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, cooperation 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

Chapter 1

Introduction

Leader election is an important primitive for distributed computing, useful as a subroutine for any application that requires the selection of a unique processor among multiple candidate processors. Applications that need a leader range from the primary-backup approach 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 mo- bile 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 dy- namic networks, including wireless and sensor networks. For instance, Wang and Wu propose a replication-based scheme for data delivery in mobile and fault-prone sen- sor 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 topol- ogy 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 di- rected 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 fin- ished 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 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 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 correct- ness. 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 algo- rithm. 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 net- works, the latter category including mobile networks. Here we mention some repre- sentative 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 elec- tion algorithms for mobile ad hoc networks; these algorithms have been evaluated solely through simulation, and lack correctness proofs. A different direction is ran- domized 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 stabil- ity properties than our causal-clocks algorithm. In particular, their algorithm ensures that, if multiple topology changes occur simultaneously after the algorithm has sta- bilized, 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 topol- ogy 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 se- quence of transformations, though, produces a complicated algorithm and incurs time and space overhead (cf. [6,13]). One issue to be overcome in transforming an algo- rithm 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]).

Algorithm 1 Put your caption here

```
1: System Initialization
2: Read the value
3: **if** $condition = True$ **then**
4:     Do this
5:     **if** $Condition \geq 1$ **then**
6:         Do that
7:     **else if** $Condition \neq 5$ **then**
8:         Do another
9:         Do that as well
10:    **else**
11:        Do otherwise
12:    **end if**
13: **end if**
14: **while** $something \neq 0$ **do**                ▷ put some comments here
15:     $var1 \leftarrow var2$                         ▷ another comment
16:     $var3 \leftarrow var4$
17: **end while**
```

Chapter 2

Preliminaries

2.1 System Model

We assume a system consisting of a set P of computing nodes and a set L of bidirectional communication links between nodes. L consists of one link for each unordered pair of nodes, i.e., every possible link is represented. The nodes are assumed to be completely reliable. The links between nodes go up and down, due to the movement of the nodes. While a link is up, the communication across it is in first-in-first-out order and is reliable but asynchronous.

We model the whole system as a set of (infinite) state machines that interact through shared events (a specialization of the IOA model [12]). Each node and each link is modeled as a separate state machine. The shared events are Link Up/Down notifications and receipt of messages, all of which are controlled and initiated by the link and responded to by the node. The sending of a message is also a shared event, but it is controlled and initiated by the node and responded to by the link; we are not explicitly modeling this.

The next subsection gives more details about how links are modeled and specifies the initial states. The algorithm executed by the nodes and its initial states are described in Section 3.

2.2 Modeling Asynchronous Dynamic Links

We now specify how communication is assumed to occur over the dynamic links, and how notification of a link's status is synchronized at the two endpoints of the link.

The state of a link Linku, v, which models the bidirectional communication link between node u and node v, consists of a status variable and two queues of messages.

The possible values of the status variable are Up, GoingDown u , GoingDown v , Down, ComingUp u , and ComingUp v . The link transitions among different values of its status variable through LinkUp and LinkDown events. Figure 1 shows the state transition diagram for Linku, v. The intuition is that if a LinkUp (resp., LinkDown) occurs at one endpoint of the link, then LinkUp (resp., LinkDown) must occur at the other endpoint before LinkDown (resp., LinkUp) can occur at either end.

The other components of the link's local state are the two message queues: mqueue u,v holds messages in transit from u to v and mqueue v,u holds messages in transit from v to u.

An attempt by node u to send a message to node v results in the message being appended to mqueue u,v if the link's status is either ComingUp u or Up; otherwise there is no effect.

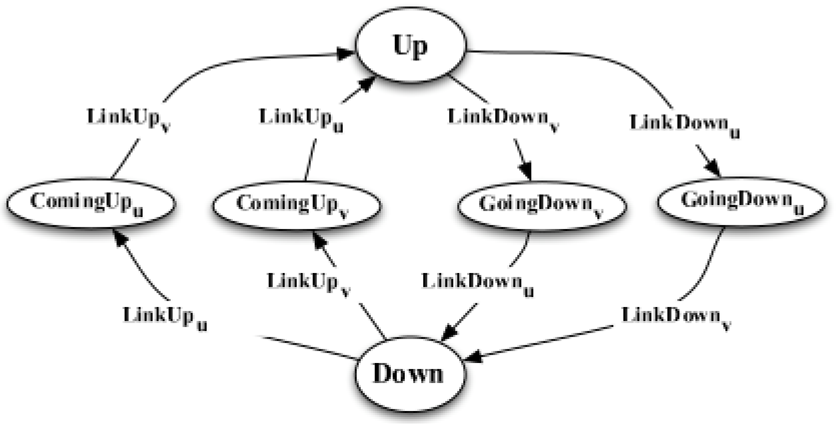

Figure 2.1: State diagram for status variable of $Link\{u, v\}$.

