 in the execution.
2. e_1 is the send event of some message from node u to node v, and e_2 is the receive event of that message by node v.
3. There exists an event e such that e_1 happens before e and e happens before e_2.

The causal clocks at all the nodes, collectively denoted T , must satisfy the following properties: spacing

– For each node u, the values of $\mathcal{T}_u$ are increasing, i.e., if e_i and e_j are events involving u in the execution with $i < j$, then $\mathcal{T}_u(e_i) < \mathcal{T}_u(e_j)$. In particular, if there is an infinite number of events involving u, then $\mathcal{T}_u$ increases without bound.

– $\mathcal{T}$ preserves the happens-before relation [16] on events; i.e., if event e_i happens before event e_j, then $\mathcal{T}(e_i) < \mathcal{T}(e_j)$.

Our description and proof of the algorithm assume that nodes have access to causal clocks. One way to implement causal clocks is to use perfect clocks, which ensure that $\mathcal{T}_u(t) = t$ for each node u and global time t. Since an event that causes another event must occur before it in real time, perfect clocks capture causality. Perfect clocks could be provided by, say a GPS service, and were assumed in the preliminary version of this paper [15]. Another way to implement causal clocks is to use Lamport's logical clocks [16], which were specifically designed to capture causality.

2.4 Problem Definition

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

In every execution that includes a finite number of topology change events, we require that the following eventually holds: Every connected component CC of the final final topology graph G_chan contains a node l, the leader, such that $lid_u = l$ for all nodes $u \in CC$, including l itself.

Chapter 3

Leader Election Algorithm

In this section, we present our leader election algorithm. The pseudocode for the algorithm is presented in Figures 1, 2 and 3. First, we provide an informal description of the algorithm, then, we present the details of the algorithm and the pseudocode, and finally, we provide an example execution. In the rest of this section, variable var of node u will be indicated as varu . For brevity, in the pseudocode for node u, variable varu is denoted by just var.

3.1 Informal Description

Each node in the system has a 7-tuple of integers called a height. The directions of the edges in the graph are determined by comparing the heights of neighboring nodes: an edge is directed from a node with a larger height to a node with a smaller height. Due to topology changes nodes may lose some of their incident links, or get new ones throughout the execution. Whenever a node loses its last outgoing link because of a topology change, it has no path to the current leader, so it reverses all of its incident edges. Reversing all incident edges acts as the start of a search mechanism (called a reference level) for the current leader. Each node that receives the newly started reference level 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 current 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 which 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. Finally, this new leader information propagates throughout the network via an extra "wave" of propagation of messages.

In our algorithm, 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 in [15], these timestamps are obtained from a global clock accessible to all nodes in the system. In this paper, we use the notion of causal clocks (defined in Section 2.3) instead.

One difficulty that arises in solving leader election 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 different heights of the two leaders are being propagated in the new connected component, eventually, some node needs to compare both and decide which one to adopt and continue propagating. Recall that when a new leader is elected, a component of the height of the leader records the time of the election which can be used to determine the more recent of two elections. Therefore, when a node receives a height with a different leader information from its own, it adopts the one corresponding to the more recent election.

Similarly, if two reference levels are being propagated in the same connected component, whenever a node receives a height with a reference level different from its current one, it adopts the reference level with the more recent timestamp and continues propagating it. Therefore, even though conflicting information may be propagating in the same connected component, eventually the algorithm ensures that as long as topology changes stop, each connected component has a unique leader.

3.2 Nodes, Neighbors and Heights

First, we describe the mechanism through which nodes get to know their neighbors. Each node in the algorithm keeps a directed approximation of its neighborhood in Gchan as follows. When u gets a ChannelUp event for the channel from u to v, it puts v in a local set variable called formingu . When u subsequently receives a message from v, it moves v from its formingu set to a local set variable called Nu (N for neighbor). If u gets a message from a node which is neither in its forming set, nor in Nu , it ignores that message. And when u gets a ChannelDown event for the channel from u to v, it removes v from formingu or Nu , as appropriate. For the purposes of the algorithm, u considers as its neighbors only those nodes in Nu . It is possible for two nodes u and v to have inconsistent views concerning whether u and v are neighbors of each other. We will refer to the ordered pair (u, v), where v is in Nu , as a link of node u.

Nodes assign virtual directions to their links using variables called heights. Each node maintains a height for itself, which can change over time, and sends its height over all outgoing channels at various points in the execution. Each node keeps track of the heights it has received in messages. For each link (u, v) of node u, u considers the link as incoming (directed from v to u) if the height that u has recorded for v is larger than u's own height; otherwise u considers the link as outgoing (directed from u to v). Heights are compared using lexicographic ordering; the definition of height ensures that two nodes never have the same height. Note that, even if v is viewed as a neighbor of u and vice versa, u and v might assign opposite directions to their corresponding links, due to asynchrony in message delays.

