An Asynchronous Leader Election Algorithm for Dynamic Networks without Perfect Clocks

Mira Radeva
Rebecca Ingram, Patrick Shields,
Dr. Jennifer Walter, Dr. Jennifer Welch
Distributed Computing Group
Parasol Lab, Texas A&M University

http://parasol.tamu.edu
Outline

• Leader election problem in dynamic networks
• Algorithm features
• System model
• Overview of the algorithm
• Example execution
• Correctness proof
• Summary and future work
Leader Election Problem

• Select a unique processor among multiple candidate processors in a dynamic network.

• Usually used as a primitive in other distributed algorithms.

• Various applications:
  ○ Primary-backup approach to replication-based fault tolerance
  ○ Group communication systems
  ○ Video conferencing
  ○ Multi-player games
Dynamic Networks

• A collection of nodes, connected together by communication links.

• Communication links go up and down frequently.

• Nodes are usually highly mobile.

• Multiple topology changes.

• Even if nodes remain static, communication links are still subject to interference.

• Example: Mobile Ad Hoc Networks.
Previous Work and Contribution

- [Gafini & Bertsekas] Introduced link reversal to construct a directed path to the leader.
- [Park & Corson] Overcome the problem of sending an infinite number of messages when disconnected from the leader.
- [Malpani et al.] Leader election algorithm for MANET, extension of TORA (Temporally Ordered Routing Algorithm).
- [Ingram et al.] Handles multiple topology changes and provides a correctness proof for the asynchronous case.
- **This paper:** Relaxes the requirement of nodes having perfect clocks; uses logical clocks instead.
Algorithm Features

• Handles **multiple** arbitrary topology changes.

• Tolerates **asynchronous** message passing.

• Does **NOT** assume perfect clocks (uses **logical clocks** instead)

• Correctness proof
System Model

- Nodes are completely **reliable**; communication across links is reliable, **asynchronous** and FIFO.

- Messages are lost only if a link goes down while a message is in transit.

- An algorithm execution is an infinite sequence of alternating configurations and events:
  - configuration: a vector of node states for each node in $P$ and a vector of link states for each link in $L$.
  - event: **three types of events** – LinkUp, LinkDown, receipt of an Update message

- There is a global time associated to each event in the system, but nodes have **no access to global time**.
Logical Clocks  [Lamport]

• Non-negative integer values (initially 0).

• Nodes timestamp all messages they send with the value of their logical clock.

• The logical clock of a node is updated in two situations:
  
  • Upon receipt of an update message, a node sets its logical clock to one more than the \text{max} of its current value and the timestamp in the message.
  • Upon LinkUp/Down events, logical clocks are incremented by 1.

\[
\text{logical clock} = \text{max} \{3, 5\} + 1 = 6
\]

\[
\text{timestamp: 5} \quad \text{logical clock: 3}
\]
Algorithm Overview

- **Main Goal**: after topology changes stop, maintain a leader-oriented DAG in each connected component.

- Links have directions determined by height values assigned to all nodes.

- A link is directed from a node with larger height to a node with smaller height.

- As the heights of nodes change, the direction of links between them may change too.

- Messages sent by nodes include their height and a logical clock timestamp.
Algorithm Overview

• Start with a clean set-up – a leader-oriented DAG and no messages in transit.

• When a node loses all of its outgoing links, it becomes a sink, and consequently it has no directed path to the leader.

• When a node becomes a sink, it is either:
  
  • in the same connected component as the leader, so it needs to find an alternative path to the leader.
  
  • partitioned from the leader, so it can possibly elect itself as the leader of this new connected component.
Algorithm Overview

• When a node becomes a sink, it starts a new reference level (RL) – a wave in search of a path to the leader.

• As the wave propagates, nodes may adopt the heights they receive, and consequently, the direction of links may reverse.

• Once the RL reaches a dead end, the wave is reflected back.

• If it finds a path to the leader, the search stops.

• When a node receives a reflected RL from all of its neighbors, it means that it is disconnected from the leader and, thus, it elects itself.
Height Data Structure

• Reference Level (RL):
  • $\tau$: time (logical clock value) when a new reference level is started.
  • $oid$: 0 or the id of the node that started the reference level.
  • $r$: reflection bit to indicate the direction of the RL.

  • $\delta$: integer value to ensure a correct direction of the links between nodes with identical RL's.

• Leader pair (LP):
  • $nlts$: negative of the time (logical clock value) when a leader is elected
  • $lid$: id of the current leader

• $id$: unique node id
Algorithm: Topology Changes

- When a link goes down:
  - increment both nodes' logical clocks by 1
  - if no neighbors, then elect self
  - else, if sink then:
    - start new reference level
    - send new height to new neighbors

- When a link comes up:
  - increment both nodes' logical clocks by 1
  - send height to new neighbor
Algorithm: Update Message

- if received LP is the same as current, then:
  - if not a sink, take no action
  - else (if sink), then:
    - if all neighbors have the same unreflected RL, then reflect RL
    - if all neighbors have same reflected RL, then elect self
    - if neighbors have different RLs, then propagate largest RL

- else (if LPs are different)
  - adopt LP if priority

- send messages with new height to all neighbors
Algorithm: Subroutines

- **start new RL**: set RL to \( (t, id, 0) \), where \( t \) is the current value of the node's logical clock.