If the status is ComingUp u , then messages in transit from u to v are held in the queue until v has been notified that the link is Up. Once the link is Up, the event by which node u receives the message at the head of mqueue v,u is enabled to occur. An attempt by node v to send a message to node u is handled analogously.

Whenever a LinkDown u or LinkDown v event occurs, both message queues are emptied. Neither u nor v is alerted to which messages in transit have been lost due to the LinkDown.

In an initial state of the link, both message queues are empty and the status is either Up or Down.

2.3 Configurations and Executions

The notion of configuration is used to capture an instan- taneous snapshot of the state of the entire system. A config- uration is a vector of node states, one for each node in P, and a vector of link states, one for each link in L . Assume that the undirected graph G = (V, E) defines the initial communication topology of the system, where V is a set of vertices corresponding to the set P of nodes, and E is a set of edges corresponding to the set of communication links that are up. In an initial configuration with respect to G, each node is in an initial state (as prescribed by the node's algorithm), each link corresponding to an edge in E is in an initial state with its status equal to Up, and every other link has its status equal to Down. Define an execution as an infinite sequence 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, that satisfies the following safety conditions:

- C 0 is an initial configuration (w.r.t. some initial topology G).
- The preconditions for event 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 $i >= 1$ (only the node and link involved in an event change state, and they change according to their state machine transitions).

An execution also satisfies the following liveness condi- tions:

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

We also assign a positive real-valued global time gt to each event e i , i 1, such that gt(e i) ¡ 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 1; we define gt(C 0) to be 0. We assume that the nodes do not have access to gt.

2.4 Problem Definition

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

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

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

Chapter 3

Leader Election Algorithm

3.1 System Model

In this section, we explain the local variables used in our leader election algorithm. The pseudocode for the algorithm is presented in Figures 3.1, 3.2 and 3.3. An overview and sample execution is given in Section 3.1. In the analysis, variable v of node i will be indicated as v i.

Each node i keeps an array of heights, height i , with an entry for itself and for each of its neighbors, in which it stores the most recent height information that it has received for those nodes.

Each height is a 7-tuple, with the following components:

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

Components (, oid, r) are referred to as the reference level, or RL; (, oid) alone are referred to as the reference level prefix; and (nlts, lid) is referred to as the leader pair or LP. The components of entry k in height i are referred to as (k , oid k , r k , k , nlts k , lid k , k) in the pseudocode.

Nodes communicate over links during the algorithm ex- ecution by sending Update messages. Each message con- tains the height tuple and the logical clock timestamp of the sending node. The link between node i and one of its neigh- boring nodes j is considered by i to be outgoing (directed from i to j) if and only if height i [i] ¿ height i [j]. That is, node i uses the information in its local state concerning it- self and node j to determine (its view of) the direction of the link to j. Because of message delays, it is not necessarily the case that i and j have consistent views of the direction of the link between them.

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

Nodes have no access to global time, but, instead, each of them has a local logical clock [11]. A logical clock is a non-negative integer, initially 0. Logical clocks can be updated in two possible ways, depending on the type of the event occurring at a node. If a LinkUp or a LinkDown event occurs at a node, its logical clock is incremented by 1. Oth- erwise, if a node receives an Update message, the logical clock is set to one more than the maximum of the node's current logical clock value and the timestamp included in the message. This way, we ensure that for each node, the value of its logical clock is non-decreasing for each subse- quent event. The logical clock of a node is referred to as LC in the pseudocode.

Given an initial connected communication graph G = (V, E), with V corresponding to the set of nodes and E to the set of communication links that are up, the initial state of each node i is defined as follows 1 .

