Topology Aware Leader Election Algorithm for Dynamic Networks

Arnaud Favier, Nicolas Guittonneau, Luciana Arantes, Anne Fladenmuller, Jonathan Lejeune, Pierre Sens

Topology Aware Leader Election Algorithm for Dynamic Networks

Arnaud Favier*, Nicolas Guittonneau*, Luciana Arantes*,
Anne Fladenmuller, Jonathan Lejeune* and Pierre Sens*
Inria*, LIP6, CNRS
Sorbonne University, Paris, France
Email: {firstname.lastname}@lip6.fr

***Abstract*—This paper proposes an algorithm that eventually elects a leader for each connected component of a dynamic network where nodes can move or fail by crash. A node only communicates with nodes in its transmission range and locally keeps a global view, denoted topological knowledge, of the communication graph of the network and its dynamic evolution. Every change in the topology or in nodes membership is detected by one or more nodes and propagated over the network, updating thus the topological knowledge of the nodes. As the choice of the leader has an impact on the performance of applications that use an eventual leader election service, our algorithm, thanks to nodes topological knowledge, exploits the closeness centrality as the criterion for electing a leader. Experiments were conducted on top of PeerSim simulator [1], comparing our algorithm to a representative flooding algorithm. Performance results show that our algorithm outperforms the flooding one when considering leader choice stability, number of messages, and average distance to the leader.**

***Index Terms*—distributed systems, dynamic networks, connected graph, leader election, topology, centrality**

I. Introduction

Eventual leader election is an essential service for many reliable applications that require coordination actions on top of asynchronous fail-prone distributed systems. Also called the Ω failure detector [2], an eventual leader election algorithm provides a primitive, denoted *Leader()* that, when invoked by a process, returns the identity of a process of the system and ensures that there exists a time after which it always returns the identity of the same nonfaulty process.

Ω allows to solve the consensus problem with the weakest assumptions on process failures considering a majority of correct processes. Consensus is one of the most fundamental problems of distributed computing [3]. Paxos [4] and Bitcoin [5] (as many other blockchains) are well-known consensus algorithms that exploit eventual leader election service, while many examples of problems in the literature, such as state machine replication or atomic broadcast, as well as many distributed systems (e.g., Kubernetes [6], ZooKeeper [7], etc.) use consensus.

We are particularly interested in providing an eventual leader election algorithm for mobile ad hoc network (MANET), a decentralized dynamic network where nodes can move and communicate by transmitting messages over wireless links. Only nodes within the transmission range of each other can communicate directly with one another. Then, they can retransmit the message to other nodes. Thus, one or more intermediate nodes may act as relays. In such a network, the communication graph evolves: nodes can join or leave the system, fail, and recover at runtime. Due to this dynamic, the network may be partitioned, i.e., composed of two or more connected components. Initially, nodes have no knowledge of the system membership, learning about it during execution time.

Many works have proposed leader election algorithms in both static [3], [8]–[10], and dynamic [11]–[18] distributed systems. However, among the latter, only a few of them take into account the above highly dynamic characteristics and membership lack of knowledge of MANETs. Furthermore, in the majority of these algorithms, the choice of the leader is based on a beforehand criterion such as the lowest or the highest nonfaulty process identity. We argue that a criterion should take into account the impact that the choice of the leader may have on the performance of algorithms that use the leader election service. Performance-related criteria are, for instance, nodes remaining battery life, nodes computation capabilities, nodes topological position (e.g., the minimum average distance from a node to all other nodes), etc. Similarly to [12], we denote *the most valued node* the one that best meets the performance-related criterion in question and, therefore, should be chosen as the leader. Hence, we believe that (1) an election algorithm for mobile networks must tolerate arbitrary, concurrent topology changes, and should eventually terminate electing a unique leader per connected component of the network [19]; (2) for the sake of performance, an elected leader should be *the most valued node* among all the nodes within its connected component.

The contribution of this paper is an eventual leader election algorithm based on *the most valued node* criterion and whose nodes have a global knowledge of the communication graph and its dynamic evolution, denoted *topological knowledge*. Our algorithm progressively builds and maintains a local knowledge of the connected graph. It relies only on broadcasts within node transmission ranges and does not require any election communication phase: with both its current *topological knowledge* and choosing *the most valued node*, each node can directly deduce at any moment which node is the current leader. In particular, the *topological knowledge* allows for the computation of the closeness centrality and the election of

per component central located leaders, which, thus, efficiently spread information across nodes of the component. If the problem of discovering a topology has been studied in various contexts [20], [21], it is to the best of our knowledge the first time this approach is used to elect an eventual leader.

We should point out that even if we target MANET, our algorithm has been designed for generic mobile dynamic networks.

We have compared the performance of our *Topology Aware* algorithm with a variant of the leader election algorithm of Vasudevan *et al.* [12], because their work is a good example of a typical flooding algorithm and is strongly referenced in the literature [13], [14], [18], [22]. Results of experiments conducted on the PeerSim simulator [1] confirms the effectiveness of our algorithm and that it outperforms the latter.

The rest of the paper is organized as follows. Section II presents research works related to the leader election problem, Section III explains the chosen model and assumptions, Section IV describes the algorithm, Section V discusses performance results and, finally, a conclusion and future work are given in Section VI.

II. Related work

There exists an extensive literature related to leader election and Ω failure detector (eventual leader election) for both static and dynamic systems.

Several existing leader election algorithms on static networks consider that nodes are logically organized in a ring [8], [9], [23], while others [3], [24], [25] dynamically build spanning trees for exchanging information, taking a decision about the elected node, and announcing the new leader.

The problem of coping with dynamics in order to implement a leader election algorithm in a dynamic system, more specifically in MANET, has been largely studied by many authors.

Malpani *et al.* have a solution [11] that builds an acyclic graph where each node has a direct path to the leader. However, contrarily to our algorithm, it does not provide enough information to elect a central leader by choosing its topological position in the network. The election criterion used by the algorithm is the lowest value of a movement-based counter. The algorithm was extended by [22] and [14] where the election of a new leader requires three diffusion waves over the network: two waves to search for a potential leader and one confirmation wave to spread the election result over the network. These waves induce a high number of message exchanges which slow down the election process. Furthermore, these algorithms do not use a broadcast communication model like our algorithm, but neighbors send point-to-point messages, increasing the number of messages sent. A final remark is that the algorithm in [22] relies on a global time, assuming that nodes have perfectly synchronized clocks.

