HAL
open science

Topology Aware Leader Election Algorithm for MANET

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

To cite this version:

Arnaud Favier, Nicolas Guittonneau, Jonathan Lejeune, Anne Fladenmuller, Luciana Arantes, et al.. Topology Aware Leader Election Algorithm for MANET. COMPAS 2020 - Conférence francophone d'informatique en Parallélisme, Architecture et Système, Jun 2020, Lyon, France. hal-03471423

HAL Id: hal-03471423

https://inria.hal.science/hal-03471423v1

Submitted on 8 Dec 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Compas'2020 : Parallélisme/ Architecture/ Système / Temps Réel
MILC - Lyon, France, du 30 juin au 3 juillet 2020

Topology Aware Leader Election Algorithm for MANET

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

LIP6, *Inria, CNRS
Sorbonne University, Paris, France
firstname.lastname@lip6.fr

Résumé
This article presents an eventual leader election algorithm for mobile dynamic networks. Each node builds knowledge of the communication graph of connected nodes, by broadcasting changes in their neighborhood. This knowledge provides the current topology of the network, used to compute the closeness centrality as the choice of the leader. Experiments were realized on PeerSim simulator [11], comparing our algorithm with static and dynamic flooding algorithms, on different network topologies and mobility patterns. Our algorithm improves leader stability up to 24%, sends half less messages and aims to an 8% shorter leader path.

Mots-clés : distributed systems, dynamic systems, leader election, centrality, MANET

---

1. Introduction

Eventual leader election is a key component for many fault tolerant services in asynchronous distributed systems. The election of an eventual leader is an agreement on the identity of a single correct node, called leader and eventually accepted by all correct nodes of a network. The leader coordinates distributed tasks in the network and all processes can potentially become leaders.

This problem is studied in dynamic systems [10, 14, 12, 5, 1, 7], but only a few articles use specificities of highly dynamic networks, such as MANET, in their leader election criteria [7].
A mobile ad hoc network (MANET) is a decentralized dynamic network where nodes can move and communicate with each other by transmitting messages over wireless links. Two nodes communicate by sending messages through a direct wireless link if they are in the same transmission range, or through a sequence of wireless links, including one or more intermediate nodes acting as relays. Initially, the system is unknown and processes it during execution time. Processes can join or leave the system, fail and recover at runtime.

In the context of MANET, the leader election problem must be specialized in two ways [4] :
- The election algorithm must tolerate arbitrary, concurrent topological changes and should eventually terminate electing a unique leader per component.
- The leader should be the most valued node of the connected component, where the value of a node is a performance-related characteristic or its central topological position.

Our contribution is an algorithm providing an eventual leader according to a topological knowledge of the network, assuming an asynchronous system and broadcast communications only.

Our algorithm progressively builds and maintains a knowledge of the connected graph formed by the neighbors of nodes at the topological level, that allows us to compute the closeness centrality and elect a central leader. A central leader efficiently spread information across the network and quickly reach a majority of nodes to solve the consensus.

2. Related work

In dynamic systems and in MANET, the leader election is a well-studied problem. To elect a new leader, [6, 5, 14, 7] require a three waves algorithm : two waves to search a potential leader and one confirmation wave to spread the election result over the network. This requires more message exchanges and slow down the election process due to network discovery.

Point-to-point communications are used in [6] and [5], requiring to know the set of direct neighbors, and to send as many messages as the number of neighbors. With a broadcast communication model, neighbors nodes do not need to be known and only one message reaches all the neighborhood at once. Rahman *et al.* [12] use a heartbeat, probe, reply and acknowledgment system, but if a specific message is lost, the leader node could not be determined. [1] uses a query-response system to verify correct nodes. All these algorithms increase the number of message exchanges and overload the network.

Malpani *et al.* [10] create a direct acyclic graph where each node has a direct path to the leader. Vasudevan *et al.* [14] uses a spanning tree to elect the leader, where each node sends back to its parent the identifier and value of the most valued node in its sub tree. Rahman *et al.* [12] also uses a spanning tree. In all cases, not enough information is available to elect a central leader in the component.

Kim *et al.* [7] use a spanning tree to elect a central leader based on the average depth of nodes, according to their own metric. However, the central leader is not always optimal, depending on the initiator node of the election. The algorithm in [6] relies on a global time and assumes that nodes have perfectly synchronized clocks, which is not suitable for a realistic MANET.

