Thesis submitted for the degree of
Doctor of Philosophy in **Computer Science**

Sorbonne Université

Institut National de Recherche en Sciences et Technologies du Numérique

Laboratoire d'Informatique de Sorbonne Université

Centre national de la recherche scientifique

École Doctorale Informatique, Télécommunications et Électronique (Paris)

Eventual Leader Elections in Dynamic Networks

Arnaud FAVIER

2 March 2022

Jury members

Reviewers	**Emmanuelle ANCEAUME**	Research Director, CNRS
	Denis CONAN	Associate Professor HDR, Télécom SudParis
Examiners	**Anne FLADENMULLER**	Professor, Sorbonne University
	Mikel LARREA	Professor, University of the Basque Country
Advisor	**Pierre SENS**	Professor, Sorbonne University
Co-advisor	**Luciana ARANTES**	Associate Professor, Sorbonne University
Invited	**Jonathan LEJEUNE**	Associate Professor, Sorbonne University

To my grandparents,
Marcel Favier
Jeanne Favier
Prosper Tournier
Marcelle Tournier

Abstract

Leader election is important for many fault-tolerant services in asynchronous distributed systems. By coordinating actions of a set of distributed processes, it allows solving agreement problems like the consensus, a fundamental problem of distributed computing. Several consensus algorithms, such as Paxos, rely on an eventual leader election service, also known as the Omega (Ω) failure detector. Omega returns the identity of a process in the system, ensuring that eventually the identity of the same correct process is always returned.

Many leadership algorithms were proposed in the literature to implement Omega. Among those that consider dynamic systems, most of them do not choose the leader according to a topological criterion. However, the position of the leader in the network directly impacts the performance of algorithms using the leader election service, since the leader must often interact with other processes, for example, to collect information from a majority of processes in consensus algorithms.

This thesis studies the eventual leader election problem in dynamic evolving networks and performance related issues. Two eventual leader election algorithms are proposed for Mobile Ad Hoc Networks. They maintain and exploit the knowledge of the network topology to eventually elect one leader per connected component with the best closeness centrality. Evaluations were conducted on simulators with different mobility models and performance results show that these algorithms present better performance than other algorithms of the literature, including fewer messages, shortest paths to the leader, and better stability.

Résumé *(Abstract in French)*

L'élection de leader est importante pour les services tolérants aux pannes dans les systèmes distribués (également appelés systèmes répartis) asynchrones. En coordonnant les actions d'un ensemble de processus distribués, elle permet de résoudre des problèmes d'accord comme le consensus, un problème fondamental en informatique distribuée. Des algorithmes de consensus, tel que Paxos, s'appuient sur un service d'élection de leader ultime, également appelé détecteur de défaillances Omega (Ω). Omega renvoie l'identité d'un processus du système et garantit qu'après un certain temps, l'identité du même processus correct est toujours renvoyée.

De nombreux algorithmes d'élection de leader ont été proposés dans la littérature pour implémenter Omega. Parmi ceux considérant les systèmes dynamiques, la plupart ne choisissent pas le leader selon un critère topologique. Or, la position du leader dans le réseau impacte directement les performances des algorithmes utilisant le service d'élection, car le leader doit souvent interagir avec les autres processus pour, par exemple, collecter les informations d'une majorité de processus dans le cas du consensus.

Cette thèse étudie le problème d'élection de leader ultime dans les réseaux dynamiques. Deux algorithmes d'élection sont proposés pour les réseaux mobiles ad hoc. Ceux-ci maintiennent et exploitent la connaissance de la topologie du réseau pour élire à terme un unique leader par composante connexe ayant la meilleure centralité de proximité. Des évaluations sur simulateurs avec différents modèles de mobilité montrent que ces algorithmes présentent de meilleures performances que d'autres algorithmes de la littérature, notamment moins de messages échangés, des chemins plus courts vers le leader, et une meilleure stabilité.

Acknowledgments

First, I am particularly thankful to my advisors Pierre Sens and Luciana Arantes, for their guidance, advice, and support throughout these years. Thank you Pierre for accepting me as your student, and for always answering my many questions. I learned a lot from you and it was always a real pleasure to work with you. Thank you Luciana for your kindness and your many precise proofreading. I really enjoyed working with you, as well as our discussions on grammar points among many other subjects. Thank you to both of you for teaching me how to do research and much more.

I am also thankful to Jonathan Lejeune, for his help and advice during this thesis. Thank you Jonathan for the opportunity to teach with you in Master, which led to the work of the first contribution with PeerSim.

Then, I would like to thank Emmanuelle Anceaume and Denis Conan for accepting to review this thesis, and for the time and efforts dedicated to evaluating it. I also want to thank Anne Fladenmuller and Mikel Larrea for accepting to be part of this jury as examiners.

I am grateful to Inria for the CORDI-S grant and to Sorbonne Université for the working and academic environment. I would like to thank my colleagues at LIP6 and especially the permanents and other Ph.D. students of the Delys team, for our interesting discussions and the good time during the Winter Schools and the Compas conference. A special thanks to Saalik, Daniel, and Mathieu for your support. I also want to thank Anne and Nicolas for their collaboration.

I would like to thank Kamel and the team from the reproduction workshop of Sorbonne Université for their kindness and their careful work.

This thesis is a milestone in my computer science journey, which started with a passion at the age of eight, and has been enriched by several people over the years.

I would like to thank Sara Bouchenak for her guidance and recommendation to meet Pierre when I was a software engineer student at INSA Lyon. I also want to thank Sonia Ben Mokhtar and Guillaume Salagnac for their advice to explore the world of scientific research.

I am thankful to all my teachers at the IUT of Bourg-en-Bresse, for their solid teaching of computer science fundamentals, for showing the possibilities of studying this field further, and for giving me the opportunity to do so.

I am also grateful to my high school teacher Jean-Baptiste Butet, my mentor in London Bertrand, my colleague in Dublin Aleksandar, my uncles Jean-Luc and Pierre, my classmate Tom, as well as my neighbor Roland, who all directly contributed to this journey.

Arriving in Paris to start a Ph.D. was a challenge, but many people made it easier.

I am thankful to thank Stéphanie for her support, trust, and career advice during these years.

I would like to thank Friedrich for his support and advice throughout our discussions during and after the lockdown.

I want to thank all my friends from INSEAD for their constant support, discussions and moments spent together. In particular, thanks to Maéva, Dalia, Marianna, Maurits and Clément.

I also would like to thank my friends from INSA: Anthony, Paul-Louis, Mathis, and Sylvain for their support and our good times in Paris.

I want to thank my friends from the IUT and Lyon: Aurélien, Corentin, Valentin, Quentin, and Florian for their support and listening, with a special thanks to Bastien for his entire trust, help, and dedication to our projects, without whom many things would not have been possible.

I also would like to thank Rémi, Olivier, Louis, Audrey, and Jean for their support and game nights.

I am thankful to Jonathan, Elsa, Leslie, Catherine, and Jean-Marc for their help and support in Paris.

I am also grateful to Louise, Monique, and Christian for their support during their visits to Paris, where I was always included with them.

I would like to thank my close friend Matthieu for his support and constant optimism during these years, with all our good times with Aurélie for many years, especially in the Morvan.

I want to thank Marie very much for her support and understanding during these years, as well as for the time off in Angers. Thank you for being by my side, you continue to impress me every day.

I also would like to thank all my family for their support and understanding during these years.

I am thankful to my sister Laëtitia and my brother-in-law Benoit for their support and precious help at the beginning in Paris, as well as for our good times in Mallorca. I also would like to thank my brother Cédric and my sister-in-law Marie-Line for their support and their visits to Paris.

I am truly grateful to my grandmother Marcelle *"Mamy Coucou"* for her happiness and enthusiasm giving me motivation and energy every day.

Last but not least, I am deeply grateful to my parents, Jean-Claude Favier and Marie Favier, for their flawless support and trust for many years, including these in Paris. Thank you for teaching me to be curious and to work, as well as giving me lucid advice and encouraging me in all situations.
Merci infiniment pour tout ce que vous faites.
Like what, bringing a computer at home, and then doing the IUT open days, was quite an idea!

I am also thankful to all the people not explicitly mentioned above, who helped me during these years.

Summary

List of Figures

List of Tables

1. Introduction

Distributed computing is present everywhere in our world and is at the heart of the information processing of our modern society. From the World Wide Web to Peer-to-Peer applications to blockchains like Ethereum [Woo+14], many applications revealed the importance of distributed computing in our daily life.

A distributed [1] system can be defined [2] as follows [VT17]:

> *A distributed system is a collection of computing elements that interact with one another in order to achieve a common goal.*

Computing elements are autonomous computational entities, each of them has its own local memory. They can be multiple software processes located on the same physical computer, or different hardware devices interconnected together, such as computer networks [VT17; And00]. Computing elements are also called *processes* or *nodes*, and to achieve their common goal, they collaborate and coordinate their actions by communicating with each other. There are two main communication paradigms: shared memory [Abr88] and messages passing [Tel00]. This thesis only considers communication by messages passing which uses a communication channel, i.e. a link between two processes of the system on which information is transmitted. Through a communication channel, a process sends and receives messages to/from another process [And00].

Since a distributed system is composed of many interconnected processes, some of them may become faulty during the execution of the system, and consequently, induce failures in the system. A failure is a deviation from the expected behavior of the system. A distributed system is considered reliable if it respects its specification, and if the system is able to provide correct services regardless of failures. Therefore, to run a reliable system in the presence of faults, the distributed system needs to be *fault tolerant*.

With the ongoing advent of mobile computers such as smartphones, intelligent vehicular, drones or mobile sensors, nodes with mobility

[Woo+14] Wood et al. (2014): 'Ethereum: A secure decentralised generalised transaction ledger'

1: The term *distributed* originally referred to computers physically distributed over a large geographical area [Lyn96].

2: Leslie Lamport gave another definition: *"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."* [Lam87]

[VT17] Van Steen et al. (2017): *Distributed Systems*

[And00] Andrews (2000): *Foundations of multithreaded, parallel, and distributed programming*

[Abr88] Abrahamson (1988): 'On achieving consensus using a shared memory'

[Tel00] Tel (2000): *Introduction to distributed algorithms*

compose a dynamic distributed system. These nodes move over a physical area and communicate directly with each other through wireless links. They can also fail as well as join and leave the system. Thus, the system evolves over time and the network is, therefore, highly dynamic.

A common type of dynamic network is the Mobile Ad Hoc Network (MANET), which does not rely on any pre-existing infrastructure to communicate, such as router or access-point. A MANET is a self-configuring network in which nodes are connected without wires and are free to move. Nodes are equipped with a radio module that enables wireless communications and participate in the network by forwarding messages to other nodes.

Many distributed applications and services, such as the distributed database Casandra [LM10] and blockchains like Bitcoin [Nak08], require that processes reach a consensus, by choosing a single value among proposed ones. The *consensus* problem is one of the most fundamental problems of distributed computing. A consensus algorithm requires that every process proposes a value to other processes in the system, and all non-faulty process eventually agree on a single value among the proposed ones. Consensus is used by many other distributed algorithms in the literature, such as state machine replication or atomic broadcast. However, Fischer, Lynch and Paterson have proved (the well-known FLP theorem [FLP85]) that it is impossible to deterministically achieve consensus in a completely asynchronous system where at least one node is prone to crash failure.

Several existing well-known algorithms in the literature solve consensus in failure-prone distributed systems by using a leader election algorithm. Examples of such algorithm are Paxos [Lam98], used by Google, Amazon and Microsoft, or Raft [OO14]. Also known as the Ω failure detector [CHT96], an eventual leader election allows to deterministically solve the consensus problem with the weakest assumptions on process failures considering a majority of correct processes. Ω provides a primitive called *Leader()*, which, when invoked, returns the identifier of a process in the system and guarantees that there is a time after which it always returns the identifier of the same correct process, i.e. the leader.

Many leadership protocols were proposed to implement Ω. Most of them consider static distributed systems and assume that the membership of the system is either known in advance [LFA00; Agu+04], or unknown [FJR06; JAF06]. Among the ones that tolerate system dynamics [Góm+13; Ara+13], only a few of them take into account the characteristics and the lack of knowledge of highly dynamic systems. Furthermore, most of these algorithms do not choose the leader according to a topological criterion, i.e. the position of the node in the network, but rather on the highest or lowest node identifier. The topological position of the leader has a strong impact on the performance of algorithms using the leader election service, since the leader must

[LM10] Lakshman et al. (2010): 'Cassandra: a decentralized structured storage system'

[Nak08] Nakamoto (2008): 'Bitcoin: A Peer-to-Peer Electronic Cash System'

[FLP85] Fischer et al. (1985): 'Impossibility of Distributed Consensus with One Faulty Process'

[Lam98] Lamport (1998): 'The part-time parliament'

[OO14] Ongaro et al. (2014): 'In search of an understandable consensus algorithm'

[CHT96] Chandra et al. (1996): 'The weakest failure detector for solving consensus'

[LFA00] Larrea et al. (2000): 'Optimal Implementation of the Weakest Failure Detector for Solving Consensus'

[Agu+04] Aguilera et al. (2004): 'Communication-efficient leader election and consensus with limited link synchrony'

[FJR06] Fernández et al. (2006): 'Eventual Leader Election with Weak Assumptions on Initial Knowledge, Communication Reliability, and Synchrony'

[JAF06] Jiménez et al. (2006): 'Implementing unreliable failure detectors with unknown membership'

[Góm+13] Gómez-Calzado et al. (2013): 'Fault-tolerant leader election in mobile dynamic distributed systems'

[Ara+13] Arantes et al. (2013): 'Eventual Leader Election in Evolving Mobile Networks'

collect information from the other nodes, such as from a majority of processes in the case of consensus. Thus, the average number of hops to reach the leader has a direct impact on the performance of consensus algorithms. A representation of a central leader is given in Figure 1.1.

This thesis studies the eventual leader election problem considering the above described dynamic evolving networks and performance issues of leader-based algorithms.

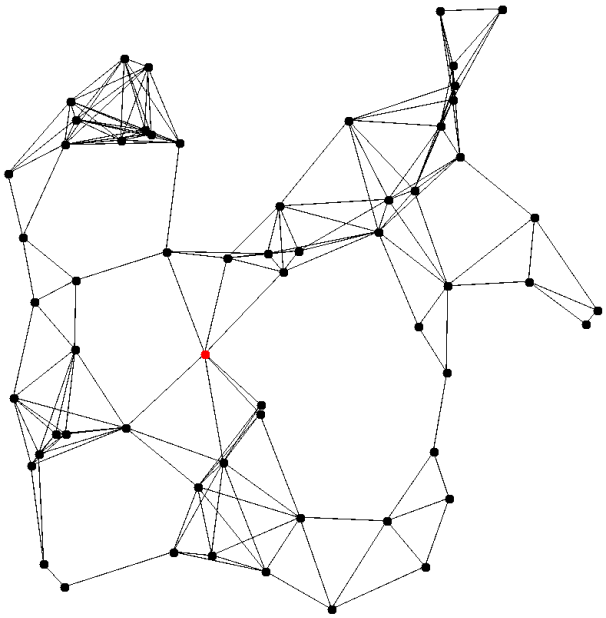

Figure 1.1.: Representation of a central leader (node in red) in a distributed network.

1.1. Contributions

Since the choice of the leader has an impact on the performance of algorithms using the leader election service, a Ω algorithm must take into account the average number of hops to reach the leader. Hence, the ideal would be to elect a non-faulty node with the best network centrality. To this end, each node could maintain a knowledge of the network topology.

This thesis thus proposes two eventual leader election algorithms, denoted *Topology Aware* and *Centrality-based Eventual Leader (CEL)*, for MANETs considering partial synchronous system and network partitions. Nodes can move, fail by crash, join and leave the system. Initially, nodes only know their respective identity, and, by exchanging messages with neighbor nodes in their transmission range, they acquire knowledge of the network topology. The algorithms exploit this knowledge to eventually elect one leader per connected component of the network with the best closeness centrality. The *Topology Aware* algorithm considers reliable channels, while the *CEL* algorithm tolerates message loss,

interference and collisions. Furthermore, *CEL* implements a cross-layer mechanism and a probabilistic gossip to reduce the number of sent messages and increase the stability of the algorithm.

To the best of my knowledge, no other Ω algorithm for dynamic networks in the literature uses the closeness centrality as a criterion to choose the eventual leader.

1.1.1. Topology Aware Leader Election Algorithm for Dynamic Networks

The first proposed algorithm, denoted *Topology Aware*, assumes reliable communication links with an underlying *probe system* to detect connection and disconnection of nodes. An incremental update mechanism is used to improve propagation cost of messages over the network.

Experiments were conducted on the PeerSim simulator [MJ09], comparing our algorithm to a flooding leader algorithm [VKT04] with Random Waypoint and a periodic single point of interest mobility models. Performance results show that the *Topology Aware* algorithm outperforms the flooding one considering leader choice stability, number of messages, and average distance to the leader metrics.

[MJ09] Montresor et al. (2009): 'PeerSim: A Scalable P2P Simulator'
[VKT04] Vasudevan et al. (2004): 'Design and analysis of a leader election algorithm for mobile ad hoc networks'

1.1.2. Centrality-Based Eventual Leader Election in Dynamic Networks

The first contribution assumes reliable communication channels, which are not suitable for communication environments prone to interference and message collisions. The second proposed eventual leader election algorithm for dynamic networks is the *Centrality-based Eventual Leader (CEL)*, that, similarly to *Topology Aware*, exploits topological information to improve the choice of a central leader and reduce message exchanges. However, rather than assuming an underlying probe system, *CEL* has a cross-layer neighbors detection, with a neighbor-aware mechanism, to improve the dissemination of topological knowledge and elect a central leader faster. It uses a self-pruning mechanism based on the topological knowledge, combined with probabilistic gossip, to improve the performance of information propagation.

Evaluations were conducted on the OMNeT++/INET environment [VH08; MVK19], simulating realistic MANET following the IEEE 802.11n specifications with interference, collision, and messages loss. *CEL* was compared to Gómez-Calzado *et al.* algorithm [Góm+13], with Random Walk and Truncated Lévy Walk mobility models. Results show that *CEL* presents better performance than the latter, including fewer message exchanges, shortest paths to the leader, and better stability.

[VH08] Varga et al. (2008): 'An overview of the OMNeT++ simulation environment'
[MVK19] Mészáros et al. (2019): 'Inet framework'
[Góm+13] Gómez-Calzado et al. (2013): 'Fault-tolerant leader election in mobile dynamic distributed systems'

1.2. Manuscript Organization

This manuscript is divided in six chapters.

Chapter 2 introduces background concepts on distributed systems and on leader election problems.

Chapter 3 presents related work on the classical and eventual leader election problems in both static and dynamic systems. Algorithms are compared and classified in tables according to different criteria.

Chapter 4 describes the first contribution of this thesis, the *Topology Aware* eventual leader election algorithm. Evaluation results on PeerSim of *Topology Aware* and a flooding algorithm [VKT04] are presented.

[VKT04] Vasudevan et al. (2004): 'Design and analysis of a leader election algorithm for mobile ad hoc networks'

Chapter 5 presents the second contribution, the *Centrality-based Eventual Leader (CEL)* election algorithm. Evaluations results on OMNeT++, using IEEE 802.11n communications, of CEL and the Ω algorithm of Gómez-Calzado *et al.* [Góm+13] are presented.

[Góm+13] Gómez-Calzado et al. (2013): 'Fault-tolerant leader election in mobile dynamic distributed systems'

Finally, Chapter 6 concludes the thesis and discusses different future research directions.

1.3. Publications

Four articles were published during this thesis: two in international conferences and two in French national conferences. Chapter 4 and Chapter 5 present the contributions from [Fav+20a] and [Fav+21] respectively.

[Fav+20a] Favier et al. (2020): 'Topology Aware Leader Election Algorithm for Dynamic Networks'

[Fav+21] Favier et al. (2021): 'Centrality-Based Eventual Leader Election in Dynamic Networks'

1.3.1. Articles in International Conferences

- [Fav+20a] Arnaud Favier, Nicolas Guittonneau, Luciana Arantes, Anne Fladenmuller, Jonathan Lejeune, and Pierre Sens. 'Topology Aware Leader Election Algorithm for Dynamic Networks'. In: *25th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2020, Perth, Australia, December 1-4, 2020*. IEEE, 2020
- [Fav+21] Arnaud Favier, Luciana Arantes, Jonathan Lejeune, and Pierre Sens. 'Centrality-Based Eventual Leader Election in Dynamic Networks'. In: *20th IEEE International Symposium on Network Computing and Applications, NCA 2021, Boston, MA, USA, November 23-26, 2021*. Ed. by Mauro Andreolini, Mirco Marchetti, and Dimiter R. Avresky. IEEE, 2021
  — Best student paper award

1.3.2. Articles in National Conferences

- [Fav+19] Arnaud Favier, Nicolas Guittonneau, Luciana Arantes, Anne Fladenmuller, and Pierre Sens. 'Un algorithme d'élection de leader cross-layer pour réseaux mobiles ad hoc (résumé)'. In: *COMPAS 2019 - Conférence d'informatique en Parallélisme, Architecture et Système*. Anglet, France, June 2019
- [Fav+20b] Arnaud Favier, Nicolas Guittonneau, Jonathan Lejeune, Anne Fladenmuller, Luciana Arantes, and Pierre Sens. 'Topology Aware Leader Election Algorithm for MANET'. In: *COMPAS 2020 - Conférence francophone d'informatique en Parallélisme, Architecture et Système*. Lyon, France, June 2020

2. Background

This chapter presents some existing concepts, taxonomy, and models, related to distributed systems, that are useful for this thesis. It is organized as follows: Section 2.2 details common models in distributed systems, Section 2.6 presents different communication assumptions with some details on the different centrality measures, Section 2.8 explains how to disseminate messages over the network, and Section 2.9 describes both the classical and eventual leader election problems.

2.1. Properties of Distributed Algorithms

In distributed algorithms, processes run concurrently and independently. Usually, each process has a limited view and information of the system. A distributed algorithm should run correctly, even if individual processes and communication channels operate at different speeds and even if some of the components fail [Lyn96]. Therefore, they must ensure some properties that guarantee progress and correctness. A property is an attribute of a program that is true for every possible execution of that program [MK99]. The two main properties for a distributed algorithm are *safety* and *liveness* [Lam77]:

[Lyn96] Lynch (1996): 'Distributed algorithms'

[MK99] Magee et al. (1999): *State models and java programs*

[Lam77] Lamport (1977): 'Proving the correctness of multiprocess programs'

- **Safety** asserts that nothing bad happens,
  i.e. a program does not reach a bad state.
- **Liveness** asserts that something good eventually happens,
  i.e. a program eventually reaches a good state.

These properties ensure users of the system to have a reasonable confidence in the services delivered by it. Other properties can be considered

by the algorithm, such as *fairness* for example. This ability of the system to provide services that can be trusted is called *dependability*.

2.2. Timing Models

As processes can take a long time to reply, can fail, or even simply leave the network during execution of the system, the communication between processes is uncertain. To take into account this uncertainty, the literature uses an underlying model that considers:

- *Latency*: the time required to transmit a message.
- *Computation*: the time for a process to execute a step.

Three main timing models were defined [Lyn96]:

[Lyn96] Lynch (1996): 'Distributed algorithms'

- **Synchronous model**: there exists a known finite time bound δ on latency, and there exists a known finite bound ϕ on computation.
- **Asynchronous model**: there are no bounds on latency neither on computation a process takes.
- **Partially synchronous model**: this model stands between the two previous models. It assumes the existence of a bound δ on latency and a bound ϕ on computation [DLS88], but their values are unknown. It considers that both bounds δ and ϕ exist, but only after a time t called *Global Stabilization Time (GST)*. Before GST, no bounds exist, such as the system is unstable and behaves asynchronously, and after the GST, both bounds exist such as the system became stable and behaves synchronously.

[DLS88] Dwork et al. (1988): 'Consensus in the presence of partial synchrony'

Among these three timing models, synchronous systems are included in partially synchronous systems, which are included in asynchronous systems, as shown in Figure 2.1 [Cam20].

[Cam20] Campusano (2020): 'Distributed eventual leader election in the crash-recovery and general omission failure models'

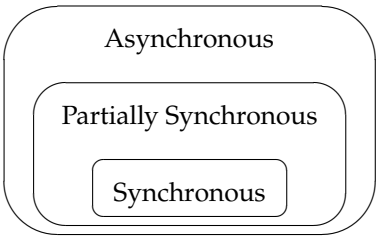

Figure 2.1.: Representation of the three main timing models.

Note that there exist many other intermediate models between the synchronous and asynchronous models.

The different types of temporal models are summarized in Table 2.1.

Table 2.1.: Table of temporal models.

	Message latency bound δ	**Computation time** bound ϕ
Synchronous model	Exist, known and finite	
Asynchronous model	Do not exist	
Partially synchronous model	Exist but unknown	

2.3. Process Failures

Processes are prone to failures during the execution of the system. A failure can be defined as a deviation from correct behavior [HT94], i.e. a behavior that does not comply with its specification. More specifically, a fault is a flaw in the system that originates from a node, which may cause unexpected behavior and result in an error. If the error is not handled and then propagates through the system, it can lead to a failure. Figure 2.2 illustrates the steps of a failure in a distributed system.

[HT94] Hadzilacos et al. (1994): *A modular approach to fault-tolerant broadcasts and related problems*

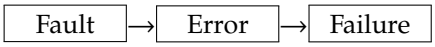

Figure 2.2.: Steps of a failure in a distributed system.

Failures can happen due to algorithm conception and programming (which can be difficult to detect, especially in distributed algorithms as processes can be at different steps of the execution of the algorithm), hardware, or hacking. A process which fails during the execution of the system is considered to be *faulty*, whereas a process which never fails during the whole execution of the system is *correct*.