- **propagate largest RL**: set RL to the largest neighboring RL, set delta to 1 less than the delta in that RL.

- **reflect RL**: set \( r \) to 1 and delta to 0.

- **elect self**: set RL to 0's, delta to 0 and LP to the current logical clock time and node id.

- **adopt LP if priority**: if new height has LP with a more recent timestamp, then adopt the new RL and LP and set delta to one more than the delta in that new height.
Example Execution

Start with a leader-oriented DAG with node H as the leader.

- ((0,0,0),3,(0,H),B)
- ((0,0,0),3,(0,H),C)
- ((0,0,0),2,(0,H),E)
- ((0,0,0),2,(0,H),D)
- ((0,0,0),0,(0,H),H)
- ((0,0,0),1,(0,H),G)
Example Execution

The link between nodes G and H goes down and their logical clocks are incremented.
Example Execution

Node G becomes a sink and starts a new reference level.
Example Execution

Nodes D and E are now sinks and have to propagate the largest reference level. They also update their logical clocks.
The msg sent by node D is slow. Node B propagates the largest reference level.
Example Execution

The link between C and D goes down. Node A propagates the largest reference level.
Example Execution

Both nodes C and D start a new reference level.
Example Execution

Node A updates its logical clock and propagates C's reference level.

Node A updates its logical clock and propagates C's reference level.
Example Execution

Node B propagates the largest reference level and updates its logical clock.

((6,C,0),-1,(0,H),A)
LC: 7

((6,C,0),-2,(0,H),B)
LC: 8

((1,G,0),-1,(0,H),E)
LC: 4

((0,0,0),0,(0,H),G)
LC: 4

((1,G,0),0,(0,H),G)

((3,D,0),0,(0,H),D)
LC: 3

((0,0,0),0,(0,H),H)
LC: 1

((0,0,0),0,(0,H),H)

Node B propagates the largest reference level and updates its logical clock.
Example Execution

Node E updates its logical clock and propagates the largest reference level.
Node G receives messages from both E and D and propagates the largest RL.
Node D receives the reference level from G, but it is a dead end, so it reflects it.
Node G propagates the largest RL and updates its logical clock.
Example Execution

Node E propagates the largest RL and updates its logical clock.
Example Execution

Node B propagates the largest RL and updates its logical clock.
Node A propagates the largest RL and updates its logical clock.
Node C receives the reflected reference level and elects itself.
Example Execution

Node A adopts the new leader pair because it has higher priority.

Node A: 
- LC: 17
- \([(6, C, 1), -3, (0, H), B]\)
- \([(0, 0, 0), 1, (-16, C), A]\)

Node B: 
- LC: 16
- \([(6, C, 1), -3, (0, H), B]\)

Node C: 
- LC: 16
- \([(0, 0, 0), 0, (-16, C), C]\)

Node D: 
- LC: 13
- \([(6, C, 1), 0, (0, H), D]\)

Node E: 
- LC: 15
- \([(6, C, 1), -2, (0, H), E]\)

Node G: 
- LC: 14
- \([(6, C, 1), -1, (0, H), G]\)

Node H: 
- LC: 1
- \([(0, 0, 0), 0, (0, H), H]\)
Example Execution

Node B adopts the new leader pair because it has higher priority.
Example Execution

Node E adopts the new leader pair because it has higher priority.
Example Execution

Node G adopts the new leader pair because it has higher priority.
Example Execution

Node D adopts the new leader pair because it has higher priority.
Node G updates its logical clock, but sends no messages because it's not a sink.
Correctness Proof

• After the last topology change, no node elects itself an infinite number of times.

• After the last topology change, no node starts an infinite number of reference levels.

• Eventually, there are no messages in transit.

• Every node has an accurate view of its neighbors' heights.

• Eventually, the connected component is a leader-oriented directed acyclic graph.
Challenges

• **Main Goal**: switch from perfect clocks to logical clocks.

• Logical clocks only capture the happens-before relationship and have no correlation to actual time.

• Suppose we have events A and B happening at two different nodes. If we know the real times when A and B happened, we can determine which happened first. However, if we only know the logical clock values of the nodes at the time, nothing can be concluded.

• This is a big problem when trying to determine whether an event happened before or after the last topology change, as that is a key point in most of the proofs of the algorithm.
Summary

• Based on TORA and link reversal.
• Handles multiple topology changes.
• Tolerates asynchronous message delays.
• Does not rely on perfect clocks (uses logical clocks).
• Proof of correctness
Future Work

- **Stability proof** – ongoing work.

  - Suppose we have a connected component which is a leader-oriented DAG.
  - Suppose at some point, a link goes down and the resulting CC containing the leader is $G$.
  - As long as there are no further topology changes in $G$, no node in $G$ elects itself.

- Better characterization of situations in which the leader is not elected unnecessarily.

- Time and message complexity analysis.