By using the highest identifier node as the criterion to elect a node, the algorithm of Rahman *et al.* [13] is based on a spanning tree construction which requires heartbeat, probe, reply, and acknowledgment messages. The number of exchanged messages is, thus, very high, overloading the network and does not provide enough information to choose a centrally positioned leader.

Kim *et al.* [18] exploit a wave algorithm to build a spanning tree to elect a centrally positioned leader, according to the average depth of nodes in the tree, and compare different centrality metrics. However, the central leader is not always optimal, depending on the initiator node of the election, and mobility of nodes is not studied.

By returning a leader identity only when it is accepted by all nodes in the network, the algorithm of Vasudevan *et al.* [12] solves, in fact, a stronger problem than the eventual leader election. The election criterion of the algorithm is static and corresponds to the highest arbitrary value assigned to a node. The wave algorithm builds a spanning tree where each node sends back to its parent the identifier of the node having the highest value in its subtree. The authors suggest the idea of electing as the leader the node with the minimum average distance to other nodes, but as it is presented, their algorithm does not permit such an election.

Since it is impossible to solve an eventual leader election (Ω failure detector) in pure asynchronous distributed systems prone to process failures [26], additional assumptions are necessary to implement it. To this end, the majority of works use one of the two orthogonal approaches: timer-based, which supposes that links are eventually timely [17], [27], [28], or message exchange pattern-based [15], [29], i.e., a query-response mechanism where eventually there is a link whose responses are always received before the others. A punishment mechanism is usually used as a criterion for electing the leader. In order to tolerate the dynamics of mobile networks, some solutions are either based on the message pattern approach with broadcasts to neighbor nodes and network stability assumptions [15], or on the construction of dynamic graphs, considering that eventually links are timely or the network topology stop changing [17], [30]. We did not compare our solution with these algorithms since their model, synchrony, stability, or mobility assumptions are different from ours.

III. System model and assumptions

In this section, we define the system model and assumptions for our eventual leader election algorithm. We assume an upper bound ϕ on the time a process takes to execute a step and eventual timely links, i.e., an upper bound δ on the transmission delay. However, both upper bounds ϕ and δ are unknown, so the system is partially synchronous. As we consider one process per node, the words node and process are interchangeable.

Node states and failures: Every node always follows the specification of the algorithm, until a potential fail. It is considered *correct* if it never fails and never leaves the system during the whole execution. Otherwise, it is considered *faulty*, until it comes back in the system.

A node can fail by crashing and a failed node can recover, joining the system again with the same *unique identifier*, i.e., two nodes cannot have the same *identifier*. Therefore, a node keeps its *identifier* regardless of its state. However, the node

does not recover its state neither the knowledge of the network membership and, thus, is initialized again.

Communication graph: As the system is composed of mobile nodes, it can be seen as an undirected graph, where vertices are nodes and edges represent the possibility of two nodes to communicate directly between each other (1-hop nodes). Two nodes can communicate directly if they are in the transmission range of each other, i.e., the emission range of the sending node intersects the reception range of the receiver node. In our system, the emission range is the same as the reception range, so if node i can communicate with node j, node j can also communicate with node i (bidirectional links).

In the graph, adjacent vertices of a node are called *neighbors* and the set of adjacent vertices represents the *neighborhood of a node*. A given node belongs to a connected graph formed by its neighbors, neighbors of its neighbors and so on, that we called a *connected component*.

If the system is divided into two (or more) different connected components due to movement or failure of nodes, each connected component is considered to be a fully-fledged network in itself, and therefore, eventually elects one leader. Both partial synchrony and algorithm ensure that, regardless of topology changes, if the latter cease, each connected component will eventually elect a single leader.

Channels: Nodes communicate by broadcast. All neighbors of the sender node receive the broadcast message. Our algorithm only uses broadcast communications on a fixed Wi-Fi channel selected beforehand. We assume reliable communication channels, possible in a wireless network where the MAC layer reliably delivers broadcast messages, even in presence of signal attenuation, collision, and interference. Otherwise, the transmitter is faulty or out of the neighborhood. There is no assumption about message order, so messages can be delivered out of order.

Membership and nodes identity: The number of nodes is unknown. Each node initially knows its own *identifier* which is unique in the system. A *probe system* is used which allows detection of neighbor nodes. By receiving messages from its neighbors, a node gets knowledge of the membership of the network.

IV. Leader Election Algorithm

In our algorithm, every node keeps a topological knowledge of the connected component to which it belongs. The algorithm builds this component knowledge during node connections and disconnections (triggered by the *probe system*), and maintains it by sending either the full knowledge (called *known*) to new neighbors, or partial modifications (called *updates*) periodically to its neighbors. The pseudo-code of the *Topology Aware* algorithm for a node i is given in Algorithm 1.

A. Data structures, variables, and messages

The two following data structures are used by node i:

- **view** (line 1) is composed of two elements: a logical *clock* [31] value only incremented by i, and a set of *identifiers* representing neighbors of i.
- **updt** (*update*, line 2) represents additions or deletions of neighbors of i. It consists of the identifier of the *source* node that has detected membership changes in its neighborhood, a set of *added* nodes (new connected neighbors), a set of *removed* nodes (new disconnected neighbors), the logical clock value of *source* node before the modifications (*old_clk*) and its logical clock value after the modifications (*new_clk*). This structure allows us to track new modifications for a given period of time.

Each node i maintains three local variables (line 3):

- **known**$_i$ (line 4): the current topological knowledge of the connected component of node i (including itself), implemented as a map of *view* indexed by node *identifier*, i.e., an entry for each node.
- **updates**$_i$ (line 5): a list of *updt* (updates) is periodically sent, used to update the knowledge of nodes by propagating modifications of the knowledge of i (new connections and disconnections), without sending the full knowledge, avoiding therefore, to send redundant information already received by neighbors.
- **pending**$_i$ (line 6): a list of pending *updates* that cannot be applied by i at the time of first reception, but applied when new information is received thereafter.

During communication, the variables **known** and **updates** are exchanged through two distinct types of messages identified by the name of the variables.

B. Algorithm

The pseudocode of the algorithm given in *Algorithm 1* is described in the following.

1) Initialization: At the beginning, node i initializes its knowledge with its own identifier (i) and its logical clock set to 0 (lines 7 to 11).

2) Periodic updates task: Periodically (every Δ milliseconds), node i broadcasts its new *updates* list if not empty, and then set it to empty (lines 12 to 16).