1. forming i is empty

2. N i contains the id of every node j such that the vertices in V corresponding to i and j are neighbors in G

3. height i [i] = (0, 0, 0, i , 0, l, i), where l is the id of a fixed node in i's connected component, the current leader

4. for each neighbor j of i, height i [j] = height j [j] (i.e., i has accurate information about j's height)

5. LC i = 0 (i.e. the logical clocks of all nodes are initially set to 0).

Furthermore, for each node i, i equals the distance from i to l; this condition ensures that every node has a directed path to l. Next we define the conditions under which a node con- siders itself to be a sink.

- SINK = ((LP i j = LP ii j N i) and (height i [i] ¡ minheight i [j] j N i) and (lid ii , i)). This pred- icate is true when, according to i's local state, i is not a leader, has all neighbors with the same LP, and has no outgoing links. If node i has links to any neighbors with different LPs, i is not considered a sink, regard- less of the directions of those links.

3.2 Overview of Algorithm

We depict the network as a DAG in which each bidirec- tional communication link points from a node with lexico- graphically higher height to another node with lexicograph- ically lower height. Nodes send algorithm messages only when they change the contents of their height tuple. The contents of the height tuple at a particular node are changed only when the node elects itself a leader, when it changes its current leader, or when it loses its last outgoing link to its current leader. The network is quiescent when there is no message in transit on any link. Messages that do not cause a node to lose its last outgoing link to its current leader or to change its current leader result only in a change to the internal data that node keeps about its neighbors' heights. Figure 3.4 shows a sample execution of the algorithm. Each part (a)–(h) is discussed below.

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

b) When non-leader node G loses its last outgoing link due to the loss of the link to node H, G increments its logical clock by 1, executes subroutine START - N EW R EF L EVEL and takes on RL (1,G,0) and = 0. Then node G sends messages with its new height to all its neighbors. By raising its height in this way, G has started a search for leader H.

c) Nodes D, E, and F receive the messages sent from node G, messages that cause each of these nodes to take on RL (1,G,0), set its to 1, ensuring that its height is lower than G's but higher than the other neighbors'. Moreover, each of these nodes also updates its logical clock to 2. Then D, E and F send messages to their neighbors.

d) Node B has received messages from both E and D with the new RL (1,G,0), and C has received a message from F with RL (1,G,0); as a result, B and C take on RL (1,G,0) with set to 2, update their logical clocks to 3 and send messages. Additionally, as a result of the messages sent by D, E, and F, node G updates its logi- cal clock to 3.

e) Node A has received message from both nodes B and C. In this situation, node A is connected only to nodes that are participating in the search started by node G for leader H. In this case, node A "reflects" the search by setting the reflection bit in the (1,G,*) reference level to 1, resetting its to 0, and sending its new height to its neighbors. Moreover, nodes A, D, E, and F update their clocks to 4.

f) Nodes B and C take on the reflected reference level (1,G,1) and set their to 1, causing their heights to be lower than A's and higher than their other neighbors'. They also update their logical clocks to 5 and send their new heights to their neighbors.

g) Nodes D, E, and F act similarly as B and C did in part (f), but set their variables to 2. As a result from the messages sent by B and C, nodes A, D, E, and F update their logical clocks to 6.

h) When node G receives the reflected reference level from all its neighbors, it knows that its search for H is in vain. G updates its logical clock to 7 and then elects itself. The new LP (-7,G) then propagates through the component, updating nodes' logical clocks along the way, assuming no further link changes occur; eventu- ally each node has RL (0,0,0) and LP (-7,G), with D, E and F having $= 1$, B and C having $= 2$, and A having $= 3$.

When $ChannelDown_u v$ **event occurs**

```
1.  $N := N \backslash \{v\}$
2.  $forming := forming \backslash \{v\}$
3.  **if** $(N = \emptyset)$
4.      $ELECTSELF$
5.      send Update(heigth[u]) to all $w \in forming$
6.  **else if**(SINK)
7.      $STARTNEWREFLEVEL$
8.      send Update(heigth[u]) to all $w \in (N \cup forming)$
9.  **else if** $(j = pred_i)$
10.     $pred_i = min\{k \;\; | \;\; k \in N_i \;\; and \;\; \delta_k = \delta_j\}$
11. **end if**
```