There exist different modes of process failures in the literature. In addition, failures can be *permanent*, where a component stays in the faulty state forever, *transient*, where a component is in the faulty state for a finite duration, or *intermittent*, where a component successively exhibits correct and faulty behavior [Dub11]. Failures can be classified from the weak to the strongest one, i.e. a failure includes previous failures and is a subset of the next failures [Ber04; GR06; CGR11; Cal15; Cam20].
As shown in Figure 2.3, crash failures are included in omission failures, which are included in crash-recovery failures, which are included in arbitrary failures.

[Dub11] Dubois (2011): 'Tolerating Transient, Permanent, and Intermittent Failures'

[Ber04] Bertier (2004): 'Service de détection de défaillances hiérarchique'
[GR06] Guerraoui et al. (2006): *Introduction to reliable distributed programming*
[CGR11] Cachin et al. (2011): *Introduction to reliable and secure distributed programming*
[Cal15] Calzado (2015): 'Contributions on agreement in dynamic distributed systems'
[Cam20] Campusano (2020): 'Distributed eventual leader election in the crash-recovery and general omission failure models'

- **Crash**: it leads to a definitive and permanent halt of the process execution, meaning that the process has stopped and will not execute any further steps of its algorithm (fail-silent).
  Before the failure, the process behaves correctly, but when the failure arises, the process is considered shutdown and will not come back in the system.
  If the system is not asynchronous, other processes may be able to detect its state, by not receiving any response from this process when invocation messages are repeatedly sent to it. However, this

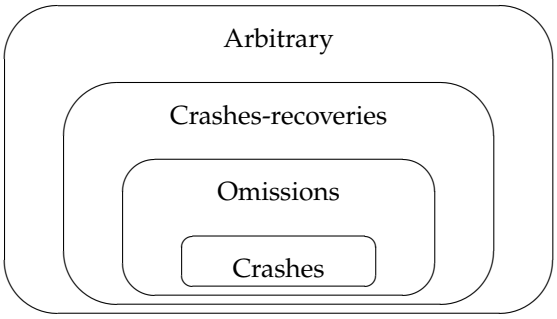

Figure 2.3.: Failure modes of a process, by [GR06].

crash failure detection mechanism relies on the use of *timeouts*, which is a period of time a process has for something to occur (assuming synchrony assumptions).

- **Omission**: this kind of failure arises when a process momentary fails to perform the actions it is supposed to achieve. The process temporarily halts its activity, and afterwards, takes back its normal behavior.
- **Crash-recovery**: also called *fail-recovery*, the process which fails stops any further steps of its algorithm, but may recover and resume its execution afterwards. If the process uses *volatile memory*, information contained in its memory before the failure will be lost upon recovery, and the process is initialized again. If the process uses *stable memory*, its memory will be recovered.
- **Arbitrary failures**: also called *Byzantine* [1][LSP19] or malicious failures, this type of failure considers that any type of error may occur, such that the behavior of processes becomes unpredictable. For instance, a process may set wrong values in its memory, returns wrong values to a message, or arbitrarily omits to reply.

1: The term *Byzantine* comes from the allegory of the *Byzantine Generals Problem*, where a group of generals of the Byzantine army, who communicate only by messenger, must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others [LSP19].

[LSP19] Lamport et al. (2019): 'The Byzantine generals problem'

The different modes of process failure are summarized in Table 2.2. Note that this list is not exhaustive and there exist other failures, such as fail-safe, fail-stop, authenticated byzantine, etc.

Table 2.2.: Table of process failures modes.

Failures	Component	Description
Crash	Process	Permanent halt of process.
Omission	Process	Temporary halt of process.
Fail-recovery	Process	Halt of process execution, but may recover and resume afterwards.
Arbitrary/Byzantine	Process	Arbitrarily omits step or reply, sets or return wrong value.

2.4. Communication Channels

When using the messages passing approach, communication is achieved by transmitting messages on a communication channel, which is a directional link between two processes, i.e. a logical abstraction of the

physical medium. A communication from process p to q requires a unidirectional communication channel from p to q. A *bidirectional* communication channel, also denoted *two-way*, assume two communication channels: a first from p to q, and a second one from q to p.

A process p sends a message m using the primitive *emit* at the application layer, which inserts m in its buffer. The buffer of process p *sends* the message m on the communication channel, which transmits m to the buffer of process q. The latter *receives* it and *delivers* it to the application layer of process q. Note that these buffers are typically offered by the operating system.

An example of communication from process p to process q is shown in Figure 2.4.

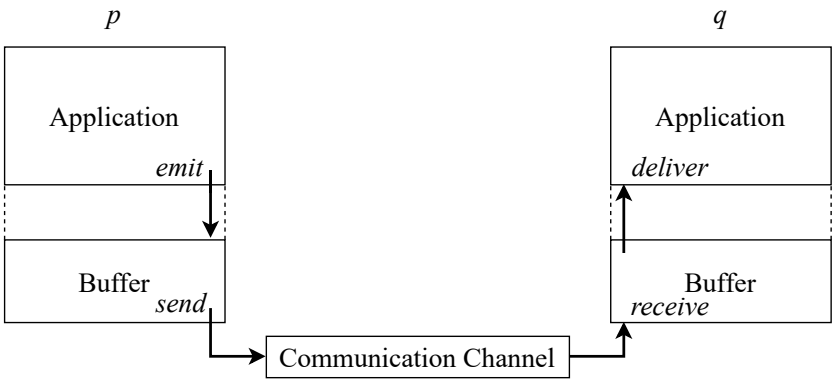

Figure 2.4.: Example of processes communication.

In the literature, there exist several types of communication channels such as *reliable*, *eventually reliable*, *fair-lossy*, and *unreliable* [FJR06]. All of them assume the three following common properties.

- No message **creation** (also called *validity* [Ray13]): if a process p receives a message from a process q, then q sent it to p.
- No message **duplication** (also called *integrity* [Ray13]): if a process p sends a message to a process q, then the process q will receive the message at most one.
- No message **alteration**: if a process p receives a message from a process q, then the message is exactly the same as the one sent by q, without any modification or corruption.

[FJR06] Fernández et al. (2006): 'Eventual Leader Election with Weak Assumptions on Initial Knowledge, Communication Reliability, and Synchrony'

[Ray13] Raynal (2013): *Distributed algorithms for message-passing systems*

Each type of communication channel offers different guarantees in terms of message loss.

- **Reliable**: channels ensure that when a correct process p sends a message to a *correct* process q, the message will eventually be delivered to q [GR06].

[GR06] Guerraoui et al. (2006): *Introduction to reliable distributed programming*

- **Eventually reliable**: channels assume the existence of a time t after which all messages sent by process p are eventually received by process q. Messages sent before that time t may be lost [BCT96].
- **Fair-lossy**: channels ensure that when a correct process p sends a message an infinity number of times to a correct process q, then q will deliver the message an infinite number of times [GR06]. This property guarantees that a link does not systematically drop every message.
- **Unreliable**: also called *lossy* channels. In this case, messages can be lost, without notifying the processes.

[BCT96] Basu et al. (1996): *Solving problems in the presence of process crashes and lossy links*

[GR06] Guerraoui et al. (2006): *Introduction to reliable distributed programming*

The type of communication channel can also specify bounds for the reception of a message, such as the *eventually timely channels* or *timely channels*, or tolerate message loss without any bound, such as *lossy asynchronous channels*:

- **Eventually timely**: a link from p to q is eventually timely if it is either a reliable or an eventually reliable channel, satisfying that there exists a bound δ and a time t such that if p sends a message to q at time t and q is correct, then q receives the message from p by time $t + \delta$ [Agu+04]. Note that if the link is eventually reliable, messages sent before time t can be lost.
- **Timely**: an eventually timely channel whose bounds hold from $t = 0$ [Agu+04].
- **Lossy asynchronous**: messages can be arbitrarily delayed (i.e. there is no bound on message delay) or even be lost (an arbitrary number of messages can be lost, if not all) [Agu+03; LMS11]. Every message that is not lost is eventually received.

[Agu+04] Aguilera et al. (2004): 'Communication-efficient leader election and consensus with limited link synchrony'

[Agu+03] Aguilera et al. (2003): 'On implementing omega with weak reliability and synchrony assumptions'

[LMS11] Larrea et al. (2011): 'Communication-efficient leader election in crash-recovery systems'

Note that in this thesis, there is no assumption on message ordering, and channels are not required to be *FIFO*. A FIFO channel stands for First-In First-Out (FIFO), where messages are received to the process by order they were sent.

The different types of communication channels explained previously are summarized in the following Table 2.3.

Table 2.3.: Table of communication channels types.

Channel types	Description
Reliable	When a correct process p sends a message to q which is correct, the message will eventually be delivered by q.
Eventually reliable	Existence of a time t after which all messages sent by p are eventually delivered by q.
Fair-lossy	A correct process p sends a message infinitely to a correct process q, q delivers it infinitely.
Unreliable/Lossy	Messages can be lost while they are in transit, without any notification.
Eventually timely	Existence of a bound δ and a time t after which all messages sent by p are delivered by q by time $t + \delta$.
Timely	Eventually timely channel whose bounds holds from $t = 0$.
Lossy asynchronous	Messages can be lost or arbitrarily delayed. Every message that is not lost is eventually received.

2.5. Failures of Communication Channels

There exist different modes of failures for communication channels in the literature.

Section 2.3 presents a similar classification of failure modes for processes that can be adapted to communication channels. As shown in Figure 2.5, crash failures are included in omission failures, which are included in arbitrary failures.

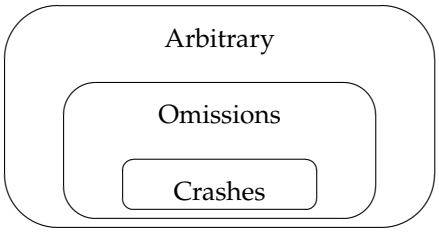

Figure 2.5.: Failure modes of a communication channel, by [GR06].

- **Crash**: in case of a crash of a communication channel, the latter is down and stops transmitting messages.

- **Channel omission failure**: it corresponds to a message loss in a communication channel, i.e. failing to transmit the message from the outgoing buffer to the incoming one.

- **Arbitrary failures**: for communication channels, arbitrary failures include messages corruption, messages omission, messages creations or multiple message deliveries. Some arbitrary failures in communication channels can be detected and fix by using *error detection and correction*, such as checksums or cyclic redundancy

checks (CRC).
Therefore, information exchanged by processes on communication channels is not reliable anymore, and security steps may be then required.

The different modes of failure are summarized in the following Table 2.4.

Table 2.4.: Table of failure modes of communication channels.

Failures	Component	Description
Crash	Channel	Stops transmitting messages.
Channel omission	Channel	Message loss.
Arbitrary/Byzantine	Channel	Arbitrarily corrupts, omits, creates or delivers message.

2.6. Distributed Systems

Distributed systems can be represented by communication graphs, composed of two elements: *vertices* and *edges*. Vertices represent processes or nodes and edges represent direct communication channels between two vertices. If the communication channel is one-way, i.e. only from vertex i to vertex j without the way back, edges are oriented and the graph is called a *directed* graph. However, this thesis assumes only bidirectional communication channels, i.e. unoriented edges in undirected graphs.

Two processes connected with a channel are called *neighbors*, such as process i is a neighbor of process j if they there is a direct edge between them. The set of processes connected to process i is called the *neighborhood* of i. For example, considering Figure 2.6, process A is a neighbor of process B, and the neighborhood of process B is the set of processes $\{A, C, F\}$.

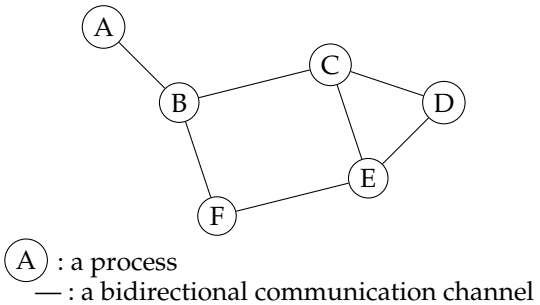

Figure 2.6.: Representation of a communication graph with bidirectional communication channels.

Neighbors of a process have a communication distance of *one hop*, and are called *1-hop neighbors*, as they can directly communicate with it. Neighbors of neighbors of a process are at a distance of *2-hop* of this process, and so on. When the distance is higher than one, the term *multi-hop* can be used. The hop count distance refers to the number of processes between two processes. For example, considering Figure 2.6,

process B is a 1-hop neighbor of process A, and process E is a 3-hop neighbor of process A.

As processes are autonomous entities, when a process needs to communicate with a process located further than 1-hop, intermediate processes located between the sender and the final receiver, are called *relays* and act as routers, establishing a path from the sender to the destination process over which the message is transmitted.

Different assumptions about network knowledge can be made: *known* networks, where nodes have a knowledge of the system, and *unknown* networks, with the weakest possible assumptions about knowledge, communication graph, channel connectivity and reliability, in order to be as close as possible to reality.

- **Known** network: system where processes know the set of processes that composed the system denoted Π.
- **Unknown** network: system where processes have no information about other nodes in the system, including the total number of processes.

There are two main types of networks: *static* and *dynamic* networks.

2.6.1. Static Systems

Originally, distributed systems were based on *static networks*. A static system is composed of a finite set of nodes, where the membership of the system can only change because of failures. The *communication graph* is static during the execution of the system, however, network partitions can arise due to failures. Nodes do not move, nor leave, neither join the system.

As an example, the links of a static system can physically be either wires between nodes, or wireless radio wave antennas with a fixed transmission range.

2.6.2. Dynamic Systems

With the advent of peer-to-peer technologies and mobile devices such as mobile phones, mobile sensors, autonomous vehicular or drones, nodes of a network are able to move, join, or leave the system. They can also fail. Therefore, the membership of the system is no longer finite but may vary over time. Communication channels between nodes are non-persistent and can also change over time, which may lead to network partitions.

There is no unique definition of dynamic systems in the literature [Agu04; KLO10; BRS12; Lar+12]. Some authors define it as a distributed system where the communication graph evolves over time. Others give the

[Agu04] Aguilera (2004): 'A pleasant stroll through the land of infinitely many creatures'
[KLO10] Kuhn et al. (2010): 'Distributed computation in dynamic networks'
[BRS12] Biely et al. (2012): 'Agreement in directed dynamic networks'
[Lar+12] Larrea et al. (2012): 'Specifying and implementing an eventual leader service for dynamic systems'

definition of a model where nodes join and leave the system at arbitrary times during the system execution.

Dynamic systems can be represented by a dynamic communication graph. Many works characterize the dynamics of graphs, such as the *Time-Varying Graph (TVG)* proposed by Casteigts *et al.* [Cas+11] (where relations between nodes take place over a time span $\mathcal{T}$, with a presence function indicating whether a given edge is available at a given time), Flocchini *et al.* [FMS09] and Tang *et al.* [Tan+10], *evolving graphs* by Ferreira [Fer04], *temporal network* by Kempe *et al.* [KKK02], *graphs over time* by Leskovec *et al.* [LKF07], or *temporal graph* by Kostakos [Kos09].

[Cas+11] Casteigts et al. (2011): 'Time-varying graphs and dynamic networks'

[FMS09] Flocchini et al. (2009): 'Exploration of periodically varying graphs'

[Tan+10] Tang et al. (2010): 'Characterising temporal distance and reachability in mobile and online social networks'

[Fer04] Ferreira (2004): 'Building a reference combinatorial model for MANETs'

[KKK02] Kempe et al. (2002): 'Connectivity and inference problems for temporal networks'

[LKF07] Leskovec et al. (2007): 'Graph evolution: Densification and shrinking diameters'

[Kos09] Kostakos (2009): 'Temporal graphs'

Physically, the links of a dynamic system can be, for example, wireless communication with radio wave antennas. A representation of a dynamic network is given in Figure 2.7.

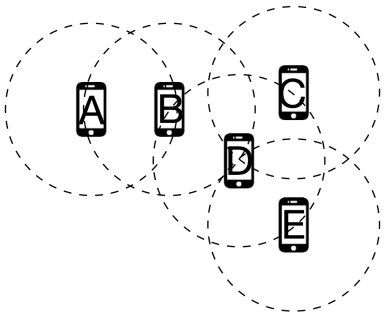

Figure 2.7.: Representation of a dynamic network.

2.7. Centralities

In a graph, some vertex can have a higher importance than others. *Importance* can have a wide range of meaning, from the type of flow through the network [Bor05], to the involvement of the cohesion in the network [BE06], or even the topological position in the network [Bav50]. Indicators of centrality can be used to assign a number or a ranking to vertices, according to their position in the network, identifying the *importance* of the vertices in a graph [Fre78; Bor05; Gha18].

[Bor05] Borgatti (2005): 'Centrality and network flow'

[BE06] Borgatti et al. (2006): 'A graph-theoretic perspective on centrality'

[Bav50] Bavelas (1950): 'Communication patterns in task-oriented groups'

[Fre78] Freeman (1978): 'Centrality in social networks conceptual clarification'

[Bor05] Borgatti (2005): 'Centrality and network flow'

[Gha18] Ghanem (2018): 'Temporal centralities: a study of the importance of nodes in dynamic graphs'

In terms of centrality, this thesis focuses on the three following centralities based on the shortest path:

- **Degree centrality**: the degree centrality is defined as the number of neighbors of a process.
- **Closeness centrality**: the closeness centrality, or closeness, of a vertex, is the average length of the shortest path between the vertex and all other vertices in the graph [Bav50]. The more central is a vertex, the closer it is to all other vertices. The closeness centrality characterizes the ability of a node to spread information over the graph.

[Bav50] Bavelas (1950): 'Communication patterns in task-oriented groups'

Alex Balevas [Bav50] defined in 1950 the closeness centrality of a vertex as follows [Sab66]:

[Bav50] Bavelas (1950): 'Communication patterns in task-oriented groups'

[Sab66] Sabidussi (1966): 'The centrality index of a graph'

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

where $d(y,x)$ is the shortest path between vertex y and vertex x.

- **Betweenness centrality**: the betweenness centrality measures the number of times a vertex acts as a relay (router) along shortest paths between other vertices. Even if previous authors have intuitively described centrality as being based on betweenness, betweenness centrality was formally defined by Freeman in 1977 [Fre77].
  The betweenness of a vertex x is defined as the sum, for each pair of vertices (s,t), of the number of shortest paths from s to t that pass through x, over the total number of shortest paths between vertices s and t,
  It can be represented by the following formula [Bra01]:

  [Fre77] Freeman (1977): 'A set of measures of centrality based on betweenness'

  [Bra01] Brandes (2001): 'A faster algorithm for betweenness centrality'

  $$C_B(x) = \sum_{s \neq x \neq t} \frac{\sigma_{st}(x)}{\sigma_{st}}$$

  where σ_{st} denotes the total number of shortest paths from vertex s to vertex t (with $\sigma_{ss} = 1$ by convention), and $\sigma_{st}(x)$ is the number of those shorter paths that pass through x.

A visual representation given in Figure 2.8 compares the degree centrality with the closeness centrality and the betweenness centrality of the same graph [Tap15].

[Tap15] Tapiocozzo-Wikipedia (2015): *Centrality - Wikipedia*

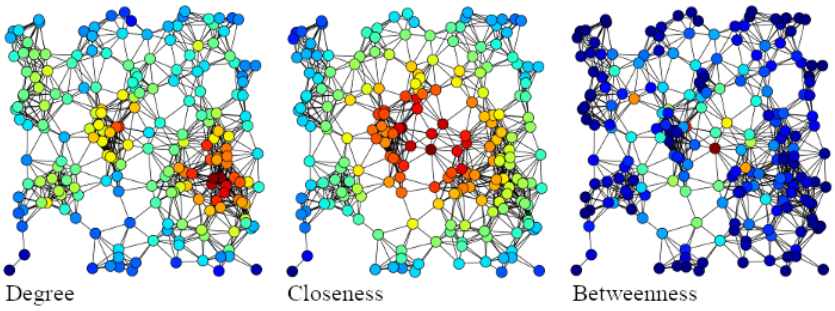

Figure 2.8.: Comparison between degree centrality, closeness centrality and betweenness centrality of the same graph, by *Tapiocozzo* [Tap15]

The more red a vertex is, the higher its centrality. The more blue a vertex is, the lower its centrality. In Figure 2.8, the most central vertex for the degree centrality is different than for the closeness and betweenness centrality. However, closeness and betweenness centrality are not always equal.

The differences between closeness and betweenness centrality are the following: closeness is generally regarded as a measure of access efficiency, i.e. how long it will take to spread information from x to all

other vertices sequentially; whereas betweenness is usually interpreted as a measure of the dependence of others on a given vertex, i.e. the number of times a vertex is present on the shortest path between two other vertices [BBF16; Du19].

[BBF16] Brandes et al. (2016): 'Maintaining the duality of closeness and betweenness centrality'
[Du19] Du (2019): 'Social network analysis: Centrality measures'

2.8. Messages Dissemination

To share messages from a source vertex to other connected vertices of a graph, communication between vertices can be achieved using different protocols. This thesis considers message flooding and probabilistic gossip protocols.

Flooding Protocol

In a flooding protocol, each received message is sent through all outgoing edges, except the one on which it arrived [TW+96]. Therefore, each vertex acts both as a transmitter and a receiver of the message, forwarding every received message to its neighbors, except the one that sent it the message.

[TW+96] Tanenbaum et al. (1996): *Computer networks*

There are two types of flooding: *uncontrolled* and *controlled* [CBL18].

[CBL18] Coutinho et al. (2018): 'Design guidelines for information-centric connected and autonomous vehicles'

- **Uncontrolled flooding**: vertices forward messages to all of their neighbors, except the one from which it received the message.
- **Controlled flooding**: vertices remember received messages and, therefore, drop already received messages.

While flooding protocols are easy to implement, they can be expensive in terms of wasted bandwidth.

To reduce the number of messages sent, flooding protocols can use a *hop count* measure contained in the header of messages, that is decremented at each hop, i.e. each time the message is sent from a vertex to all its neighbors. When the hop counter reaches 0, the message stops being sent and is discarded.

Gossip Protocol

Gossip protocols can be used to reduce the cost of flooding protocol, based on the way epidemics spread [Dem+87]. Also called *epidemic* protocols, gossip protocols spread information in a manner similar to a viral infection in a biological population, where a virus plays the role of a piece of information, i.e. a message, and an infection plays the role of learning about the information, i.e. receiving a message [SGK11].

[Dem+87] Demers et al. (1987): 'Epidemic algorithms for replicated database maintenance'

[SGK11] Serugendo et al. (2011): *Self-organising Software*

There are two main types of gossip protocols:

- Random *probabilistic* dissemination, suitable for Peer-to-Peer (P2P) systems, where a node randomly chooses which of its neighbors to send the information. Kermarrec *et al.* [KMG03] consider that each node has a randomized partial knowledge of the system, i.e. a list of process identifiers stored locally. Then, a process that must forward a received information, randomly picks k processes from its local list and sends them the information. This mechanism is called *peer-sampling* service and provides a primitive that returns a random process drawn from the list [Jel+07].

[KMG03] Kermarrec et al. (2003): 'Probabilistic reliable dissemination in large-scale systems'

[Jel+07] Jelasity et al. (2007): 'Gossip-based peer sampling'

- Haas *et al.* [HHL02] proposed several gossip protocols for *ad hoc networks* that use probabilities. Combined with the number of hops or the number of times the same message is received, the protocols choose if a node broadcast a message to all its neighbors or not, reducing thus the number of messages propagated in the system. The authors show that gossiping with a probability between 0.6 and 0.8 ensures that almost every node of the system gets the message, with up to 35% fewer messages in some networks compared to flooding.

[HHL02] Haas et al. (2002): 'Gossip-based ad hoc routing'

The difference between Kermarrec *et al.* algorithms and Haas *et al.* algorithms, is that Haas *et al.* algorithms gossip a message to all its neighbors using a probability, whereas Kermarrec *et al.* algorithms gossip a message to a subset of neighbors randomly chosen.

In this thesis, a probabilistic broadcast from Haas *et al.* is used to reduce the number of messages exchanged. This type of gossip is more suitable to wireless networks, since a message is received by all nodes within the wireless transmission range of the sending node.

2.9. Leader Election

Electing a leader is one of the fundamental problems in distributed computing [Pel90]. Also called *coordinator election* or *leader finding*, a leader election selects a single node among the ones of a distributed system, according to some election criterion.

[Pel90] Peleg (1990): 'Time-optimal leader election in general networks'

The leader election problem was proposed for the first time [2] in 1977 by Gérard Le Lann [Le 77], a French Computer Researcher.

2: Gallager [Gal77] and Kanellakis [Kan78] also defined the leader election problem in 1977 at the Massachusetts Institute of Technology (MIT). However, both papers are not accessible online and some sources indicates that they might come from 1978.

In this thesis, two leader election problems are considered, denoted the *classical* leader election problem (Section 2.9.1), and the *eventual* leader election problem (Section 2.9.2).

[Le 77] Le Lann (1977): 'Distributed Systems-Towards a Formal Approach.'

2.9.1. Classical Leader Election

In 1980, Angluin shows that there is no deterministic leader election algorithm in *anonymous* and *uniform* networks [Ang80], where anonymous means that processes do not have a unique identifier, so there is no way

[Ang80] Angluin (1980): 'Local and global properties in networks of processors'

to distinguish a process i from another process j, and uniform means that the number of processes in the system is not known in advance by processes, i.e. not hardcoded in the algorithm [AW04]. To circumvent this impossibility, the symmetry, i.e. the fact that all processes have the same local state which makes a leader election impossible [Ray13], needs to be broken. Therefore, the literature proposes two solutions:

[AW04] Attiya et al. (2004): *Distributed computing: fundamentals, simulations, and advanced topics*

[Ray13] Raynal (2013): *Distributed algorithms for message-passing systems*

- Nodes in the system have unique identifiers and, initially, can be either aware or unaware of the identifier of other nodes.
- Randomized algorithms where each process has an independent random number generator which allows to break the symmetry, such as the algorithm proposed by Itai *et al.* [IR81] which is a Las Vegas algorithm [Bab79], i.e. it always gives a correct answer but have a probabilistic running time.