3) Connection: When a new node j appears in the transmission range of node i, it is detected thanks to the *probe system* and the *Connection* method is triggered (line 20). Node j is considered as a new neighbor and is added to the knowledge of node i (line 21). As the latter has been updated, the logical clock of node i is incremented (line 22). Then, both nodes broadcast their current knowledge to share information about their component with each other, and also to inform neighbors about the new node. Therefore, node i broadcasts its *known* map (line 23) to its neighbors: node j then acquire topological knowledge about the component, while the other neighbors of i are informed about the new connection with j. Same process is executed by j in regard to node i and its neighbors.

4) Disconnection: When a certain number (γ) of probes from node j are not received by node i, node j is considered disconnected by i (line 24). An *update* structure is created with the following information (line 25):

- the identifier of node i, considered as the source of the modification;

Algorithm 1: The *Topology Aware* eventual leader election algorithm for a node i

```
1  Typedef view: ⟨clk: int, neigh: set(id)⟩
2  Typedef updt: ⟨src: int, add: set(id), rmv: set(id),
     old_clk: int, new_clk: int⟩
3  Local variables of node i:
4      known: map(key: id, value: view)
5      updates: list(updt)
6      pending: list(updt)
7  Initialization of node i:
8      known[i].neigh ← {i}
9      known[i].clk ← 0
10     updates ← ∅
11     pending ← ∅
12 Periodic Updates Task:
13     if updates ≠ ∅ then
14         Broadcast (updates)
15         updates ← ∅
16     Wait Δ milliseconds
17 Invocation of Leader():
18     component ← Reachable (known[i])
19     return Max (ClosenessCentrality (component))
20 Connection of node j:
21     known[i].neigh ← known[i].neigh ∪ {j}
22     known[i].clk ← known[i].clk + 1
23     Broadcast (known)
24 Disconnection of node j:
25     updates ← updates ∪ {⟨i, ∅, {j}, known[i].clk,
         known[i].clk + 1⟩}
26     known[i].neigh ← known[i].neigh \ {j}
27     known[i].clk ← known[i].clk + 1
28 Receive known_j from node j:
29     ∀ ⟨id, view⟩ ∈ known_j do
30         if ∄ ⟨id, –⟩ ∈ known then
31             updates ← updates ∪ {⟨id, view.neigh, –, 0,
                 view.clk⟩}
32             known[id] ← ⟨view.clk, view.neigh⟩
33         else if view.clk > known[id].clk then
34             add ← view.neigh \ known[id].neigh
35             rmv ← known[id].neigh \ view.neigh
36             updates ← updates ∪ {⟨id, add, rmv,
                 known[id].clk, view.clk⟩}
37             known[id] ← ⟨view.clk, view.neigh⟩
38     PendingUpdates()
39 Receive updates_j from node j:
40     ∀ updt⟨src, add, rmv, old_clk, new_clk⟩ ∈ updates_j
         do
41         if ∄ ⟨src, –⟩ ∈ known then
42             if old_clk = 0 then
43                 known[src] ← ⟨new_clk, add⟩
44                 updates ← updates ∪ updt
45             else
46                 pending ← pending ∪ updt
47         else if old_clk = known[src].clk then
48             known[src].neigh ← (known[src].neigh ∪
                 add) \ rmv
49             known[src].clk ← new_clk
50             updates ← updates ∪ updt
51         else if old_clk > known[src].clk then
52             pending ← pending ∪ updt
53     PendingUpdates()
54 Invocation of PendingUpdates():
55     ∀ updt⟨src, add, rmv, old_clk, new_clk⟩ ∈ pending
         do
56         if old_clk = 0 then
57             if ∄ ⟨src, –⟩ ∈ known then
58                 known[src] ← ⟨new_clk, add⟩
59                 pending ← pending \ updt
60         else if old_clk = known[src].clk then
61             known[src].neigh ← (known[src].neigh ∪
                 add) \ rmv
62             known[src].clk ← new_clk
63             pending ← pending \ updt
64         if old_clk < known[src].clk then
65             pending ← pending \ updt
```

- an empty value for the *add* set (no new connection);
- the identifier of the disconnected node j for the removed set;
- the current clock of node i;
- the new clock, whose value is equal to the clock value of node i increased by 1.

This tuple is added to the list of *updates* to be propagated later by the periodic updates task. Node j is then removed from the knowledge of node i (line 26) and the clock of node i is incremented (line 27).

5) Knowledge reception: When node i receives the *known* map of node j (line 28), it checks each node id included in known$_j$ (line 29). If id is a new node for it (line 30), node i creates an *update* containing the neighbors of node id with an old clock valued at 0, meaning that all neighbors of id are in the *add* set (line 31). The *update* is added to the list of *updates* to be propagated later by the periodic updates task. Then, the clock and neighbors of id are added to the knowledge of node i (line 32).

If id is known by i and the clock value of id is greater than the clock value known by node i for id (line 33), it means that id made some connections and/or disconnections of which node i is not aware. Hence, node i creates an *update* and computes the *add* set (line 34), which will be composed

of the new neighbors for i that id informed in *view*, minus the neighbors of id which i already knew. The result represents new neighbors of id since its last received view. Then, node i computes the removed nodes of the *update*, by removing the received neighbors from the known neighbors of id (line 35), which represents disconnections since the last received view. The value of the *old clock* in the *update* is set to the clock value of id in the knowledge of node i, and the new clock value is set to the value of the received clock (line 36).

The *update* is added to the list of *updates* to be propagated later by the periodic updates task (line 12), and eventually, thanks to their previous knowledge and update exchanges, neighbors of node i will have the same knowledge as node i with identical clocks, thus, they will be able to apply this new update in their respective knowledge.

Finally, the clock value and neighbor identifiers of id are added to the knowledge of i (line 37) and the *PendingUpdates* method is called to apply previously received updates (line 38).

6) Updates reception: When node i receives *updates* from node j (line 39), each update adds or removes neighbors of a *source* node src (line 40). Following the *old clock* value, an update can be applied, saved in the *pending* list to be applied later, or discarded.

If the *old clock* is equal to 0 (line 42), the update contains all the neighbors of node src (see *Knowledge reception* IV-B5), and the update is applied (line 43) if node i does not have any information about node src (line 41). If the *old clock* is equal to the clock of node src in the current knowledge of node i (line 47), the update corresponds to new information. The update is then applied, i.e., neighbors are updated (line 48) as well as the clock (line 49). In both cases, the updates are added to the updates set of node i (lines 44 and 50), to be propagated later to neighbors through the periodic updates task.