Most works elect the most valuable node of the network, considering a static value such as the highest node identifier [12], a movement based counter [10, 6, 5, 1], or the highest arbitrary value [14]. Vasudevan *et al.* [14] suggest the idea of a leader with the minimum average distance to other nodes, but their algorithm does not.
We compare our algorithm with a variant of Vasudevan *et al.* [14], because it is a good example of a typical flooding algorithm, and is recognized in the literature [6, 12, 13, 5, 3, 7].

3. System model and assumptions

Nodes always follow the specification of the algorithm. A node is correct if it never fails and never leaves the system. A node is faulty if it fails, or leaves the system, until it comes back in the system again, with the same *id* and knowledge of the membership as before. A node can fail by stopping early, shutting down or by crashing.

3.1. Communication graph

The system is a collection of mobile nodes, seen as an undirected connected graph, where vertices are nodes and edges represent direct communication links between nodes (1-hop dis-

tance). Two nodes can communicate directly if they are in the transmission range of each other.Emission range and the reception range are identical, so if node i can communicate with node j, node j can also communicate with node i. Adjacent nodes are called neighbors and the set of adjacent nodes represents the neighborhood of a node.
If the system is divided into two different connected components due to movements, each connected component is considered to be a fully fledged network, and therefore eventually contains one leader. The algorithm ensures that if the topological changes cease, each connected component will contain a single leader.

3.2. Messages

Nodes communicate by exchanging messages in a MANET supporting broadcast. Nodes are directly reachable if they are physically located in the wireless transmission range of each other. Our algorithm only uses broadcast communications on a fixed Wi-Fi channel decided beforehand. There is no assumption about the execution time of nodes or message transfer time, so the system is asynchronous, but we assume reliable communication channels.

3.3. Membership and identity

The number of nodes is unknown. Each node initially knows its own identifier, unique in the system. By receiving messages from its neighbor, a node gets knowledge of the membership of the network. Each node periodically emits *probes* with its identifier, every τ millisecond, but *probes* are not used in the algorithm. When node i receives probes from node j because they are in the same transmission range, the method *Connection* of the algorithm is triggered. After not receiving α *probes* from node j, node i considers node j as not a neighbor anymore, and the method *disconnected* of the algorithm is triggered.

4. Algorithm

Every node builds a topological knowledge of its connected component, formed by its neighbors, neighbors of its neighbors and so on ; during node connections and disconnections. The algorithm maintains it by sending the knowledge (called *known*) to new neighbors, or partial modifications (called *updates*) periodically to its neighbors. The pseudo code for the algorithm is given in *Algorithm 1*.

4.1. Types, variables and messages

Each node knows its identifier and maintains three local variables (line 3) :

- **known** (line 4) : the actual knowledge of the connected component of a node (including itself), implemented as a map of *views* (a logical *clock* [8] and a set of direct neighbors *identifiers*, line 1) indexed by node *identifier*, i.e., an entry for each node.
- **updates** (line 5) : a list of *updates* is periodically sent to update the knowledge, by propagating new connections and disconnections, without sending the entire knowledge, reducing message sizes and avoiding sending information already received by neighbors. An update consists of the identifier of the *source* node having the modifications in its neighborhood, a set of *added* nodes (new direct connected neighbors), a set of *removed* nodes (direct disconnected nodes), the logical clock value of *source* node before the modifications (*old_clk*) and the logical clock value after modifications (*new_clk*).
- **saved** (line 6) : a list of *updates* which cannot be applied at the time of first reception, but which could be applied when new information is received thereafter.

known and **updates** are exchanged through two distinct kinds of messages with the same name.

4.2. Connection

When a new node *j* appears in the transmission range of node *i*, the probe system calls the *Connection* method (line 21). Node *j* is considered as a new neighbor and is added to the knowledge of node *i* (line 22). As the knowledge of node *i* has just been updated, the logical clock of node *i* is incremented by one (line 23). Both nodes *i* and *j* exchange their knowledge to share information about the component (line 24).