[IR81] Itai et al. (1981): 'Symmetry breaking in distributive networks'

[Bab79] Babai (1979): 'Monte-Carlo algorithms in graph isomorphism testing'

This thesis focuses on the first solution called *non-anonymous*, where nodes have unique identifiers in the system, but initially do not have knowledge of the identity of the other nodes in the system.

At the beginning of the leader election algorithm, all nodes in the system are unaware of which node is the leader [Hal15]. At the end of the election, there is exactly one correct node that has been elected as the leader among the set of nodes, and each correct node throughout the network recognizes this same and unique node as the leader. The number of nodes in the system can be known or unknown by the algorithm, as seen in Section 2.6.

[Hal15] Haloi (2015): *Apache zookeeper essentials*

Formally, the classical leader election problem has the two following properties [MWV00; AW04]:

[MWV00] Malpani et al. (2000): 'Leader election algorithms for mobile ad hoc networks'

[AW04] Attiya et al. (2004): *Distributed computing: fundamentals, simulations, and advanced topics*

- There should never be more than one leader (i.e. there is zero or one process considered as the leader by the processes of the system).
- Eventually, there is a leader (i.e. all processes of the system eventually agree on the identity of a single leader).

During the execution of a distributed system, failures may arise and if the system is dynamic, processes may join and leave the system overtime, as seen in Section 2.6.2. Therefore, the system may be partitioned into multiple connected components. Every connected component of the system has a unique leader. Therefore, Malpani *et al.* [MWV00] modify the definition of a classical leader election to take into account multiple components:

[MWV00] Malpani et al. (2000): 'Leader election algorithms for mobile ad hoc networks'

- Any component whose topology is static sufficiently long will eventually have exactly one leader (i.e. every process will eventually agree on the identity of the same leader of its component).

In a classical leader election algorithm, during the election, nodes are aware that they do not currently have the identify of the leader, and therefore, are in an *unstable* state. This state can be indicated by a *done* flag, as suggested by Raynal [Ray13], or by returning $\perp$ like in the

[Ray13] Raynal (2013): *Distributed algorithms for message-passing systems*

algorithm of Vasudevan *et al.* [VKT04].

[VKT04] Vasudevan et al. (2004): 'Design and analysis of a leader election algorithm for mobile ad hoc networks'

2.9.2. Eventual Leader Election

The eventual leader election is a key component for many fault-tolerant services in asynchronous distributed systems. An eventual leader election considers that the uniqueness property of the elected leader is **eventually** satisfied for all **correct** nodes in the system. Several consensus algorithms such as Paxos [Lam98] or Raft [OO14], adopt a leader-based approach. They rely on an eventual leader election service, also known as the Ω failure detector [CHT96]. Consensus is a fundamental problem of distributed computing [Pel90], used by many other problems in the literature, like state machine replication or atomic broadcast.

[Lam98] Lamport (1998): 'The part-time parliament'
[OO14] Ongaro et al. (2014): 'In search of an understandable consensus algorithm'
[CHT96] Chandra et al. (1996): 'The weakest failure detector for solving consensus'
[Pel90] Peleg (1990): 'Time-optimal leader election in general networks'

Problem Definition

A first definition of the eventual leader election problem is given. Then, a second definition is presented based on the Ω failure detector. Finally, this second definition is enhanced to tolerate asynchronous dynamic systems.

Formally, an eventual leader election algorithm has the two following properties:

- Several (correct or not) leaders can coexist before a single correct process is elected as the leader [Lar+12].
- Every correct process of the system eventually agrees on the identity of the same correct leader.

[Lar+12] Larrea et al. (2012): 'Specifying and implementing an eventual leader service for dynamic systems'

The Ω failure detector introduced by Chandra *et al.* in 1996 [CHT96], considering a static distributed system with reliable communication links and known membership, satisfies the following property [Cal15]:

- **EL** (Eventual Leadership): there is a time after which all correct processes trust the same correct process, i.e. the leader.

[CHT96] Chandra et al. (1996): 'The weakest failure detector for solving consensus'
[Cal15] Calzado (2015): 'Contributions on agreement in dynamic distributed systems'

In a dynamic system, as processes can join and leave the system, the size of the system may increase or decrease overtime. Therefore, Larrea *et al.* [Lar+12] have defined the Dynamic Omega failure detector class denoted $\Delta\Omega$, which provides an eventual leader election algorithm in an asynchronous dynamic system (denoted $\Delta\mathcal{AS}$). $\Delta\Omega$ have the two following properties, assuming that there is at least one process in the system at any time:

[Lar+12] Larrea et al. (2012): 'Specifying and implementing an eventual leader service for dynamic systems'

- **EL_NI** (Eventual Leadership in Non-Increasing systems): if after some time, no process joins the system, a leader must eventually be elected.

- **EL_ND** (Eventual Leadership in Non-Decreasing systems): if after some time, no process leaves the system, a leader must eventually be elected, and the new processes joining after a leader has been elected eventually adopt it.

Note that, by definition, $\Omega \subset \Delta\Omega$ [Cal15].

[Cal15] Calzado (2015): 'Contributions on agreement in dynamic distributed systems'

According to Larrea *et al.* [Lar+12] and considering dynamic (denoted $\Delta\mathcal{S}$) systems, the class $\Delta\Omega$ of failure detectors includes all failure detectors which satisfies both properties *EL_NI* and *EL_ND*.

[Lar+12] Larrea et al. (2012): 'Specifying and implementing an eventual leader service for dynamic systems'

Note that the dynamics of the system may imply that no process stays "long enough" in the system, in which case it is possible that no common process will ever be elected [Lar+12]. Therefore, a leader, which can be temporary, is required to be elected only when the size of the system does no longer increase or decrease, during "long enough" period of time. The two properties *EL_NI* and *EL_ND* satisfy this requirement [Lar+12].

Implementing Eventual Leader Election

Solving the eventual leader election problem is possible by implementing an eventual leader service, also called a *leader oracle* [FJR06]. Such a service consists in providing the processes with a primitive denoted *Leader()* that [Ray07]:

[FJR06] Fernández et al. (2006): 'Eventual Leader Election with Weak Assumptions on Initial Knowledge, Communication Reliability, and Synchrony'

[Ray07] Raynal (2007): 'Eventual leader service in unreliable asynchronous systems: Why? how?'

1. Returns the identity of a process of the system each time it is called.
2. Ensures that there exists a time after which it always returns the identity of the same correct process.

As there is no knowledge of when the leader is elected, several leaders can coexist at time t, such as two processes can have different leaders, but eventually, the same correct process is elected as the leader and its identity is known by all correct nodes. The Ω failure detector provides such a *Leader()* primitive satisfying these properties [CHT96].

[CHT96] Chandra et al. (1996): 'The weakest failure detector for solving consensus'

[GR04] Guerraoui et al. (2004): 'The information structure of indulgent consensus'

[GL08] Guerraoui et al. (2008): 'A general characterization of indulgence'

[Lar+12] Larrea et al. (2012): 'Specifying and implementing an eventual leader service for dynamic systems'

A distributed algorithm based on Ω is indulgent, i.e. it never violates the safety property of the consensus, if the algorithm never produces incorrect outputs regardless of the behavior of Ω [GR04; GL08; Lar+12]. Therefore, if Ω behaves correctly (its behavior corresponds to its specification), the algorithm produces correct outputs.

Ω cannot be implemented in pure asynchronous distributed systems prone to process crashes. Otherwise, it would solve the consensus proven to be impossible in such systems [FLP85; MRT04]. There exist two main approaches to implement Ω in the literature:

[FLP85] Fischer et al. (1985): 'Impossibility of Distributed Consensus with One Faulty Process'

[MRT04] Mostefaoui et al. (2004): 'Crash-resilient time-free eventual leadership'

[LFA00] Larrea et al. (2000): 'Optimal Implementation of the Weakest Failure Detector for Solving Consensus'

[Agu+04] Aguilera et al. (2004): 'Communication-efficient leader election and consensus with limited link synchrony'

1. **Timer-based**: it considers additional synchrony assumptions, where some or all links are eventually synchronous [LFA00; Agu+04]. In this case, other assumptions are usually included such as the maximum number of processes that can crash, the number of eventually synchronous links, etc.

2. **Time-free**: it considers a property on the message exchange pattern based on query-response, with a maximum number of processes that can crash [MMR03; Ara+13]. It assumes that responses from some nodes eventually and permanently arrive among the first ones.

[MMR03] Mostefaoui et al. (2003): 'Asynchronous implementation of failure detectors'
[Ara+13] Arantes et al. (2013): 'Eventual Leader Election in Evolving Mobile Networks'

Many algorithms use the maximum number of faults, i.e. the number of processes that can crash, denoted f, where $1 \leq f < n$ (n being the number of processes $n = |\Pi|$) [MOZ05; FJR06; Hut+08]. An important result by Aguilera *et al.* [Agu+04] shows that implementing the Ω failure detector in a partially synchronous system with n process and up to f process crashes, requires the existence of some correct processes called $\diamond f$-source, (whose identity does not have to be known) with f outgoing links that are eventually timely. If $f = 1$, implementing Ω requires only one eventual timely link. Ω can be implemented if there is at least one correct $\diamond f$-source (the identity of the $\diamond f$-source does not have to be known).

[MOZ05] Malkhi et al. (2005): 'Ω meets paxos: Leader election and stability without eventual timely links'
[FJR06] Fernández et al. (2006): 'Eventual Leader Election with Weak Assumptions on Initial Knowledge, Communication Reliability, and Synchrony'
[Hut+08] Hutle et al. (2008): 'Chasing the weakest system model for implementing Ω and consensus'
[Agu+04] Aguilera et al. (2004): 'Communication-efficient leader election and consensus with limited link synchrony'

Leader Based Consensus

Ω allows to solve the consensus problem with the weakest assumptions on process failures considering a majority of correct processes. In consensus problems, each process proposes a value, and all correct processes should eventually decide on a common value among the proposed ones [FLP85; CT96].

[FLP85] Fischer et al. (1985): 'Impossibility of Distributed Consensus with One Faulty Process'
[CT96] Chandra et al. (1996): 'Unreliable failure detectors for reliable distributed systems'

The following properties are defined:

- **Termination** (a *liveness* property): All correct processes eventually decide on a proposed value.
- **Validity** (a *safety* property): the decided value is one of the proposed values.
- **Agreement** (a *safety* property): All correct process must agree on the same value.

Fisher *et al.* [FLP85] proved the impossibility to deterministically achieve consensus in a completely asynchronous system, if at least one node is prone to crash failure, due to the inherent difficulty of determining whether a process has crashed, or is slow to reply/compute.

[FLP85] Fischer et al. (1985): 'Impossibility of Distributed Consensus with One Faulty Process'

Several consensus algorithms use a leader election to solve the consensus, such as Paxos [Lam98] and Raft [OO14].

[Lam98] Lamport (1998): 'The part-time parliament'
[OO14] Ongaro et al. (2014): 'In search of an understandable consensus algorithm'

Lamport introduced in 1989 Paxos [Lam98], which uses an eventual leader election to guarantee the safety properties with weak assumptions. The uniqueness property of the leader is required to ensure the protocol to make progress, otherwise, two processes thinking they are leaders may stall the protocol by continuously proposing conflicting updates. Once a single leader is elected, it is the only one that tries to issue proposals.

In 2004, Ongaro and Ousterhout introduced Raft [OO14], a leader-based consensus algorithm which uses randomized timers to elect leaders, responsible for log replication to the follower nodes. A leader election is triggered at the initialization of the algorithm, when the existing leader fails or disconnects, or if no heartbeat is received by the followers of the leader after a timeout. Note that Raft, like other leader-based consensus algorithms, is not tolerant to Byzantine fault, as the nodes trust the elected leader.

[OO14] Ongaro et al. (2014): 'In search of an understandable consensus algorithm'

2.10. Conclusion

This chapter has presented well-known concepts of distributed computing, systems, and algorithms such as the definition of the safety and liveness properties, the timing models, which vary from synchronous to asynchronous, communication channel types related to reliability and time bounds, and some failure models that range from crash to arbitrary failures. The communication graph with the specificity of static and dynamic systems were described, including a discussion of some centrality approaches, especially the closeness one, as well as the flooding and probabilistic gossip protocols.

Both the classical and eventual leader election problems were described, the latter including the Ω failure detector implementation. The consensus problem and its properties are described and an overview of a few leader based consensus algorithm is given. The concepts and models introduced in this chapter will be useful in the following chapters.

Related Work 3.

This chapter summarizes some related work on both the classical (Section 3.1) and the eventual (Section 3.2) leader election problems. Each of these sections presents existing algorithms of the literature for static and dynamic systems. In static systems, nodes can fail and recover, as presented in Section 2.6.1, whereas in dynamic systems, nodes can move, join and leave the system, as well as fail and recover, as described in Section 2.6.2.

3.1. Classical Leader Election Algorithms

The classical leader election problem defined in Section 2.9.1 has the two following properties:

- There should never be more than one leader
- Eventually there is a leader.

In this section, the presented works assume a unique identifier for every process in the system, where n and e correspond to the number of processes in the network, and the number of edges respectively.

3.1.1. Static Systems

There exist several works in the literature on classical leader election problem, essentially distinguishable on the topology and complexity. Usually, they are extrema-finding, i.e. the highest or lowest process identifier is used as the election criterion.

As a reminder, the complexity of an algorithm is provided using the mathematical Big O notation, describing the limiting behavior of a function when the argument tends towards infinity [Cor+09]:

[Cor+09] Cormen et al. (2009): *Introduction to algorithms*

- $\Theta(g(n))$ gives an asymptotically tight bound (i.e. bounds both from above and below).
- $O(g(n))$ gives an asymptotic upper bound.

- $\Omega(g(n))$ gives an asymptotic lower bound.

Note that the $\Omega(g(n))$ complexity should not be confused with the Ω failure detector introduced in Section 2.9.2.

Ring Topologies

The leader election problem was studied for ring topologies since 1977 with the first solution proposed by Le Lann [Le 77] which required $O(n^2)$ messages. In 1979, Chang and Roberts [CR79] improved the algorithm of Le Lann by reducing the number of messages to $O(n \log n)$ on average, still with a worst-case scenario of $O(n^2)$ messages. Both solutions consider unidirectional communication. In Chang and Roberts algorithm, a process compares its identifier with the one received by message from its right neighbor in the ring. If the former is lower than the identifier in the message, the process forwards the message to its left neighbor, otherwise, it discards it. When at phase i a process receives both of its messages, if the latter traveled overall the ring, the node is elected as the leader, otherwise the process starts phase $i + 1$.

[Le 77] Le Lann (1977): 'Distributed Systems-Towards a Formal Approach.'

[CR79] Chang et al. (1979): 'An improved algorithm for decentralized extrema-finding in circular configurations of processes'

Hirschberg and Sinclair proposed in 1980 [HS80] an algorithm requiring $O(n \log n)$ messages in the worst case, by assuming bidirectional communication on the ring, i.e. processes can send messages to the left and right sides. The algorithm executes by phases. In phase i, processes send messages along paths of length 2^i. If the process identifier is lower than the received one by message from its left (right) neighbor, it forwards the message to its right (left) neighbor. Otherwise, it drops the message. If a message of a process travels over all the processes, it has won the election. They also conjectured that any unidirectional solution must be $\Omega(n^2)$. However, Dolev [DKR82] and Peterson [Pet82] both shown in 1982 that the conjecture of Hirschberg and Sinclair is false, by presenting unidirectional algorithms requiring at most $O(n \log n)$ messages. By improving the algorithm of Peterson, Dolev obtained a $1.356n \log n + O(n)$ messages algorithm, which was improved to $1.271n \log n + O(n)$ by Higham *et al.* [HP93], and to $0.693n \log n + O(n)$ by Pachl *et al.* [PKR82].

[HS80] Hirschberg et al. (1980): 'Decentralized extrema-finding in circular configurations of processors'

[DKR82] Dolev et al. (1982): 'An O (n log n) unidirectional distributed algorithm for extrema finding in a circle'

[Pet82] Peterson (1982): 'An O (n log n) unidirectional algorithm for the circular extrema problem'

[HP93] Higham et al. (1993): 'A simple, efficient algorithm for maximum finding on rings'

[PKR82] Pachl et al. (1982): 'A technique for proving lower bounds for distributed maximum-finding algorithms (Preliminary Version)'

Burns [Bur80] and Franklin [Fra82] improved the algorithm of Chang and Roberts. Burns proposed an algorithm that saves messages by alternating the direction in which messages are sent. He also formally defined the model and the problem, giving a $\Omega(n \log n)$ lower bound for bidirectional communication.

[Bur80] Burns (1980): 'A formal model for message-passing systems'

[Fra82] Franklin (1982): 'On an improved algorithm for decentralized extrema finding in circular configurations of processors'

Leeuwen *et al.* proposed in 1987 [LT87] a $1.441n \log n + O(n)$ message complexity algorithm for a deterministic solution on bidirectional, but unoriented rings of size n. Frederickson *et al.* [FL84] makes a distinction between algorithms doing computation or comparison on the values of process identifiers in synchronous rings. He presented an algorithm doing comparison with a lower bound of $O(n \log n)$ messages. Gafni [Gaf85] improved the time complexity of the Frederickson

[LT87] Leeuwen et al. (1987): 'An improved upperbound for distributed election in bidirectional rings of processors'

[FL84] Frederickson et al. (1984): 'The impact of synchronous communication on the problem of electing a leader in a ring'

[Gaf85] Gafni (1985): 'Improvements in the time complexity of two message-optimal election algorithms'

algorithm in synchronous rings, with two algorithms: a first one with $\Theta(n)$ messages and $\Theta(n2^n + |T|^2)$ time, where $|T|$ is the cardinality of the set of the processes identifiers, and a second algorithm achieving $O(n \log n)$ messages and $O(\alpha^{-1}(\log n)|T|)$ time, where $\alpha^{-1}()$ is the functional inverse of log .

General Networks and Spanning Trees

The problem of finding a leader is reducible to the problem of finding a spanning tree [Awe87], as any distributed algorithm that constructs a spanning tree can be transformed into an election algorithm [Afe85], where the root vertex of the spanning tree is the leader of the system. Spanning tree algorithms are usually based on the *Echo* algorithm, which is a wave algorithm for networks of arbitrary topology [Tel00]. In the Echo algorithm, defined by Chang [Cha82] in 1982, an initiator node sends messages to all its neighbors, which forward messages to all their neighbors (except the sender node, i.e. the father), and so on. After receiving a message from all its neighbors, each node sends an echo message back to its father. When, the initiator receives an echo message from all its neighbors, and decides.

In 1977, Spira investigated distributed algorithms to find the Minimum Spanning Tree (MST), based on an asynchronous algorithm of Dalal [Dal77] by which the MST can be found and maintained in a completely distributed manner. Spira proposed then an algorithm using $O(n \log 2n + e)$ messages, where e is the number of edges [Spi77].

Humblet [Hum83] presented in 1983 a distributed algorithm for minimum weight directed spanning trees, taking into account costs (i.e. weights) associated with links in the network, achieving a complexity of $O(n^2)$ on both messages and also time. For instance, the cost could represent latency or some physical distance between two processes.

One notable distributed algorithm for minimum weight spanning trees in static networks which requires $O(5n \log n + 2e)$ messages and $O(n \log n)$ time in a connected undirected graph was proposed by Gallager *et al.* in 1983 [GHS83]. The algorithm of Chin *et al.* [CT85] for an undirected weighted connected graph improved the time complexity of this algorithm, with $O(5n \log n + 2e)$ messages and $O(nG(n))$ time, where $G(n)$ is the number of times that the log function must be applied to n to get a result smaller than or equal to 1. Gafni [Gaf85] also presented in 1985 an algorithm in asynchronous general networks with a $\Theta(e + n \log n)$ messages and $\Theta(n \log n)$ time. Awerbuch [Awe87] proposed in 1987 an optimal distributed algorithm in messages and time to find a MST in asynchronous networks, which requires $O(e + n \log n)$ messages and $O(n)$ time. Peleg suggested an alternative to MST in 1990 [Pel90], with an algorithm for general networks inspired from Afek [Afe85] that achieves a complexity of $O(de)$ messages and $O(d)$ time, with d being the diameter of the network.

[Awe87] Awerbuch (1987): 'Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems'

[Afe85] Afek (1985): 'Distributed algorithms for election in unidirectional and complete networks (traversal, synchronous, leader, asynchronous, complexity).'

[Tel00] Tel (2000): *Introduction to distributed algorithms*

[Cha82] Chang (1982): 'Echo algorithms: Depth parallel operations on general graphs'

[Dal77] Dalal (1977): *Broadcast protocols in packet switched computer networks.*

[Spi77] Spira (1977): 'Communication complexity of distributed minimum spanning tree algorithms'

[Hum83] Humblet (1983): 'A distributed algorithm for minimum weight directed spanning trees'

[GHS83] Gallager et al. (1983): 'A distributed algorithm for minimum-weight spanning trees'

[CT85] Chin et al. (1985): 'An almost linear time and O (nlogn+ e) messages distributed algorithm for minimum-weight spanning trees'

[Gaf85] Gafni (1985): 'Improvements in the time complexity of two message-optimal election algorithms'

[Awe87] Awerbuch (1987): 'Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems'

[Pel90] Peleg (1990): 'Time-optimal leader election in general networks'

[Afe85] Afek (1985): 'Distributed algorithms for election in unidirectional and complete networks (traversal, synchronous, leader, asynchronous, complexity).'

Garcia-Molina [Gar82] proposed the fault-tolerant *bully* algorithm for synchronous general networks which requires $O(n^2)$ messages in the worst case, and where the highest identifier process forces smaller identifier processes into accepting it as the leader. Initially, a process p attempts to contact all processes with higher identifier: if any of these processes respond, then process p waits until the process with a higher identifier becomes the new leader. Otherwise, if all processes with higher identifiers do not respond after a time limit t as they have failed, process p elects itself as the leader.

[Gar82] Garcia-Molina (1982): 'Elections in a distributed computing system'

As proposed by Afek [Afe85] and Awerbuch [Awe87], $\Omega(e)$ is a lower bound to construct a spanning tree in asynchronous general networks (including rings topologies), since an algorithm requires to send at least one message over each edge to traverse the network. Burns [Bur80] proved an $\Omega(n \log n)$ lower bound on the worst-case number of messages sent to find a leader in an asynchronous ring. Therefore, $\Omega(e + n \log n)$ messages are required to construct a spanning tree in asynchronous general network which solves the election problem [Afe85; Awe87], and is an optimum as observed by Gallager *et al.* [GHS83].

[Bur80] Burns (1980): 'A formal model for message-passing systems'

Complete network

Korach *et al.* [KMZ84] shown in 1984 that leader election in asynchronous complete networks has a lower bound of $\Omega(n \log n)$ messages presenting an algorithm of $5n \log k + O(n)$ messages where k is the number of processes starting the algorithm, and $O(n \log n)$ time.

[KMZ84] Korach et al. (1984): 'Tight lower and upper bounds for some distributed algorithms for a complete network of processors'

Afek *et al.* [AG85] proposed in 1985 two leader election algorithms for synchronous and asynchronous complete networks, with $O(\log n)$ and $O(n)$ time complexity for the synchronous and asynchronous algorithms respectively, and with $O(n \log n)$ messages for both algorithms. In synchronous complete networks, the authors also proved a lower bound of $\Omega(n \log n)$ messages and that $\Omega(\log n)$ time is required for any message-optimal synchronous algorithm.

[AG85] Afek et al. (1985): 'Time and message bounds for election in synchronous and asynchronous complete networks'

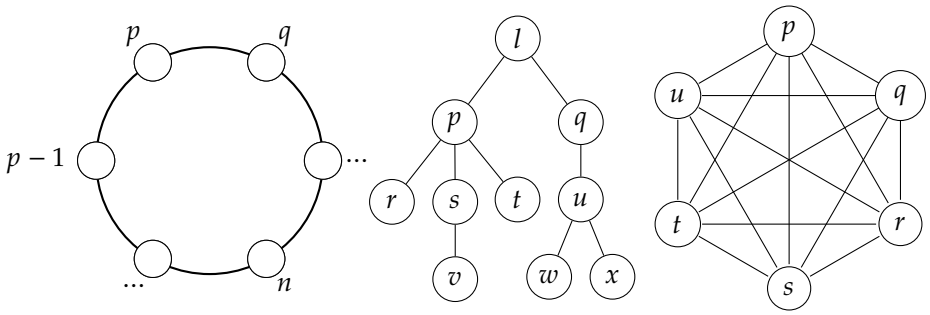

Figure 3.1.: Representation of a ring topology, a spanning tree and a complete graph.

The presented algorithms are summarized in Table 3.1.

Table 3.1.: Comparison of some classical leader election algorithms in static systems.