An update cannot be applied when it is more recent than other updates not yet received by i, which should have applied before the former. This out of order update receptions might happen if the component contains cycles, i.e., when the *old clock* is greater than the clock of src in the knowledge of i (line 51), or when node i does not have information about node src and the *old clock* is greater than 0 (line 45). In those cases, node i saves the update in a *pending updates* list (lines 46 and 52), and will try to apply it in the future, after new updates will be received (line 53).

7) Pending updates: PendingUpdates (line 54) checks the updates that can be applied in the *pending* list (line 55). To reduce message exchanges and improve performance, updates that cannot be applied when first received are saved, and the algorithm tries to apply them after new information is received.

When an update is applied (i.e., the knowledge of i changes, lines 58 and 61-62), the latter is removed from the *pending* list (lines 59 and 63). If the clock value of the current knowledge is greater than the *old clock* value of the *update* (line 64), the *update* is also removed for the *pending* list (line 65), meaning that node i receives a knowledge or updates from a node with more recent information.

8) Leader(): When a process running on node i requires a leader, it calls the local *Leader* method (line 17) which computes and returns, based on the knowledge of i, the best leader according to the closeness centrality (line 19). We use the closeness centrality rather than the betweenness centrality, because it is faster computed and requires fewer computational steps, so use less energy from the mobile nodes. The closeness of a node is the inverse of the sum of all shortest paths to other nodes, characterizing the ability of a node to spread information over the graph. We use the closeness centrality formula defined by Alex Bavelas [32] for a node x as :

$$C(x) = \frac{1}{\sum_y d(y,x)}$$

where $d(y,x)$ is the shortest path between nodes y and x. In order to compute the closeness centrality, node i, starting from itself, get the set of reachable nodes according to its topological knowledge of the component (line 18). Then, for each reachable node, it computes the shortest distance between this node and the other reachable ones, obtains the closeness centrality, and deduces the most central node as the leader (line 19). The highest node identifier is used to break ties among identical centrality values.

If all nodes of the component have the same knowledge of the topology, the *Leader()* call returns the same leader node to all of them. Otherwise, it may return different leaders for distinct nodes. However, if topology changes cease, our algorithm ensures that all nodes of a connected component will eventually have the same topology knowledge and, therefore, will have the same leader node [14].

V. Results

The objective of the experiments is to compare our *Topology Aware* algorithm with a flooding algorithm.

A. Simulation environment

We have conducted evaluation experiments on PeerSim [1], a Java peer-to-peer network simulator. Each experiment lasts 30 minutes, with a simulated unit of time corresponding to one millisecond, and simulates 60 nodes placed in a 900 $\times$ 900 meters obstacle-free area. Message sending latency follows a Poisson distribution with parameter $\lambda = 10$.

B. Mobility models

We consider two different mobility patterns for the experiments: the *random waypoint* and a periodic disc positioning pattern around a *single point of interest*. For both mobility patterns, the minimum node speed is set to 5 m/s and the maximum node speed is set to 15 m/s. The chosen speed follows a uniform distribution between the minimum and the maximum speed.

1) Random waypoint: Nodes are randomly placed in the area and move according to the *Random Waypoint* mobility model [33]. Nodes wait 10 seconds before choosing the next random destination. We point out that this mobility pattern is largely used in the literature [12], [33].

2) Periodic single point of interest: First, nodes are placed to create concentric circles whose center is the same unique point of interest, thus, composing a disc, like the one shown in Figure 1, such that there exists a path between any two nodes in the disc.

After 10 seconds, nodes start moving to a randomly chosen destination. Once reached, nodes wait 10 seconds before going back to their initial position in the circle, waiting for each other nodes to reach its initial position. Note that nodes can wait a long time for the other nodes to return to their initial position in the disc, due to various node speeds. When every node is at its initial circle position (the disc is reconstructed again), they wait 10 seconds before moving to another random destination, and repeat this behavior continually until the end of the experiment.

The shape of the disc is independent of the node transmission range, i.e., it always looks like the one in Figure 1 regardless of the diameter of the transmission range. This pattern could model user activities, where nodes represent people visiting regularly just a few places [34].

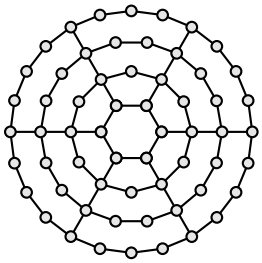


Figure 1. Single point of interest disc

C. Algorithms

We compared two versions of our *Topology Aware* algorithm with a variant of the Vasudevan et al. algorithm [12], since it is representative of flooding algorithms. Vasudevan et al. algorithm returns $\perp$ during the election phase, if the leader has not been elected yet. In order to be fairly comparable with our algorithm *Topology Aware*, we have considered a variant of this algorithm that never returns $\perp$, but a possible current leader. Indeed, in our algorithm, a correct node is always able to render a leader according to its topological knowledge. We denoted this variant *Flooding* because each node periodically broadcasts *leader messages* informing its current leader to neighbor nodes.

Note that *Flooding* is also a variant of the *OptFloodMax* algorithm that Nancy A. Lynch introduced in her book *Distributed Systems* [10]. In *OptFloodMax*, processes send their unique identifier to their neighbors, whenever a process obtains a new maximum unique identifier (which will eventually be elected as the leader).

We adapted the *Flooding* algorithm for MANET assuming an underlying *probe system* that detects connections and disconnections. Exchanged *leader messages* contain the node *identifier* and an election criterion, called *value*. The leader is the node with the highest *value*. It periodically broadcasts *leader messages* to neighbor nodes every α milliseconds, and each node forwards this information to neighbors. When the leader fails, it stops sending *leader messages*, and after a non-reception of β *leader messages* from the leader, nodes trigger a new election by setting themselves as their own new leader. Thus, new *leader messages* from different nodes are propagated, and eventually, the node with the highest value is elected.

In order to elect a leader with good local connectivity, we consider that the *value* is equal to the number of direct neighbors of a node, equivalent to node degree in a graph, which is updated at each topology change, thanks to the probe system (new connection or disconnection). The highest node *identifier* is used to break ties among identical *values*. We denote this algorithm *Flooding Degree*.