`../../Pictures/algorithm_2.png`

Figure 3.1: Code triggered by link changes.

```
ELECTSELF
1.   $height[i] := (0,0,0,0,-LC_i, i, i)$

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

PROPAGATELARGESTREFLEVEL
1.   $(\tau^i, oid^i, r^i) := max\{(\tau^k, oid^k, r^k) | \ k \in N\}$
2.   $\delta^i := min\{ \ \delta^k \ | \ k \in N$ and $(\tau^i, oid^i, r^i) = (\tau^k, oid^k, r^k)\} - 1$

STARTNEWREFLEVEL
1.   $height[i] := (LC_i, i, 0, 0, nlts^i, lid^i, i)$

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

Figure 3.2: Subroutines

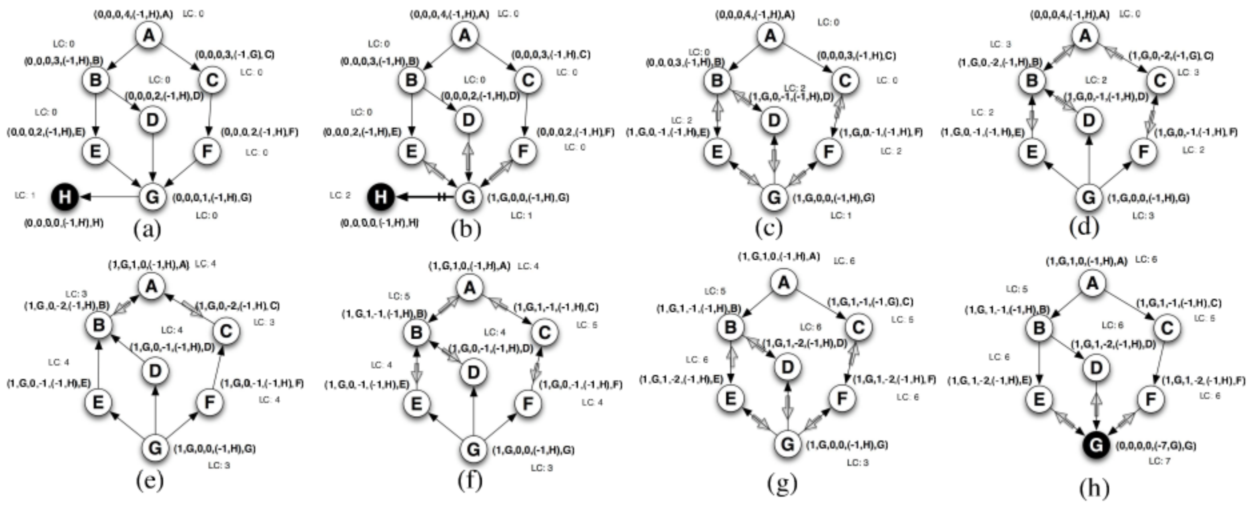

Figure 3.3: Simple execution when leader H becomes disconnected (a), with time increasing from (a)– (h). With no other link changes, every node in the connected component will eventually adopt G as its leader.

Chapter 4

Correctness

In this section, we show that, once topology changes cease, the algorithm eventually terminates with each con- nected component forming a leader-oriented DAG. First, we make some definitions regarding the information concerning nodes' heights that exists in the system and prove some properties about it. Then we prove that, after the last topology change, each node elects itself a finite number of times and a finite number of new reference levels are started. As a result, we show that eventually no messages are in transit and at that point we have a leader-oriented DAG. Throughout the proof, consider an arbitrary execution of the algorithm in which the last topology change occurs at some time t_{LTC} , and consider any connected component of the final topology.

4.1 Height Tokens and Their Properties

Property A : If h is a height token for a node v in the (v, u) height sequence, then: (i) nlts(h) LC v and (h) LC v (ii) If h is in transit, then nlts(h) T and (h) T where T is the logical timestamp on the Update mes- sage from v to u containing h. (iii) If h is in u's height array then nlts(h) LC u and (h) LC u .

Property B: Let h 0 , h 1 , . . . , h m be the (u, v) height sequence for any Linku, v whose status is Up or ComingUp u . Then the following are true: (1) h 0 = h 1 . (2) For all l, 0 l ¡ m, LP(h l) LP(h l+1). (3) For all l, 0 l ¡ m, if LP(h l) = LP(h l+1), then RL(h l) RL(h l+1).

4.2 Bounding the Number of Elections

Property C: For each height token h with RL (t, p, r), either t = p = r = 0, or t ¿ 0, p is a node id, and r is 0 or 1.

Property D: Let h be a height token for some node u. If LP(h) = (s, l), where LC l (t) = s at time t and t t LTC , then RL(h) = (0, 0, 0) and (h) is the distance in LT (s, l) from l to u.

Lemma 1: Any node u that adopts leader pair (s, l) for any l and any s, where LC l (t) = s and t ¿ t LTC , never sub- sequently becomes a sink.

Lemma 2: No node elects itself more than a finite number of times after t LTC.

4.3 Bounding the Number of New Reference Levels

Property E: If h and h are two height tokens for the same node u with RL(h) = RL(h) and LP(h) = LP(h), then (h) = (h). **Property F:** If there is a height token for node u with RL prefix (t, p), where LC u (t) = t and t   t LTC , then u is in RD(t, p). **Property G:** If there is a height token for node u with RL (t, p, 1), where LC u (t) = t and t   t LTC , then all neighbors of u are in RD(t, p). **Property H:** Suppose node u has height h u , neighboring node v has height h v , and u's view of v's height is h v , all with the same LP. If h u ¡ h v , then h v = h v . Property I: Consider two height tokens, h u for a node u with RL(h u) = (t, p, r u) and (h u) = d u , and h v for a neigh- boring node v with RL(h v) = (t, p, r v) and (h v) = d v , where LC p (t) = t and t   t LTC . Then the following are true: (1) r u r v if and only if u is a predecessor of v in RD(t, p). (2) If r u = r v = 0, then d u ¿ d v if and only if u is a prede- cessor of v. (3) If r u = r v = 1, then d v ¿ d u if and only if u is a prede- cessor of v. Lemma 3 Every node starts a finite number of new RLs after t LTC .