Article	Topology	Complexity
[Le 77]	Ring	$O(n^2)$ message
[CR79]	Ring	Average $O(n \log n)$ message, worst: $O(n^2)$ message
[HS80]	Ring	Worst: $O(n \log n)$ message
[Bur80]	Ring	Lower bound: $\Omega(n \log n)$ message
[Fra82]	Ring	Worst: $O(n \log n)$ message
[Pet82]	Ring	$1.44n \log n + O(n)$ message
[DKR82]	Ring	$1.356n \log n + O(n)$ message
[HP93]	Ring	$1.271n \log n + O(n)$ message
[PKR82]	Ring	$0.693n \log n + O(n)$ message
[FL84]	Ring	$O(n \log n)$ message
[Gaf85]	Ring	$\Theta(n)$ message, $\Theta(n2^n + \lvert T \rvert^2)$ time
[Gaf85]	Ring	$O(n \log n)$ message, $O(\alpha^{-1}(\log n)\lvert T \rvert)$ time
[Spi77]	MST	Average: $O(n \log^2 n + e)$ message
[Hum83]	MST	Worst: $O(n^2)$, $O(n^2)$ time
[GHS83]	MST	$O(n \log n + e)$ message, $O(n \log n)$ time
[CT85]	MST	$O(5n \log n + 2e)$ message, worst: $O(nG(n))$ time
[Gaf85]	MST	$\Theta(e + n \log n)$ message, $\Theta(n \log n)$ time
[Awe87]	MST	$O(e + n \log n)$ message, $O(n)$ time
[Pel90]	General Networks	$O(de)$ message, $O(d)$ time
[Gar82]	General Networks	Worst: $O(n^2)$ message
[KMZ84]	Complete async.	$5n \log k + O(n)$ message, $O(n \log n)$ time
[AG85]	Complete sync.	$O(n \log n)$ message, $O(\log n)$ time
	Complete async.	$O(n \log n)$ message, $O(n)$ time

3.1.2. Dynamic Systems

As presented in Section 2.6.2, the communication graph of dynamic systems evolves over time since nodes can fail, recover, join or leave the system at an arbitrary time during execution.

In this type of system, network partitions can happen resulting in disjoint connected component, and algorithms have to handle communication changes.

Election Criterion

Like in static systems, many of the leader election algorithms are *extrema finding*, meaning that they use the highest node identifier as an election criterion [Hat+99; MWV00; RAC08]. Some other ones use different criteria to elect a leader, like election time [Ing+09; Ing+13], or some feature such as remaining battery or computation power [VKT04; KW13].

Hatzis *et al.* [Hat+99] presented two leader election algorithms, where both elect the node with the highest identifier. They solve a stronger problem by specifying that once elected, the leader should also be aware of the size of the system. In the algorithm of Malpani *et al.* [MWV00], the leader is the node with the lowest identifier of the component.

[Hat+99] Hatzis et al. (1999): 'Fundamental control algorithms in mobile networks'

[MWV00] Malpani et al. (2000): 'Leader election algorithms for mobile ad hoc networks'

Rahman *et al.* [RAC08] proposed an algorithm for static and dynamic systems aiming at reducing the number of leader election rounds and therefore saving energy: each node maintains a list of leaders, and the leader is the node with the highest identifier. Each time a new node joins the system, after recovering from a crash for example, it starts a new leader election round.

[RAC08] Rahman et al. (2008): 'Performance analysis of leader election algorithms in mobile ad hoc networks'

Algorithms proposed by Ingram *et al.* [Ing+09; Ing+13] use clocks to record the time the election took place, and the leader is the process that wins the most recent election.

[Ing+09] Ingram et al. (2009): 'An asynchronous leader election algorithm for dynamic networks'
[Ing+13] Ingram et al. (2013): 'A leader election algorithm for dynamic networks with causal clocks'

Vasudevan *et al.* [VKT04] proposed an algorithm where the election criterion is some value related to the node, i.e. a performance-related characteristic, such as remaining battery life or computation capabilities. The authors suggest the idea of electing as the leader the node with the minimum average distance to other nodes, but no implementation is given.

[VKT04] Vasudevan et al. (2004): 'Design and analysis of a leader election algorithm for mobile ad hoc networks'

Similarly, Masum *et al.* [MA+06] proposed an algorithm for electing a local extrema among the nodes participating in the election, based on an arbitrary value called *priority*. Each node has a *priority* indicating its attractiveness to be the leader, which can be a performance-related attribute of the node, such as the battery life or computational capabilities.

[MA+06] Masum et al. (2006): 'Asynchronous leader election in mobile ad hoc networks'

By exploiting the structure of a spanning tree, Kim *et al.* algorithm [KW13] elects a centrally positioned leader, according to the average depth of nodes in the tree. They called their centrality measure *tree-based centrality*, and compare it with different centrality measures such as degree, closeness, and betweenness. Compared to the closeness centrality, the tree-based central leader is not always optimal because it depends on the node that initiates the election algorithm.

[KW13] Kim et al. (2013): 'Leader election on tree-based centrality in ad hoc networks'

Information Spreading

Dynamic classical leader election algorithms have different structures of communication, which are mainly a leader-oriented **Directed Acyclic Graph (DAG)** where each node has a direct path to the leader, or a **spanning tree** directed towards the initiator node of the election. Some other algorithms use a different communication structure, based on counters, for example.

Directed Acyclic Graph. The algorithm proposed by Malpani *et al.* [MWV00] is based on a routing algorithm for mobile wireless networks called TORA [PC97], which creates a leader-oriented DAG. A mechanism is used to detect network partitions, and nodes that no longer have a path to a destination stop sending unnecessary messages. Each node creates a 6-uplet from the 5-uplet used in TORA, adding the identifier of the current assumed leader of the node partition. This

[MWV00] Malpani et al. (2000): 'Leader election algorithms for mobile ad hoc networks'
[PC97] Park et al. (1997): 'A highly adaptive distributed routing algorithm for mobile wireless networks'

6-uplet is modified during topological changes and is used for the election.

Ingram *et al.* [Ing+09] improved Malpani *et al.* algorithm [MWV00] and requires a three waves algorithm [Tel00] on all nodes of the system to elect a new leader (two waves to search a potential leader and one confirmation wave). Like Malpani *et al.* [MWV00], a leader-oriented DAG structure is used in each connected component, where every node has a directed path to the leader. Leader stability is also studied to reduce new leader elections while a path to the old leader still exists. Their algorithm requires that nodes have perfectly synchronized clocks, which is made possible by using a global time accessible by all nodes in the system.

[Ing+09] Ingram et al. (2009): 'An asynchronous leader election algorithm for dynamic networks'

[Tel00] Tel (2000): *Introduction to distributed algorithms*

A more complete version of the previous work was published in 2013 by Ingram *et al.* [Ing+13], with causal clocks instead of a global clock accessible to all nodes, which can be implemented using perfect clocks or logical clocks. Performance of the algorithm depends on the type of clocks used to implement a causal time and the authors specify that their algorithm is not correct for approximately synchronized clocks unless they preserve causality.

[Ing+13] Ingram et al. (2013): 'A leader election algorithm for dynamic networks with causal clocks'

Spanning Tree. Vasudevan *et al.* [VKT04] use a wave mechanism to build a spanning tree, where each node sends back to its parent the identifier of the node having the highest value in its subtree. Their algorithm returns a leader identity *only* when it is accepted by all nodes in the network, returning $\perp$ in the rest of the time. Note that links are assumed to be bidirectional and heartbeat messages are periodically sent from the leader node. Based on Vasudevan *et al.* [VKT04], the algorithm of Rahman *et al.* [RAC08], also builds a spanning tree, where nodes periodically send *probe* messages to their neighbors, and get *reply* messages from them. As in the Echo algorithm previously presented, an acknowledgement system is used where nodes reply to *election* messages with *ack* messages along with their identifier. The leader sends heartbeat messages every twenty seconds, and after a timeout of six messages, a new leader election is triggered. Like in Vasudevan *et al.*, the algorithm also assumes that communication links are bidirectional.

[VKT04] Vasudevan et al. (2004): 'Design and analysis of a leader election algorithm for mobile ad hoc networks'

[RAC08] Rahman et al. (2008): 'Performance analysis of leader election algorithms in mobile ad hoc networks'

Kim *et al.* [KW13] exploit a three wave algorithm to build a spanning tree, used to elect a central leader. Each node periodically broadcasts *hello* messages, in order to check connectivity with its neighbors, and to create a neighbor information table. *Election* messages are used to dynamically build the spanning tree and are propagated when an election is triggered.

[KW13] Kim et al. (2013): 'Leader election on tree-based centrality in ad hoc networks'

Counter-based. Hatzis *et al.* [Hat+99] presented an algorithm with a different communication structure, where each node maintains a local counter representing the number of other mobile nodes met. When two mobile nodes meet, they exchange their identities: the one

[Hat+99] Hatzis et al. (1999): 'Fundamental control algorithms in mobile networks'

with the higher identifier wins and receives the counter of the loser, which is added to its local counter. The loser node changes its state to inactive, and no longer responds to messages about the execution of the algorithm. Information is only transmitted when a new node joins a component, thereby reducing the number of broadcasts needed by the algorithm and saving, therefore, battery power consumed for message transmissions. Each node also keeps a local list of node identifiers that it has defeated. When two nodes meet, the winner concatenates its local list with the list transmitted by the loser node. That way, the final winner will know the network size and the identifiers of nodes in the network.

Other approaches. Masum *et al.* [MA+06] consider that communication links are reliable, bidirectional, and FIFO. Their algorithm does not rely on a specific communication structure, and tolerates intermittent failures, such as link failures, sudden crash, as well as recovery of mobile nodes. A node is considered faulty if the communication links between the node and each of its neighbors have failed, while a node recovery is the recovery of the communication links between the recovering node and its neighbors. Message delivery is only guaranteed if the sender and receiver remain both connected (not partitioned) for the entire duration of the message transfer.

[MA+06] Masum et al. (2006): 'Asynchronous leader election in mobile ad hoc networks'

Connectivity Assumption

To handle frequent topology changes, election algorithms must tolerate arbitrary, concurrent changes and should eventually terminate electing a unique leader within the connected component. Due to the dynamics of the network, it is impossible to guarantee a unique leader all the time, because when a network partition occurs or when two components merge, it will take some time to elect a new leader. In order to satisfy the agreement property of the leader election and eventually elect a single leader, algorithms designed for dynamic systems make different assumptions about connectivity of nodes of the system.

The two leader election algorithms of Hatzis *et al.* [Hat+99] are designed to handle dynamic topology changes with mobility of some or all nodes. Both leader election algorithms require that nodes know in advance the type and dimensions of the area in which they move, and nodes need to measure the distance that they cover when moving. The first algorithm presented might never elect a single leader, if nodes never meet to exchange information, while the second one assumes nodes with no sense of orientation that follow random walks and is based on a Las Vegas algorithm. Nodes have a unique identifier, but a variation of the second algorithm allows anonymous nodes.

[Hat+99] Hatzis et al. (1999): 'Fundamental control algorithms in mobile networks'

Malpani *et al.* [MWV00] supposed a synchronous system, where executions take place in stages during finite phases. When a partition occurs, the system is separated in two or more components and the authors

[MWV00] Malpani et al. (2000): 'Leader election algorithms for mobile ad hoc networks'

consider that any component whose topology is static long enough will eventually have exactly one leader.

Similarly, Ingram *et al.* [Ing+09; Ing+13] consider an asynchronous system where messages are delivered in FIFO order through reliable asynchronous communication channels, and where nodes are assumed to be completely reliable. If topology changes cease, then eventually each connected component of the network has a unique leader.

[Ing+09] Ingram et al. (2009): 'An asynchronous leader election algorithm for dynamic networks'
[Ing+13] Ingram et al. (2013): 'A leader election algorithm for dynamic networks with causal clocks'

Vasudevan *et al.* [VKT04] modified the requirements that eventually every connected component has a unique leader, by proposing an algorithm that ensures that after a finite number of topology changes, eventually each node has the same leader.

[VKT04] Vasudevan et al. (2004): 'Design and analysis of a leader election algorithm for mobile ad hoc networks'

Masum *et al.* [MA+06] assume one or multiple simultaneous topological changes can occur, but each node remains in the network for a sufficiently long time. Any connected component of the network whose topology remains static sufficiently long will eventually have exactly one unique leader.

[MA+06] Masum et al. (2006): 'Asynchronous leader election in mobile ad hoc networks'

The algorithm of Rahman *et al.* [RAC08] does not make the assumption that topology changes eventually stop. Nodes permanently send periodic *probe* messages and wait for the *reply* message from the neighbors of the spanning tree, to maintain the connectivity.

[RAC08] Rahman et al. (2008): 'Performance analysis of leader election algorithms in mobile ad hoc networks'

Note that most of the presented works do not specify which mobility model or pattern they use.

The presented algorithms are summarized in Table 3.2.

Table 3.2.: Comparison of classical leader election algorithms in dynamic systems.

Article	Election Criterion	Information Spreading	Connectivity Assumption
[Hat+99]	Highest identifier	Counter of met nodes	Nodes need to meet with random walks
[MWV00]	Lowest identifier	Leader-oriented DAG	Component is static long enough
[Ing+09]	Most recent election	Leader-oriented DAG	Topology changes cease
[Ing+13]	Most recent election	Leader-oriented DAG	Topology changes cease
[VKT04]	Highest arbitrary value	Spanning tree with waves	Finite number of topology changes
[MA+06]	Highest arbitrary value	(not specified)	Static long enough
[RAC08]	Highest identifier	Spanning tree with waves	(not specified)
[KW13]	Centrality positioned	Spanning tree with waves	(not specified)

3.2. Eventual Leader Election Algorithms

As presented in Section 2.9.2, Chandra *et al.* [CHT96] introduced in 1996 the Ω failure detector, which provides a primitive *Leader()* satisfying the following property: there is a time after which all correct processes trust the same correct process p, i.e. the process returned by the *Leader()* primitive.

[CHT96] Chandra et al. (1996): 'The weakest failure detector for solving consensus'

As Ω cannot be implemented in an asynchronous distributed system where process crashes arise (otherwise it would solve the consensus in

such system, which is known to be impossible as seen in Section 2.9.2), any asynchronous system has to be enriched with some assumptions in order to implement Ω [Lar+12].

Note that an algorithm is *communication efficient* if after some time, the eventual leader is the only process to send messages to all other processes, forever [Agu+01].

[Lar+12] Larrea et al. (2012): 'Specifying and implementing an eventual leader service for dynamic systems'

[Agu+01] Aguilera et al. (2001): 'Stable Leader Election'

3.2.1. Static Systems

Eventual leader election algorithms designed for static networks can be divided in two categories, *known* and *unknown* networks, based on assumptions about network knowledge, as presented in Section 2.6. The literature contains two main approaches: *timer-based* and *time-free* as defined in Section 2.9.2.

Known Networks

In *known* networks, nodes have some knowledge about the system.

Timer-based. Chandra *et al.* [CT96], as well as Larrea *et al.* [LFA00], proposed the first Ω solutions by considering a fully-connected network and reliable eventually timely links.

[CT96] Chandra et al. (1996): 'Unreliable failure detectors for reliable distributed systems'

[LFA00] Larrea et al. (2000): 'Optimal Implementation of the Weakest Failure Detector for Solving Consensus'

Aguilera *et al.* [Agu+01] presented in 2001 a weaker condition on channel reliability and synchrony, by relaxing the need of time constraints on all links. In their algorithm, the authors assume that there is at least one process whose links are eventually timely with all the other processes of the system. They later proposed, in 2003, an algorithm [Agu+03] implementing Ω in weak systems, where all processes may be arbitrarily slow or may crash, except for some timely processes s whose identities are not known. Only the output links of s are eventually timely, and all other links can be arbitrarily slow and lossy. Then in 2004, the authors presented an article where there exist some correct processes, whose identities are not known, with f outgoing links that are eventually timely [Agu+04]. They show that when $f = 1$, it is sufficient to have only one eventually timely link to implement Ω and solve consensus. The authors also give a simple communication efficient algorithm that implements Ω for systems with up to f crashes, and with at least one correct $\diamond f - source$, i.e. a correct process with eventually f outgoing links that are timely.

[Agu+01] Aguilera et al. (2001): 'Stable Leader Election'

[Agu+03] Aguilera et al. (2003): 'On implementing omega with weak reliability and synchrony assumptions'

[Agu+04] Aguilera et al. (2004): 'Communication-efficient leader election and consensus with limited link synchrony'

Malkhi *et al.* [MOZ05] proposed an alternative solution without having any eventual timely links, by considering eventually *accessible* links. Their algorithm considers that eventually one process can send messages such that every message obtains f timely responses, denoted $\diamond f - accessible$. This property implies that the f responders do not need to be fixed, and may change from one message to another, i.e. to vary in time. Compared to Aguilera *et al.* [Agu+04], where in $\diamond f - source$, the

[MOZ05] Malkhi et al. (2005): 'Ω meets paxos: Leader election and stability without eventual timely links'

set of f links is fixed throughout the execution, Malkhi *et al.* algorithm presents a weaker assumption with $\diamond f - accessible$ that allows the f links to vary in time.

By unifying the assumptions made in [Agu+03], [Agu+04] and [MOZ05], an even weaker model was proposed by Hutle [Hut+08] in 2008. The authors show that Ω can also be implemented in a system with at least one process with f outgoing moving eventually timely links (moving means that the set of neighbors may change over time), assuming either unicast or broadcast steps. They assume a model of a fully-connected network where processes are partially synchronous. Their model requires at least one correct process that is a $\diamond moving{-}f{-}source$, i.e. it must have timely outgoing links with an unknown delay bound to a moving set of f receivers at any time.

[Hut+08] Hutle et al. (2008): 'Chasing the weakest system model for implementing Ω and consensus'

As seen in Section 2.3, in some systems, a crashed process can recover with the same identifier and goes on executing. The algorithm of Fernández-Campusano *et al.* [Fer+17] implements Ω in a crash-recovery and omissive system. They consider a totally ordered set of n processes which are known by each node, and nodes have access to a stable storage. Every pair of nodes is connected by two unidirectional communication links, one in each direction. Each node elects as leader the node with the smallest penalty value among the nodes that communicate with a majority of processes. A node increases its penalty value when it recovers after a crash, or if it is not well connected with a majority of processes. Each node keeps track of its communication with every other node by periodically exchanging messages, which includes its penalty value, its current leader, and information about its connectivity with the receiver node.

[Fer+17] Fernández-Campusano et al. (2017): 'A distributed leader election algorithm in crash-recovery and omissive systems'

Time-free. Mostéfaoui *et al.* [MMR03] use a query-response mechanism, which assumes that the responses from some processes to a query arrive among the $(n - f)$ first ones. The authors show that the algorithm implements the $\diamond\mathcal{S}$ failure detector, which is known to be equivalent to Ω when the membership is known. In [MRT06], Mostéfaoui *et al.* extend the previous algorithm to implement Ω, considering a star communication structure if the following property is satisfied: there is a correct process p and a set Q of f processes q ($p \notin Q$, and processes in Q can crash), such that eventually, either 1) each time q broadcasts a query, q receives a response from p among the $(n - f)$ first responses to this query (and such a first response is called a winning response), or 2) the channel from p to q is timely.

[MMR03] Mostefaoui et al. (2003): 'Asynchronous implementation of failure detectors'

[MRT06] Mostéfaoui et al. (2006): 'Time-Free and Timer-Based Assumptions Can Be Combined to Obtain Eventual Leadership'

Unknown Networks

Other works aim to implement Ω in *unknown* networks and look for models with the weakest possible assumptions on the knowledge and communication graph. They share a common assumption on reachability communication between every pair of correct processes.

The presented works consider a timer-based approach, as defined in Section 2.9.2.

The work of Jimenez *et al.* [JAF06] shows that it is possible to implement Ω without knowledge of the system membership. To this end, they present an algorithm implementing a Ω failure detector, which requires minimal reliability and synchrony assumptions in systems whose links are only of two types: either eventually timely (where messages are received by time $t + \Delta$ after an unknown GST, with Δ being an unknown bound) or lossy asynchronous (where messages can be lost or arbitrarily delayed), as seen in Section 2.4.

[JAF06] Jiménez et al. (2006): 'Implementing unreliable failure detectors with unknown membership'

Two algorithms were proposed by Fernandez *et al.* [FJR06; AJR10] to implement Ω with weak assumptions on the initial knowledge of each process and the behavior of the underlying network. The first one considers a partial unknown network, where each process knows the lower bound α on the number of correct processes ($\alpha = n - f$). This algorithm assumes fair-lossy links and a strongly connected graph, where there is a correct process connected to $f - c$ other correct processes through eventually timely paths (with c being the actual number of crashes in the considered run, and eventually timely paths are paths made up of correct processes and eventually timely links). Note that this first algorithm is not communication-efficient, as each correct process has to send messages forever by fair-lossy links (one in each direction). The second algorithm considers an unknown network with a complete communication graph, where each pair of correct processes is connected by fair-lossy links (one in each direction). It also assumes that there is a correct process with output links to every correct process that are eventually timely. The authors also present an important impossibility result which consists of a lower bound theorem: in a system where processes know neither α (a lower bound on the number of correct processes) nor c (a lower bound on the actual number of crashes) in their initial knowledge, there is no eventual leader election algorithm with less than $n - c - 1$ eventually timely links [AJR10].

[FJR06] Fernández et al. (2006): 'Eventual Leader Election with Weak Assumptions on Initial Knowledge, Communication Reliability, and Synchrony'
[AJR10] Anta et al. (2010): 'Eventual Leader Election with Weak Assumptions on Initial Knowledge, Communication Reliability, and Synchrony'

Martín *et al.* [MLJ09] proposed some algorithms in 2009 to implement Ω in systems not necessarily fully connected, where two algorithms of them assume an unknown membership. The system does not require that every pair of processes is connected by a direct communication link, and some links can be lossy asynchronous. The first algorithm assumes that eventually all processes are reachable timely from the process that crashes and recovers a minimum number of times. The second algorithm assumes that all processes are eventually reachable timely from some correct process. The eventual leader is the process with the lowest identifier in the set of processes that no longer crashes after some time.

[MLJ09] Martín et al. (2009): 'Implementing the omega failure detector in the crash-recovery failure model'

The presented algorithms are summarized in Table 3.3.

Table 3.3.: Comparison of eventual leader election algorithms in static systems.

Article	Membership	Approach
[CT96]	Known	Timer-based
[LFA00]	Known	Timer-based
[Agu+01]	Known	Timer-based
[Agu+03]	Known	Timer-based
[Agu+04]	Known	Timer-based
[MOZ05]	Known	Timer-based
[Hut+08]	Known	Timer-based
[Fer+17]	Known	Timer-based
[MMR03]	Known	Time-free
[MRT06]	Known	Time-free
[JAF06]	Unknown	Timer-based
[FJR06; AJR10]	Unknown	Timer-based
[MLJ09]	Unknown	Timer-based

3.2.2. Dynamic Systems

The eventual leader election problem in dynamic systems has been studied in the literature, particularly for MANET networks. Since this thesis proposes eventual leader election algorithms for mobile networks, this section aims to discuss some related work organized following the approach used by the related algorithms: *timer-based* or *time-free*.

Timer-based

The algorithm presented by Larrea *et al.* [Lar+12] elects the node that has been in the system for the longest period of time, i.e. the node that joined the system first and has not yet left nor crashed. Each node has a local clock and the time when a node joins the system is timestamped using its current local clock value. If two nodes have the same timestamp, the lower identifier breaks the tie. Clocks are not required to be synchronized, but the local clock value of a node joining the system should be as large as the clock value of the nodes already in the system. A new election is started when a timer related to the current leader expires. The authors assume that communication links are eventually timely among nodes in the system [Agu+08], and their algorithm is communication-efficient. The proposed algorithm is of class $\Delta\Omega$ and satisfies the **EL-NI** and **EL-ND** properties introduced in Section 2.9.2, i.e. no more nodes join or leave the system during long enough periods of time.

[Lar+12] Larrea et al. (2012): 'Specifying and implementing an eventual leader service for dynamic systems'

[Agu+08] Aguilera et al. (2008): 'On implementing omega in systems with weak reliability and synchrony assumptions'

Similarly to the previous algorithm, the algorithm of Gómez-Calzado *et al.* [Góm+13] uses the timer-based approach and elects as the leader the oldest node of the connected component with the highest identifier. Each node maintains a set with the identity of the nodes that belong to its component. When a node does not belong to any component, it periodically broadcasts *join* messages. On the other hand, if it is the leader of a component, it sends a leader message with the size of its

[Góm+13] Gómez-Calzado et al. (2013): 'Fault-tolerant leader election in mobile dynamic distributed systems'

component. When a node receives a *join* message, it adds the new node to its component, and propagates this new information over the network. When it receives a *leader* message, it changes its leader if the received component size is higher than its current leader component size (when components merge, for example), and propagates the result over the network. Communication channels are bidirectional, and since a minimum stability of the graph is required, they use a notion of link duration, i.e. a time interval that ensures communication among nodes. Periodically, all links act as eventually timely channels. Nodes are mobile and the system alternates periods of good and bad behaviors. During a good period, i.e. in a stable and long enough connected interval, the communication paths guarantee that graph connectivity corresponds to a spanning tree. The authors also consider the **EL-NI** and **EL-ND** properties which guarantee stability of the system after some time, extending them to tolerate nodes mobility, i.e. graph partitioning and merging. To this end, they formally define Mobile Dynamic Omega, denoted $\Delta^*\Omega$. In short, $\Delta^*\Omega$ requires that eventually and for a sufficiently long time, there is a bidirectional path between the leader and the rest of nodes in case of fragmentation of the network or merging of multiple partitions (which then act as independent networks). They also introduced a formalism with a framework to classify mobile and dynamic distributed systems [Cal15].

[Cal15] Calzado (2015): 'Contributions on agreement in dynamic distributed systems'

Tucci-Piergiovanni *et al.* [TB10] consider an unknown system where nodes are *up* or *down*, with a bounded number of concurrently *up* nodes. Nodes have a local oracle called HB^* which provides a list of *up* nodes, used to implement Ω. HB^* sorts nodes according to a sequence number of heartbeats, which are periodically sent by *up* nodes. Nodes that are eventually and permanently *up*, i.e. the oldest nodes, are in the first positions of the sorted list. HB^* permanently exchange the set of the first nodes of their respective lists, updating them. The leader is the node with the lowest identifier among the b first nodes in the list. The authors do not consider any specific communication structure, assuming unknown but finite bounds on message losses and message delay. Network partitions are temporary. Each node periodically sends heartbeats with its identifier, so that the other *up* nodes consider it as a participant of the system. An infinite number of nodes may join and leave over time. Each node has a local clock, which is not synchronized among each other nodes.

