A hierarchical leader election algorithm for dynamic networks with causal clocks*

João André Pestre†
Télécom Paris
SLR209 Seminar
(Dated: June 7, 2020)

An algorithm for electing a 2 level hierarchy of leaders in an asynchronous network with dynamically changing communication topology is presented. The algorithm ensures that, for most patterns of topology change, if topology changes cease, then eventually every connected component contains a well formed hierarchy of leaders. The algorithm works as an extension of A leader election algorithm for dynamic networks with causal clocks (Radeva et al. in Distib. Comput. 26, pp. 75-97 2013) usign a height-based mechanism for reversing the direction of communication links (Gafni and Bertsekas in IEEE TransCommun C-29(1), pp. 11-18 1981). Furthermore, a general notion of event precedence is needed, which can be achieved using totally-ordered values that preserve the causality of events, such as logical clocks and perfect clocks. A network simulator was implemented and used to argue the correctness of the algorithm for certain configurations and that in certain well-behaved situations, elections are limited to the local variety, that is, the hierarchical structure works in favor of the network.

Keywords: Distributed algorithms, Leader election, Leader hierarchy, Link reversal, Dynamic networks

I. INTRODUCTION

Leader election is an important primitive for distributed computing in situations that require the selection of a unique process to perform specific functions. As for a hierarchy of leaders, it is useful to reduce the amount of messages in the network, since most leader actions would occur is smaller partitions of the network and most topology changes would cause local elections instead of global ones.

A dynamic network is one in which communication channels are not stable, going up and down frequently. Such changes may be caused by physical movement of nodes in mobile networks or failure and repair of links in wired ones. Research in recent years has focused on developing distributed applications on dynamic networks and many of this applications require the action of leaders. Hence the need for leader election algorithms for dynamic networks.

A hierarchical leader structure differs from a simple leadership in the multiplicity of leaders and in the presence of levels of leadership. Regular nodes in the network respond to leaders in the lowest level, which in turn respond to the leaders one level above and so on recursively until the single global leader for the network (or connected component, more precisely). On a fixed level of leadership, leaders rule over different partitions of the network and the partitioning can be defined in different ways.

This algorithm tackles the problem of ensuring that, if topology changes cease, then each connected component of the network will have a hierarchical leader structure. This algorithm is an extension of the leader election algorithm in [1] which in turn uses ideas from [2].

Gafni and Bertsekas [2] present a generalized numbering system and a general class of algorithms based on it to transform directed graphs into acyclic directed graphs(ADG's) that are destination oriented (all nodes have a path to the only sink in the graph). The numbering system consists in assigning to each node in the network a unique number from a totally ordered set in a way that, for all pairs of neighbor nodes *n1* and *n2*, the directed communication link between them goes from *n1* to *n2* if and only if the number assigned to *n1* is greater than the one assigned to *n2*. The general class of algorithms proposed consists in generating a sequence of numbering for the graph in such a way that after a finite number of steps, the numbering stops changing and the graph has become a destination oriented ADG. Notice that, since the direction of the links is defined by the numbering of the graph, as this numbering changes certain links will change direction. This is the basic technique used in the graph transformation (and in the following algorithms) and it is called *link reversal*.

Radeva et al. [1] uses both of these concepts to define a leader election algorithm for dynamic networks that is the basis for this algorithm. Each node is assigned a 7-tuple of integers that is lexicographically compared to those of other nodes, consisting of a 3-tuple called the *reference level*, a *delta* component, a pair called the *leader pair* and the node's unique id. In the description of the algorithm, this 7-tuple is called a *height* and will henceforward be referred to as such. The *reference level* is used when a node loses its path to the leader and needs to search for an alternate one. This 3-tuple is started as all zeroes and, when a search needs to be started, the node starts a new reference level by setting the first component to its causal clock, the second to its id and

* Preliminary version of the article written for the course SLR209 at Télécom Paris

† joao.pestre@telecom-paris.fr

the third to 0. The reference level is propagated through the connected component as nodes lose their outgoing links because of updating heights. The propagation of a reference level is stopped when it finds an alternate path to the leader, in which case the algorithm is done, or when it reaches a dead end. In the latter, the reference level is *reflected* by changing the third component from 0 to 1 and propagated back through the connected component until reaching the node that initiated the search. When this node is reached, it identifies that the former leader is no longer in the connected component and therefore elects itself as new leader by changing the *leader pair*. The first component of the pair is changed to the negative of the causal clock of the node and the second one to its unique id. This change in height is also propagated through the network and all nodes adopt the new leader as their own. Lastly, the *delta* component of the height is used to orient the links in two different manners. During the search portion of the algorithm, the delta starts at 0 and decreases as the search propagates to orient the links outward from the search origin. When a leader is elected, the delta is reset to 0 and increases as the message propagates so that when the algorithm converges the deltas represent the distance from the nodes to the leader, in number of hops. This algorithm works in an asynchronous system with arbitrary topology changes.

This paper presents an extension of the previous algorithm to account for a 2 level hierarchy of leaders by adding 4 components to the height. One integer component called *local hops* is added to the end of the reference level to differentiate local and global searches. When a node starts a search for a global leader this component is set to 0 and it stays constant for the whole search. When the search is local the component is set to 1 and it is incremented each time the search is forwarded, counting the number of hops taken by the search. A new delta called *local delta* and a new leader pair called *local leader pair* are added after the previous leader pair and work in a similar manner as described above but are associated to local leaders. The former delta and leader pair are appropriately renamed *global delta* and *global leader pair*. The mechanisms for leader adoption and message propagation are similar to those of [1] but some new rules are introduced to account for the hierarchy. For instance, in [1] the criteria for adopting a new leader is taking the most recent one, while in the algorithm here presented the criteria for adopting a new local leader include comparing distances to global and local leaders as well as timestamps.