Next, we examine the structure of a node's height in more detail. The height for each node is a 7-tuple of integers $((\tau, oid, r), \delta, (nlts, lid), id)$, where the first three components are referred to as the reference level (RL) and the fifth and sixth components are referred to as the leader pair (LP). In more detail, the components are defined as follows: spacing

- τ , a non-negative timestamp which is either 0 or the value of the causal clock time when the current search for an alternate path to the leader was initiated.
- oid, is a non-negative value that is either 0 or the id of the node that started the current search (we assume node ids are positive integers).
- r, a bit that is set to 0 when the current search is initiated and set to 1 when the current search hits a dead end.
- δ , an integer that is set to ensure that links are directed appropriately to neighbors with the same first three components. During the execution of the algorithm δ serves multiple purposes. When the algorithm is in the stage of

searching for the leader (having either reflected or unreflected RL), the δ value ensures that as a node u adopts the new reference level from a node v, the direction of the edge between them is from v to u; in other words it coincides with the direction of the search propagation. Therefore, u adopts 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 from node v, it adopts the entire height of v and sets the δ value to one more than v's. $--$ $nlts$, a non-positive timestamp whose absolute value is the causal clock time when the current leader was elected. –*lid*, the *id* of the current leader. $--$ id, the node's unique ID.

Each node u keeps track of the heights of its neighbors in an array $height_u$, where the height of a neighbor node v is stored in $height_u[v]$. The components of $height_u[v]$ are referred to as $(\tau^v, oid^v, r^v, \delta^v, nlts^v, lid^v, v)$ in the pseudocode.

3.3 Initial States

The definition of an initial configuration for the entire system from Section 2.3 included the condition that each node be in an initial state according to its algorithm. The collection of initial states for the nodes must be consistent with the collection of initial states for the channels. Let G_{init} chan be the undirected graph corresponding to the initial states of the channels, as defined in Section 2.3. Then in an initial configuration, the state of each node u must satisfy the following: spacing

– formingu is empty, – Nu equals the set of neighbors of u in Ginit chan

– heightu[u] $= (0, 0, 0, \delta_u, 0, l, u)$ where l is the id of a fixed node in $u's$ connected chan (the current leader), and δ_u equals the distance from u to l in component in Ginit Ginit chan ,

– for each v in Nu , heightu[v] = heightv [v] (i.e., u has accurate information about v's height), and

– Tu is initialized properly with respect to the definition of causal clocks.

The constraints on the initial configuration just given imply that initially, each connected component of the communication topology graph has a leader; furthermore, by following the virtual directions on the links, nodes can easily forward information to the leader (as in TORA). One way of viewing our algorithm is that it maintains leaders in the network in the presence of arbitrary topology changes. In order to establish this property, the same algorithm can be executed, with each node initially being in a singleton connected component of the topology graph prior to any ChannelUp or ChannelDown events.

3.4 Goal of the Algorithm

The goal of the algorithm is to ensure that, once topology changes cease, eventually each connected component of G_{chan}^{final} is "leader-oriented", which we now define. Let CC be any connected component of G_{chan}^{final} . First, we define a directed version of CC, denoted $CC^{\longleftrightarrow}$, in which each undirected edge of CC is directed from the endpoint with larger height to the endpoint with smaller height. We say that CC is leader-oriented if the following conditions hold:

1. No messages are in transit in CC.
2. For each (undirected) edge u, v in CC, if (u, v) is a link of u, then u has the correct view of v's height.
3. Each node in CC has the same leader id, say l, where l is also in $CC^{\longrightarrow}$.
4. CC is a directed acyclic graph (DAG) with l as the unique sink.

A consequence of each connected component being leader-oriented is that the leader election problem is solved.

3.5 Description of the Algorithm

The algorithm consists of three different actions, one for each of the possible events that can occur in the system: a channel going up, a channel going down, and the receipt of a message from another node. Next, we describe each of these actions in detail.

First, we formally define the conditions under which a node is considered to be a sink: spacing

– SINK = $((\forall v \in N_u, LP_u^v = LP_u^u)$ and $(\forall v \in N_u, height_u[u] < height_u[v])$ and $(lid_u^u = u))$. Recall that the LP component of node $u's$ view of $v's$ height, as stored in $u's$ height array, is denoted LP_u^v , and similarly for all the other height components. This predicate is true when, according to $u's$ local state, all of $u's$ neighbors have the same leader pair as u, u has no outgoing links, and u is not its own leader. If node u has links to any neighbors with different LPs, u is not considered a sink, regardless of the directions of those links.

$ChannelDown$ event: When a node u receives a notification that one of its incident channels has gone down, it needs to check whether it still has a path to the current leader. If the $ChannelDown$ event has caused u to lose its last neighbor, as indicated by u's N variable, then u elects itself by calling the subroutine $ELECTSELF$. In this subroutine, node u sets its first four components to 0, and the LP component to $(nlts, u)$ where $nlts$ is the negative value of u's current causal clock time. Then, in case u has any incident channels that are in the process of forming, u sends its new height over them. If the $ChannelDown$ event has not robbed u of all its neighbors (as indicated by $u'sN$ variable) but u has lost its last outgoing link, i.e., it passes the SINK test, then u starts a new reference level (a search for the leader) by setting its τ value to the current clock time, oid to u's id, the r bit to 0, and the δ value to 0, as shown in subroutine $STARTNEWREFLEVEL$. The complete pseudo-code for the $ChannelDown$ action is available in Figure 1.