4.3. Knowledge reception

When node *i* receives the known map of node *j* (line 44), it checks each node *id* included in $known_j$ (line 45). If *id* is a new node (line 46), an updated is created with neighbors of *id* and an old clock valued at 0 (i.e., all neighbors are in the *add* set, line 47), and the knowledge of *i* is updated (line 48).
If *id* is already known and has a greater clock than the clock known by *i* (line 49), *id* had new connections and/or disconnections, so node *i* creates an update, computes new neighbors (line 50) and removed nodes (representing disconnections, line 51) since the last received view. Clock values are set in the update (line 52). Eventually, due to previous knowledge and update exchanges, neighbors of node *i* will have the same knowledge as node *i* with identical clocks, and will be able to apply this new update in their knowledge.
Clock and neighbors are updated in the knowledge of *i* (line 53). The *SavedUpdates* method is called to apply previous saved updates (line 54).

4.4. Disconnection

When α probes from node *j* are not received by node *i*, node *j* is considered disconnected (line 25). An update is created (line 26) with : the identifier of *i* (source of the modification) ; 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 *i* ; the new clock, which is the clock of *i* increased by 1. The update will be propagated later (line 26). Node *j* is then removed from knowledge of *i* (line 27) and the clock is increased (line 28).

4.5. Updates reception

Each update adds or removes neighbors of a *source* node *k* (line 30). If *old clock* is equal to 0 (line 32), the update contains all the neighbors of *k*, and is applied (line 33) if node *i* does not have any information about *k* (line 31). If *old clock* equals the clock of *k* in the knowledge of *i* (line 37), i.e., new information, neighbors (line 38) and clock (line 39) are updated. Updates are propagated later by the periodic updates task (lines 34 and 40). An update may not be applied because it is too recent and node *i* has not received one or more previous updates yet (line 41 and 35). Thus, node *i* saves the update (line 36 and 42) to apply it in the future, after new updates will be received (line 43).

4.6. Updates saved

SavedUpdates (line 55) checks the updates that can be applied in the *saved* list (line 56).
To reduce message exchanges and improve performances, we save updates that cannot be applied when first received, and we try to apply them after new information is received.

4.7. Leader()

When the upper layer needs a leader, it calls the *Leader* method (line 17) which computes the best leader of the knowledge, according to the closeness centrality (line 20). The closeness of a node is the inverse of its distance to all other nodes, and characterize the capacity of a node to spread information over the graph. The closeness centrality formula for a node x used was de-

fined by Alex Bavelas in 1950 [2] is : $C(x) = \frac{1}{\sum_y d(y,x)}$. From the set of reachable nodes (line 18) (nodes of the component), we compute the closeness centrality. The leader is the node with maximum centrality (line 20), and node identifiers are used to break ties. In the knowledge, unreachable nodes are deleted to improve future performances of component computing (line 19).

5. Results

The objective of the experiments is to compare our *TopologyAware* algorithm with other classic flooding algorithms. Thanks to the knowledge of the topology, our algorithm can choose a better leader, i.e., more central to spread information more efficiently over the graph.

5.1. Simulation environment

Experiments used PeerSim [11], a peer-to-peer network simulator. Each experiment lasts 30 minutes, with a simulated unit of time of one millisecond, and simulates 60 nodes placed in a 900m $\times$ 900m obstacle free area. We assumed reliable communications, no message loss, no failures and consider two different mobility patterns.

5.1.1. Random waypoint

Nodes are randomly placed and move according to the Random Waypoint mobility pattern, with a minimum speed of 5m/s, a maximum speed of 15m/s, and a 20s pause before moving to a next random destination (speeds follow a uniform random law).

5.1.2. Periodic single point of interest

Nodes are circularly placed around a center point of interest **Fig. 1.** After a random timeout, nodes start moving to a random destination. Once the destination has been reached, node wait a few seconds before coming back to their initial position, and start to move randomly again.

5.2. Probes system

A *probe* message is sent every 400 milliseconds and contains the unique identifier of the node. After not receiving one probe from node j (reliable communications with no messages loss are assumed), node i considers node j as out of range and trigger the *disconnected* method.

5.3. Algorithms

Topology Aware is compared with *Beacon*, a typically representative of flooding algorithms.