As argued in [1], this algorithm uses a generic notion of time that respects the causality between events called a *causal clock*. Logical clocks and perfect clocks may be used to implement this causal clock.

Finally, this paper does not provide a proof of correctness for the proposed algorithm. Instead, correctness is explored in certain key situations by simulations implemented using the Akka framework in Java.

II. PRELIMINARIES

A. System model

Since this algorithm is an extension of the one developed by [1], we assume a system similar to one defined in that paper. That is, a set $\mathcal{P}$ of completely reliable computing nodes and a set $\mathcal{X}$ of unreliable directed communication channels between nodes. $\mathcal{X}$ contains every possible channel in the network, that is a channel for every ordered pair of nodes. Channels can go up and down and, while up, the communication across them is asynchronous and in first-in-first-out order.

The system as a whole works as a set of state machines that interact through events. Every node and every channel is a state machine. Events can be notifications of a channel going up or down, from a channel to a node, or message passing, from a node to channel that then forwards it to another node.

1. Nodes

The state of a node u is defined by the value of its variables:

- *id*: An unique node identifier.
- *forming*: A set of nodes. A node v is in this set if *Channel(u,v)* is up and u has not received any messages from v.
- N: A set of nodes. A node v is in this set if *Channel(u,v)* is up and u has received a message from v.
- *glid*: Global leader id. The unique identifier of the global leader.
- *llid*: Local leader id. The unique identifier of u's local leader.
- $\mathcal{T}$: A causal clock. Can be implemented using logical clocks or perfect clocks.
- *heights*: A set of heights including u's height and those of all nodes v belonging to N.

2. Channels

A channel from node u to node v, denoted *Channel(u,v)* is defined by the following variables:

- *status*: Depicts the status of the channel, either *Up* or *Down*.
- *mqueue*: The queue of messages from u to v.

While the status of *Channel(u,v)* is *Up*, messages received from u will be added to end of *mqueue* and messages in the queue will be forwarded to v in order. When the channel goes down, the message queue is emptied and received messages start to be ignored. The nodes u and v do not know which messages were delivered and which were lost.

3. Events

The state machines of the nodes and channels interact through shared events. When a *Channel(u,v)* changes status it sends a $ChannelDown_{uv}$ or $ChannelUp_{uv}$ event to node u as appropriate and when a node u sends a message to node v it sends an *Update* event to *Channel(u,v)* which in turn forwards the event to node v.

B. Hierarchical leadership structure

The structure of leadership used in this paper is a 2 level hierarchy made of set $\mathcal{L}$ of local leaders, one for each *local neighborhood*, and one global leader belonging to $\mathcal{L}$. For simplicity of development, the partitioning of the network into local neighborhoods is defined using a fixed value for the maximum distance from a node to its local leader. That is, there is a system wide constant called *MAX HOPS* and, for each node u, if node v is its local leader then there is a directed path from u to v of length at most *MAX HOPS* hops. Furthermore, if a node v belongs to $\mathcal{L}$ then v must be its own local leader.

This choice of partitioning is perhaps too naive and not ideal. It is easy to see that, if *MAX HOPS* is too high there might be too few local leaders in the network and every topology change may cause a global leader election for the whole network. On the other hand, if it is too low there might be too many local leaders and too many local elections whenever the best suitable local leader is one hop too far. It is easy to see as well that the number of local leader varies as the topology changes. Different partitioning strategies that might be worth exploring are fixing the amount of local leaders or defining a fixed set of local neighborhoods.

C. Configurations

A *configuration*, as defined in [1], is an instantaneous snapshot of the state of the network. That is, the state of every node in $\mathcal{P}$ and every channel in $\mathcal{X}$. For every configuration it can be assigned a graph G depicting the status and logical direction of every communication channel.

A *well formed* configuration is as follows:

- Every node has a local variable *llid* holding the unique id of a node within *MAX HOPS* of distance.
- Every node in a connected component holds the unique id of the same node for the local variable *glid* depicting their view of the global leader.
- A node that is a local leader holds its own unique id for the variable *llid*.
- A node that is a global leader must also be a local leader
- The *global delta* and *local delta* components of a node's height hold the number of hops in the shortest direct path from it to its global leader and local leader respectively.
- For every channel the message queue is empty.
- For every pair of nodes u and v, *Channel(u,v)* and *Channel(v,u)* hold the same value for the local variable *status*.

D. Problem statement

Let *net* be a network in an initial configuration respecting the *well formed* constraints defined above. If *net* goes through a finite number of topology events, it eventually returns to a *well formed* configuration.

III. HIERARCHICAL LEADER ELECTION ALGORITHM

In this section the hierarchical leader election algorithm is presented with an informal and a formal description and the support of pseudocode.

A. Height

This algorithm extends the 7-tuple height used in [1] to an 11-tuple height $((\tau, oid, r, lh), g\delta, (nglts, glid), l\delta, (nllts, llid), id)$, where the first 4 components are the *reference level* (RL) the sixth and seventh are the *global leader pair* (GLP) and the ninth and tenth are the *local leader pair* (LLP). The components are defined as follows:

- τ: A non-negative timestamp which is either 0 or the value of the causal clock of when the search was started.
- *oid*: A non-negative value that is either 0 or the id of the node that started the search.
- r: A bit that is set to 0 when a search starts and to 1 when it is reflected.
- *lh*: A non-negative value that is either fixed at 0 for a global search or set to 1 when a local search is started and incremented each time the search is propagated.