For fairness comparison with *Flooding Degree*, we also implemented a variant of the *Topology Aware* algorithm, called *Topology Aware Degree*, which uses the node degree as the election criterion. The version presented in Section IV-B8 using the closeness centrality as an election criterion is called *Topology Aware Closeness*.

D. Algorithms settings

In the *Flooding Degree* algorithm, nodes send messages every $\alpha = 250$ milliseconds. Every node triggers a new election if $\beta = 1$ message from the current leader was not received after a timeout of 300 milliseconds.

In both *Topology Aware* versions, updates are kept in a list acting as a buffer, before being sent every Δ milliseconds. The value of Δ is based on the transmission range (on x-axis of figures), considering that the larger the transmission range, the higher number of nodes potentially reached. Thus, to avoid burst effect after topology changes, the value of Δ is computed according to the following formula:

$$\Delta = 70 \times log_{10}(range) - 60$$

With this formula, Δ has a low value for small ranges and increases proportionally to the size of the range, with a fast increase for low ranges and a slow increase for higher ranges. This aims to transmit information quickly on smaller components, but more slowly on larger components due to the high dynamics of mobile nodes induced by larger transmission ranges.

When a *probe* message is received by node i, the *Connection* method of the algorithm is triggered. A *probe* message contains the unique identifier of the node and is sent every $\tau = 400$ milliseconds. If $\gamma = 1$ probe message from node j is not received after a timeout of 450 milliseconds, node i considers node j as out of range and trigger the *Disconnection* method.

E. Metrics

We considered the following three metrics:

1) Instability: it is the percentage of average time during which a node takes as the leader a node that is not equal to the eventually unique elected leader of the component. The latter

is computed by an oracle based on nodes degree for *Flooding Degree* and *Topology Aware Degree*, or closeness centrality for *Topology Aware Closeness*. *Instability* at time t is computed by the following formula:

$$Instability_t = \frac{\sum_{i=0}^{N} \begin{cases} 0, leader(i) = oracle(i) \\ 1, leader(i) \neq oracle(i) \end{cases}}{N} \times 100$$

where N is the number of nodes in the system and i the node identifier. Then, we compute the *Instability* over the whole time of an experiment, which is the average $Instability_t$ $\forall t$, from $t = 0$ to the end of the experiment.

2) Number of messages sent per second: it is the average of the total number of messages sent per second. This metric does not consider *probe* messages, since the same number of *probes* is sent for all algorithms every τ milliseconds.

3) The longest leader path relative to the component diameter: This metric characterizes how fast a leader can reach nodes of its component. First, we compute the *longest path* of all shortest paths from every node of the component to their current leader. Then, since it depends on the number of nodes in the component, we divide the longest path by the diameter of the component.

F. Performance

The goal is to compare the performance of *Flooding Degree* algorithm with the *Degree* and *Closeness* versions of our *Topology Aware* algorithm, for different diameters of transmission range that vary from 10 meters to 200 meters. Note that, there is a strong correlation between the transmission range and network connectivity, i.e., the number of components in the system.

1) Instability: The random waypoint pattern in Figure 2 shows that the instability percentage of Flooding Degree and both Topology Aware versions varies according to the transmission range, with a stabilization starting at a transmission range of 120 meters when the majority of nodes are in a few large connected components. However, nodes in Flooding Degree spend on average 55% more time with the wrong leader than nodes in Topology Aware Degree. We observe that the Closeness version of Topology Aware is slightly less stable than the Degree version.

For the periodic single point of interest pattern, compared to Topology Aware Degree, Flooding Degree spends on average 82% more time with a wrong leader, i.e., 1.5 times more than the previous mobility pattern. In Flooding Degree, leader unavailability is progressively detected by all nodes of the component, upon expiration of *leader messages* timeout. In this case, each node of the component triggers a new election by setting itself as the leader, and starts broadcasting *leader messages*. However, some nodes located further away from the old leader might still receive, from their neighbors, *leader messages* related to this old leader and, thus, will take more time to start a new election. Furthermore, the greater the number of nodes in the component, the higher the spreading of *leader message* (as we can observe in Figure 2).

