Performance Comparison of Existing Leader Election Algorithms for Dynamic Networks

Plamen Ivanov and Jennifer Walter
Computer Science Department
Vassar College
{plivanov, jewalter}@vassar.edu

Abstract—Leader election is an important problem in distributed networks for any application that requires a distinguished node (the leader) to be selected during execution. We have written a discrete event simulator to emulate the topological changes of dynamic networks. Our simulator is designed to allow simulated networks to have a finite number of topology changes. In this paper we present the results of simulating and comparing the performance of two leader election algorithms presented in [4] and [2]. The algorithm presented in [4] is designed to work in an asynchronous dynamic network and to guarantee that once topology changes stop, each connected component of the network will have a unique leader. The authors of the algorithm described in [2] claim that a connected component will converge to a legitimate state within a finite amount of time even if topological changes occur during the convergence time. Both algorithms can tolerate multiple concurrent topology changes. We compare the two algorithms by running them on a set of randomly generated topology records and measuring relevant parameters such as number of messages generated and leaders elected. We also test the performance based on varied network conditions such as different number of nodes, different message delays, etc. We present some conclusions on the robustness and extent to which these algorithms meet their objectives.

Index Terms—leader election

I. INTRODUCTION

Leader election is an important primitive for distributed computing in dynamic mobile networks. Mobile networks are composed of a collection of dynamic computing devices. These devices usually communicate through a wireless medium. The typical structure of such a network does not have any centralized or fixed infrastructure to manage the network. Another key characteristic of dynamic mobile networks is that their topology changes as the participating nodes move. This mobility causes the frequent formation and loss of connections between nodes. Such a network model can be applied even to networks with nodes that do not move since wireless communication is subject to a lot of interference. Even wired networks can have a frequently changing topology if the connections between the nodes are unreliable or subject to interference.

The goal for leader election algorithms is to ensure the selection of a unique processor in a distributed system. More specifically, the algorithms ensure that each connected group of nodes selects a unique leader as efficiently as possible. The efficiency is determined by the time it takes to elect a new leader as well as the number of messages required for the successful election.

Proposed applications for leader election algorithms include: primary-backup approach to replication-based fault-tolerance, group communication systems, video conferencing, and multiplayer games.

II. RELATED WORK

The algorithms compared in this paper are both extensions of the work done by Malpani, et al. [5]. These authors present an algorithm that is shown to satisfy the correctness requirements for leader election when a single link failure occurs, after which the network quiesces. A harder problem is addressed in [4] and [2] because they require robust leader election in the presence of multiple concurrent topology changes. Other researchers [1] have compared the performance of various routing protocols on ad hoc wireless networks using mobile network simulators such as NS-2 [3]. In [2], the performance of the leader election algorithms of [5] and [2] are compared using GloMoSim [6]. We used an in-house discrete-event simulator that has similar functionality to these open-source simulation programs.

III. SYSTEM ASSUMPTIONS AND PROBLEM STATEMENT

Assumptions about the system that we make when when simulating the two algorithms:

- There are N nodes.
- Nodes have perfect clocks.
- Nodes communicate through message passing.
- Messages are only lost if they are in transit on a link that goes down.
- Messages are delivered in sending order over each link.
- Message delays are asynchronous or synchronous, depending on settings.
- Nodes move at a random constant speed in random directions.
- All movement stops at some point in time, allowing the network to quiesce.

The algorithms in [4] and [2] use a leader-oriented directed acyclic graph (DAG) to map the topology of the network and maintain a unique leader in each connected component of the network. A leader-oriented DAG has one node with only
incoming links (the leader), one or more nodes with only outgoing links and all other nodes have at least one outgoing link. This property guarantees that every node has a link that is pointing in the direction of the leader. The link’s direction is determined by a comparison between a set of variables in each node. This set is collectively referred to as the height tuple of a node. Height tuples are compared lexicographically and a link is directed from a higher to lower node height.

The algorithms maintain the leader oriented DAG by using link reversals and message passing as a response to changes in the topology and the heights of nodes.

The correctness requirement for a leader election algorithm on a dynamic network is as follows: Every connected component of the network eventually has a unique leader and the logical directions on the links form a leader-oriented DAG.

IV. OVERVIEW OF INGRAMS ALGORITHM [4]

In this section we have included a description of the variables used in the algorithm from [4] as well as a general description of how the algorithm works. The pseudo-code for the algorithm can be found in [4].