[TB10] Tucci-Piergiovanni et al. (2010): 'Eventual leader election in infinite arrival message-passing system model with bounded concurrency'

Melit *et al.* [MB12] proposed an algorithm that also elects the node which has the highest *priority* value among all nodes within its connected component. The lowest node identifier is used to break ties in case of equal priority value. A timer is used to detect loss of communication with the leader, i.e. when no more *leader* messages are received. At the expiration of the timer of a node, the node elects itself as the new leader and starts sending *leader* message periodically. Links are either fair-lossy or eventual timely. The algorithm uses the flooding approach for the *leader* messages, which is initiated by the leader and contains the identifier and the priority value of the leader. Nodes forward received

[MB12] Melit et al. (2012): 'An Ω-Based Leader Election Algorithm for Mobile Ad Hoc Networks'

leader messages to all direct neighbors, except the sender node. A finite number of topological changes can take place, i.e. after a time t, the topology does not change anymore and becomes static.

Time-free

Mostefaoui *et al.* [Mos+05] extends the algorithm for static systems [MRT04], for dynamic systems. The system can have infinitely many nodes, but at each run has only a finite number of nodes. Nodes are asynchronous, i.e. there is no bound on the execution of a computation step, and they enter the system by executing a *join()* operation that provides them an identity. The system is characterized by a succession of unstable and stable periods, where progress can only be guaranteed during the stable periods if they last long enough. The set of nodes which are correct after entering the system, i.e. neither crash nor leave, is called $stable$, and the elected leader is a $stable$ node. The following progress condition must be satisfied: $|stable| > \alpha$, where $\alpha = n - f$. The leader is the node with the lowest identifier among a set of trusted nodes. This set of trusted nodes eventually contains all nodes of the component that replied among the α first response to queries. Each node maintains a logical date and a set of trusted node identifiers, such that eventually, the set of all nodes have the same value, hence allowing each node to elect the same leader from its set. Nodes use gossiping to disseminate their set of trusted nodes as well as a logical date defining the age of set. This logical date indicates if the received set of trusted nodes is more recent than the current one. A query-response mechanism is used to get knowledge about nodes in the system. Upon receiving α responses, the query terminates.

[Mos+05] Mostefaoui et al. (2005): 'From static distributed systems to dynamic systems'

[MRT04] Mostefaoui et al. (2004): 'Crash-resilient time-free eventual leadership'

Arantes *et al.* [Ara+13] propose a time-free algorithm for unknown network where nodes can move, fail by crashing, join and leave the system. The authors consider the finite arrival model, i.e. the network is a dynamic system composed of infinitely many mobile nodes but each run consists of a finite set of n nodes. Communication channels are fair-lossy and the dynamics of the network is modeled by using the recurrent connectivity class of Time-Varying Graph (TVG) that ensures that at any point in time the communication graph remains connected over time [Cas+11]. For diffusing information, the algorithm uses local query-response mechanism [MMR03]: at each query-response round, node p_i systematically broadcasts a query message to the nodes in its neighborhood until it possibly crashes or leaves the system. When p_i receives α_i messages, the query-response terminates. α_i is defined locally as a function of the expected number of correct known neighbors with whom p_i may communicate at the time t in which the query is issued. The algorithm elects the leader based on a punishment procedure and on the periodic exchange of query-response messages. If a neighbor p_j of p_i does not respond within the α_i responses, p_i punishes it by incrementing a counter associated to it. The algorithm thus will eventually elect a correct process that has the smallest punish

[Ara+13] Arantes et al. (2013): 'Eventual Leader Election in Evolving Mobile Networks'

[Cas+11] Casteigts et al. (2011): 'Time-varying graphs and dynamic networks'

[MMR03] Mostefaoui et al. (2003): 'Asynchronous implementation of failure detectors'

counter. To ensure that all the nodes will elect the same leader, query messages of p_i carry information about the view it has of the system and the values of its process punished counters. In order to tolerate mobility of nodes and avoid false suspicions in case of mobility, messages are timestamped: as soon as p_i gets the information (by the contents of a received message) that another node has received a message from p_j with a greater timestamp, p_i stops punishing p_j, which was falsely suspected as faulty by p_i. To ensure eventual stability of the algorithm, the system assumes both the *Stabilized Responsiveness Property* ($\mathcal{SRP}$), which defines the ability of a correct process to eventually always reply to a query sent among the first processes, and the *Stable Termination Property* ($\mathcal{SatP}$), which guarantees that information from/to a process is going to be sent/received to/from at least one correct other process in its neighborhood.

The presented algorithms are summarized in Table 3.4.

Table 3.4.: Comparison of eventual leader election algorithms in dynamic systems.

Article	Approach	Election Criterion	Information Spreading	Connectivity Assumption
[Lar+12]	Timer-based	Oldest node with lower identifier	Periodic leader flooding (spanning tree)	Periodic stable system
[Góm+13]	Timer-based	Oldest node with higher identifier	Periodic leader flooding (spanning tree)	Periodic stable system
[TB10]	Timer-based	Oldest node with lower identifier	Fully connected graph	Only temporary partition
[MB12]	Timer-based	Highest arbitrary value	Periodic leader flooding (spanning tree)	Eventually static
[Mos+05]	Time-free	Lowest identifier	(not specified)	Periodic stable system
[Ara+13]	Time-free	Lowest punishment counter	Connected graph	Recurrent connectivity (TVG)

3.3. Conclusion

This chapter has presented several works on the classic and eventual leader election problems for static and dynamic systems.

First, a few pioneer articles are presented on the classical leader election problem in static systems for specific topologies such as rings, spanning trees, or complete communication graphs. Then some papers on the same problem but for dynamic systems are described, following three criteria: the election choice, information spreading and connectivity assumptions.

Then, the problem of the eventual leader election, also called the Ω failure detector is addressed. For static systems, the corresponding section organizes the related work according to the knowledge or not

of the system membership, while for dynamic systems, some existing Ω algorithms are presented by implementation choice (timer-based or time-free).

Topology Aware Leader Election Algorithm for Dynamic Networks

4.

This chapter presents the first contribution of this thesis: an eventual leader election for Mobile Ad Hoc Networks (MANET), which exploits the knowledge that nodes maintain of the network in the decision of the leader choice. The mobile nodes 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 move, join and leave the system, fail, and recover at runtime. Due to these dynamics, 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.

As seen in Chapter 3, many works have proposed leader election algorithms in both static and dynamic 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 MANET [Mos+05; MA+06; MB12]. 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. However, it is important that the 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 [VKT04], *the most valued node* denotes the one that best meets the performance-related criterion in question and, therefore, should be chosen as the leader. Hence:

[Mos+05] Mostefaoui et al. (2005): 'From static distributed systems to dynamic systems'
[MA+06] Masum et al. (2006): 'Asynchronous leader election in mobile ad hoc networks'
[MB12] Melit et al. (2012): 'An Ω-Based Leader Election Algorithm for Mobile Ad Hoc Networks'

[VKT04] Vasudevan et al. (2004): 'Design and analysis of a leader election algorithm for mobile ad hoc networks'

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 [Fer+17].
2. For the sake of performance, an elected leader should be *the most valued node* among all the nodes within its connected component.

[Fer+17] Fernández-Campusano et al. (2017): 'A distributed leader election algorithm in crash-recovery and omissive systems'

The proposed algorithm is an eventual leader election algorithm called *Topology Aware*, which is 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*. The 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* makes the computation of the closeness centrality possible, as well as the election of per component central located leaders, which, thus, efficiently spread information across nodes of the component. Even if the problem of discovering network topology has been studied in various contexts [NT09; BO99], this approach has never been used to elect an eventual leader.

[NT09] Nesterenko et al. (2009): 'Discovering network topology in the presence of byzantine faults'
[BO99] Bellur et al. (1999): 'A reliable, efficient topology broadcast protocol for dynamic networks'

It is worth pointing out that even if the presented work target MANET, the *Topology Aware* algorithm has been designed for generic mobile dynamic systems.

The rest of the chapter is organized as follows: Section 4.1 explains the chosen model and assumptions, while Section 4.2 describes the algorithm. Section 4.4 discusses performance results and, finally, a conclusion is given in Section 4.5.

4.1. System Model and Assumptions

The system model considers an upper bound ϕ on the time a process takes to execute a step and eventually timely links, i.e. an upper bound δ

on the transmission delay. However, both bounds ϕ and δ are unknown and hold after an unknown *Global Stabilization Time (GST)*, so the system is partially synchronous, as described in Section 2.2 and Section 2.4.

There exists one process per node. Therefore, the words *node* and *process* are interchangeable.

4.1.1. 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 to 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*, as detailed in Section 2.3. 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.

4.1.2. Communication graph

The system is modeled as an undirected graph, as presented in Section 2.6. Two nodes can communicate directly if they are in the transmission range of each other, i.e. a receiver node is located inside the emission range of a sender node. The system assumes that the emission range is the same as the reception range. Therefore, if node i can communicate with node j, node j can also communicate with node i (bidirectional links).

A given node belongs to a connected graph formed by its neighbors, neighbors of its neighbors, and so on, which is called a *connected component*. Due to the movement, failure, and disconnection of nodes, the system can be divided into two (or more) different connected components. Each of them 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 changes cease, each connected component will eventually elect a single leader.

4.1.3. Channels

Nodes only communicate by broadcast on a fixed Wi-Fi channel selected beforehand. All neighbors of the sender node receive the broadcast message. Reliable communication channels are assumed, possible in a wireless network where the MAC layer reliably delivers broadcast messages, even in the 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.

4.1.4. 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.

4.2. Topology Aware Leader Election Algorithm

In the *Topology Aware* 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 a *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.

4.2.1. Pseudo-code

The pseudo-code of the *Topology Aware* algorithm for a node i is given in Algorithm 1 and is described in the following.

4.2.2. Data structures, variables, and messages (lines 1 to 6)

The two following data structures are used by node i:

- **view** (line 1) is composed of two elements: a logical *clock* [Lam78] 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):

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
```

- **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.