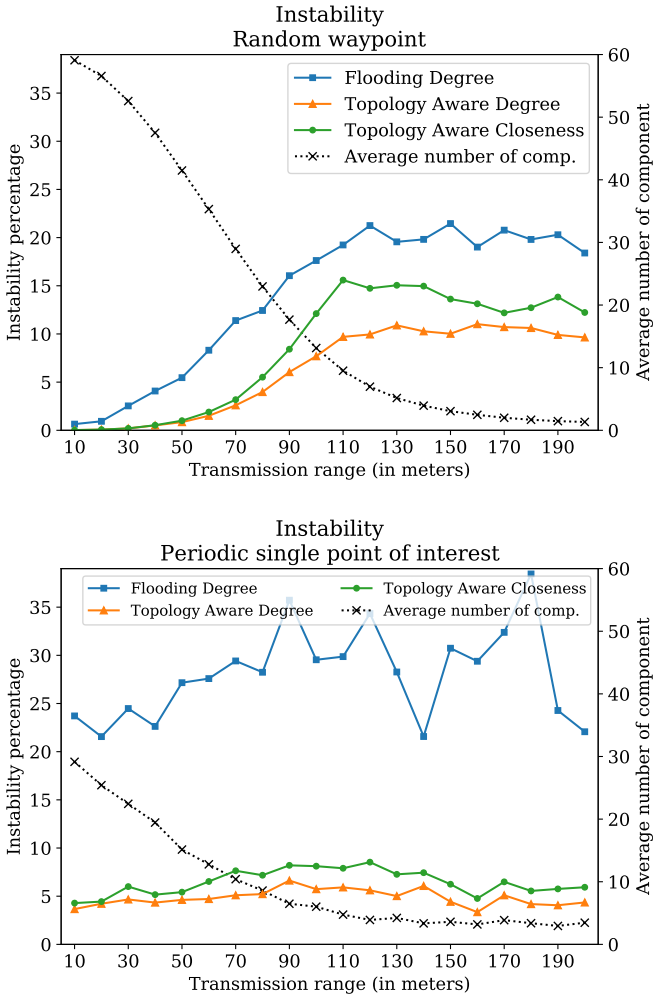


Figure 2. Instability percentage

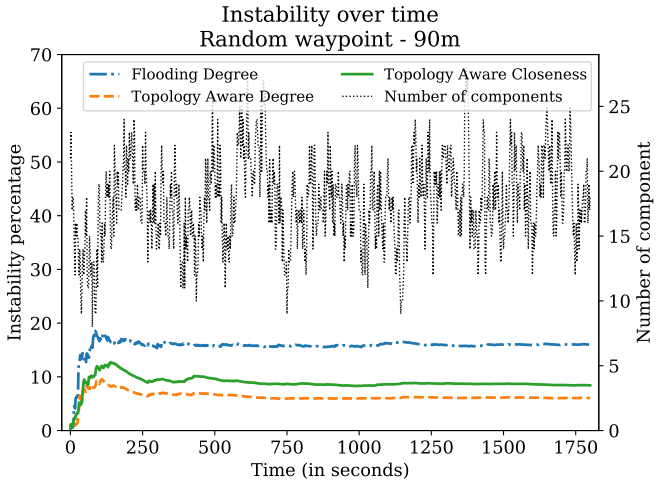
In Flooding Degree, the spreading of *leader messages* from all the nodes that have started a new election slows down the election convergence and thus increases the average stability. On the other hand, both Topology Aware versions only need to spread updates to each node.

The two versions of Topology Aware are more stable than Flooding Degree, especially in large components where there is low nodes movement, because it needs fewer steps to elect a new leader once its topological knowledge is built.

Instability over time at 90 meters: In Figure 3, we show the evolution of instability over time, considering a transmission range of 90 meters which is quite realistic. The instability at time t is the average cumulative instability from time 0 to time t, and the right y-axis is the exact number of components at time t.

As shown on top of Figure 3, the random waypoint mobility pattern has on average 18 connected components, and Topology Aware Degree is on average 62% more stable than Flooding Degree.

In the periodic single point of interest pattern at the bottom of Figure 3, nodes are gathering at their initial position on



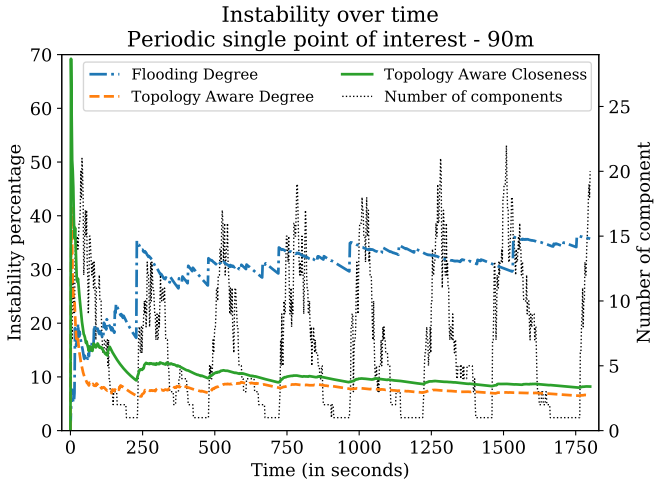
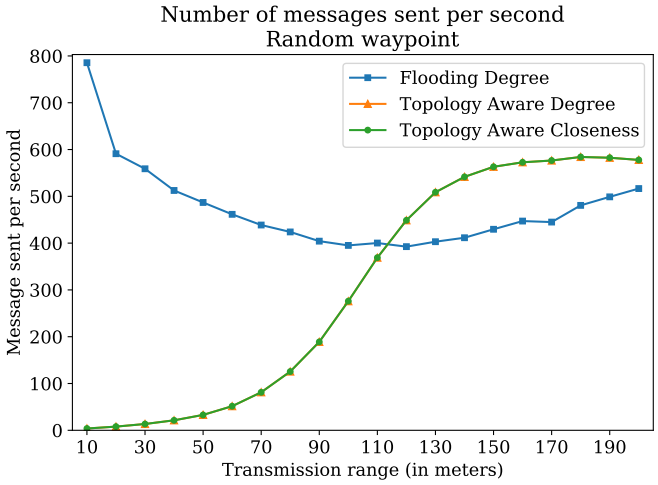


Figure 3. Evolution of instability at a transmission range of 90 meters



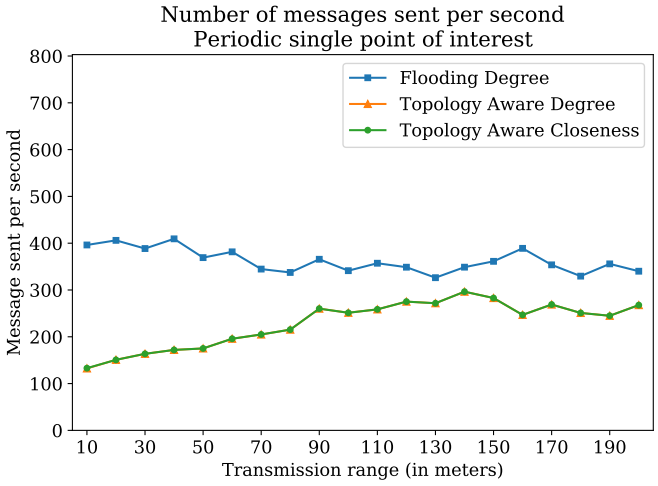


Figure 4. Number of messages sent per second

the disc in Figure 1, creating a unique connected component, then moving to random positions, also having an average maximum of 18 components. At the beginning of the experiment, nodes in both Topology Aware versions exchange messages to build their knowledge, inducing many leader errors, hence, an instability rate of 41% for the Degree version and 69% for the Closeness one. However, Flooding Degree quickly finds the correct leader, thus, showing low instability. After the second gathering and until the end of the experiment, the two Topology Aware versions are on average 79% more efficient to elect the correct leader than Flooding Degree, which increases its instability after each gathering, when nodes start to randomly move again.

2) Number of messages sent per second: In the previous section, we observed that the smaller the transmission range, the higher the number of components. Therefore, in the case of low transmission ranges, there are more components, and consequently, more leaders. This higher number of leaders explains why Flooding Degree presents bad performance with low transmission ranges, since each leader floods its component with *leader messages*. When the transmission range increases, the number of leaders decreases, thus reducing the number of flooding messages. On the other hand, Topology Aware algorithms behave inversely: when the transmission range increases, each node observes more topological movements, therefore, increasing the amount of new knowledge and update messages. These behaviors are well characterized in Figure 4, especially for the random waypoint pattern. Note that the number of messages sent in both Topology Aware versions is the same because the election criterion does not impact the number of messages.