- $g\delta$: An integer value used to orient links when the nodes have the same RL. During a global search, this value is initially set to 0 and it decreases as the search is propagated, orienting the links outward from the search origin. When a new global leader is elected, this value is set to 0 and it increases as the message is propagated, orienting the links towards the new leader and representing the number of hops from the node to the global leader.
- *nglts*: A non-positive timestamp whose absolute value is the causal clock of when the global leader was elected.
- *glid*: The unique id of the global leader.
- $l\delta$: Analogous to $g\delta$ but for local searches and local leaders. A point in which it differs from its global counterpart is in the initial value for searches. Instead of being set to 0 it is set to -1 to avoid conflicts when comparing distances to local leaders.
- *nllts*: A non-positive timestamp whose absolute value is the causal clock of when the local leader was elected.
- *llid*: The unique id of the local leader.
- *id*: The node's unique id.

B. Informal description

Each node in the network has a height as described above. The direction of the links in the network is determined by comparing the heights of neighboring nodes, an edge going from a 'higher' node to a lower one. When topology changes occur, nodes may lose or gain incident links. Whenever a node loses its last outgoing link it also loses its path to the leaders. It is the interest of this node to reverse its incoming links in order to find a path to its leaders and, to do so, it needs to raise its own height. This is done by starting a new *reference level*, the part of the height related to the search for a leader. If the node starting the search is a regular node, it starts a search for a local leader. If the node is a local leader, it starts a search for a new path to the global leader. The node then sends a message to its neighbors to update them of its new height. When a node receives this update it may also lose its last outgoing link, since at least one of its links have been reversed. In this case it should also raise its height, and it does so by adopting the reference level of the search, changing the deltas appropriately and updating the neighbors. Following this propagation, a search may end in 2 different ways. If it reaches a node u that has outgoing links then a new path to the leader was found in which case u responds with an update message so the searching nodes can update their respective deltas with their new distances to the leaders. If a new path is not found, meaning that the old leader is no longer reachable, then the search will reach a *dead end*. A dead end is a node in the network whose neighbors all have the same *reference level*, that is, they all now about the same search. When a dead end is reached the search is reflected and propagated back through the network in the same manner as before. When this reflected search reaches the node from which it originated, this node will elect itself as a new leader and update its neighbors who then accept it as their new leader according to certain priority criteria.

The steps described above are generic for searches for local and global leaders. There are, however, more situations in which a local search may end. If a local search goes too far, that is, if the *lh* component of the *reference level* reaches a value greater than the constant *MAX HOPS*, the search is reflected as if it had reached a dead end. This is done to limit local searches and local elections to *local neighborhoods*. Furthermore, if a different local leader is found within the limit of hops it will adopted and the search will be over even though no path to the old local leader was found.

At any point in the previously described process a node may receive an update message with different leaders than the ones it holds. In such situations the node will adopt the leaders if the priority criteria hold. For global leaders the criterion is the election timestamp, that is, the most recently elected global leader will be adopted as it is done in [1]. For local leaders the criteria include distance to global leader, distance to self and election timestamp. The node will accept the local leader who is closer to the global leader among the two (given that it respects the *MAX HOPS* constraint). This is done so communication in the network is kept in the general direction of the global leader in the final configuration. If this criterion is insufficient, then the node will accept the local leader closer to itself, limiting the number of hops it needs to send a message to its local leader. Finally, if it is still unresolved, the node will accept the local leader that was more recently elected.

C. Pseudocode

When node u **receives a** $ChannelUp_{uv}$ **event:**
$forming := forming \bigcup \{v\}$
send $Update(heights[u])$ to v

When node u receives a $ChannelDown_{uv}$ event:
$N := N \backslash \{v\}$
$forming := N \backslash \{v\}$
if $N = \emptyset$ **then**
  $ElectSelfGlobal$
  $\forall w \in forming$ send $Update(heights[u])$
else if $Sink \bigwedge glid \neq u$ **then**
  if $llid = u$ **then**
    $StartNewRefLevelGlobal$
  else
    $StartNewRefLevelLocal$
  end if
  $\forall w \in (N \bigcup forming)$ send $Update(heights[u])$
else
  $myOldHeight := heights[u]$
  $UpdateLocalDelta$
  if $myOldHeight \neq heights[u]$ **then**
    $\forall w \in (N \bigcup forming)$ send $Update(heights[u])$
  end if
end if

When node u receives $Update(h)$ event from v:
// if v is not in forming nor in N the message is ignored
$height[v] := h$
$forming := forming \backslash \{v\}$
$N := N \bigcup \{v\}$
$myOldHeight := heights[u]$
if $(nglts^u, glid^u) = (nglts^v, glid^v)$ **then**
  // global leader pairs are the same
  if $((nllts^u, llid^u) = (nllts^v, llid^v)) \bigvee (\tau^v > 0 \bigwedge lh^v = 0)$ **then**
    // local leaders are the same or the search is global
    if $Sink$ **then**
      // u has no outgoing links
      if $glid^u \neq u$ **then**
        if $llid^u = u \bigwedge lh^v > 0$ **then**
          // local search found local leader
          $StartNewRefLevelGlobal$
        end if
        if $r^v = 0 \bigwedge lh^v > MAXHOPS$ **then**
          // local search has gone too far
          $ReflectRefLevel$
        else if $\exists(\tau, oid, r, lh) | (\tau^w, oid^w, r^w, lh^w) = (\tau, oid, r, lh) \forall w \in N$ **then**
          // if all neighbors have the same RL. Every $lh > 0$ is considered equal for this comparison
          if $\tau > 0 \bigwedge r = 0$ **then**
            // dead end
            $ReflectRefLevel$
          else if $\tau > 0 \bigwedge r = 1 \bigwedge oid = u$ **then**
            // reflected search reached origin
            if $lh = 0$ **then**
              // the search was global
              $ElectSelfGlobal$
            else
              // the search was local
              $ElectSelfLocal$
            end if
          else
            // there is no search happening or second dead end
            if $llid^u = u$ **then**

```
            StartNewRefLevelGlobal
          else
            StartNewRefLevelLocal
          end if
        end if
      else
        // neighbors have different RL
        PropagateLargestRefLevel
      end if
    end if
  else
    // u has outgoing links
    if (lδ^u > 0 ⋀ gδ^u > 0) ⋁ llid^u = u then
      // u's deltas aren't dirty
      if lδ^v ≤ 0 ⋁ gδ^v ≤ 0 then
        // At least one of v's deltas is dirty
        send Update(heights[u]) to v
      end if
    end if
    UpdateLocalDelta
  end if
else
  // local leaders are different
  if llid^u ≠ u ⋀(∀w ∈ N|lδ^w + 1 > MAXHOPS) then
    // u is not a local leader and there are no local leaders close enough
    ElectSelfLocal
  else
    // there are possible leaders close
    AdoptLLPIfPriority(v)
  end if
end if
else
  // global leader pairs are different
  AdoptGLPIFPriority(v)
end if
if myOldHeight ≠ heights[u] then
  ∀w ∈ (N ⋃ forming) send Update(heights[u])
end if

Subroutines
Sink:
((∀v ∈ N_u, GLP^v = GLP^u) ⋀(∀v ∈ N_u, heights_u[u] < heights_u[v]))

StartNewRefLevelGlobal:
heights[u] := ((T_u, u, 0, 0), 0, (nglts^u, glid^u), lδ^u, (nllts^u, llid^u), u)

StartNewRefLevelLocal:
heights[u] := ((T_u, u, 0, 1), gδ^u, (nglts^u, glid^u), −1, (nllts^u, llid^u), u)

ReflectRefLevel:
if lh = 0 then
  heights[u] := ((τ, oid, 1, lh), 0, (nglts^u, glid^u), lδ^u, (nllts^u, llid^u), u)
else
  heights[u] := ((τ, oid, 1, lh), gδ^u, (nglts^u, glid^u), −1, (nllts^u, llid^u), u)
end if

PropagateLargestRefLevel:
if llid^u = u then
```