4.4 Bounding the Number of Messages

Lemma 4: Eventually every node in the connected compo- nent has the same leader pair.

Lemma 5: Eventually there are no messages in transit.

Lemma 6: Eventually every node has an accurate view of its neighbors' heights.

4.5 Leader-Oriented DAG

Property J: A node is never a sink in its own view. **Property K:** Consider any height token h for node u. If RL(h) = (0, 0, 0), then (h) 0. Furthermore, (h) = 0 if and only if u is a leader.

Theorem 7 Eventually the connected component is a leader-oriented DAG.

Chapter 5

Implementation

5.1 The Tool used

JBotSim is an open source simulation library that is dedicated to distributed algorithms in dynamic networks. I developed it with the purpose in mind to make it possible to implement an algorith- mic idea in minutes and interact with it while it is running (e.g., add, move, or delete nodes). Besides interaction, JB OT S IM can also be used to prepare live demos of an algorithm and to show it to colleagues or students, as well as to assess the algorithm per- formance. JB OT S IM is not a competitor of mainstream simulators such as NS3 [5], OMNet [10], or The One [8], in the sense that it does not aim to implement real-world networking protocols. Quite the opposite, JB OT S IM aims to remain technology-insensitive and to be used at the algorithmic level, in a way closer in spirit to the ViSiDiA project (a general-purpose platform for distributed algo- rithms). Unlike ViSiDiA, however, JB OT S IM natively supports mobility and dynamic networks (as well as wireless communica- tion). Another major difference with the above tools is that it is a library rather than a software: its purpose is to be used in other pro- grams, whether these programs are simple scenarios of full-fledged software. Finally, JB OT S IM is distributed under the terms of the LGPL licence, which makes it easily extensible by the community. Whether the algorithms are centralized or distributed, the natu- ral way of programming in JB OT S IM is event-driven: algorithms are specified as subroutines to be executed when particular events occur (appearance or disappearance of a link, arrival of a message, clock pulse, etc.). Movements of the nodes can be controlled ei- ther by program or by means of live interaction with the mouse (adding, deleting, or moving nodes around with left-click, right- click, or drag and drop, respectively). These movements are typi- cally performed while the algorithm is running, in order to visualize it or test its behavior in challenging configurations. The present document offers a broad view of