```
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
```

4.2.3. Initialization (lines 7 to 11)

At the beginning, node i initializes its knowledge with its own identifier (i) and its logical clock set to 0.

4.2.4. Periodic updates task (lines 12 to 16)

Periodically, i.e. every Δ milliseconds, node i broadcasts its new *updates* list if not empty, and then set it to empty.

4.2.5. Connection (lines 20 to 23)

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.

4.2.6. Disconnection (lines 24 to 27)

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 then created with the following information (line 25):

- the identifier of node i, considered as the 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 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 (line 12). Node j is then removed from the knowledge of node i (line 26) and the clock of node i is incremented (line 27).

4.2.7. Knowledge reception (lines 28 to 38)

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).

4.2.8. Updates reception (lines 39 to 53)

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* paragraph), 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.

Note that an update cannot be applied when it is more recent than other updates not yet received by i, which should have been 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 any 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).

4.2.9. Pending updates (lines 54 to 65)

PendingUpdates (line 54) checks the updates that can be applied from 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.

4.2.10. Leader election (lines 17 to 19)

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). The closeness centrality defined in Section 2.7 is used rather than the betweenness centrality, because it is computed faster and requires fewer computational steps, so use less energy from the mobile nodes. 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, the algorithm ensures that all nodes of a connected component will eventually have the same topology knowledge and, therefore, will have the same leader node [Ing+13].

[Ing+13] Ingram et al. (2013): 'A leader election algorithm for dynamic networks with causal clocks'

4.2.11. Execution examples

This section presents examples of the execution of *Topology Aware* algorithm, showing the connection between node i and node j.

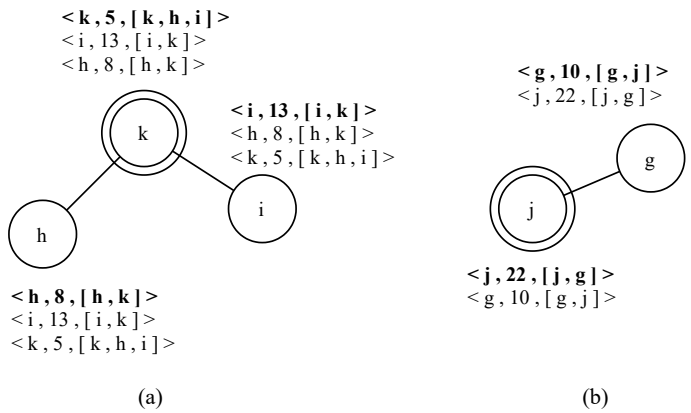

Figure 4.1.: Example Topology Aware: Initial state.

Initially, in Figure 4.1, the system is composed of two connected components: nodes h, i, and k in Figure 4.1.a, and nodes j and g in Figure 4.1.b. Each node has its own knowledge composed of $< identifier, clock, set(neighbors) >$, with arbitrary initial values used for the sake of the example. One leader is currently elected per component, respectively nodes k and j. Nodes j and g have the same closeness centrality, therefore, the highest node identifier is used to break the tie, as presented in Section 4.2.10, and since $g < j$, node j is the leader.

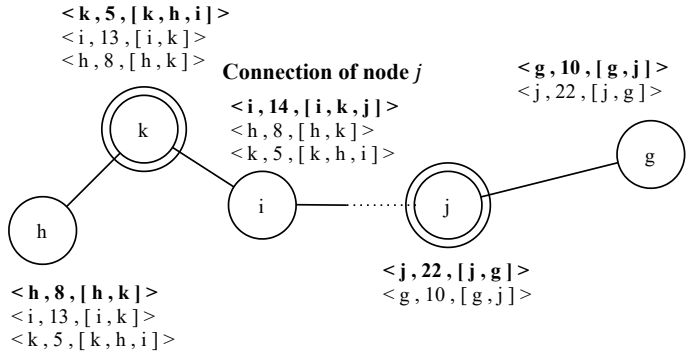

Figure 4.2.: Example Topology Aware: Connection of node j.

In Figure 4.2, the underlying probe system of node i first detects node j and triggers the *Connection* method described in Section 4.2.5. Therefore, node i adds node j in its neighbors list, and increments by one its clock value, previously sets to 13.

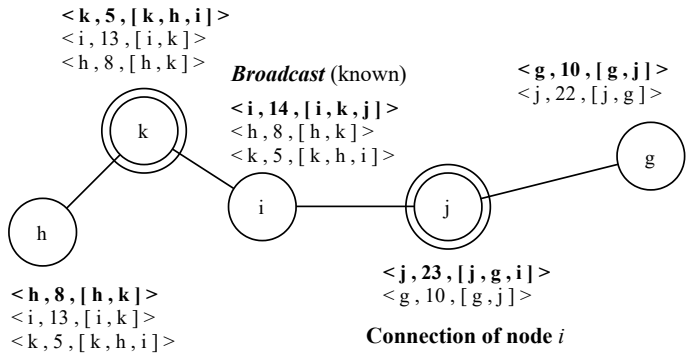

Figure 4.3.: Example Topology Aware: Broadcast knowledge of node i and connection of node i.

Then, in Figure 4.3, node i broadcasts its knowledge in order to share its view of the network with node j. Meanwhile, the probe system of node j has detected node i and triggers the *Connection* method of node j. Node i is added to the neighbor list of node j and the latter increments its clock value from 22 to 23.

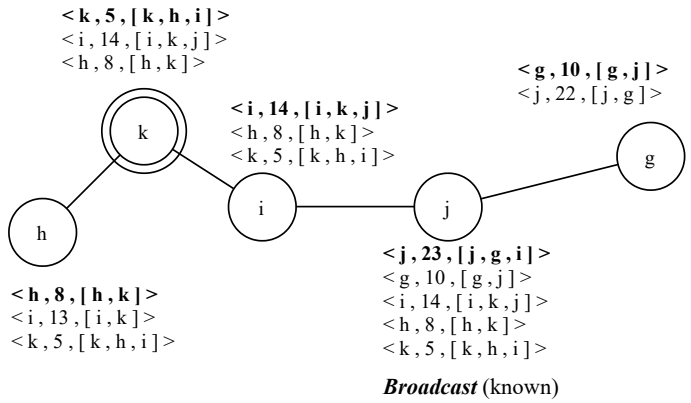

Figure 4.4.: Example Topology Aware: Broadcast knowledge of node j and knowledge reception of node j.

In Figure 4.4, node j broadcasts its knowledge to complete the *Connection* method, and it receives a knowledge from node i. According to this new knowledge, node j is not the leader anymore. Since node j received a knowledge from node i, it creates an update. However, in this case, the update will not be usefull since its neighbors already received its knowledge. Note that node k also received the knowledge of node i, therefore, it follows the steps described in Section 4.2.7: the received clock of node i is higher than the known clock of i in the knowledge of node k ($14 > 13$, line 33), meaning that new information is received. Therefore, node k updates the entry of node i in its knowledge by adding node j as a new neighbor and updating the clock of node i from 13 to 14. It also creates an update $< i, add(j), rmv(-), 13, 14 >$ that will be sent by the *Periodic Updates Task*.

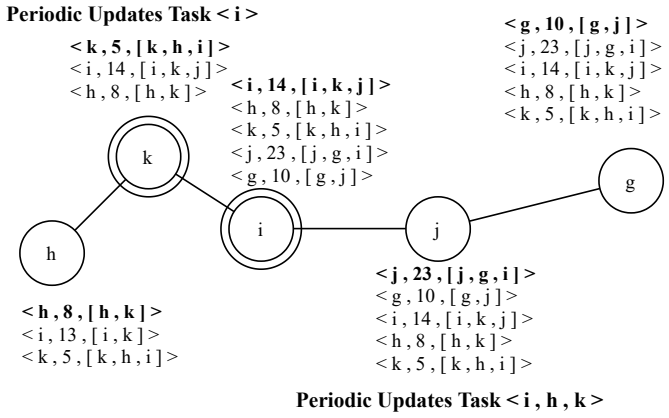

Figure 4.5.: Example Topology Aware: Knowledge reception of nodes i and g.

Then, in Figure 4.5, nodes i and g receive a knowledge from node j. Both nodes update their knowledge and create updates with the new received information, that will be broadcast later. According to this new knowledge, node i considers itself as the leader since it has the highest closeness centrality.

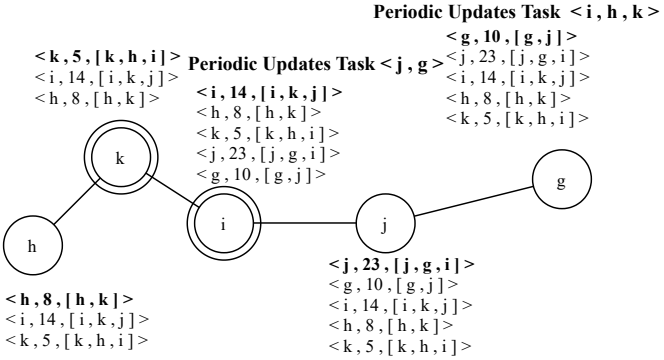

Figure 4.6.: Example Topology Aware: Periodic Updates Task of nodes i and g.

In Figure 4.6, the *Periodic Updates Task* of both nodes i and g send their update. Node h received the update of node k and updates the entry of node i in its knowledge with the more recent information received. Therefore, it creates an update with the new information received, that will be sent later.

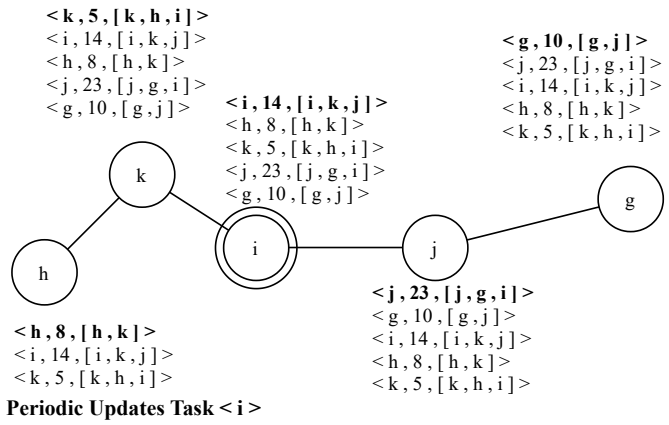

Figure 4.7.: Example Topology Aware: Update reception of node k.

Then, in Figure 4.7, node k received the update sent by node i, with information about nodes j and g. Thus, it updates its knowledge and creates an update containing this new received information. The *Periodic Updates Task* of node h sends the update with information about node i.

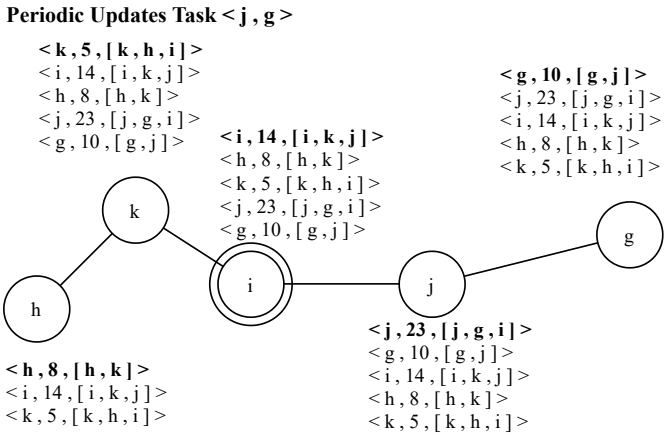

Figure 4.8.: Example Topology Aware: Periodic Updates Task of node k.

In Figure 4.8, the *Periodic Updates Task* of node k sends the update with information about nodes j and g.

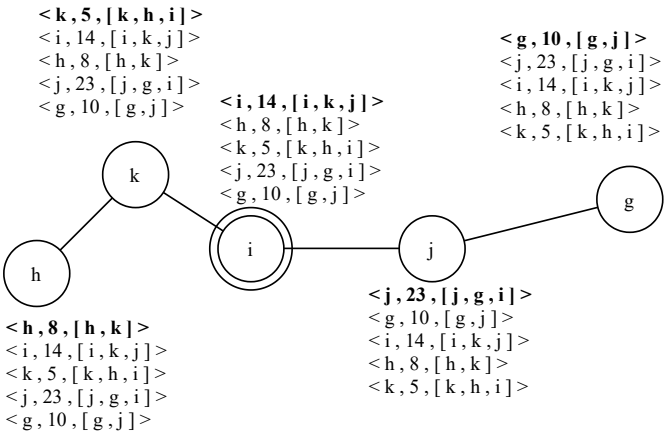

Figure 4.9.: Example Topology Aware: Update reception of node h.

Finally, in Figure 4.9, node h receives the update from node k with information about nodes j and g. It updates its knowledge and creates an update with the new received information, that will be sent later by the *Periodic Updates Task* (not shown in the example). All nodes have the same knowledge of the component. Hence, the invocation of *Leader()* method of Ω failure detectors returns the most central node according to the closeness centrality: node i, which is the leader.

4.3. Simulation Environment

The objective of the simulations is to compare the *Topology Aware* algorithm with a flooding one.

Several evaluation experiments were conducted on PeerSim [MJ09], a Java peer-to-peer network simulator with an event-based engine and modularity. Each experiment lasts 30 minutes, with a simulated unit of time corresponding to one millisecond, and simulates 60 nodes placed in a 900×900 meters obstacle-free area, with a fixed diameter of transmission range between 10 and 200 meters decided beforehand and identical for all nodes. Message sending latency follows a Poisson distribution with parameter $\lambda = 10$. The experiments were conducted using console-mode on remote servers, but a graphical interface was used primarily for development purposes, as shown in Figure 4.10.

[MJ09] Montresor et al. (2009): 'PeerSim: A Scalable P2P Simulator'

4.3.1. Algorithms

Two versions of the *Topolgy Aware* algorithm were considered:

- *Topology Aware Closeness*, which uses the closeness centrality as the election criterion, and elects as the leader the node having the highest closeness in the connected component, as presented in Section 4.2.

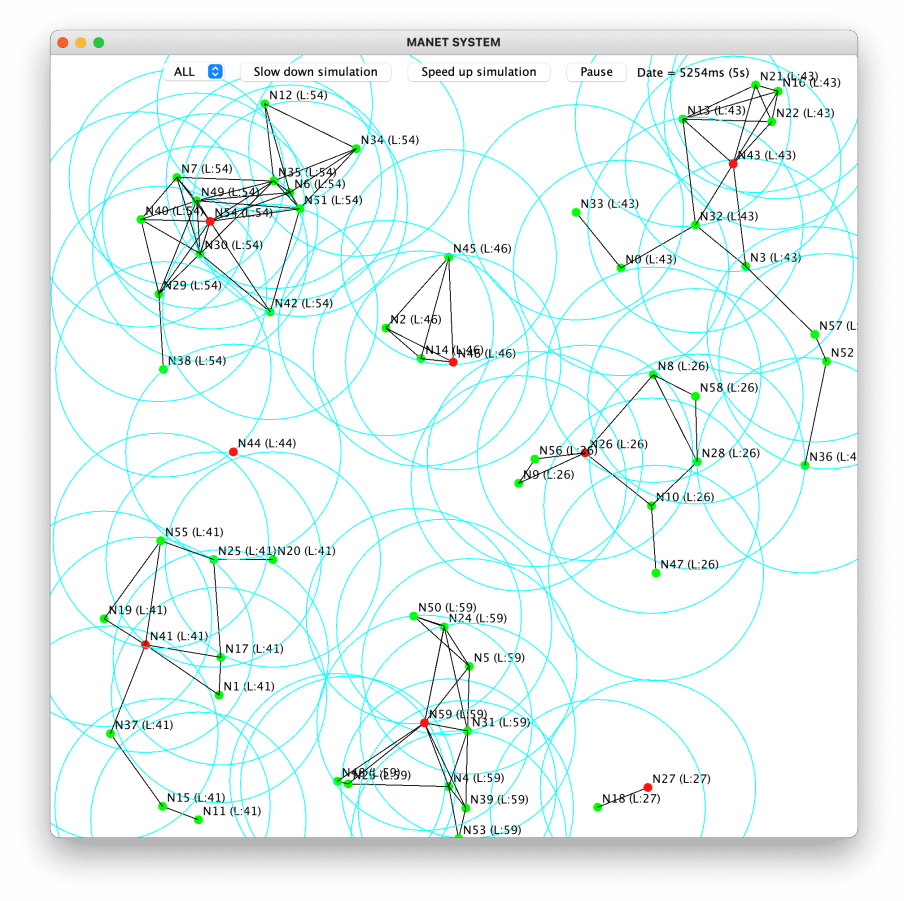

Figure 4.10.: Screenshot of a graphical PeerSim experiment running the *Topology Aware* algorithm with a transmission range of 120 meters. Leaders are in red.

- *Topology Aware Degree*, which uses the node degree, i.e. the number of direct neighbors of a node, as the election criterion, and elects as the leader the node having the highest degree in the connected component.

These two versions of the *Topology Aware* algorithm were compared with a variant of the Vasudevan *et al.* algorithm [VKT04], which is based on flooding. 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 the *Topology Aware* algorithm, a variant of this algorithm that never returns $\perp$ but a possible current leader, is considered. Indeed, in *Topology Aware*, a correct node is always able to render a leader according to its topological knowledge. This variant is denoted *Flooding* because each node periodically broadcasts *leader messages* informing its current leader to neighbor nodes.

[VKT04] Vasudevan et al. (2004): 'Design and analysis of a leader election algorithm for mobile ad hoc networks'

Note that *Flooding* is also a variant of the *OptFloodMax* algorithm that Nancy A. Lynch introduced in her book *Distributed Systems* [Lyn96]. 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).

[Lyn96] Lynch (1996): 'Distributed algorithms'

The *Flooding* algorithm was adapted 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, it is considered 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*. This algorithm is denoted *Flooding Degree*.

4.3.2. Algorithms Settings

In the *Flooding Degree* algorithm, nodes send *leader messages* every $\alpha = 250$ milliseconds. Every node triggers a new election if $\beta = 1$ *leader 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 if the list is not empty. 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 should make information transmission faster on small components, but slower on lager components, due to the high dynamics of mobile nodes induced by larger transmission ranges. To this aim, the following formula was deduced empirically:

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

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, as shown in Figure 4.11. Therefore, information is transmitted quickly on small components, but more slowly on larger components. As a result, updates are sent every 60 milliseconds on average for lower transmission ranges, and every 90 milliseconds on average for larger transmission ranges.

In the three algorithms, 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 $\tau = 400$

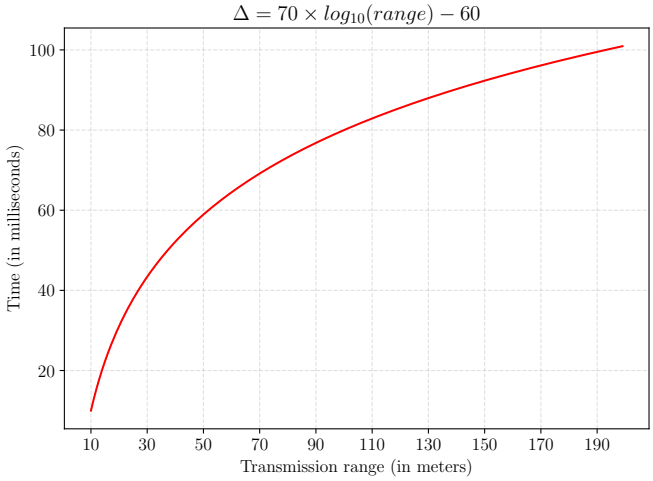

Figure 4.11.: Representation of Δ in Periodic Updates Task.

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.

4.3.3. Mobility Models

Two different mobility patterns have been considered in 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.

Random waypoint

Nodes are randomly placed in the area and move according to the *Random Waypoint* mobility model [CBD02]. Nodes wait 10 seconds before choosing the next random destination. It is worth noting that this mobility pattern is largely used in the literature [VKT04; CBD02]. A representation of the *Random Waypoint* is given in Figure 4.12.

[CBD02] Camp et al. (2002): 'A survey of mobility models for ad hoc network research'
[VKT04] Vasudevan et al. (2004): 'Design and analysis of a leader election algorithm for mobile ad hoc networks'
[CBD02] Camp et al. (2002): 'A survey of mobility models for ad hoc network research'

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 4.13, 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

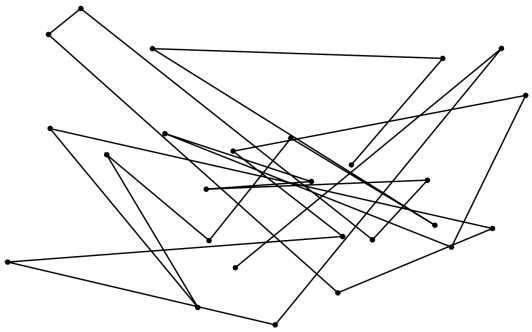

Figure 4.12.: Representation of the Random Waypoint mobility model [CBD02].

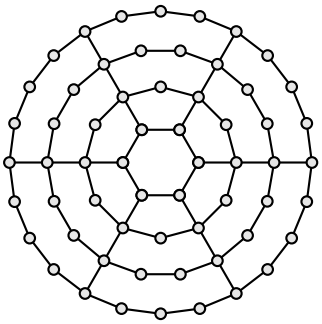

Figure 4.13.: Representation of the single point of interest disc.

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 4.12 regardless of the diameter of the transmission range. This pattern could model user activities, where nodes represent people visiting regularly just a few places [Pap+16].

[Pap+16] Papandrea et al. (2016): 'On the properties of human mobility'

4.4. Evaluation

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

4.4.1. Metrics

For evaluation, the following three metrics have been considered: instability, number of messages sent per second, and longest leader path relative to the component diameter.

Instability

This is the percentage of the average time that a node elects a different leader from the eventually unique elected leader of its 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*.

First, the $CurrentInstability$ at time t is computed with the following formula:

$$CurrentInstability_t = \frac{\sum_{i=0}^{N} \begin{cases} 0 \text{ if } leader_t(i) = oracle_t(i) \\ 1 \text{ if } leader_t(i) \neq oracle_t(i) \end{cases}}{N}$$

where N is the number of nodes in the system, and i the node identifier. Then, the *Instability* is computed over the entire experiment time, which is the average $CurrentInstability_t$ from 0, to the end of the experiment (1 800 seconds).

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 by all three algorithms every τ milliseconds.

Longest leader path relative to the component diameter

This metric characterizes how fast a leader can reach nodes of its component. First is computed 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, the longest path is divided by the diameter of the component, i.e. the greatest distance between any pair of nodes in the component.

4.4.2. Instability

The random waypoint pattern in Figure 4.14 shows that the instability percentage of *Flooding Degree* and of 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*. Note that the *Closeness* version of *Topology Aware* is slightly less stable than the *Degree* version.

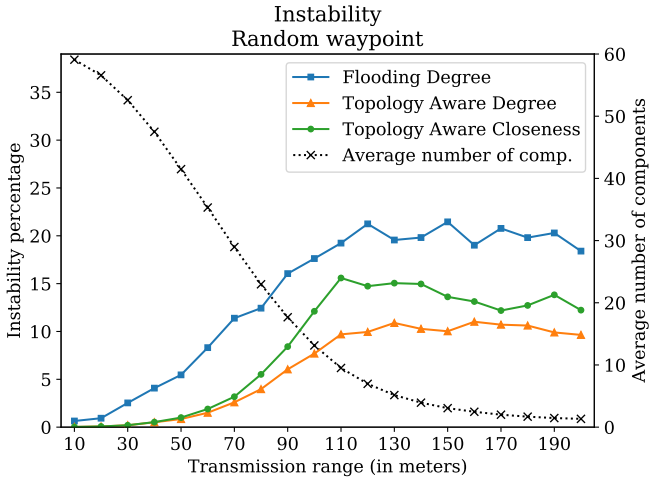

Figure 4.14.: Instability percentage with random waypoint (lower is better).

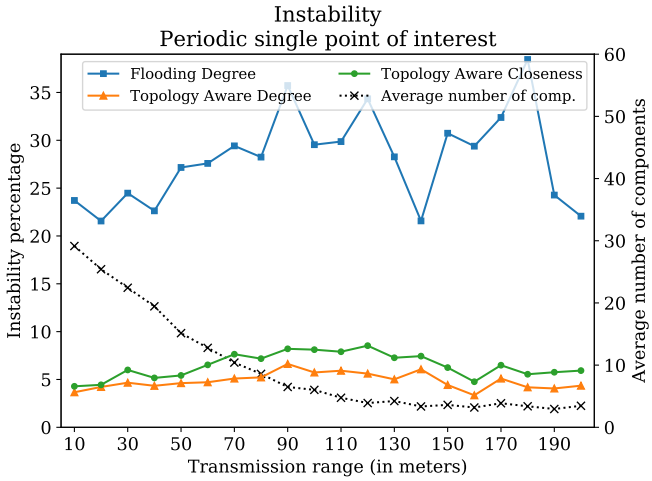

Figure 4.15.: Instability percentage with periodic single point of interest (lower is better).

For the periodic single point of interest pattern in Figure 4.15, 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 observed in Figure 4.15).

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 they need fewer steps to elect a new leader once their topological knowledge is built.

Instability over time at 90 meters

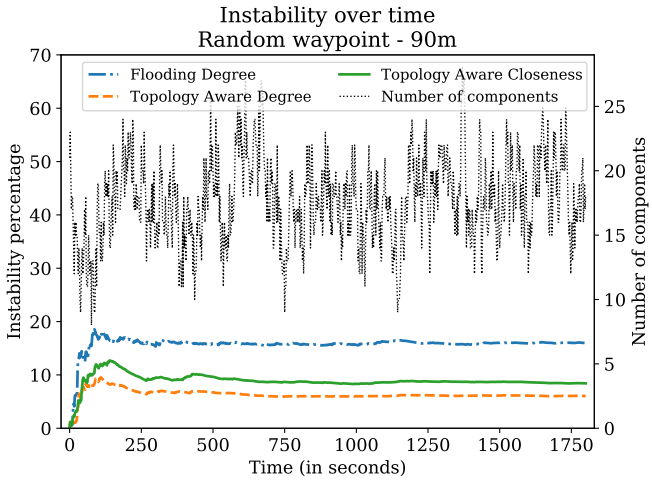

Figure 4.16.: Evolution of instability at a transmission range of 90 meters with random waypoint (lower is better).

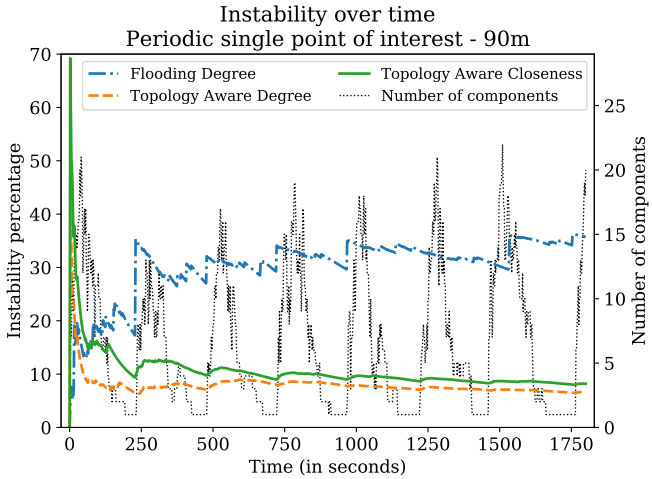

Figure 4.17.: Evolution of instability at a transmission range of 90 meters with periodic single point of interest (lower is better).

Figure 4.16 and Figure 4.17 both 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 in Figure 4.16, the random waypoint mobility pattern has on average 18 connected components, and *Topology Aware Degree* is on average 62% more stable than *Flooding Degree*.

For the periodic single point of interest pattern in Figure 4.17, nodes are gathering at their initial position on the disc presented in Figure 4.13, creating a unique connected component, then moving to random positions, also having on average a 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.

4.4.3. Number of messages sent per second

The previous section shows 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 4.18, 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.

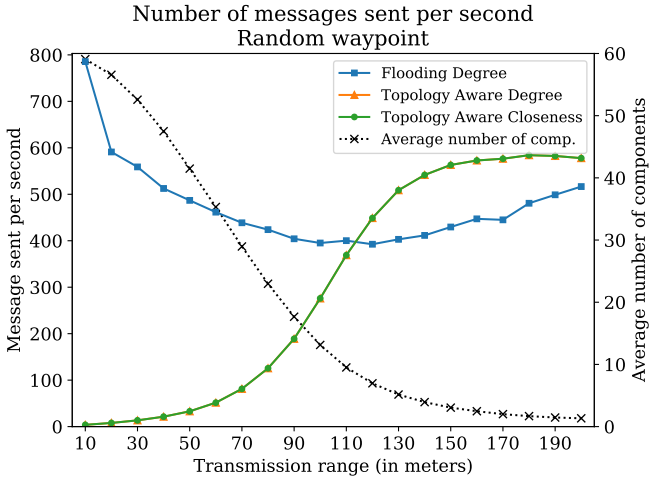

Figure 4.18.: Number of messages sent per second with random waypoint (lower is better).

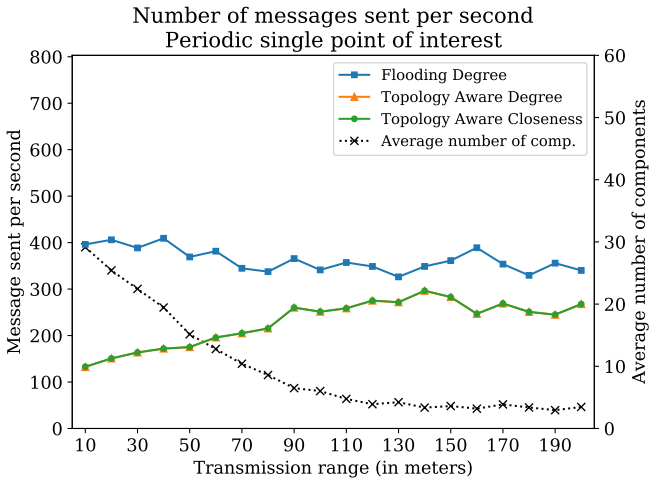

Figure 4.19.: Number of messages sent per second with periodic single point of interest (lower is better).

An interesting threshold effect is observed 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.

Table 4.1.: 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

For the periodic single point of interest pattern in Figure 4.19, 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 for a while, a flooding algorithm continues to periodically send information about its current leader.

However, the size of messages exchanged in both *Topology Aware* algorithms is larger than in the *Flooding Degree* algorithm. In Table 4.1, 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 values of message sizes in Table Table 4.1 contain additional information needed by the simulator, which is identical to the three algorithms. Furthermore, since the size of messages remains below the MTU value of wireless networks, a message fits within a single packet.

4.4.4. Path to the leader

Figure 4.20 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.

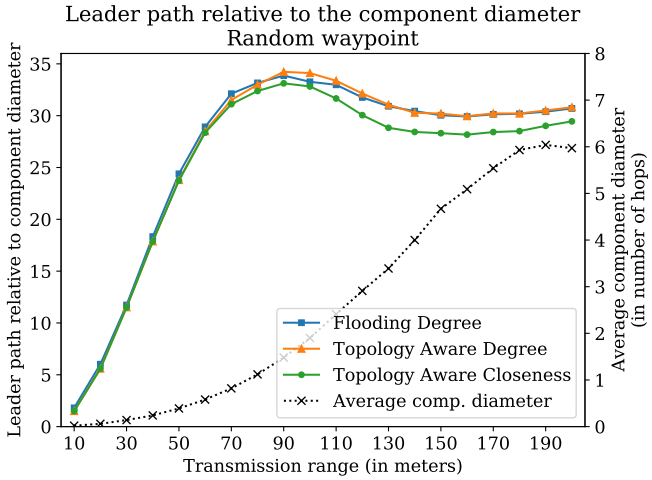

Figure 4.20.: Longest leader path relative to component diameter (lower is better).

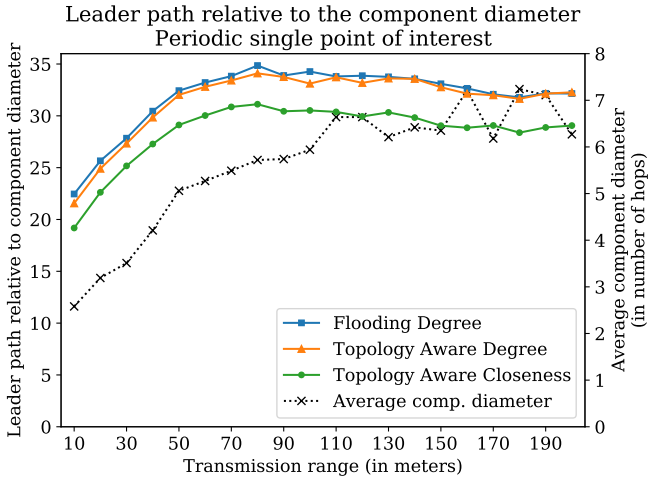

Figure 4.21.: Longest leader path relative to component diameter (lower is better).

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.

For the periodic single point of interest pattern in Figure 4.21 however, since the shape of the disc is independent of the transmission range, there is periodically only one component comprising the entire network. It can be observed 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.

Table 4.2.: Average election time with fault injection (in milliseconds).

	Flooding Degree	Topology Aware Degree	Topology Aware Closeness
Average election time (in ms)	422 (σ = 38)	820 (σ = 188)	731 (σ = 154)

4.4.5. Fault injection

Some experiments were conducted aiming at injecting fault 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 4.2 show the average election time to elect the new leader. It can be observed that for both *Topology Aware* versions the election time is larger than *Flooding Degree*, because the buffering of update messages slows down message transmission time of each node as the transmission range increases. However, this time remains low (smaller than 1 second).

4.5. Conclusion

This chapter has presented a per component eventual leader election algorithm for dynamic networks that shows the advantages of all nodes using network topology knowledge 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, both versions of the *Topology Aware* algorithm (based on closeness and degree centralities respectively) and a flooding algorithm with a local topological election criterion, were evaluated on PeerSim [MJ09] simulator.

A performance comparison of the *Topology Aware* algorithm with a variant of the leader election algorithm of Vasudevan *et al.* [VKT04] is presented, because their work is a good example of a typical flooding algorithm and is strongly referenced in the literature [RAC08; Ing+13; KW13]. The results confirm the effectiveness of the *Topology Aware* algorithm and that it outperforms the latter. 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. The latter is less sensitive to the component size and sends fewer messages than *Flooding Degree*. When compared to *Flooding Degree*, both *Topology Aware* versions improve the leader stability

[MJ09] Montresor et al. (2009): 'PeerSim: A Scalable P2P Simulator'

[VKT04] Vasudevan et al. (2004): 'Design and analysis of a leader election algorithm for mobile ad hoc networks'

[RAC08] Rahman et al. (2008): 'Performance analysis of leader election algorithms in mobile ad hoc networks'

[Ing+13] Ingram et al. (2013): 'A leader election algorithm for dynamic networks with causal clocks'

[KW13] Kim et al. (2013): 'Leader election on tree-based centrality in ad hoc networks'

up to 82% depending on mobility models, sends half as many messages, and nodes reach the leader by 11% shorter paths. It is worth pointing out that the size of messages in *Topology Aware* could be reduced using compression, for example.

One limitation of the work presented in this chapter is the assumption of reliable channels. Therefore, the second algorithm presented in the next chapter considers eventually reliable communication channels, with interference, collision, and messages loss.

Centrality-Based Eventual Leader Election in Dynamic Networks

5.

This chapter proposes a new eventual leader election algorithm for dynamic systems, which implements a cross-layer neighbor detection, and a neighbor-aware mechanism that improves the sharing of topological knowledge, electing a central leader faster. Furthermore, in order to improve the performance of information propagation, the algorithm uses a topological knowledge based on a self-pruning mechanism combined with probabilistic gossip. Evaluations were conducted on the OMNeT++ [Var10] environment, simulating realistic MANET including interference, collision, and messages loss.

[Var10] Varga (2010): 'OMNeT++'

Nodes are mobile and communicate by sending messages over wireless links. The system membership is not known in advance. The communication graph can evolve over time, therefore, the network is not always fully connected but composed of one or more connected components. The algorithm chooses the leader according to a topological criterion: for every component, the leader is eventually the node having the best closeness centrality in the component. Each node progressively builds and maintains a local knowledge of the component communication

graph. This knowledge is then used to locally determine a central leader, well located to be reached by a majority of processes.

The current chapter brings three main contributions:

1. A new *Centrality-based Eventual Leader* election algorithm for dynamic systems, called *CEL*, where the leader eventually has the best centrality. *CEL* has a cross-layer neighbors detection which exploits the broadcast features of the underlying wireless network. The neighbor-aware mechanism improves the sharing of the topological knowledge and elects a central leader faster.
2. *CEL* uses the topological knowledge through a self-pruning mechanism, combined with probabilistic gossip, to reduce global information propagation costs.
3. An extensive evaluation on the OMNeT++ environment [Var10] using two mobility patterns to simulate Mobile Ad Hoc Network (MANET) with interference, collision, and messages loss. Comparison with the closest algorithm [Góm+13] shows that *CEL* has a good trade-off considering the number of messages exchanged, stability of the leader (i.e. the percentage of the average time that nodes adopt the expected leader), and the closeness of the leader to the other nodes.

[Var10] Varga (2010): 'OMNeT++'

[Góm+13] Gómez-Calzado et al. (2013): 'Fault-tolerant leader election in mobile dynamic distributed systems'

Compared to the algorithm in the previous chapter, the system model of *CEL* assumes unreliable communication channels, which are suitable for realistic environments with interference and message collisions. Message loss is taken into account by the CSMA/CA protocol included in IEEE 802.11 [Com99]. Rather than assuming an underlying probe system, *CEL* uses a cross-layer mechanism to leverage already existing data link layer messages, and to access the MAC addresses of the nodes, which are used to construct the topological knowledge of the connected component. In addition, the topological knowledge is used to improve communication performance, and the sharing of this knowledge between nodes is optimized by taking advantage of the bidirectional links assumption (neighbor-aware mechanism), whereas in the previous chapter, the algorithm does not provide such mechanisms. Evaluation compare *CEL* with the Ω eventual leader election algorithm of Gómez-Calzado *et al.* [Góm+13], which also assumes eventually reliable communication channels and is more recent than Vasudevan *et al.* algorithm [VKT04]. Experiments use two more realistic mobility models: the *Random Walk* and the *Truncated Lévy Walk*.

[Com99] Comnittee (1999): *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*

[Góm+13] Gómez-Calzado et al. (2013): 'Fault-tolerant leader election in mobile dynamic distributed systems'

[VKT04] Vasudevan et al. (2004): 'Design and analysis of a leader election algorithm for mobile ad hoc networks'

The rest of the chapter is organized as follows: Section 5.1 presents the system model and assumptions, Section 5.2 describes the algorithm, Section 5.4 discusses performance results, and finally, a conclusion is given in Section 5.5.

5.1. System Model and Assumptions

The system considered is the same as in Section 4.1.

5.1.1. Node states and failures

Nodes always follow the specification of the algorithm until they fail. They can fail by crashing and a node can recover, joining the system again with the same *unique identifier* as before the failure. Hence, a node keeps its identifier regardless of its state, and two nodes cannot have the same *identifier*. However, a node does not recover its state neither its knowledge of the network membership, thus, is initialized again.

Initially, all nodes in the system are in the *correct* state. A node is considered *faulty* if it fails and does not recover, or if it leaves the system forever. Otherwise, if present in the system, it is considered *correct*.

5.1.2. Communication graph

The assumptions for the communication graph are the same as in Section 4.1.2.

5.1.3. Channels

Nodes can only communicate by broadcasting local messages, which are received by all neighbors of the sending node. Communication is based on a fixed Wi-Fi channel, chosen beforehand. *Eventually reliable* communication channels are considered, with messages losses induced by messages interference and collisions. The CSMA/CA protocol included in IEEE 802.11 [09], is used to handle messages losses. There are no assumptions about message ordering, i.e. messages can be delivered out of order.

[09] (2009): *IEEE Std 802.11n-2009 – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput*

5.1.4. Membership and nodes identity

Initially, each node only knows its unique *identifier* in the system. This means that nodes do not know the total number of nodes, neither the membership of the system. Nodes detect their neighbors through a *cross-layer* mechanism described in Section 5.2.4, using already existing *beacon* messages of the data link layer. A node gets knowledge of the network membership by receiving *knowledge* messages from its neighbors.

5.2. Centrality-Based Eventual Leader Election Algorithm

This section presents the *Centrality-based Eventual Leader (CEL)* election algorithm. In *CEL*, every node maintains a topological knowledge of the connected component to which it belongs. The algorithm builds this knowledge during node connections and disconnections (triggered by the *cross-layer* mechanism), and by sending *knowledge* messages to its neighbors. Nodes spread *knowledge* messages using *probabilistic gossip*, combined with a *self-pruning* mechanism that exploits the topological knowledge to reduce the number of messages sent. A new *knowledge* message is only sent after a connection or disconnection. Based on the component knowledge, the algorithm eventually elects one leader per component, which is placed at the center of the component.

5.2.1. Pseudo-code

The pseudo-code of *CEL* for node i is given in Algorithm 2.

5.2.2. Data structures, messages, and variables (lines 1 to 4)

CEL uses a data structure called a **view** (line 1). A *view* associated to node i is composed of two elements:

1. A logical *clock* value, acting as a timestamp and incremented at each connection and disconnection.
2. A set of node *identifiers*, which are the current neighbors of i.

Each node i maintains a local variable (line 3) called **known**. This variable represents the current topological knowledge that i has of its current component (including itself). It is implemented as a map of *view* indexed by node *identifier* (line 4).

The only type of message exchanged between neighbors is the **knowledge** message (line 2). It contains the current topological knowledge that the sender node has of the network, i.e. its *known* variable.

5.2.3. Initialization (lines 5 to 7)

Firstly, node i initializes its *known* variable with its own identifier (i), and sets its logical clock to 0.

Algorithm 2: *Centrality-based Eventual Leader (CEL)* election algorithm for node i