// if u is a local leader, only propagate largest global search
$(\tau^u, oid^u, r^u, lj^u) := max\{(\tau^w, oid^w, r^w, lh^w) | w \in N \bigwedge lh^w = 0\}$
else
$(\tau^u, oid^u, r^u, lj^u) := max\{(\tau^w, oid^w, r^w, lh^w) | w \in N\}$
end if
if $lh^u > 0$ **then**
// if the largest ref level is a local search
$l\delta^u := min\{l\delta^w | w \in N \bigwedge (\tau^u, oid^u, r^u, lh^u) = (\tau^v, oid^v, r^v, lh^v)\} - 1$
$lh^u := lh^u + 1$
else
$g\delta^u := min\{g\delta^w | w \in N \bigwedge (\tau^u, oid^u, r^u, lh^u) = (\tau^v, oid^v, r^v, lh^v)\} - 1$
end if

ElectSelfGlobal:
$heights[u] := ((0,0,0,0),0,(-\mathcal{T}_u, u),0,(-\mathcal{T}_u, u),u)$

ElectSelfLocal:
$heights[u] := ((0,0,0,0), g\delta^u, (nglts^u, glid^u), 0, (-\mathcal{T}_u, u), u)$

AdoptGLPIFPriority(v):
if $(nglts^v < nglts^u) \bigvee ((nglts^v = nglts^u) \bigwedge (glid^v < glid^u))$ **then**
$heights[u] := ((\tau^v, oid^v, r^v, lh^v), g\delta^v + 1, (nglts^v, glid^v), l\delta^u, (nglts^u, llid^u), u)$
if $l\delta^v + 1 \leq MAXHOPS$ **then**
AdoptLLPIfPriority(v)
end if
else
send $Update(heights[u])$ to v
end if

AdoptLLPIFPriority(v):
if $l\delta^v \geq 0 \bigwedge g\delta^v \geq 0$ **then**
// v's deltas aren't dirty
if $(l\delta^u < 0) \bigvee (g\delta^v + 1 < g\delta^u) \bigvee ((g\delta^v + 1 = g\delta^u) \bigwedge (l\delta^v + 1 < l\delta^u)) \bigvee ((g\delta^v + 1 = g\delta^u) \bigwedge (l\delta^v + 1 \geq l\delta^u) \bigwedge (nllts^v < nllts^u)) \bigvee ((g\delta^v + 1 = g\delta^u) \bigwedge (l\delta^v + 1 \geq l\delta^u) \bigwedge (nllts^v = nllts^u) \bigwedge (llid^v < llid^u))$ **then**
// if u's local delta is dirty or v's leader is closer to the global leader or v's leader is closer to u
$heights[u] := ((\tau^v, oid^v, r^v, lh^v), g\delta^v + 1, (nglts^v, glid^v), l\delta^v + 1, (nglts^v, llid^v), u)$
end if
else
send $Update(heights[u])$ to v
end if

UpdateGlobalDelta:
$g\delta^u := min\{g\delta^w | w \in N \bigwedge glid^w = glid^u \bigwedge g\delta^w \geq 0\} + 1$
if $glid^u = u$ **then**
$g\delta^u = 0$
end if

UpdateLocalDelta:
$l\delta^u := min\{l\delta^w | w \in N \bigwedge llid^w = llid^u \bigwedge l\delta^w \geq 0\} + 1$
if $llid^u = u$ **then**
$l\delta^u = 0$
end if
if $l\delta^u > MAXHOPS$ **then**
ElectSelfLocal
end if
UpdateGlobalDelta

D. Description of the algorithm

The algorithm works handling the three possible events that can occur in the system: a channel going up, a channel going down and an update message. The pseudocode

for this event handlers and the subroutines needed is available above. Next, a formal description of them.

1. ChannelUp event

When node u receives a $ChannelUp_{uv}$ event it adds node v to its $forming$ set and sends an update message to v with its current height.