We can observe an interesting threshold effect from the transmission range of 130 meters. At this range, components are bigger and start to become more stable in terms of topology changes. Topology Aware algorithms benefit from the topology stability since: (1) they generate fewer messages (due to fewer connections and disconnections, so fewer knowledge and updates exchanges); (2) they are less sensitive to the size of components. On the opposite, Flooding Degree is punished by the size of the components, because it is not sensitive to topology changes and flooding is more costly when the number of links increases. This explains why the curves of Topology

Aware versions stabilize while the curve of Flooding Degree increases.

For the periodic single point of interest pattern, both versions of Topology Aware send fewer messages, because they do not need to communicate when the topology is motionless. As the number of nodes in the component increases, the number of messages also increases, depending on the value of Δ that impacts the number of messages sent per second. On the other hand, Flooding Degree sends more messages than the two Topology Aware versions, because even if the topology is static, a flooding algorithm continues to periodically send information about its current leader.

However, the size of messages exchanged in Topology Aware is larger than in the Flooding Degree algorithm. In Table I, the Flooding Degree algorithm presents the same average message size since messages contain just a node identifier and a value, i.e., two integers. On the other hand, the size of messages in Topology Aware varies according to the number of nodes in the connected component. Therefore, messages in both Topology Aware versions have larger sizes when compared to the Flooding Degree algorithm. Note that the value of message sizes in Table I contain additional information needed by the simulator, which is identical to the three algorithms. Also, note that the size of messages remains below the MTU value of wireless networks, then messages can be included in single packets.

Table I
AVERAGE MESSAGE SIZE (IN BYTES)

	Flooding Degree	Topology Aware Degree	Topology Aware Closeness
Random waypoint	196	705 to 1284	681 to 1260
Periodic single POI	196	786 to 942	760 to 915

3) The longest leader path relative to the component diameter: Figure 5 gives the average longest path to the leader (y-axis left) and the average component diameter (y-axis right) for both mobility patterns. It shows that an election criterion based on Closeness centrality shortens paths to the leader compared to an election criterion only based on Degree.

For the random waypoint pattern, low transmission ranges lead to small components with only some nodes. Therefore, paths to the leader are most of the time direct links or contain only a few nodes. When the transmission range increases, Topology Aware Closeness is better than Flooding Degree, thanks to its component central leader choice.

In the periodic single point of interest pattern however, as the shape of the disc is independent of the transmission range, there is periodically only one component comprising the entire network. We observe that Topology Aware Closeness is 11% better than Flooding Degree. Topology Aware Degree has a similar behavior to Flooding Degree as both have the same election criterion.

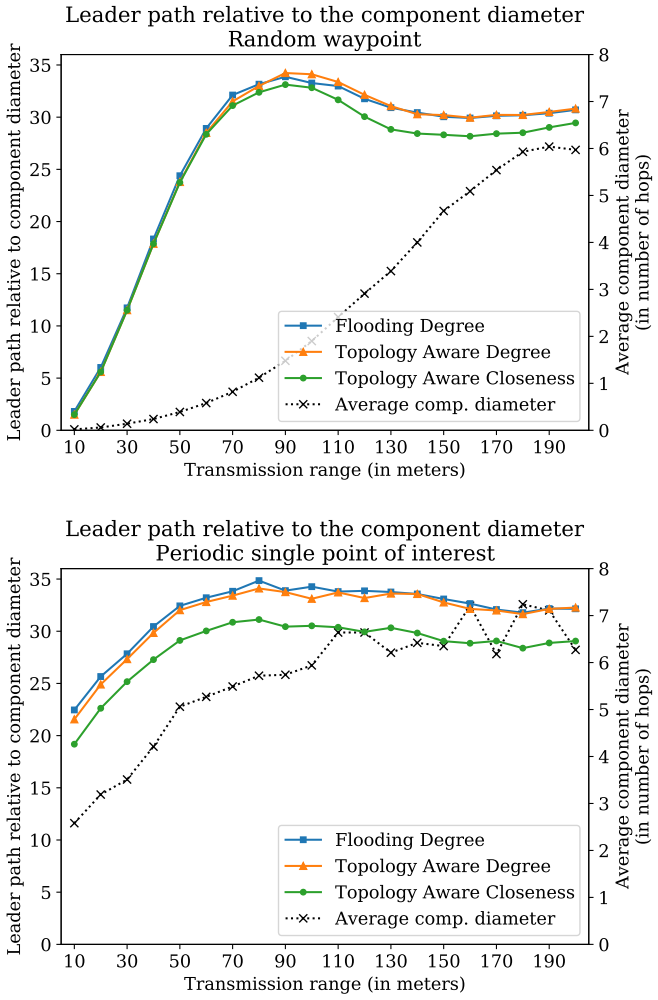


Figure 5. Longest leader path relative to component diameter

G. Fault injection

We have conducted fault injection experiments on the leader node in a static configuration, evaluating then the average time to elect a new leader when the transmission range varies from 10m to 200m. After nodes have exchanged information to elect the eventual leader of the component, the leader crashes and recovers periodically. Results in Table II show the average election time to elect the new leader. We observe that for both Topology Aware versions the election time is larger than Flooding Degree, because the buffering of messages slows down message transmission time of each node as the transmission range increases. However, this time remains low (less than 1 second).

Table II
AVERAGE ELECTION TIME (IN MILLISECONDS)

Flooding Degree	Topology Aware Degree	Topology Aware Closeness
422	820	731

VI. Conclusion and future work

This article has presented a per component eventual leader election algorithm for dynamic networks, that shows the advantages of a network topology knowledge used by all nodes for the choice of the leader. To this end, by exchanging messages, every node maintains a local knowledge of the communication graph of connected nodes and exploits such knowledge to elect as the leader the node having the highest closeness centrality. This leader can, therefore, spread information faster over its connected component than flooding algorithms. Considering the random waypoint and a periodic single point of interest mobility models, we evaluated on the PeerSim [1] simulator both our algorithm and a flooding algorithm with a local topological election criterion. Both *Topology Aware* algorithms are more stable than *Flooding Degree* and the closeness version has a shorter path to the leader, especially on large components with low movements of nodes. It is less sensitive to the component size and sends fewer messages than *Flooding Degree*. When compared to *Flooding Degree*, our algorithm improves the leader stability up to 82% depending on mobility models, sends half as many messages, and nodes reach the leader by 11% shorter paths. In the *Topology Aware* algorithm, the size of messages could be reduced using compression, for example.