JB OT S IM 's main features and design traits. I start with preliminary information in Section 2 regarding installation and documentation. Section 3 re- views JB OT S IM 's main components and specificities such as pro- gramming paradigms, clock scheduling, user interaction, or global architecture. Section 4 zooms on key features such as the exchange of messages between nodes, graph-level APIs, or the creation of online demos. Finally, I discuss in Section 5 some extensions of JB OT S IM , includ- ing TikZ exportation feature and edge-markovian dynamic graph generator. Besides its features, the main asset of JBotSim is its simplicity of use – an aim that I pursued at the cost of re-writing it several times from scratch (the API is now stable).

5.2 Features and Architecture

This section provides an overview of JB OT S IM 's key features and discusses the reason why some design choices were made. I review topics as varied as programming paradigms, clock schedul- ing, user interaction, and global architecture.

5.2.1 Basic features of nodes and links

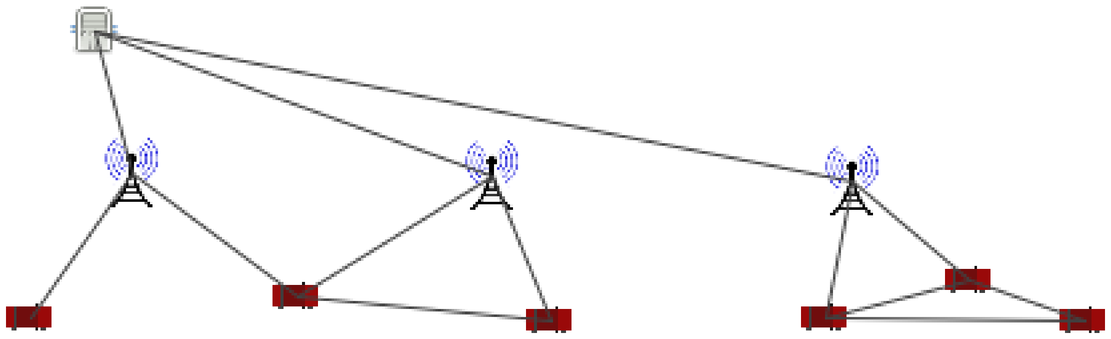

Figure 5.1: A highway scenario composed of vehicles, road-side units, and central servers. Part of the network is ad hoc (and wireless); the rest is infrastructured (and wired)

JB OT S IM consists of a small number of classes, the most cen- tral being Node, Link, and Topology. The contexts in which dynamic networks apply are varied. In order to accommodate a majority of cases, these classes offer a number of conceptual varia- tions around the notions of nodes and links. Nodes may or may not possess wireless communication capabilities, sensing abilities, or self-mobility. They may differ in clock frequency, color, commu- nication range, or any other user-defined property. Links between the nodes account for potential communication among them.

The nature of links varies as well; a link can be directed or undirected, as well as it can be wired or wireless – in the latter case JB OT S IM 's topology will update the set of links automatically, as a function of nodes distances and communication ranges. Figures 1 and 2 illustrate two different contexts. Figure 1 depicts a highway scenario where three types of nodes are used: vehicles, road-side units (towers), and central servers. This scenario is semi- infrastructured: Servers share a dedicated link with each tower. These links are wired and thus exist irrespective of distance. On the other hand, towers and vehicles communicates through wire- less links that are automatically updated.

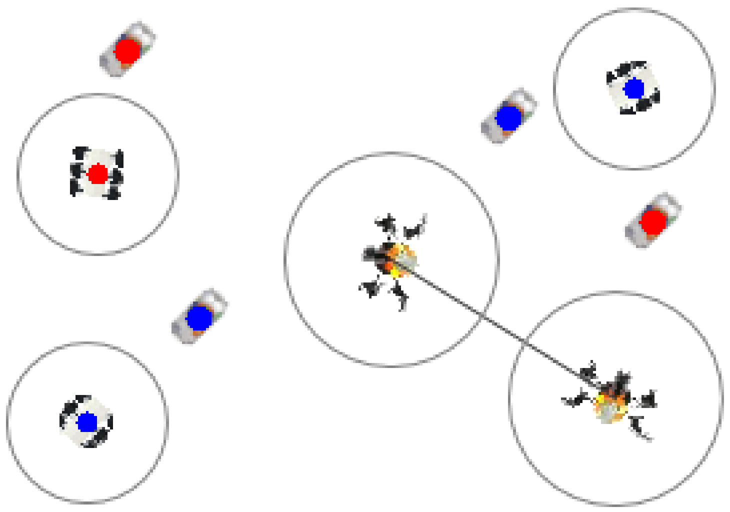
Figure 5.2: A swarming scenario, whereby mobiles robots and UAVs collaborate in order to clean a public park

Figure 2 illustrates a purely ad hoc scenario, whereby a hetero- geneous swarm of UAVs and robots strives to clean a public park collectively. In this scenario, robots can detect and clean wastes of a certain type (red or blue) only if these are within their sens- ing range (depicted by a surrounding circle). However, they are pretty slow to move and cannot detect remote wastes. In the mean- time, a set of UAVs is patrolling over the park at higher speed and with larger sensing range. Whenever they detect a waste of some type, they store its position and start searching for a capable robot. In addition to sensing capabilities, UAVs can exchange messages with each other to optimize the process. Besides nodes and links, the concept of topology is central in JB OT S IM . Topologies can be thought of as containers for nodes and links, together with dedicated operations like updating wireless links. They also play a central role in JB OT S IM 's event architec- ture, as explained later on.

5.2.2 Distributed vs. centralized algorithms