2. ChannelDown event

When node u receives a $ChannelDown_{uv}$ event it removes node v from its $forming$ and N sets as appropriate. If after removing v from its neighborhood u has an empty N set, that is, if u is alone in its connected component, then it elects itself as the global leader and updates the nodes in its $forming$ set. If u has become a sink, that is, if it has no outgoing links and it is not the global leader then it needs to start a new reference level to search for an alternate path to its leader. If u is a local leader it starts a global reference level, if it isn't it starts a local reference level. Then u sends an update message to all nodes in its $forming$ and N sets.

If none of the previous conditions are met, then u still has outgoing links, but since it lost a link then the deltas might not reflect the length of the shortest path to the leaders anymore. In this case, u updates its deltas and sends an update message to all nodes in its $forming$ and N sets.

3. Update event

When node u receives a $Update(h)$ event from node v it takes the following actions.

If v is not in $forming$ or N the message is ignored. If it is in the $forming$ set then v is moved to the N set. u also updates its view of v's height. Then, u checks if both nodes have the same global leader pair (GLP). If they don't, u checks if v's global leader is more recent than its own and, if it is, it adopts it by changing its own height and, if v's local leader respects the *MAX HOPS* constraint, it checks its priority as described bellow and adopts it if appropriate. If u's global leader happen to be the most recent elected one, than u sends v an update with its height. If both nodes have the same global leader, u then checks if they have the same local leader. If they don't, u checks if there are possible local leaders respecting the *MAX HOPS* constraint in its neighbors heights and, if there aren't any, u elects itself as local leader by changing its height. If there are possible local leaders in the neighborhood, u checks if v's local leader has priority. That is, if v's local leader is closer to the global leader or if both leaders are equidistant to the global leader but v's local leader is closer to u or if both conditions are inconclusive and v's local leader is more recent. If v's local leader has priority, u adopts it and v's global leader by changing its height and, if it doesn't, u sends an update to v with its height.

After checking that the leaders match, u needs to check if this update has caused it to lose its outgoing links, that is, it it has become a *sink*. If it hasn't, this might mean that a search was started and it ended when it reached u. In this case, u must send an update message back to v so v can change its deltas to represent its new distance to the leaders. In this same scenario, u updates its own deltas if necessary. If u did become a sink, then it performs one of the following actions:

1. If u is a local leader and the search was for a local leader, then this search is over and u must look for a new path to the global leader by starting a new reference level.

2. If the search hasn't been reflected yet and the lh component of the reference level is greater than *MAX HOPS*, then this is a local search that has gone too far and so it is reflected by changing the r component of the reference level from 0 to 1.

3. If all of u's neighbors have the same reference level, that is, if all neighbors know about the same search (notice that for this comparison all values of lh greater than 0 are considered to be equal) then this is a dead end and one of the following conditions may apply.

   - If τ is greater than 0 and r is 0, then this is the first dead end and the search is reflected by changing its r component to 1.
   - If τ is greater than 0, r is 1 and u is the origin node for the search, then the reflected search has reached the node that started it and, so, u elects itself as leader (local leader if the search was local and global leader if the search was global).
   - If none of this conditions apply, then either τ is 0, in which case there isn't a search happening, or r is 1 and u isn't the origin of the search, which means this is a second dead end. In either case, u does not have a path to its leaders and there is no reference level to propagate. In this situation, u starts a new reference level (global if u is a local leader and local if it's a regular node).

4. If none of the previous conditions are met, then there are multiple possible reference levels in u's neighbors. In this case, u propagates the most recent search, decrementing the appropriate delta to orient the links in the direction of the search and, if the search is local, incrementing the lh component to represent the number of hops the search has taken so far.

After performing all of this actions, u's height might have changed. If it did, it sends an update message with its new height to all of the nodes in its $forming$ and N sets.

IV. SIMULATIONS

The correctness of this algorithm was explored using simulations to observe how it behaves in certain key situations.

A. The simulator

First, we describe how the simulator was implemented. The system described in II A and the event handlers and subroutines in III C were implemented in Java using the Akka framework [3] to simulate asynchronous message passing between the nodes and channels. Every node and channel is an actor in the Akka actor system. The nodes' $forming$ and N sets contain references to the channel actors to its neighbors. That is, if node u at a point in the algorithm has node v in its N set, then the node actor in the Akka system for node u will hold in its N set the actor for *Channel(u,v)*.

For the purposes of network construction, an extra $SetUp_{uv}$ event was created to establish the initial topology of the network. This new event is handled similarly to the $ChannelUp_{uv}$ event, but without sending update messages. This is done to avoid the triggering of the algorithm before the initial network is completely formed.

For the causal clocks needed for the algorithm, both logical clocks and perfect clocks can be used. For this implementation, Lamport's logical clocks [4] were used.

Finally, a network visualization system and initial setup menu were created using the Processing library [5] in order to make the simulations more comprehensible. In this visualizations, the nodes are drawn as circles with their unique identifiers inside them and their heights and causal clocks next to them. Nodes with a second circle around them are local leaders and black nodes are global leaders. Communication links are drawn as an arrow from node u to node v if *Channel(u,v)* is up and $heights_u[u] > heights_u[v]$. Since communication is asynchronous, as the algorithm progresses the nodes' views of the heights get asymmetrical so a *Channel(u,v)* may be up and not be drawn if nodes u and v both see their own height as the lowest in the pair, or it may be drawn as an arrow in both directions if both of the nodes see their own height as the highest in the pair. As update messages are delivered, this asymmetry is resolved and so is the visualization of the channels.

The code for the simulator can be found in the github repository for the project.

B. Sample executions

In this section we will be looking at some sample executions to see when the algorithm works and when it doesn't.

The first three executions we are gonna explore were done in *network 1*, a fully connected 3 node network with *MAX HOPS* set to 1 and only node 0 as local and global leader. The initial configuration can be seen in figure 1. We explore these execution in more detail because they are simple enough to follow and get a better understanding of the algorithm.