In the future, we plan to work on a cross-layer implementation for MANET, using information from the MAC layer to improve communication performance. We will also consider wireless message collisions. In order to detect inconsistencies, nodes will periodically broadcast a checksum of their topological knowledge, in probe messages for example. Lastly, we intend to implement the Topology Aware algorithm on real devices such as Raspberry Pis, conducting experiments in real conditions.

References

[1] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *The 9th Int. Conference on Peer-to-Peer (P2P'09)*, 2009, pp. 99–100.

[2] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *Journal of the ACM (JACM)*, vol. 43, no. 4, pp. 685–722, 1996.

[3] D. Peleg, "Time-optimal leader election in general networks," *Journal of parallel and distributed computing*, vol. 8, no. 1, pp. 96–99, 1990.

[4] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[5] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Oct. 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[6] H. V. Netto, L. C. Lung, M. Correia, A. F. Luiz, and L. M. S. de Souza, "State machine replication in containers managed by kubernetes," *Journal of Systems Architecture*, vol. 73, pp. 53–59, 2017.

[7] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8, no. 9, 2010.

[8] G. Le Lann, "Distributed systems-towards a formal approach." in *IFIP congress*, vol. 7, 1977, pp. 155–160.

[9] E. Chang and R. Roberts, "An improved algorithm for decentralized extrema-finding in circular configurations of processes," *Communications of the ACM*, vol. 22, no. 5, pp. 281–283, 1979.

[10] N. A. Lynch, *Distributed algorithms*. Elsevier, 1996, ch. 4.1, pp. 52–56.

[11] N. Malpani, J. L. Welch, and N. Vaidya, "Leader election algorithms for mobile ad hoc networks," in *The 4th int. workshop on Discrete algorithms and methods for mobile computing and communications*. ACM, 2000, pp. 96–103.

[12] S. Vasudevan, J. Kurose, and D. Towsley, "Design and analysis of a leader election algorithm for mobile ad hoc networks," in *The 12th int. Conference on Network Protocols, ICNP.* IEEE, 2004, pp. 350–360.

[13] M. Rahman, M. Abdullah-Al-Wadud, and O. Chae, "Performance analysis of leader election algorithms in mobile ad hoc networks," *Int. J. of Computer Science and Network Security*, vol. 8, no. 2, pp. 257–263, 2008.

[14] R. Ingram, T. Radeva, P. Shields, S. Viqar, J. E. Walter, and J. L. Welch, "A leader election algorithm for dynamic networks with causal clocks," *Distributed computing*, vol. 26, no. 2, pp. 75–97, 2013.

[15] L. Arantes, F. Greve, P. Sens, and V. Simon, "Eventual leader election in evolving mobile networks," in *Int. Conference On Principles Of Distributed Systems*. Springer, 2013, pp. 23–37.

[16] C. Shea, B. Hassanabadi, and S. Valaee, "Mobility-based clustering in vanets using affinity propagation," in *GLOBECOM*. IEEE, 2009, pp. 1–6.

[17] C. Gómez-Calzado, A. Lafuente, M. Larrea, and M. Raynal, "Fault-tolerant leader election in mobile dynamic distributed systems," in *19th Pacific Rim Int. Symposium on Dependable Computing*. IEEE, 2013, pp. 78–87.

[18] C. Kim and M. Wu, "Leader election on tree-based centrality in ad hoc networks," *Telecommunication Systems*, vol. 52, no. 2, pp. 661–670, 2013.

[19] C. Fernández-Campusano, M. Larrea, R. Cortiñas, and M. Raynal, "A distributed leader election algorithm in crash-recovery and omissive systems," *Information Processing Letters*, vol. 118, pp. 100–104, 2017.

[20] M. Nesterenko and S. Tixeuil, "Discovering network topology in the presence of byzantine faults," *Transactions on Parallel and Distributed Systems*, vol. 20, no. 12, pp. 1777–1789, 2009.

[21] B. Bellur and R. G. Ogier, "A reliable, efficient topology broadcast protocol for dynamic networks," in *INFOCOM'99. Conference on Computer Communications. 18th Annual Joint Conference of the IEEE Computer and Communications Societies.*, vol. 1. IEEE, 1999, pp. 178–186.

[22] R. Ingram, P. Shields, J. E. Walter, and J. L. Welch, "An asynchronous leader election algorithm for dynamic networks," in *Int. Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–12.

[23] D. S. Hirschberg and J. B. Sinclair, "Decentralized extrema-finding in circular configurations of processors," *Communications of the ACM*, vol. 23, no. 11, pp. 627–628, 1980.

[24] P. Humblet, "A distributed algorithm for minimum weight directed spanning trees," *IEEE Transactions on Communications*, vol. 31, no. 6, pp. 756–762, 1983.

[25] B. Awerbuch, "Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems," in *The 19th annual ACM symposium on Theory of computing*, 1987, pp. 230–240.

[26] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, p. 374–382, Apr. 1985.

[27] M. Larrea, A. Fernández, and S. Arévalo, "Optimal implementation of the weakest failure detector for solving consensus," in *19th IEEE Symposium on Reliable Distributed Systems, SRDS'00, Proc.*, 2000, pp. 52–59.

[28] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, "On implementing omega with weak reliability and synchrony assumptions," in *The 22nd annual symposium on Principles of distributed computing*, 2003, pp. 306–314.

[29] A. Mostéfaoui, E. Mourgaya, M. Raynal, and C. Travers, "A time-free assumption to implement eventual leadership," *Parallel Process. Lett.*, vol. 16, no. 2, pp. 189–208, 2006.

[30] L. Melit and N. Badache, "An Ω-based leader election algorithm for mobile ad hoc networks," in *Networked Digital Technologies - 4th Int. Conference. Part I*, 2012, pp. 483–490.

[31] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558–565, 1978.

[32] A. Bavelas, "Communication patterns in task-oriented groups," *The journal of the acoustical society of America*, vol. 22, no. 6, pp. 725–730, 1950.

[33] T. Camp, J. Boleng, and V. Davies, "A survey of mobility models for ad hoc network research," *Wireless communications and mobile computing*, vol. 2, no. 5, pp. 483–502, 2002.

[34] M. Papandrea, K. K. Jahromi, M. Zignani, S. Gaito, S. Giordano, and G. P. Rossi, "On the properties of human mobility," *Computer Communications*, vol. 87, pp. 19–36, 2016.