JB OT S IM supports the manipulation of centralized or distributed algorithms (possibly simultaneously). The natural way to imple- ment a distributed algorithm is by extending the Node class, in which the desired behavior is implemented. Centralized algorithms are not constrained to a particular model, they can take the form of any standard java class. *Distributed Algorithm* JB OT S IM comes with a default type of node that is implemented in the Node class. This class provides the most general features a node could have, including primitives for moving, exchanging messages, or tuning basic parameters (e.g. communication range and sensing range). Distributed algorithms are naturally imple- mented through adding specific features to this class. Listing 2 provides a basic example in which the nodes are endowed with self-mobility. The class relies on a key mechanism in JB OT S IM : performing periodic operations that are triggered by the pulse of the system clock. This is done by overriding the on-Clock() method, which is called periodically by JBotSim's engine (by default, at every pulse of the clock). The rest of the code is responsible for moving the node, setting a random direction at construction time (in radian), then moving in this direction periodically. (More de- tails about the movement API can be found online.)

5.2.3 Architecture of the event system

So far, we have seen one type of event: clock pulses, to be lis- tened to through the ClockListener interface. JB OT S IM offers a number of such events and interfaces, some of which become are ubiquitous. The main ones are depicted on Figure 3 on the fol- lowing page. This architecture allows one to specify dedicated op- erations in reaction to various events. For instance, one may ask to be notified whenever a link appears or disappears somewhere. Same for messages, which are typically listened to by the nodes themselves or can be watched at a global scale (e.g. to keep a log of all communications). In fact, every node is automatically noti- fied for its own events; it just needs to override the corresponding methods from the parent class Node in order to specify event han- dlings (e.g. onClock(), onMessage(), onLinkAdded(), onSensingIn(), onSelection(), etc.). Explicit listeners, on the other hand, like the ones in Figure 3, are meant to be used by centralized programs which do not extend class Node. Listing 6 gives one such example, consisting of a mobility trace recorder. This program listens to topological events of various kinds, including appearance or disappearance of nodes or links, and movements of the nodes. Upon each of these events, it outputs a string representation of the event using a dedicated human readable format called DGS [7]. Similar code could be written for Gephi [3]. Other events exist besides those represented in Figure 3, such as the SelectionListener interface, which makes it possible to be notified when a node is selected (middle-click) and make it initiate some tasks, for instance broadcast, distinguished role, etc.

5.2.4 Single threading: why and how?