The other executions were done in *network 2*, a 6 node network with a more complex initial topology and with *MAX HOPS* set to 2. Local leaders are initially 2 and 5, with 5 acting as the global leader as well.The initial configuration for this network can be seen in figure 14. These executions involve more nodes and more steps, so we analyse them by looking at initial and final configurations only.

1. *Execution 1*

Dropping *Channel(0,1)* and *Channel(1,0)* triggers a local search from node 1 as seen in figures 2 and 3. A new path is found through node 2 and the deltas are updated. Since node 0 is now 2 hops away from 1, this violates the *MAX HOPS* constraint, which makes node 1 elect itself as a local leader and send it to node 2. Node 2 does not accept node 1 as its new leader because its old leader, node 0, is closer to the global leader. This can be seen in figure 4.

The final configuration is *well formed*, so this execution is correct.

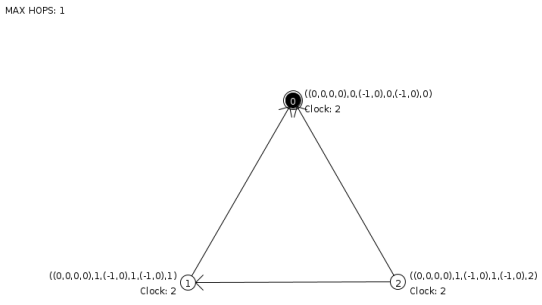

FIG. 1. Initial configuration of network 1.

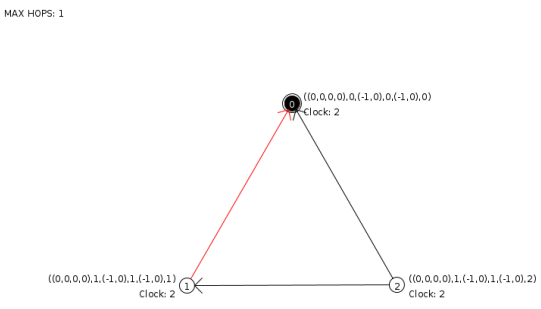

FIG. 2. Dropping channel between 0 and 1.

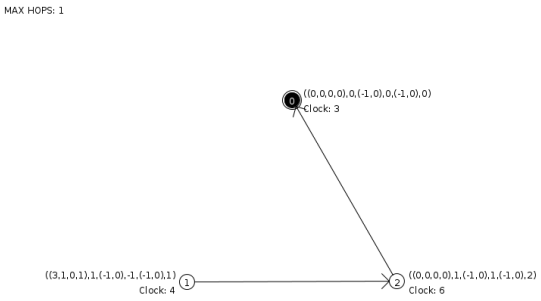

FIG. 3. Node 1 starts a local search and sends it to 2.

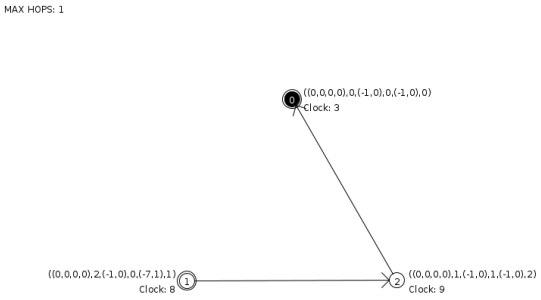

FIG. 4. Node 1 finds a path to 0 through 2.

2. *Execution 2*

The initial configuration for this execution is the final one from the last execution, seen in figure 4. This execution is triggered by dropping *Channel(0,2)* and *Channel(2,0)* in figure 5. Node 2 becomes a sink and starts a local search as in figure 6, sending an update to node 1. Node 1 receives this update from 2 and sees the different local leader pair. Since 2's local delta is negative, its leader does not have priority and, therefore, node 1 sends an update to 2. 2 receives this update and adopts 1 as its local leader, as seen in figure 7, and sends an update of its new height to node 1. Node 1 is now a sink and a local leader, so it starts a search for the global leader in figure 8. Node 2 reflects the search in figure 9 and node 1 receives this reflected search and elects itself as a global leader in figure 10. Finally, node 2 adopts 1 as a global leader.

Again, the final configuration is *well formed* and the execution is correct.

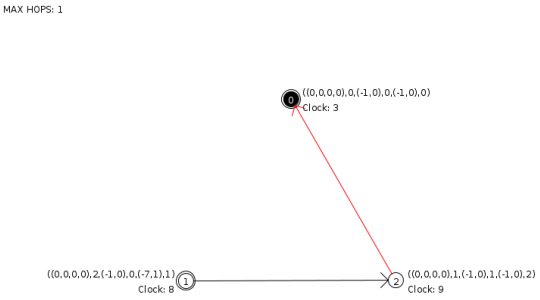

FIG. 5. Dropping channel between 0 and 2.

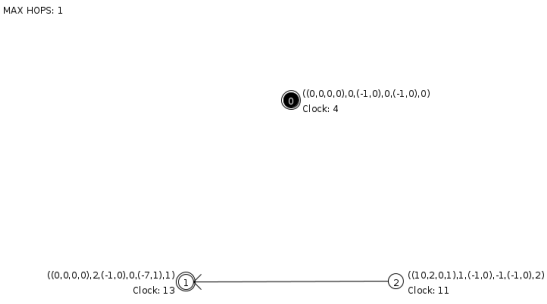

FIG. 6. Node 2 starts a local search and sends it to 1.

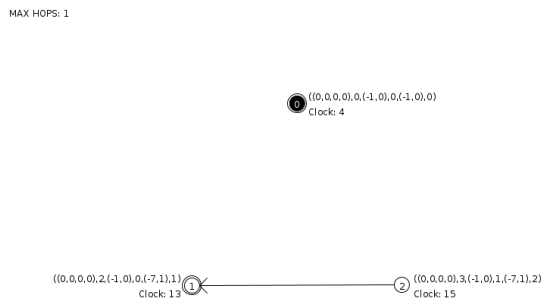

FIG. 7. Node 2 adopts 1 as local leader.