5.3.1. Beacon

Vasudevan et al. proposed an algorithm [14] returning $\perp$ when no leader is decided in the election phase. We made a variant without $\perp$, to be fairly comparable with *Topology Aware*, and called it *Beacon*, because it periodically broadcasts information about its current leader. *Beacon* is also a variant of *OptFloodMax* algorithm of Nancy A. Lynch, *Distributed Systems* [9].

Beacon was also adapted for MANET assuming an underlying *probe system*. Exchanged messages contain an election criteria called *value* and a node identifier. *Beacon* sends each 250 milliseconds information about the current leader, which is the highest received value compared to node value. The leader fails after a non-reception within 600 ms (2 $\times$ 250 + margin), of one *leader* message by direct neighbors (reliable communications with no messages loss are assumed), and nodes trigger a new election by setting themselves as their own new leader. Thus, new leader messages are propagated and eventually, the highest valued node will be elected.

We made two versions of *Beacon*. *Beacon Static* uses a fixed node value, set at launch time with a random number. *Beacon Dynamic* uses a local topological election criteria evolving over time :

the number of direct neighbors (i.e., node degree in a graph), updated at each topology change.

5.3.2. Topology Aware

We bufferize sent messages with a Δ buffer value equals to transmission range (on x-axis of figures), considering that larger transmission ranges potentially reach more nodes, to avoid burst effect after a topology change.

5.4. Performances

The goal is to compare static and dynamic versions of *Beacon* algorithms, with our *Topology Aware* algorithm, for diameters of transmission range from 10m to 200m, representing the connectivity indices, i.e., the number of components in the system. We consider 3 metrics in the experiments.

5.4.1. Instability

Instability is the average percentage of time a node has a wrong leader, according to an oracle. On **Fig. 3.**, the instability of the random waypoint pattern ceases to increase at a 130-meter range, as the majority of nodes often form a single component. In periodic POI, *Topology Aware* is 24% more stable than *Beacon*, because flooding in a circular configuration spread slower than knowledge and updates exchange. A static election criteria leads to low linear instability.

5.4.2. Messages sent

It is the *number of messages sent* for each node, divided by the experiment time in seconds. On **Fig. 4.**, low transmission ranges lead to more components (more leaders), so *Beacon Dynamic* sends more messages, but when the range increases, broadcasts reach more nodes. As the range increases, each *Topology Aware* node observes more topological movements, increasing the amount of knowledge and update messages. In periodic POI, *Topology Aware* does not need to communicate when the topology is motionless. *Beacon Dynamic* sends more messages even in static topology, because flooding algorithms periodically send information about the current leader. *Beacon Static* has a sending frequency independent of topology changes.
Message sizes in *Topology Aware* are higher than *Beacon Dynamic* **Fig. 2**, because *Beacon* only sends one integer (the degree; a value for *Beacon Static*). *Topology Aware* message sizes vary according to the number of nodes in the knowledge. This is the trade-off of *Topology Aware* : sending fewer but heavier messages.

5.4.3. Longest leader path relative to component diameter

We compute the *longest path* of all shortest paths from every node of the component to their current leader, then divide by the diameter of the component. **Fig. 5.** shows that a dynamic election criteria aims to a shorter leader path on average 8% compared to a static criteria, on both mobility patterns. In random waypoint, low transmission ranges lead to small components, i.e., leader paths are mainly direct. The second pattern does not depend on the transmission range because periodically motionless, so components periodically contain the whole network.

6. Conclusion

This article explains the advantages of building a local knowledge of the network topology, to improve the leader choice in MANET, using the closeness centrality to elect a leader that spread information faster over the graph. Our algorithm improves the leader stability up to 24% depending on mobility patterns, sends half less messages than a classical flooding algorithm, and aims to an 8% shorter leader path. The trade off is higher message sizes that can be improved.