It seems convenient at first, to assign every node a dedicated thread, however JB OT S IM was designed differently. JB OT S IM is single-threaded, and definitely so. This section explains the why and the how. Understanding these aspects are instrumental in de- veloping well-organized and bug-free programs. In JB OT S IM , all the nodes, and in fact all of JB OT S IM 's life (GUI excepted) is articulated around a single thread, which is driven by the central clock. The clock pulses at regular interval (whose period can be tuned) and notifies its listeners in a specific order. JB OT S IM 's internal engines, such as the message engine, are served first. Then come those nodes whose wait period has expired (re- mind that nodes can choose to register to the clock with different periods). These nodes are notified in a random order. Hence, if all nodes listen to the clock at a rate of 1 (the default value), they will all be notified in a random order in each round, which makes JB OT S IM 's scheduler a non-deterministic 2-bounded fair sched- uler. (Other policies will be available eventually.) A simplified version of the current scheduling process is depicted on Figure 4. One consequence of single-threading is that all computations (GUI excepted) take place in a sequential order that makes it pos- sible to use unsynchronized data structures and simpler code. This also improves the scalability of JB OT S IM when the number of nodes grows large. One can rely on other user-defined threads in the program, however one should be careful that these thread do not interfer with JB OT S IM 's. The canonical example is when a sce- nario is set up by program from within the thread of the main() method. If the initialization makes extensive use of JB OT S IM 's API from within that thread and the clock starts triggering events at the same time, then problems might occur (and a Concurrent ModificationException be raised). The easy way around is to pause the clock before executing these instructions and to re- sume it after (using the pause() and the resume() methods on the topology, respectively).

5.2.5 Interactivity

I designed JB OT S IM with a clear separation in mind between GUI and internal features. In particular, it can be run without GUI (i.e. without creating the JViewer object), and things will work exactly the same, though invisibly. As such, JB OT S IM can be used to perform batch simulations (e.g. sequences of unattended runs that log the effects of some varying parameter). This also enables to withstand heavier simulations in terms of the number of nodes and links. This being said, one of the most distinctive features of JB OT S IM remains interactivity, e.g., the ability to challenge the algorithm in difficult configurations through adding, removing, or moving nodes during the execution. This approach proves useful to think of a problem visually and intuitively. It also makes it possible to explain someone an algorithm through showing its behavior. The architecture of JB OT S IM 's viewer is depicted on Figure 5. As one can see, the viewer relies heavily on events related to nodes, links, and topology. The influence also goes the other way, with mouse actions being translated into topological operations. These features are realized by a class called JTopology. This class can often be ignored by the developer, which creates and manip- ulates the viewer through the higher JViewer class. The latter adds external features such as tuning slide bars, popup menus, or self-containment in a system window. While natural to JB OT S IM 's users, the viewer remains, in all technical aspects, an independent piece of software. Alternative viewers could very well be designed with specific uses in mind.

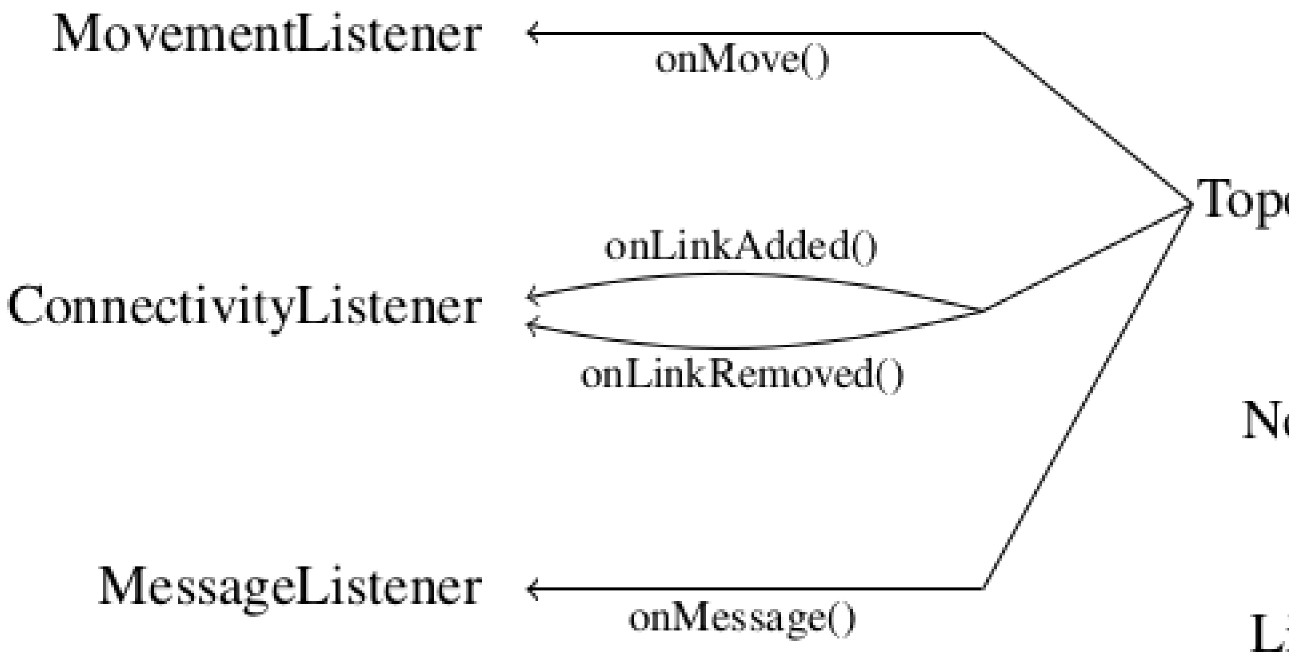

Figure 5.3: Main sources of events and corresponding interfaces in JB OT S IM

Chapter 6

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