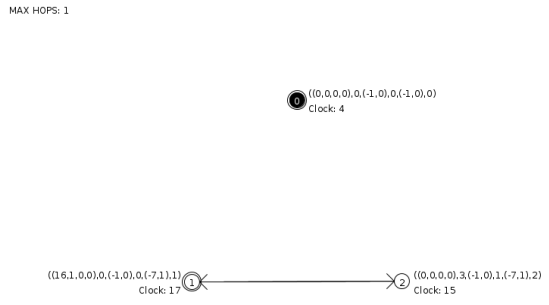

FIG. 8. Node 1 starts a global search and sends it to 2.

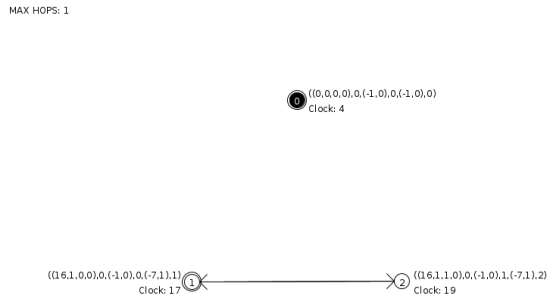

FIG. 9. Node 2 reflects the search.

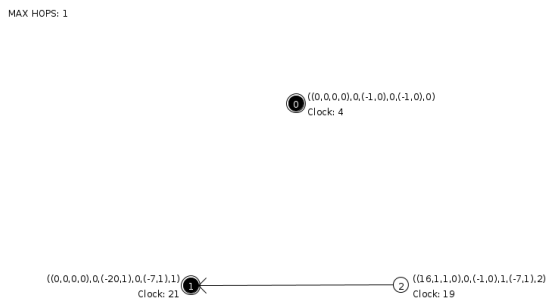

FIG. 10. Node 1 elects itself as global leader.

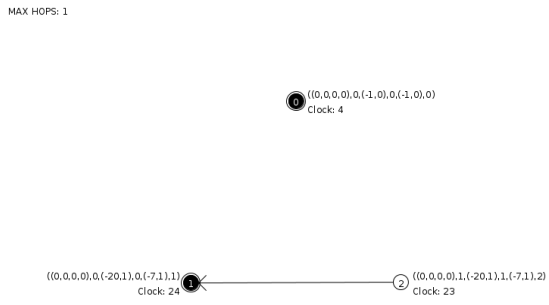

FIG. 11. Node 2 adopts 1 as global leader.

3. Execution 3

From the final configuration of the previous execution, figure 11, starts this execution. It is triggered by the *Channel(0,1)* and *Channel(1,0)* going up seen in figure 12. Since none of the nodes involved have a view of the other's height, the direction of this link is not determined until the next step, which implies that it is not drawn in the visualization, but we can see that the channel has gone up from the difference in the clocks from the initial configuration. The nodes update each other of their heights and node 0 adopts node 1 as its local and global leader, since it is the most recent one.

The final configuration is *well formed* and the execution is correct.

4. Execution 4

This execution was done on *network 2*, from its initial configuration in figure 14. After *Channel(0,2)* and

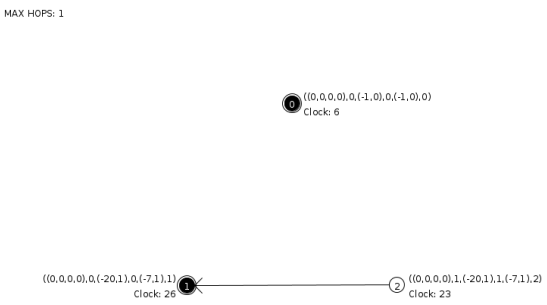

FIG. 12. Remaking the channel between 0 and 1.

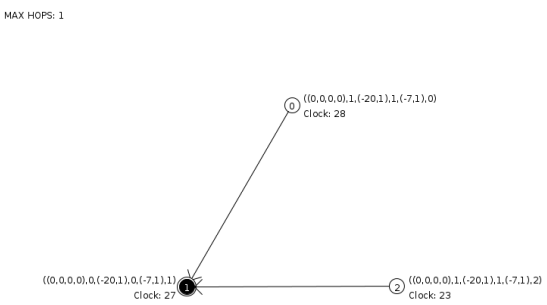

FIG. 13. Node 0 adopts 1 as global leader.

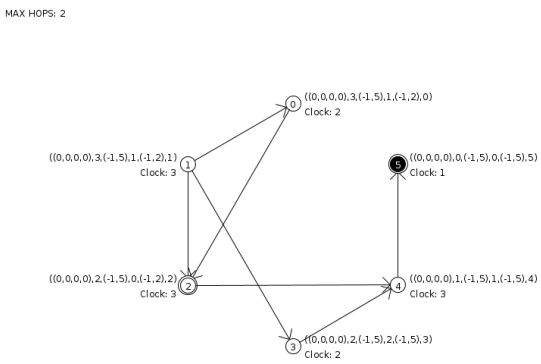

FIG. 14. Initial configuration of network 2.

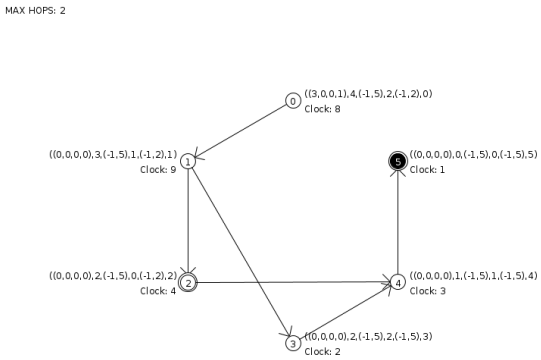

FIG. 15. Dropping channel between 0 and 2, final configuration

Channel(2,0) go down, node 0 becomes a sink and starts a search for a local leader. It finds a path to node 2, its old local leader, through node 1 and updates its deltas to represent its new distance to the leader. The final *well formed* configuration can be found in figure 15. This execution is correct.