Algorithm 1: Topology Aware Eventual Leader Election for node i

```
Typedef view : ⟨clk : int, neigh : set(id)⟩
Typedef updt : ⟨src : int, add : set(id), rmv : set(id), old_clk : int, new_clk : int⟩
Local variables of node i :
    known : map(key : id, value : view)
    updates : list(updt)
    saved : list(updt)
Initialisation of node i :
    known[i].neigh ← {i}
    known[i].clk ← 0
    updates ← ∅
    saved ← ∅
Periodic UpdateTask :
    if updates ≠ ∅ then
        Broadcast (updates)
        updates ← ∅
    Wait Δ seconds
Invocation of Leader() :
    component ← Reachable (known[i])
    known ← RetainAll (known, component)
    return Max (ClosenessCentrality (component))
Connection of node j :
    Add (known[i].neigh, {j})
    known[i].clk ← known[i].clk + 1
    Broadcast (known)
Disconnection of node j :
    updates ← updates ∪ {⟨i, –, {j}, known[i].clk, known[i].clk + 1⟩}
    known[i].neigh ← known[i].neigh \ {j}
    known[i].clk ← known[i].clk + 1
```

```
29 Receive updates_j from node j :
30     for each updt :⟨src, add, rmv, old_clk, new_clk⟩ ∈ updates_j do
31         if ∄ ⟨src, –⟩ ∈ known then
32             if old_clk = 0 then
33                 known[src] ← ⟨new_clk, add⟩
34                 updates ← updates ∪ updt
35             else
36                 saved ← saved ∪ updt
37         else if old_clk = known[src].clk then
38             known[src].neigh ← (known[src].neigh ∪ add) \ rmv
39             known[src].clk ← new_clk
40             updates ← updates ∪ updt
41         else if old_clk > known[src].clk then
42             saved ← saved ∪ updt
43     SavedUpdates()
44 Receive known_j from node j :
45     for each ⟨id, view⟩ in known_j do
46         if ∄ ⟨id, –⟩ ∈ known then
47             updates ← updates ∪ {⟨id, view.neigh, –, 0, view.clk⟩}
48             known[id] ← ⟨view.clk, view.neigh⟩
49         else if view.clk > known[id].clk then
50             add ← view.neigh \ known[id].neigh
51             rmv ← known[id].neigh \ view.neigh
52             updates ← updates ∪ {⟨id, add, rmv, known[id].clk, view.clk⟩}
53             known[id] ← ⟨view.clk, view.neigh⟩
54     SavedUpdates()
55 Invocation of SavedUpdates() :
56     for each updt :⟨src, add, rmv, old_clk, new_clk⟩ ∈ saved do
57         if old_clk = 0 then
58             if ∄ ⟨src, –⟩ ∈ known then
59                 known[src] ← ⟨new_clk, add⟩
60                 saved ← saved \ updt
61         else if old_clk = known[src].clk then
62             known[src].neigh ← (known[src].neigh ∪ add) \ rmv
63             known[src].clk ← new_clk
64             saved ← saved \ updt
65         if old_clk < known[src].clk then
66             saved ← saved \ updt
```

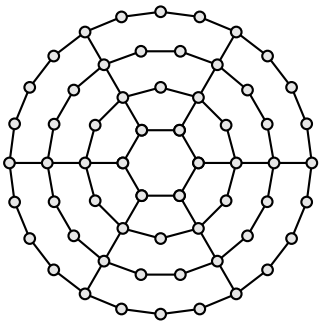

FIGURE 1 – Topology of single point of interest

	Beacon Static	Beacon Dynamic	Topology Aware
Random Waypoint	196	196	681-1720
Single POI	196	196	738-942

FIGURE 2 – Average message size (in bytes)