$ChannelUp$ 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 set $forming_u$. The pseudo-code for the $ChannelUp$ action is available in Figure 1.

Algorithm 1 When $ChannelDown_{uv}$ event occurs:

```
1: $N := N \setminus v$
2: $forming := forming \setminus v$
3: **if** $N = \emptyset$ **then**
4:     $ELECTSELF$
5:     send $Update(height[u] to all w \in forming)$
6: **else if** $SINK$ **then**
7:     $STARTNEWREFLEVEL$
8:     send Update($height[u]$) to all $w \in (N \cup forming)$)
9: **end if**
```

Algorithm 2 When $ChannelUp_{uv}$ event occurs:

```
1: $forming := forming \cup v$
2: send Update($height[u]$) to $v$
```

Receipt of an update message: When a node u receives a message from another node v, containing v's height, node u performs the following sequence of rules (shown in Figure 2).

First, if v is in neither $forming_u$ nor N_u , then the message is ignored. If $v \in forming_u$ but $v \ni N_u$ then v is moved to N_u. Next, u checks whether v has the same leader pair as u. If v knows about a more recent leader than u, node u adopts that new LP (shown in subroutine $ADOPTLPIFPRIORITY$ in Figure 3). If the LP's of u and v are the same, then u checks whether it is a sink using the definition above. If it is not a sink, it does not perform any further action (because it already has a path to the leader). Otherwise, if u is a sink, it checks the value of the RL component of all of its neighbors' heights (including v's). If some neighbor of u, say w, knows of a RL which is more recent than u's, then u adopts that new RL by setting the RL part of its height to the new RL value and changing the δ component to one less than the δ component of w. Therefore, the change in u's height does not cause w to become a sink (again) and so the search for the leader does not go back to w and it is thus prop-agated in the rest of the connected component. The details are shown in subroutine PROPAGATE L ARGEST R EF L EVEL in Figure 3.

If u and all of its neighbors have the same RL component of their heights, say (τ, oid, r), we consider three possible cases:

1. If $\tau > 0$ (indicating that this is a RL started by some node, and not the default value 0) and r = 0 (the RL has not reached a dead end), then this is an indication of a dead end because u and all of its neighbors have the same unreflected RL. In this case u changes its height by setting the r component of its height to 1 (shown in subroutine REFLECT R EF L EVEL in Figure 3).

2. If $\tau > 0$ (indicating that this is a RL started by some node, and not the default value 0), r = 1 (the RL has already reached a dead end) and oid = u (u started the current RL), then this is an indication that the current leader may not be in the same connected component anymore. In other words, all the branches of the RL started by u reached dead ends. Therefore, u elects itself as the new leader by setting its first 4 components to 0, and the LP component to (nlts, u) where nlts is the negative value of u's current causal clock time (shown in subroutine ELECT S ELF in Figure 3). Note that this case does not guarantee that the old leader is not in the connected component, because some recent

topology change may have reconnected it back to u's component. We already described how the leader information of two different leaders is handled.

3. If neither of the two conditions above are satisfied, then it is the case that either $\tau = 0$ or $\tau > 0$, r = 1 and oid 6= u. In other words, all of u's neighbors have a different reflected RL or contain an RL indicating that various topology changes have interfered with the proper propagation of RL's, and so node u starts a fresh RL by setting τ to the current causal clock time, oid to u's id, the r bit to 0, and the δ value to 0 (shown in subroutine STARTNEWREFLEVEL in Figure 3).

Finally, whenever a node changes its height, it sends a message with its new height to all of its neighbors. Additionally, whenever a node u receives a message from a node v indicating that v has different leader information from u, then either if u adopts v's LP or not, u sends an update message to v with its new (possibly same as old) height. This step is required due to the weak level of coordination in neighbor discovery.

3.6 Sample execution

Next, we provide an example which illustrates a particular algorithm execution. Figure 4, parts (a)-(h), show the main stages of the execution. In the picture for each stage, a message in transit over a channel is indicated by a light grey arrow. The recipient of the message has not yet taken a step and so, in its view, the link is not yet reversed.

a this is item a

b another item

Chapter 4

Conclusion

We have described and proved correct a leader election algorithm using logical clocks for asynchronous dynamic networks. A set of circumstances were identified under which the algorithm does not elect a leader unnecessarily, but it remains to give a more complete characterization of such circumstances. Also, the time and message complexity of the algorithm needs to be analyzed. It would be interest- ing to compare the efficiency of the algorithm in the cases of perfect clocks and logical clocks.

Bibliography