5. Execution 5

The initial configuration for this execution is the final state of the last execution, in figure 15. When the channels between nodes 4 and 5 go down, node 4 starts a search for a local leader, finds node 2 and adopts it as its new local leader. Node 2 becomes a sink and, since it is a local leader, it starts a search for a global one. The nodes propagate the search until reaching dead ends and reflecting it. When the reflected search reaches node 2 it elects itself and the other nodes adopt it as a global and local leader. Notice that in the final configuration of this execution, figure 16, the network has less local leaders than before, but since all nodes respect the *MAX HOPS* constraint this configuration is still *well formed* and the execution correct.

6. Execution 6

This execution has the same initial configuration as the previous one and is triggered in the same manner. However, as we can see in figure 17, its final configuration is not the same. Moreover, its final configuration is not *well formed* since there isn't one global leader per connected component, there isn't a path from any of the nodes to their view of the global leader and some of the nodes also do not have a path to their local leaders.

This happened because during the execution node 3 adopted node 2 as a local leader and then, because of the reversal of some links, node 3's path to 2 became too long and node 3 had to elect itself as a local leader. This caused conflicts in orienting links later on and, when the network converged (nodes stopped sending messages), no

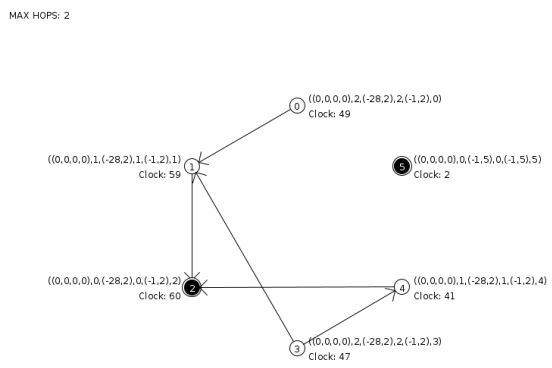

FIG. 16. Dropping channel between 4 and 5, final configuration.

node was elected as a global leader.

This execution shows two important problems. Two executions from the same initial configuration and same topology change triggering the algorithm yielded two different results. This means that the algorithm depends on the order of certain events in different nodes, which is not ideal in an asynchronous setting. Moreover, it shows that an execution from an initial configuration that is *well formed* may end, after a finite number of events, in a configuration that does not respect multiple constraints in the *well formed* definition.

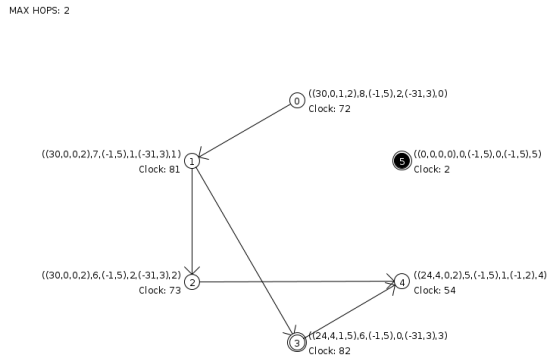

FIG. 17. Dropping channel between 4 and 5, failed execution.

V. ISSUES

As shown by the sixth sample execution, the algorithm is not correct for some situations. Some of the problems encountered were:

1. Local election: There are situations in which a node elects itself as a local leader even though it was not strictly necessary. This happens when messages are received in a specific order and the node takes precipitated actions that other messages in transit would show were not necessary. This actions in turn make the algorithm converge to an incorrect final configuration.

2. Local leader unreachable: Let u be a node and v be its local leader. When v loses an outgoing link, it may happen that the only path from v to the global leader goes through u. In this situation, the link between u and v is reverted so v can reach the global leader. By consequence, u can no longer reach its local leader, but at no point it changed its local leader.

VI. CONCLUSION

A hierarchical leader election algorithm for dynamic networks was proposed and exemplified through simulations. Some of this simulations revealed problems with the algorithm that were noted in the *Issues* section of this paper.

Even though problems were encountered, the simulations showed for many scenarios the algorithm working as expected which implies that with further research the issues faced might be solved. Moreover, the simulations also showed that in many situations elections are limited to the local variety and restricted to a smaller portion of the network.

Another point worth exploring is changing the partitioning of the network. In this paper, an initial partitioning was defined by hand and roughly kept by the use of a constant upper bound on the distance from a node to its local leader. Alternatives might include fixing the partitioning from the start, and adapting the algorithm to maintain one local leader per partition, or fixing the number of local leaders in a connected component while allowing the partitions to change as topology changes.

Finally, it is important to provide a proof of correctness for the algorithm after the issues are resolved.

Acknowledgements: We thank Petr Kuznetsov and Patrick Bellot for bringing this research topic to our attention and Ada Diaconescu for the multiple meetings that were immensely helpful in the developing and testing of the ideas here presented.

[1] R. Ingram, T. Radeva, P. Shields, S. Viqar, J. Walter, and J. Welch, A leader election algorithm for dynamic networks with causal clocks, Distrib. Comput. **26**, 75 (2013).
[2] E. Gafni and D. Bertsekas, Distributed algorithms for generating loop-free routes in networks with frequently changing topology, IEEE Transactions on communications **COM-29**, 11 (1981).
[3] Akka framework, https://akka.io/ (2011–2020).
[4] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM **21**, 558 (1978).
[5] B. Fry and C. Reas, Processing, https://processing.org/ (2001–2012).