In this paper, variable $v$ of node $i$ will be indicated as $v_i$, when necessary for clarity. 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 neighbors.

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

1) $\tau$, a nonnegative timestamp that is either 0 or the time when this reference level was initiated
2) $oid$, a nonnegative value that is either 0 or the id of the node that started this reference level
3) $refl$, a bit that is set to 0 when the reference level is initiated and set to 1 when it is reflected
4) $\delta$, an integer that is set to ensure that links are directed appropriately to neighbors with the same reference level
5) $-lts$, 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 $(\tau, oid, refl)$ are referred to as the reference level, or RL, $(\tau, oid)$ alone are referred to as the reference level prefix, and $(-lts, lid)$ is referred to as the leader pair or LP. Nodes communicate over links during the algorithm execution by sending Update messages, where each message contains only the height tuple of the sending node. Nodes lexicographically compare the height tuple received in messages with their own height tuple to determine logical link direction (from higher to lower height).

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 node $i$ has received 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 $N$. If $i$ has not yet received a message from $j$, then the link is considered to be still forming, and $i$ stores the id of $j$ in its local variable forming; in this case, $j$ is not considered a neighbor of $i$ until $i$ receives an Update message from $j$.

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:

- forming, $i$ is empty
- $N_i$ contains the id of every node $j$ such that the vertices in $V$ corresponding to $i$ and $j$ are neighbors in $G$
- $height_i = (0, 0, 0, \delta, 0, -lts, i)$, where $-lts$ is the id of a fixed node in the graph, the current leader
- for each neighbor $j$ of $i$, $height_j[i] = height_j[j]$ (i.e., $i$ has accurate information about $j$’s height and vice versa.

Furthermore, at the initial leader $l$, $\delta_l = 0$, and the $\delta$ values of all the other nodes are positive integers such that every node has a directed path to $l$. (For instance, $\delta_i$ could be the distance in $G$ between $i$ and $l$.)

- $SINK_i$: This predicate is true when $i$ 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, regardless of the directions of those links.

In [4], the network is depicted as a DAG in which each bidirectional communication link points from a node with lexicographically higher height to another node with lexicographically 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 stores about its neighbors’ heights.

V. OVERVIEW OF DREHAB’S ALGORITHM [2]

In this section we provide a brief description of the algorithm presented in [2]. Similarly to [4], this algorithm also models the network as graph $G = (V, E)$, with $V$ corresponding to the set of nodes and $E$ to the set of communication links that are up.

Each node $i$ maintains a set of variables that form a 9-tuple with the following content:

1) $certain$: 1 if node has a path to leader, 0 otherwise.
2) $Te$: time at which the leader has started the creation of its DAG.
3) $lid$: the identifier of the node considered to be the leader.
4) $Tb$: beginning of the Reference Level Interval.
5) $Te$: end of the Reference Level Interval.
6) $oid$: 0 or the identifier of the node that started this reference level.
7) $refl$: 0 normally, 1 for reflected reference level.
8) $\delta$: distance from leader.
9) $id$: the identifier of a node.
The first three variables are referred to as the partition index. The last six variables are the height of the node. The tuple \((Tb_i, Te_i, oid_i, refl_i)\) is called the extended reference level (ERL). The triple \((Tb_i, oid_i, refl_i)\) is known as the reference level (RL). The variables \([Tb, Te]\) together form the reference level interval (RLI), which is the knowledge of node \(i\) about the time period during which the reference level is propagating.

A definition of the comparison criteria between heights is given in [2]. In brief, it works by determining the relationship between the RLIs in each height to determine the direction of each link.

A definition of the comparison criteria between partition indices from [2] is:

\[
P_i > P_j \equiv ((\text{Certain}_i > \text{Certain}_j) \vee (\text{Certain}_i = \text{Certain}_j \wedge ((Tc_j, \text{lid}_j) > (Tc_i, \text{lid}_i)))
\]

The formation of a link will cause the node at both endpoints to send a LinkInfo message to compare its PI with the one of its new neighbor. The neighbor’s PI is greater, node \(i\) adopts the neighbor’s height and propagates it with a CreateDag message, otherwise it does nothing. The same procedure is used to handle the reception of a CreateDag message that contains a different lid from the one in the current node.

A node that receives a CreateWeakDag message containing a different lid will check if the sender’s old lid is the same as the current node’s lid. If that is true, the node will broadcast a FakeLeader message containing identification of the fake leader. Otherwise the node propagates the CreateWeakDag messages or, if appropriate, elects itself as a leader and broadcasts a CreateDag message.