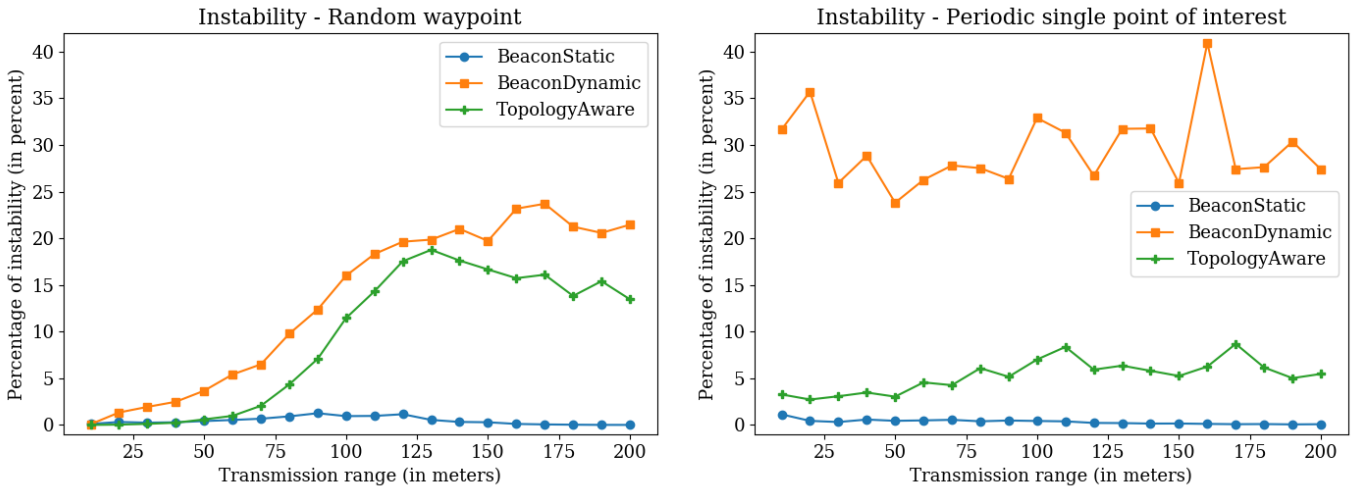

FIGURE 3 – Instability results

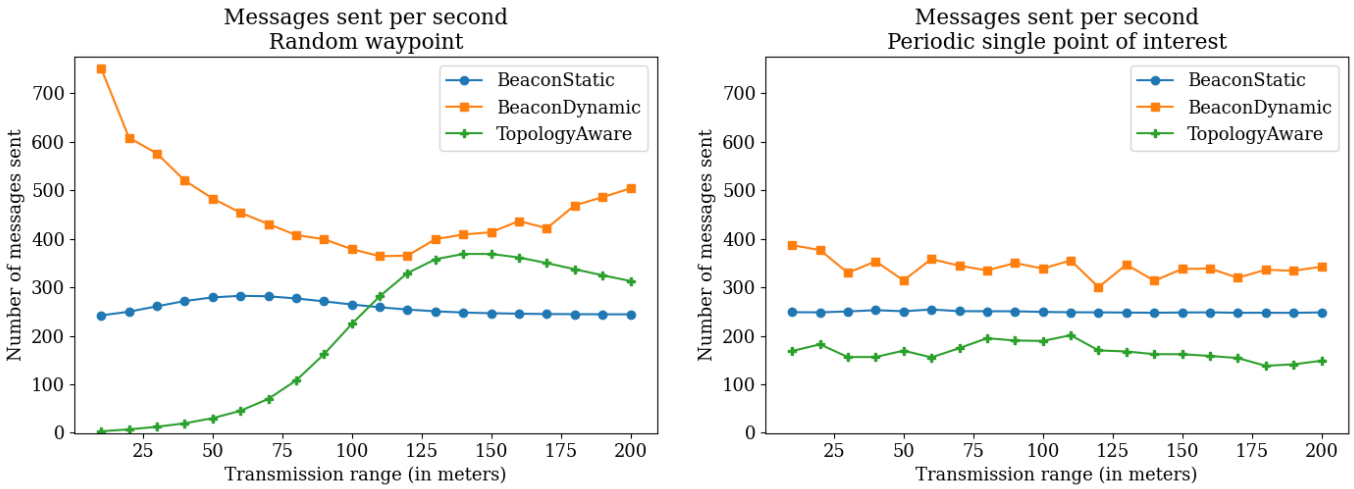

FIGURE 4 – Messages sent per second

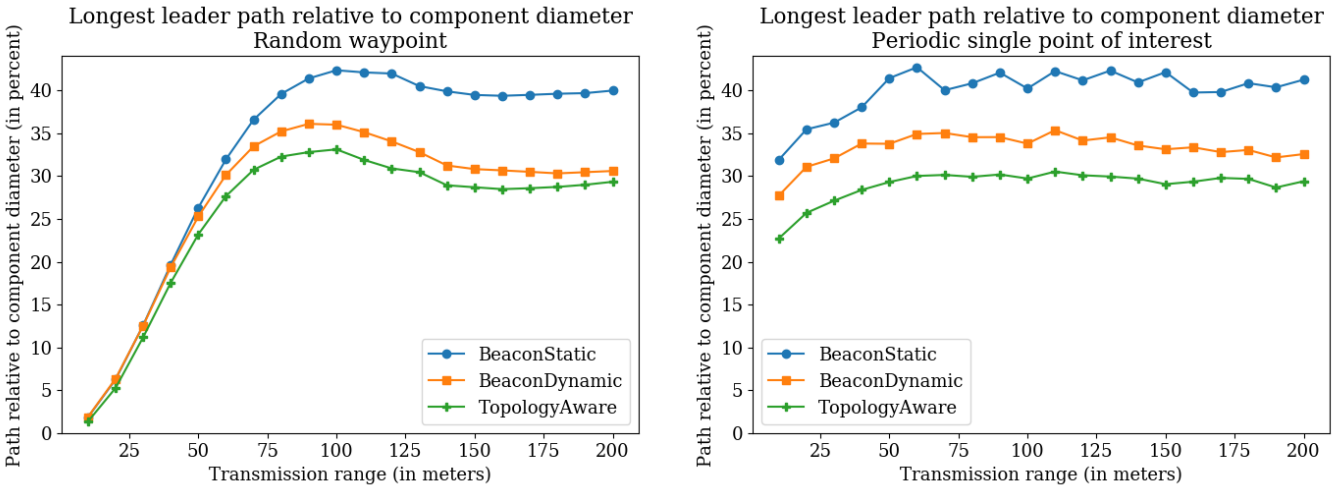

FIGURE 5 – Longest path to leader

Bibliographie

1. Arantes (L.), Greve (F.), Sens (P.) et Simon (V.). – Eventual leader election in evolving mobile networks. – In *International Conference On Principles Of Distributed Systems*, pp. 23–37. Springer, 2013.
2. Bavelas (A.). – Communication patterns in task-oriented groups. *The journal of the acoustical society of America*, vol. 22, n6, 1950, pp. 725–730.
3. Bhoir (S.) et Vidhate (A.). – A modified leader election algorithm for manet. *International Journal on Computer Science and Engineering*, vol. 5, n2, 2013, p. 78.
4. Fernández-Campusano (C.), Larrea (M.), Cortiñas (R.) et Raynal (M.). – A distributed leader election algorithm in crash-recovery and omissive systems. *Information Processing Letters*, vol. 118, 2017, pp. 100–104.
5. Ingram (R.), Radeva (T.), Shields (P.), Viqar (S.), Walter (J. E.) et Welch (J. L.). – A leader election algorithm for dynamic networks with causal clocks. *Distributed computing*, vol. 26, n2, 2013, pp. 75–97.
6. Ingram (R.), Shields (P.), Walter (J. E.) et Welch (J. L.). – An asynchronous leader election algorithm for dynamic networks. – In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–12. IEEE, 2009.