```
1  Typedef view: ⟨clock: int, neigh: set(id)⟩
2  Message knowledge: ⟨map(key: id, value: view)⟩
3  Local variables:
4      known: map(key: id, value: view)
5  Initialization:
6      known[i].neigh ← {i}
7      known[i].clock ← 0
8  Connection of node j:
9      known[i].neigh ← known[i].neigh ∪ {j}
10     known[i].clock ← known[i].clock + 1
11     if j ∉ known then
12         known[j].neigh ← {j, i}
13         known[j].clock ← 1
14     else
15         known[j].neigh ← known[j].neigh ∪ {i}
16         known[j].clock ← known[j].clock + 1
17     LocalBroadcast (knowledge⟨known⟩, 1)
18 Disconnection of node j:
19     known[i].neigh ← known[i].neigh \ {j}
20     known[i].clock ← known[i].clock + 1
21     known[j].neigh ← known[j].neigh \ {i}
22     known[j].clock ← known[j].clock + 1
23     LocalBroadcast (knowledge⟨known⟩, 1)
24 Receive knowledge message known_j from node j:
25     ∀ n ∈ known_j do
26         if n ∉ known or
27         known_j[n].clock > known[n].clock then
28             known[n] ← known_j[n]
29             UpdateNeighbors (known_j, n)
30         else if known_j[n].clock = known[n].clock then
31             known[n].neigh ← known[n].neigh ∪ known_j[n].neigh
32             UpdateNeighbors (known_j, n)
33     if known was updated then
34         TopologicalBroadcast ()
```

5.2.4. Node connection (lines 8 to 17)

When a new node j appears in the transmission range of i, the cross-layer mechanism of i detects j, and triggers the *Connection* method (line 8). Node j is added to the neighbors set of node i (line 9). As the knowledge of i has been updated, its logical clock is incremented (line 10).

Since links are assumed bidirectional, i.e. the emission range equals the reception range, if node i has no previous knowledge of j (line 11), the

```
35 Call of UpdateNeighbors(known_j, n):
36     ∀ k ∈ known_j[n].neigh do
37         if k ∉ known or
38         known_j[k].clock > known[k].clock then
39             known[k] ← known_j[k]
40         else if known_j[k].clock = known[k].clock then
41             known[k].neigh ← known[k].neigh ∪ known_j[k].neigh

42 Call of TopologicalBroadcast():
43     ∀ n ∈ known[i].neigh do
44         if known[n].neigh = known[i].neigh then
45             if n < i then
46                 return
47     LocalBroadcast (knowledge⟨known⟩, ρ)