When a node loses its last outgoing link and it is uncertain or it has just received a Failure message and no longer has any outgoing links it calls a function called Handle-UncertainReverse. The HandleUncertainReverse procedure takes different actions, depending on the reference levels, the reflect bit, the lid, and the oid at the endpoints of a link. It detects partitions in the network using the same technique as is used in [4] and [5], and ensures that the lid with the oldest start time and with certain bit set to 1 is favored for propagation. This procedure uses a complex set of comparisons to determine the type of message propagated by a node. The procedure causes a node to broadcast either a Failure, CreateWeakDag, or CreateDag message.

For a more detailed overview and pseudocode refer to [2].

VI. Simulator

We implemented the algorithms in [4] and [2] using our own discrete event simulator. The simulator is written in Java and models the node mobility, the handling of link failures and formations by an underlying protocol, dynamic topology, and synchronous or asynchronous message delays. The simulator generates and records topology changes and replays them using different algorithms. The simulator can have varying network settings such as different message delays (synchronous or asynchronous), different node speed, and different node transmission range. The output of the simulator includes all the information needed to debug and analyze the performance of the running algorithm.

When running the algorithms we start with an initial configuration in which all the nodes are evenly spaced in a vertical straight line in the middle of the network area. The nodes are initialized as part of a leader-oriented DAG. The movement of the nodes begins from this initial configuration and proceeds to simulate module movement in a random manner. The nodes move for 100 time steps and then all movement halts. The simulation is run on a 600 by 600 pixel grid. The nodes have a wireless range of 100 pixels. These parameters can be varied, as can the number of nodes. However, varying the number of nodes is sufficient to give us an idea of how different types of network topologies work. For example, a network with 10 nodes running on an area of this size can be classified as sparse and it will most likely have several small disjoint components at the end of the simulation. Similarly, a network with 40 nodes can be classified as dense and it will most likely have a few large components with high degree of node interconnectivity at the end of the simulation.

VII. Results and Conclusions

We hypothesize that an important idea from the algorithm presented in [2] can potentially be applied to the algorithm presented in [4]. This idea pertains to which computation is
stopped from propagation. Currently in [4], the algorithm stops the propagation of the older leader. This could be changed so that older leaders are sometimes chosen over the newer ones. However, there is a potential problem with selecting an older leader, as it is less likely to be up-to-date as topology changes in the network are still occurring.

During the implementation of the algorithm in [2] we discovered that the algorithm will incur significant calculation costs in the process of comparing the heights of nodes. These comparisons will require sorting and comparison of time intervals, which will have a complexity of $O(n\log n)$. This can lead to increases in computational complexity if the network topology is very dense (there are many neighbors in range) or if there are many concurrent topological changes leading to many comparisons between the height of a node and its neighbors. We also discovered a case that may not be covered by the algorithm of [2] or may have been omitted by the authors. Action 4, as described in [2], is executed when a node has lost its last outgoing link after the reception of a Failure message. We discovered that when the first condition is not met, Action 4 is not executed, and the originator of the Failure message will not be informed of its neighbor’s height change and will remain in an uncertain state. This could result in a poorly formatted DAG and can potentially compromise the results of the algorithm. We have also encountered other issues such as network components that are left without a leader after the topology changes cease and several cases of infinite message passing. We still need to conduct more tests and analysis to determine if these problems are caused by our implementation or by problems with the algorithm in [2].

A preliminary comparison between the performance of the two algorithms on the same topology records shows that the algorithm presented in [2] is only marginally better with regards to the number of messages it uses to find a unique leader for each connected network component. Both algorithms, on average, take roughly the same amount of time to find unique leaders after the topology changes have ceased.

In light of these results, we conjecture that the two algorithms have similar performance and that, even though the algorithm of Derhab and Badache uses less messages than the algorithm of Ingram, et al., it does so at the expense of higher computational cost and higher algorithm complexity, and therefore may not be an efficient nor correct solution to the leader election problem on distributed networks with highly asynchronous message delays.

We also feel that the algorithm presented in [2] is described ambiguously at places and there are some omissions, such as the LinkInfo message.

VIII. Future Work

Future work can focus on a more thorough investigation of the performance and computational complexity of the two algorithms, particularly to ensure that our simulation of [2] is an accurate implementation of their algorithm. Work can also be done on testing a modification of the algorithm in [4] that would give priority to leaders that are older.

REFERENCES