7. Kim (C.) et Wu (M.). – Leader election on tree-based centrality in ad hoc networks. *Telecommunication Systems*, vol. 52, n2, 2011, pp. 661–670.
8. Lamport (L.). – Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, vol. 21, n7, juillet 1978, p. 558–565.
9. Lynch (N. A.). – *Distributed algorithms*. – Elsevier, 1996.
10. Malpani (N.), Welch (J. L.) et Vaidya (N.). – Leader election algorithms for mobile ad hoc networks. – In *Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications*, pp. 96–103. ACM, 2000.
11. Montresor (A.) et Jelasity (M.). – PeerSim : A scalable P2P simulator. – In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pp. 99–100, Seattle, WA, septembre 2009.
12. Rahman (M.), Abdullah-Al-Wadud (M.) et Chae (O.). – Performance analysis of leader election algorithms in mobile ad hoc networks. *Int'l J. of Computer Science and Network Security*,

vol. 8, n2, 2008, pp. 257–263.

13. Shayeji (M. H. A.), Al-Azmi (A. R.), Al-Azmi (A. R.) et Samrajesh (M.). – Analysis and enhancements of leader elections algorithms in mobile ad hoc networks. *arXiv preprint arXiv :1210.1686*, 2012.
14. Vasudevan (S.), Kurose (J.) et Towsley (D.). – Design and analysis of a leader election algorithm for mobile ad hoc networks. – In *Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004.*, pp. 350–360. IEEE, 2004.