48 Invocation of Leader():
49     component ← known[i].neigh
50     ∀ j ∈ component do
51         component ∪ known[j].neigh
52     return Max (ClosenessCentrality (component))
```

neighbor-aware mechanism adds both i and j in the set of neighbors of j (line 12). Then, i sets the clock value of j to 1 (line 13), as i was added to the knowledge of node j. On the other hand, if node i already has information about j (line 14), i is added to the neighbors of j (line 15), and the logical clock of node j is incremented (line 16).

Finally, by calling *LocalBroadcast* method (line 17), node i shares its knowledge with j and informs its neighborhood of its new neighbor j. Note that such a method sends a knowledge message to the neighbors of node i, with a gossip probability ρ, as seen in Section 2.8 [HHL02]. However, for the first hop, ρ is set to 1 to make sure that all neighbors of i will be aware of its new neighbor j. Note that the cross-layer mechanism of node j will also trigger its *Connection* method, and the respective steps will also be achieved on node j.

[HHL02] Haas et al. (2002): 'Gossip-based ad hoc routing'

5.2.5. Node disconnection (lines 18 to 23)

When a node j disappears from the transmission range of node i, the cross-layer mechanism stops receiving beacon messages at the MAC level, and triggers the *Disconnection* method (line 18). Node j is then removed from the knowledge of node i (line 19), and its clock is incremented as its knowledge was modified (line 20). Then, the neighbor-aware mechanism assumes that node i will also disconnect from j. Therefore, i is removed from the neighborhood of j in the knowledge of node i, and the corresponding clock is incremented

(lines 21- 22). Finally, node i broadcasts its updated knowledge to its neighbors (line 23).

5.2.6. Knowledge update (lines 24 to 34)

When node i receives a knowledge message $known_j$, from node j (line 24), it looks at each node n included in $known_j$ (line 25). If n is an unknown node for i (line 26), or if n is known by node i and has a more recent clock value in $known_j$ (line 27), the clock and neighbors of node n are updated in the knowledge of i (line 28).

Note that a clock value of n higher than the one currently known by node i (line 27) means that node n made some connections and/or disconnections of which node i is not aware. Then, the *UpdateNeighbors* method is called to update the knowledge of i regarding the neighbors of n (line 29). If the clock value of node n is identical to the one of both the knowledge of node i and $known_j$ (line 30), the neighbor-aware mechanism merges the neighbors of node n from $known_j$ with the known neighbors of n in the knowledge of i (line 31).

Remark that the clock of node n is not updated by the neighbor-aware mechanism, otherwise, n would not be able to override this view in the future with more recent information. The *UpdateNeighbors* method is then called (line 32). Finally, node i broadcasts its knowledge only if this latter was modified (lines 33-34).

5.2.7. Neighbors update (lines 35 to 41)

The *UpdateNeighbors* method (line 35) updates the knowledge of i with information about the neighbors of node n (line 36). If the neighbor k is an unknown node for i (line 37), or if k is known by i but has a more recent clock value in $known_j$ (line 38), the clock and neighbors of node k are added or updated in the knowledge of node i (line 39). Otherwise, if the clock of node k is identical in the knowledge of node i and in $known_j$ (line 40), the neighbor-aware mechanism merges the neighbors of node k in the knowledge of i (line 41).

5.2.8. Information propagation (lines 42 to 47)

The *TopologicalBroadcast* method (line 42) uses a self-pruning approach [LK01] to broadcast or not the updated knowledge of node i, after the reception of a knowledge from a neighbor j. To this end, node i checks whether each of its neighbors has the same neighborhood as itself (lines 43 to 44). In this case, node n is supposed to have also received the knowledge message from neighbor node j. Therefore, among the neighbors having the same neighborhood than i, only the one with the smallest identifier will broadcast the knowledge (line 45), with a

[LK01] Lim et al. (2001): 'Flooding in wireless ad hoc networks'

gossip probability ρ (line 47). Note that this topological self-pruning mechanism reaches the same neighborhood as multiple broadcasts.

5.2.9. Leader election (lines 48 to 52)

The leader is elected when a process running on node i calls the *Leader* function (line 48). This function returns the most central leader in the component according the closeness centrality (line 52), as seen in Section 2.7, using the knowledge of node i. The closeness centrality is chosen instead of the betweenness centrality, because it is faster to compute and requires fewer computational steps, therefore consuming less energy from the mobile node batteries than the latter.

First, node i rebuilds its component according to its topological knowledge. To do so, it computes the entire set of reachable nodes, by adding neighbors, neighbors of its neighbors, and so on (lines 49 to 51). Then, it evaluates the shortest distance between each reachable node and the other ones, and computes the closeness centrality for each of them. Finally, it returns the node having the highest closeness value as the leader (line 52). The highest node identifier is used to break ties among identical centrality values. If all nodes of the component have the same topological knowledge, the *Leader()* function will return the same leader node when invoked. Otherwise, it may return different leader nodes. However, when the network topology stops changing, the algorithm ensures that all nodes of a component will eventually have the same topological knowledge and therefore, the *Leader()* function will return the same leader node for all of them [Ing+13].

[Ing+13] Ingram et al. (2013): 'A leader election algorithm for dynamic networks with causal clocks'

5.3. Simulation Environment

Realistic simulations were carried over in order to compare the *Centrality-based Eventual Leader (CEL)* algorithm, with the Ω eventual leader election algorithm of Gómez-Calzado *et al.* [Góm+13] (see Section 3.2.2).

[Góm+13] Gómez-Calzado et al. (2013): 'Fault-tolerant leader election in mobile dynamic distributed systems'

Experiments were conducted on a C++ discrete event simulator called OMNeT++ [VH08], with the INET framework [MVK19] to model wireless protocols and mobile networks. This environment allows simulation of unreliable communication channels and realistic layers of the OSI communication model. Each experiment involves 60 moving nodes placed in a 500×500 meters obstacle-free area during 30 simulated minutes. The experiments were conducted using OMNeT++ in console-mode on remote servers, but a graphical interface shown in Figure 5.1 was mainly used for development purposes.

[VH08] Varga et al. (2008): 'An overview of the OMNeT++ simulation environment'

[MVK19] Mészáros et al. (2019): 'Inet framework'

Simulations consider a full MANET network stack, with the physical and data-link layers following the IEEE 802.11n specifications [09]. The use of a cross-layer mechanism at the MAC level allows the application layer to access neighbor's MAC addresses. Therefore, the identifier of

[09] (2009): *IEEE Std 802.11n-2009 – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput*

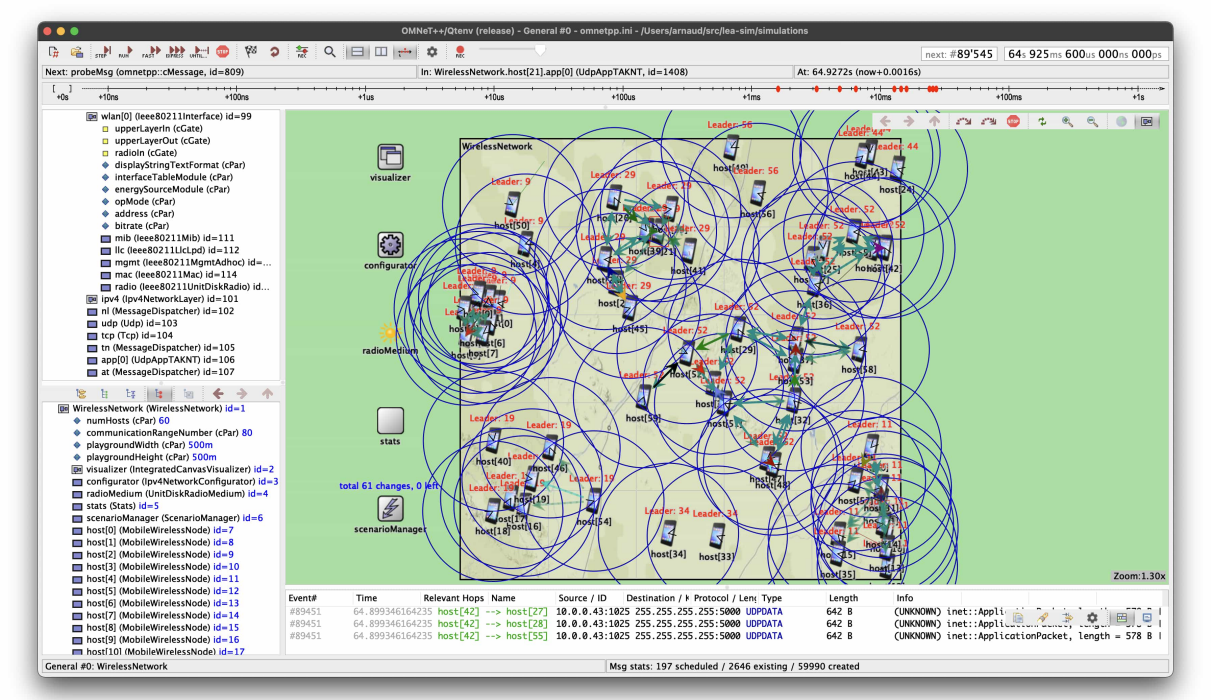

Figure 5.1.: Screenshot of an OMNeT++ experiment running the CEL algorithm.

a node is a MAC address, encoded on 3 bytes rather than usually 6 bytes, as it is assumed that all nodes have network components from the same manufacturer. Every node uses a single transceiver, with a fixed transmission range decided at the beginning of the experiment between 20 and 80 meters, and identical for all nodes. This transceiver uses the 2.4 GHz frequency band, with a nominal bitrate of 52 Mbps.

5.3.1. Algorithms Settings

In *CEL*, beacon messages are sent by the data-link layer every τ = 102.4 milliseconds (usual interval value of the Target Beacon Transmission Time), and are detected at the MAC level by the cross-layer mechanism.

The following two configurations of the algorithm were used for evaluation:

- In *CEL-1*, the probability ρ to gossip a knowledge message in the *TopologicalBroadcast* method is set to 1. Therefore, messages are flooded in the network, if the neighborhood of the sender node is different from the receiver's one.
- In *CEL-0.7*, ρ is set to 0.7, i.e. a message is retransmitted with 0.7 probability [HHL02], if the sender and receiver neighborhoods are different.

[HHL02] Haas et al. (2002): 'Gossip-based ad hoc routing'

In Gómez-Calzado *et al.* algorithm, the frequency to send either $join$ messages when a node is unconnected, or leader messages, is 102.4ms, as both are considered beacon messages. The timers detecting leader failure and node disconnection have an initial value (100ms) which is increased by 500ms when they expire. Since the original paper does not give any indication on the parameter values, these values were chosen after running several experiments using different values, as they were the most favorable to Gómez-Calzado *et al.* algorithm.

5.3.2. Mobility Models

Experiments use two mobility models from BonnMotion [Asc+10], a Java mobility scenario generation tool:

1. *Random Walk* [CBD02], based on the Brownian motion (mathematically described by Einstein in 1926 [SM01; Ein56]), where a node moves from its current location to a new location by randomly choosing a direction in the interval $[0, 2\pi]$ and a speed between 0.1m/s and 1m/s, with a pause time of 10 seconds once the destination is reached. A representation of the Random Walk is given in Figure 5.2.

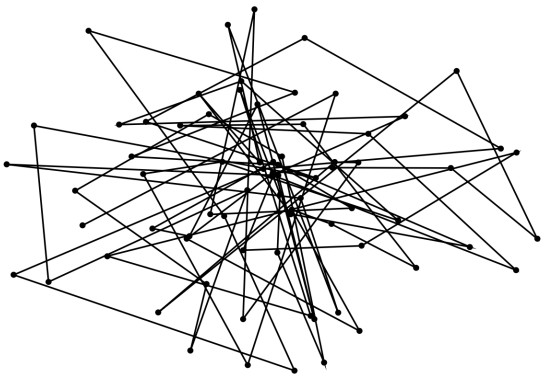

Figure 5.2.: Representation of the Random Walk mobility model [CBD02].

2. *Truncated Lévy Walk* [Rhe+11], which characterizes human mobility. Lévy walks are continuous-time random walks whose turning points are the visit points associated with the Lévy flights model. Parameters are a Levy exponent for flight length distribution α sets to 1, and a Levy exponent for pause time distribution β sets to 1. A representation of the Lévy Walk is given in Figure 5.3.

[Asc+10] Aschenbruck et al. (2010): 'Bonnmotion: a mobility scenario generation and analysis tool'

[CBD02] Camp et al. (2002): 'A survey of mobility models for ad hoc network research'

[SM01] Sánchez et al. (2001): 'ANEJOS: a Java based simulator for ad hoc networks'

[Ein56] Einstein (1956): *Investigations on the Theory of the Brownian Movement*

[Rhe+11] Rhee et al. (2011): 'On the levy-walk nature of human mobility'

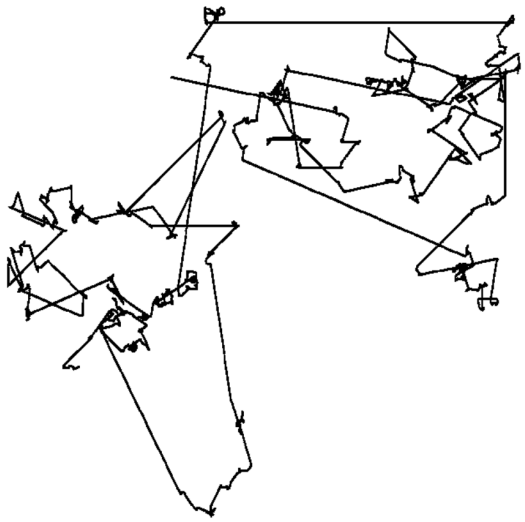

Figure 5.3.: Representation of the Lévy Walk mobility model [Rhe+11].

5.4. Evaluation

The goal is to compare the performance of both versions of the *CEL* algorithm, with Gómez-Calzado *et al.* algorithm [Góm+13], using different transmission ranges on both mobility patterns. Note that the number of components in the system is strongly correlated with the transmission range.

[Góm+13] Gómez-Calzado et al. (2013): 'Fault-tolerant leader election in mobile dynamic distributed systems'

5.4.1. Metrics

The following three metrics have been considered: average number of messages sent per second per node, average of the median path to the leader and instability.

Average number of messages sent per second per node

Similarly as in Section 4.4.1, this metric does not consider *beacon* messages, since the same number of *beacons* is sent every τ milliseconds by the underlying data-link layer in all three algorithms. In both *CEL* versions, *beacon* messages are used at the MAC layer by the cross-layer mechanism.

Average of the median path to the leader

Compared to the metric in Section 4.4.1, this metric characterizes how fast a leader can be reached by a majority of nodes (at least 50%) in its component. First is computed the *longest path* of all shortest paths from every node to their current leader, except for single node components,

as its null path would unfairly improve the metric. Then, the average of all medians over time is computed. Note that this metric is expressed in number of hops, and is not a ratio between the longest leader path and the component diameter as in Section 4.4.1.

Instability

This metric is the same as the one defined in Section 4.4.1. The oracle is based on the closeness centrality for *CEL*, and on the oldest node of the connected component with the highest identifier for Gómez-Calzado *et al.* algorithm.

5.4.2. Average number of messages sent per second per node

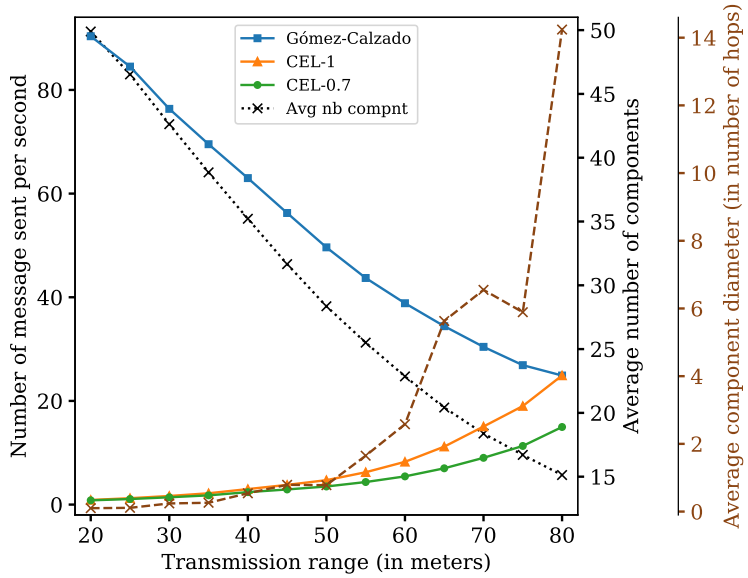

Figure 5.4.: Messages sent (lower is better) Random Walk.

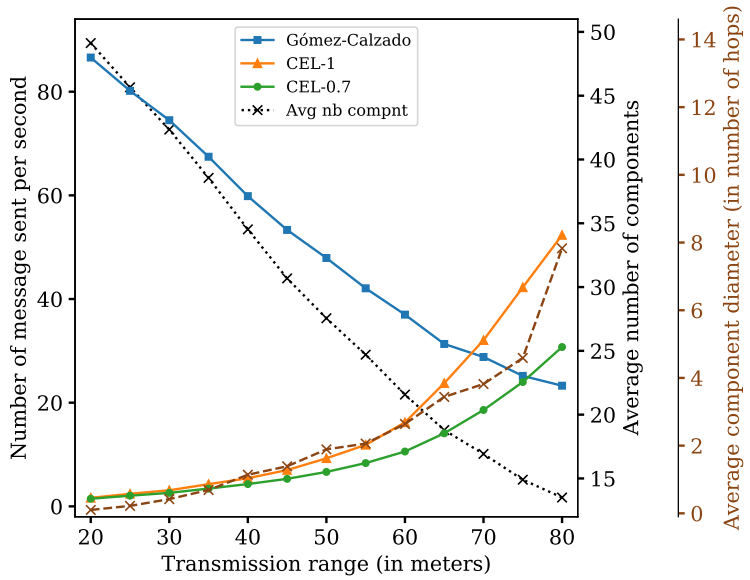

Figure 5.5.: Messages sent (lower is better) Truncated Lévy Walk.

Average number of messages sent per second is shown in Figures 5.4 and 5.5 for both mobility patterns. Right y-axes give the average number of components and their average diameter. In the Gómez-Calzado algorithm, nodes periodically send *join* messages when they are alone, in order to connect with a bigger component.

On both mobility models, *CEL-1* sends more messages than *CEL-0.7*, as it floods the network by broadcasting every received knowledge message. *CEL-0.7* reduces the number of messages sent per second, especially in larger transmission ranges, where more messages are broadcast at each topological change. There is an average reduction of 36% when ρ is set to 0.7 compared to ρ sets to 1, on both mobility patterns for a transmission range from 20m to 80m.

Note that the average message sent size varies from 4.56 to 6.58 bytes in the Gómez-Calzado algorithm, and from 263.03 to 1322.69 bytes for both versions of the *CEL* algorithm, as they share the topological knowledge of the component, and fits into the MTU of a single network packet.

5.4.3. Average of the median path to the leader

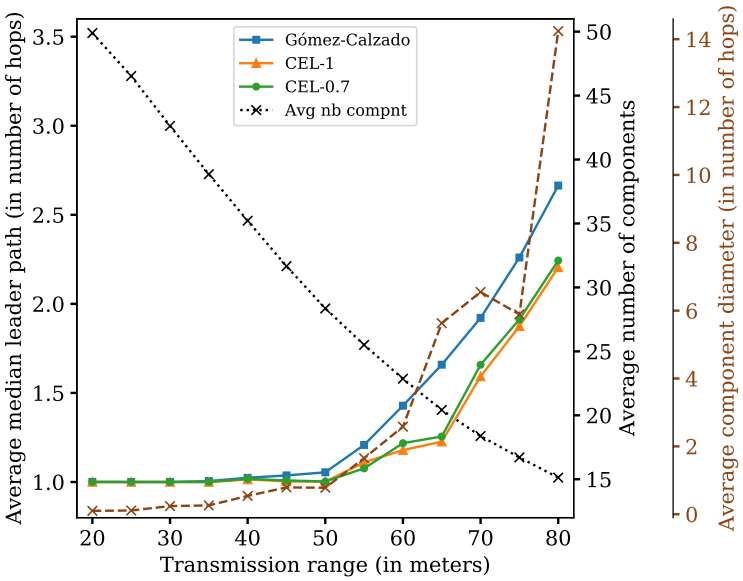

Figure 5.6.: Leader path (lower is better) Random Walk.

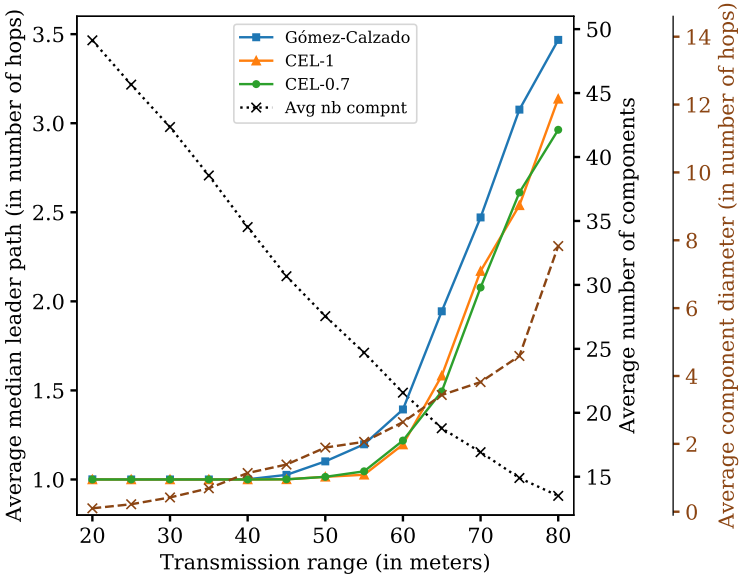

Figure 5.7.: Leader path (lower is better) Truncated Lévy Walk.

Average of the median path to the leader is shown in Figures 5.6 and 5.7 for both mobility patterns. Figure 5.6 shows that the average of the median path from every nodes of the components to the leader, is shorter in *CEL* than in Gómez-Calzado algorithm for the Random Walk mobility model, with a gain of up to 26% in larger transmission ranges. Interestingly, the probabilistic gossip version of *CEL* has a low impact on the leader path. These results are interesting, because the median represents half of the component nodes, which is the size of a quorum in Paxos-type consensus algorithms.

The Truncated Lévy Walk pattern shows in Figure 5.7 the impact of flying nodes which disrupt the component, by quickly moving in and out, therefore, modifying potential paths to the leader. Therefore, the difference between the *CEL* algorithm with ρ sets to 0.7 and the Gómez-Calzado algorithm, leads to a shortest path up to 15%.

Sharing a topological knowledge like in the *CEL* algorithm, allows the election of a central leader per component. Consequently, the results confirm that the number of hops to reach the leader by the nodes of its component is reduced.

5.4.4. Instability

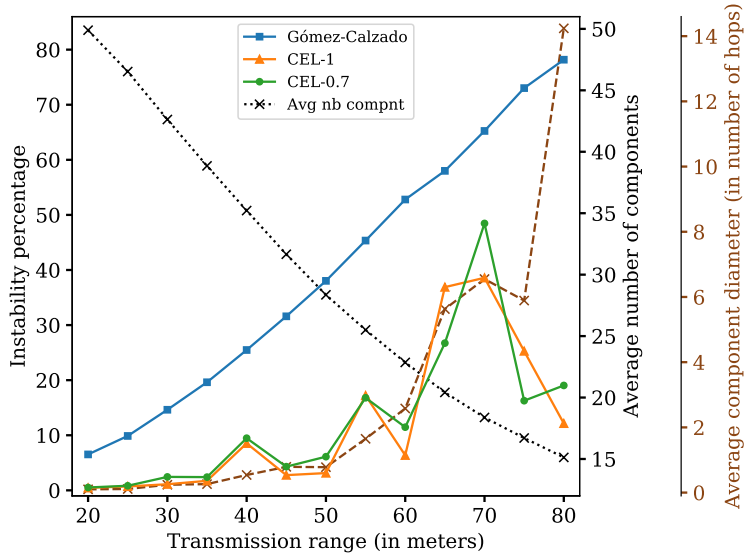

Figure 5.8.: Instability (lower is better) Random Walk.

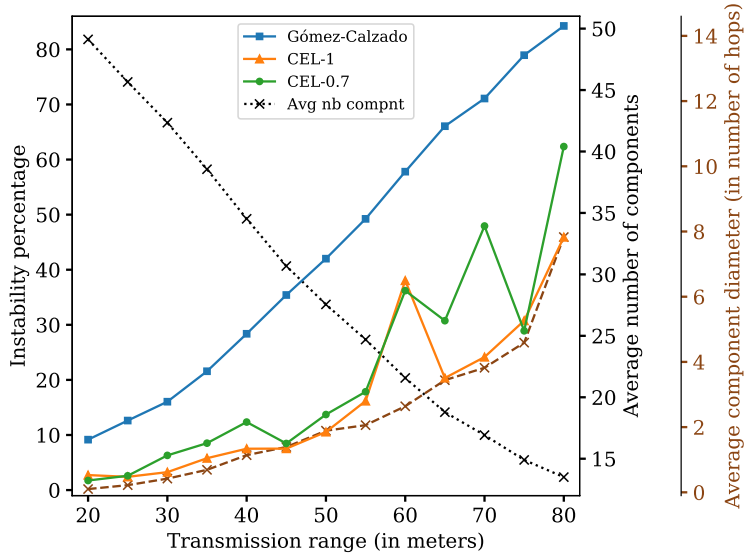

Figure 5.9.: Instability (lower is better) Truncated Lévy Walk.

Instability evolution is shown in Figures 5.8 and 5.9, according to the transmission range, and for both patterns. It is observed that the average instability increases when the transmission range increases, since components are composed of more nodes with a larger diameter. Hence, it takes a longer time to elect a new leader for all the algorithms.

In Figure 5.8, the percentage of instability in Gómez-Calzado algorithm is on average 69% higher than on both *CEL* versions. There is no significant instability difference between the *CEL* versions.

The instability for the Truncated Lévy Walk pattern in Figure 5.9, shows that the *CEL* algorithm is more stable than Gómez-Calzado algorithm when the transmission range increases. On average, *CEL* versions are 57% more stable than Gómez-Calzado algorithm. It can also be observed that the probabilistic gossip version of *CEL* with ρ sets to 0.7, is slightly less stable than the flooding version with ρ sets to 1. This is induced by a lower number of broadcast messages, making disrupting changes caused by flying nodes, to take more time to be spread over large components diameter.

5.4.5. Focusing on the 60 meters range over time

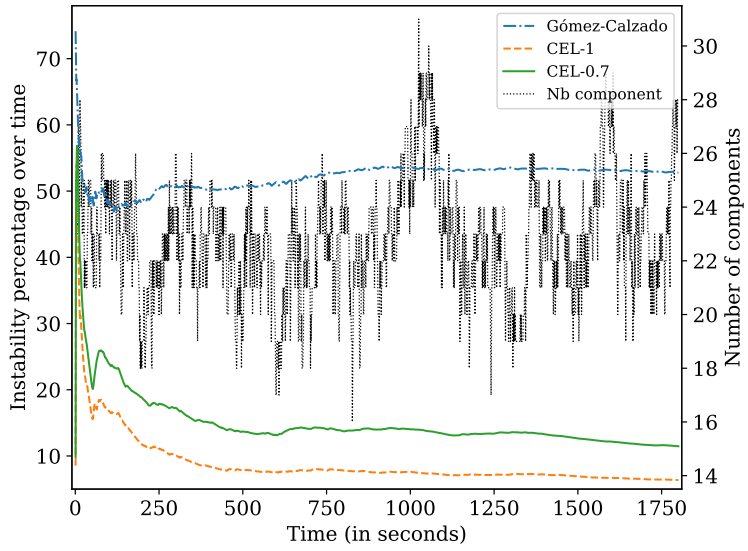

Figure 5.10.: Instability at 60m (lower is better) Random Walk.

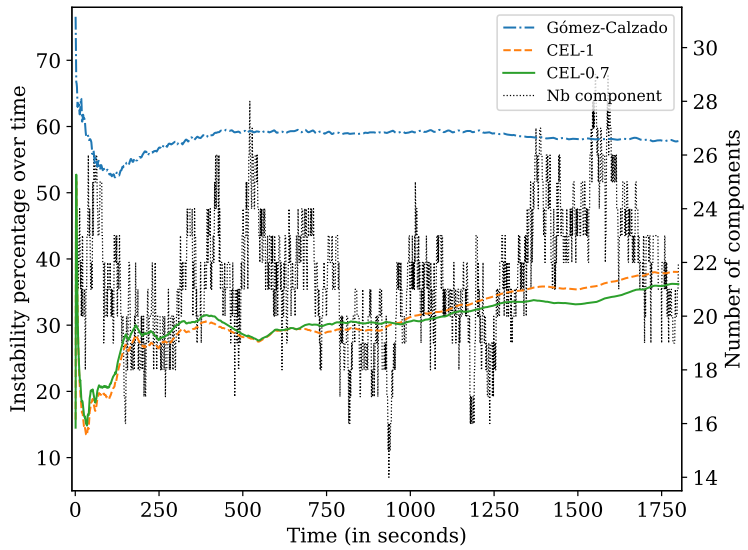

Figure 5.11.: Instability at 60m (lower is better) Truncated Lévy Walk.

Focusing on the 60 meters range over time is interesting to understand in detail the differences between the algorithms behaviors, on an approximate range of usual Wi-Fi indoor devices. Figures 5.10 and 5.11 show the average instability from time 0 to time t for both mobility patterns. The right y-axis gives the exact number of components at time t.

In Figure 5.10, at the beginning of the experiment on the Random Walk pattern, Gómez-Calzado algorithm has a higher instability rate, which quickly decreases to reach a threshold of 50% at 240 seconds, with a slight increase over time. Both *CEL* algorithm versions need a few seconds to stabilize, before reaching a threshold of around 430 seconds. The probabilistic gossip version (*CEL-0.7*) is less stable than the flooding version, as some knowledge messages are not broadcast by nodes to their neighborhood.

Figure 5.11 shows that the Truncated Lévy Walk model increases the instability of Gómez-Calzado algorithm, where an instability threshold of 59% is reached after 416 seconds. On the other hand, both *CEL* versions have a common instability evolution over time, with a small difference at the end of the experiment following the rebroadcast probability.

5.4.6. A comparative analysis with Topology Aware

A comparative analysis with the *Topology Aware* algorithm presented in Chapter 4 shows the performance gain by both the neighbor-aware and self-pruning mechanisms which exploit the topological knowledge, and by the probabilistic gossip.

The *Topology Aware* algorithm elects a central leader, but assumes reliable communication channels, which are not suitable for realistic

environments with message interference and collisions. A global view of the network is exchanged using probes and an update mechanism, leading to message collisions and losses. Furthermore, the knowledge of the topology is not used to improve communication performance, and the bidirectional links assumption is not taken into account to optimize knowledge sharing.

Table 5.1.: Random Walk (lower is better)

	Topology Aware	CEL-1	CEL-0.7
Messages sent	53.32/s	24.91/s	14.97/s
Leader path (in hop)	2.44	2.20	2.24
Instability	21.75%	12.15%	19.04%

Table 5.2.: Truncated Levy Walk (lower is better)

	Topology Aware	CEL-1	CEL-0.7
Messages sent	84.06/s	52.35/s	30.74/s
Leader path (in hop)	3.50	3.14	2.96
Instability	54.53%	45.92%	62.36%

Experiments ran in the realistic OMNeT++ environment described in Section 5.3, with a probe frequency $\tau = 102.4$ms for *Topology Aware*. The analysis is focused on a Wi-Fi transmission range of 80 meters, as it is the highest range of the experiments and a complex configuration with large components diameters.

Results for the Random Walk mobility pattern in Table 5.1, show that exploiting the topological knowledge reduces the number of messages sent per second by 71.92% comparing the *Topology Aware* algorithm to *CEL-0.7*, while having a shorter leader path. Instability is 44.14% lower comparing *Topology Aware* to *CEL-1*.

For the Truncated Levy Walk pattern in Table 5.2, the comparison between *Topology Aware* and *CEL-0.7* shows a reduction of the number of messages sent per second by 63.43%, and an average median leader path lower by 15.43%. Note that while the instability percentage is lower for *CEL-1*, the reduction of the number of messages by the probabilistic gossip version (*CEL-0.7*) may lead to lower stability than *Topology Aware*, especially when high transmission ranges imply large components.

5.5. Conclusion

This chapter proposed *CEL*, a new distributed eventual leader election algorithm for dynamic wireless networks, which exploits topological information to improve the choice of the leader and reduce message exchanges. A leader is eventually elected in each connected component of the network, and it is the node having the highest closeness centrality in the communication graph of the connected component.

CEL implements a cross-layer approach: when the underlying network layer (MAC) detects a change in the current neighborhood, the node updates its knowledge and spreads its new view of the network. In order to reduce the cost of message propagation, *CEL* uses a probabilistic gossip approach and local topological information to avoid redundant broadcasts.

Evaluation results from experiments on the OMNet++ environment with two mobility models, *Random Walk* and *Truncated Lévy Walk*, confirm that *CEL* algorithm reduces the number of messages and the path to the leader, when compared to Gómez-Calzado *et al.* algorithm.

6. Conclusion and Future Work

This thesis has addressed the eventual leader election problem in dynamic networks.

It presents two eventual leader election algorithms, *Topology Aware* and *Centrality-based Eventual Leader (CEL)*, for dynamic networks, which elect a leader as the process that presents the best closeness centrality in each connected component of the system. Having a central leader is essential to quickly reach a majority of nodes and communicate with a quorum of processes, as required by Paxos-type consensus algorithms, as well as to communicate faster with nodes of the component.

6.1. Contributions

Topology Aware and *Centrality-based Eventual Leader (CEL)* election algorithms consider dynamic networks, where nodes can move, fail by crash, join and leave the system, and partial synchronous system where network partitions can happen.

The algorithms progressively build and maintain a local knowledge of the network topology and rely only on broadcasts within the transmission range of nodes. They do not require any election communication phase: by using its current *topological knowledge*, each node can directly deduce at any moment which node is the current leader, by choosing the most central node of its connected component according to the closeness centrality.

The *Topology Aware* algorithm assumes reliable communication channels, and an underlying *probe system* to detect connection and disconnection of nodes. It uses an update mechanism to improve propagation cost of messages over the network. Evaluation experiments on top of PeerSim simulator [MJ09] were conducted with *Topology Aware* to a flooding algorithm based on [VKT04]. Performance results show that, compared to the flooding leader algorithm, *Topology Aware* 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.

[MJ09] Montresor et al. (2009): 'PeerSim: A Scalable P2P Simulator'
[VKT04] Vasudevan et al. (2004): 'Design and analysis of a leader election algorithm for mobile ad hoc networks'

Compared to *Topology Aware*, the *Centrality-based Eventual Leader (CEL)* algorithm assumes eventually reliable communication channels, closer to realistic environments with interference and message collisions. Communication follows the IEEE 802.11 [09] standard and message loss is handled by CSMA/CA protocol, included in the standard. *CEL*

[09] (2009): *IEEE Std 802.11n-2009 – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput*

implements a cross-layer neighbors detection which exploits the beacon messages of the underlying data-link layer and accesses the MAC addresses of the nodes, which are used to construct the topological knowledge of the connected component. In addition, *CEL* improves communication performance through a self-pruning mechanism combined with a probabilistic gossip. Finally, a neighbor-aware mechanism reduces the sending of redundant knowledge information on the network, which was not the case in the first contribution.

CEL was extensively evaluated on the OMNeT++/INET [Var10; MVK19] environment using both Random Walk and Truncated Lévy Walk mobility models, simulating, therefore, a realistic Mobile Ad Hoc Network (MANET) with interference, collision, and messages loss. Results of performance comparison with [Góm+13] show that *CEL* presents a good trade-off between the number of messages exchanged, stability of the leader, and the closeness of the leader to the other nodes.

[Var10] Varga (2010): 'OMNeT++'
[MVK19] Mészáros et al. (2019): 'Inet framework'

[Góm+13] Gómez-Calzado et al. (2013): 'Fault-tolerant leader election in mobile dynamic distributed systems'

6.2. Future Directions

As future research directions, a few possibilities are summarized in the following:

Short term

- Proof of correctness for *Topology Aware* and *CEL* algorithms.
- Implementation and evaluation of a leader-based consensus algorithm, which could take advantage of the good leader connectivity offered both in *Topology Aware* and *CEL*.
- Render publicly available the source code for the implementation of the *Topology Aware* and *CEL* algorithms, with the benchmarking environments, allowing, thus, other researchers to easily implement and evaluate their leader election algorithms through realistic experimentation.
- Evaluation of the energy consumption of nodes running *Topology Aware* and *CEL* algorithms, on both wireless communication and computation. Since communication has a high energy consumption, extending the cross-layer approach by using existing data on different levels of the OSI model, should minimize the number of message exchanged in the algorithm, and, therefore, reduce the energy consumption of nodes. Preliminary works on the energy consumption are presented in Appendix A.1.
  Furthermore, the choice of the eventual leader could be based on energy consumption. For instance, elect a leader with higher remaining battery, as it will probably exchange more messages than the other component nodes.

Mid-term

- Implementation and evaluation of *Topology Aware* and *CEL* algorithms on real networks like the ones composed by devices such as Raspberry Pi.
- A mechanism to reduce the size of messages exchanged, since larger message sizes also have an important impact on the collision rate. Such a reduction could be achieved by using compression algorithms, or by restricting the knowledge of the network to a limited number of hops, either locally for the centrality computation or for exchanged messages. Therefore, message exchanges would be smaller, and communication would be more reliable with less energy consumption.
- Exploration of other centrality metrics, such as the one proposed by Kim *et al.* [KW13], the eigenvector used by Google PageRank algorithm [BP98], or an estimation of the closeness [Coh+14], taking advantage of network dynamics and reducing the number of required computation steps. Temporal centralities [Gha18] could also be considered.
- A collaborative approach of nodes of the same component in order to exchange information about current centrality computation, aiming at reducing the number of required computation steps, redundant computation, and energy consumption. Examples, such as the algorithm of Wang *et al.* [WT15] or Lulli *et al.* [Lul+15], could be considered.

[KW13] Kim et al. (2013): 'Leader election on tree-based centrality in ad hoc networks'

[BP98] Brin et al. (1998): 'The anatomy of a large-scale hypertextual web search engine'

[Coh+14] Cohen et al. (2014): 'Computing classic closeness centrality, at scale'

[Gha18] Ghanem (2018): 'Temporal centralities: a study of the importance of nodes in dynamic graphs'

[WT15] Wang et al. (2015): 'Distributed estimation of closeness centrality'

[Lul+15] Lulli et al. (2015): 'Distributed current flow betweenness centrality'

A. Appendix

A.1. Energy consumption per node

Some preliminary results about the energy consumption of nodes executing *Centrality-based Eventual Leader (CEL)* and Gómez-Calzado *et al.* algorithms (Chapter 5) are presented in the following.

A.1.1. Simulation environment

Experiments were conducted on the OMNeT++ [VH08]/INET [MVK19] environment described in Section 5.3, with the same configuration. However, the transmission range is fixed between 30 meters (instead of 20 meters) and 80 meters.

[VH08] Varga et al. (2008): 'An overview of the OMNeT++ simulation environment'
[MVK19] Mészáros et al. (2019): 'Inet framework'

The power consumption consumer model for IEEE 802.11 used the following constant parameter values approximately based on a CC3220 transceiver:

- Sleep is set to 0.05mW
- Idle receiver is set to 0.5W
- Idle transmitter is set to 1W
- Switching mode is set to 100mW
- Busy receiver is set to 0.5W
- Reception is set to 1W
- Transmission is set to 2.5W

Local computation (CPU) and mobility are not considered in the energy consumption model, neither the power transmission range fixed at each experiment. Each experiment lasts 30 simulated minutes.

A.1.2. Algorithms settings

The two versions of *CEL*, i.e. CEL-1 and CEL-0.7, and Gómez-Calzado *et al.* algorithm [Góm+13] are compared using the same settings has in Section 5.3.1.

[Góm+13] Gómez-Calzado et al. (2013): 'Fault-tolerant leader election in mobile dynamic distributed systems'

A.1.3. Mobility Models

The two mobility models defined in Section 5.3.2 are used, i.e. Random Walk and Truncated Lévy Walk.

A.1.4. Metric

The energy consumption is computed using the radio power consumer model describe in A.1.1, where transmission, reception, or idle states consume energy from the node battery. At the end of each experience, the **energy consumption per node** for each algorithm is computed and measured in watts (W).

A.1.5. Performance Results

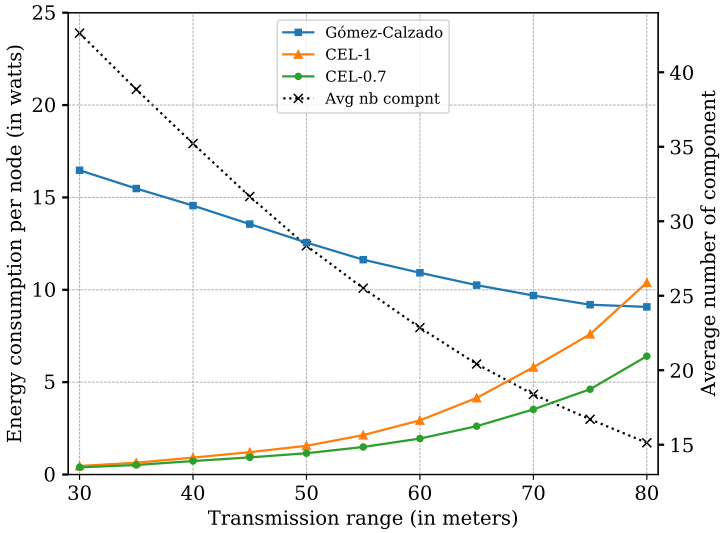

Figure A.1.: Energy consumption per node (lower is better) Random Walk.

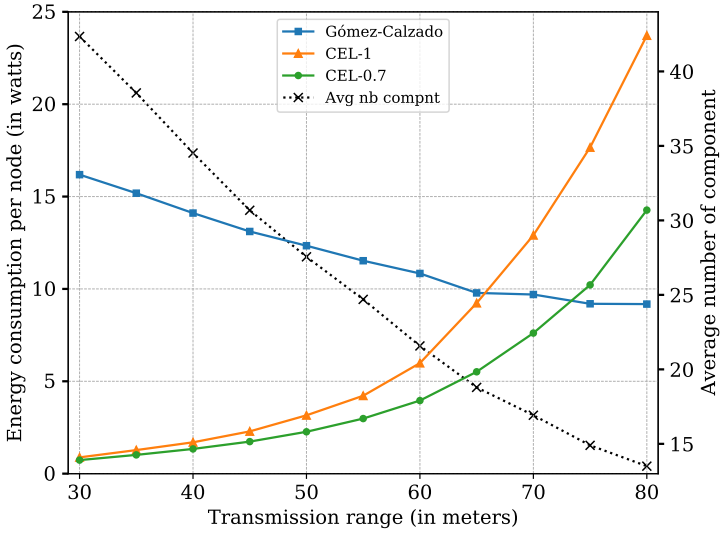

Figure A.2.: Energy consumption per node (lower is better) Truncated Lévy Walk.

Overall, the energy consumption per node is correlated with the number of messages sent, presented in Section 5.4.2, as the power consumption consumer model considers only communication.

For the Random Walk mobility model in Figure A.1, the power consumption of nodes in the Gómez-Calzado *et al.* algorithm decreases as the transmission range increases, since more nodes are in the same component and timeouts of leader messages are higher. The power consumption of both versions *CEL* algorithms increase as the transmission range increase, since movements in larger connected component lead to more messages sent. In lower transmission ranges, the energy consumption per node executing Gómez-Calzado *et al.* algorithm is higher than both *CEL* versions. For instance, at a transmission range of 30 meters, nodes executing Gómez-Calzado *et al.* algorithm consume on average 16.47W, while nodes in CEL-1 and CEL-0.7 consume on average 0.46W and 0.39W respectively. The probabilistic gossip version of *CEL* with ρ sets to 0.7 reduces the energy consumption compared to CEL-1, since less messages are sent as seen in Section 5.4.2, due to the self-pruning approach used in the *TopologicalBroadcast* method described in Section 5.2.8. Nodes in CEL-0.7 have an energy consumption of 6.41W on average at a transmission range of 80 meters, while nodes in CEL-1 and Gómez-Calzado *et al.* algorithms are up to 10.39W and 9.07W respectively.

Figure A.2 shows a higher average energy consumption for nodes executing CEL-1 than Gómez-Calzado *et al.* in the Truncated Lévy Walk mobility model when the transmission range is larger than 65 meters. A similar observation can be made concerning the average number of messages sent on this mobility model as presented in Section 5.4.2. At a transmission range of 80 meters, nodes executing CEL-0.7 have an average energy consumption up to 14.27W, while nodes executing CEL-1

are up to 23.72W, and nodes in Gómez-Calzado *et al.* algorithm are up to 9.18W. These increase are due to flying nodes that induce many connections and disconnections among existing connected components. However, the energy consumption per node remains lower for both *CEL* versions than Gómez-Calzado *et al.* algorithm on smaller ranges such as 30 meters.

Note that this preliminary evaluation is a work in progress.

Bibliography

[09] *IEEE Std 802.11n-2009 – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput*. 2009.

[Abr88] Karl Abrahamson. 'On achieving consensus using a shared memory'. In: *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*. 1988, pp. 291–302.

[Afe85] Yehuda Afek. 'Distributed algorithms for election in unidirectional and complete networks (traversal, synchronous, leader, asynchronous, complexity).' PhD thesis. University of California, Nov. 1985.

[AG85] Yehuda Afek and Eli Gafni. 'Time and message bounds for election in synchronous and asynchronous complete networks'. In: *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*. 1985, pp. 186–195.

[Agu+01] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. 'Stable Leader Election'. In: *Distributed Computing, 15th Int. Conference, DISC 2001, Proc.* 2001, pp. 108–122.

[Agu+03] Marcos K Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. 'On implementing omega with weak reliability and synchrony assumptions'. In: *The 22nd annual symposium on Principles of distributed computing*. 2003, pp. 306–314.

[Agu+04] Marcos K Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. 'Communication-efficient leader election and consensus with limited link synchrony'. In: *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*. 2004, pp. 328–337.

[Agu+08] Marcos K Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. 'On implementing omega in systems with weak reliability and synchrony assumptions'. In: *Distributed Computing* 21.4 (2008), pp. 285–314.

[Agu04] Marcos K Aguilera. 'A pleasant stroll through the land of infinitely many creatures'. In: *ACM Sigact News* 35.2 (2004), pp. 36–59.

[AJR10] Antonio Fernández Anta, Ernesto Jiménez, and Michel Raynal. 'Eventual Leader Election with Weak Assumptions on Initial Knowledge, Communication Reliability, and Synchrony'. In: *J. Comput. Sci. Technol.* 25.6 (2010), pp. 1267–1281.

[And00] Gregory R Andrews. *Foundations of multithreaded, parallel, and distributed programming*. Vol. 11. Addison-Wesley Reading, 2000.

[Ang80] Dana Angluin. 'Local and global properties in networks of processors'. In: *Proceedings of the twelfth annual ACM symposium on Theory of computing*. 1980, pp. 82–93.

[Ara+13] Luciana Arantes, Fabíola Greve, Pierre Sens, and Véronique Simon. 'Eventual Leader Election in Evolving Mobile Networks'. In: *Principles of Distributed Systems - 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings*. Ed. by Roberto Baldoni, Nicolas Nisse, and Maarten van Steen. Vol. 8304. Lecture Notes in Computer Science. Springer, 2013, pp. 23–37.

[Asc+10] Nils Aschenbruck, Raphael Ernst, Elmar Gerhards-Padilla, and Matthias Schwamborn. 'Bonnmotion: a mobility scenario generation and analysis tool'. In: *Proceedings of the 3rd international ICST conference on simulation tools and techniques*. 2010, pp. 1–10.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*. Vol. 19. John Wiley & Sons, 2004.

[Awe87] Baruch 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.

[Bab79] László Babai. 'Monte-Carlo algorithms in graph isomorphism testing'. In: *Université tde Montréal Technical Report, DMS* 79-10 (1979).

[Bav50] Alex Bavelas. 'Communication patterns in task-oriented groups'. In: *The journal of the acoustical society of America* 22.6 (1950), pp. 725–730.

[BBF16] Ulrik Brandes, Stephen P Borgatti, and Linton C Freeman. 'Maintaining the duality of closeness and betweenness centrality'. In: *Social Networks* 44 (2016), pp. 153–159.

[BCT96] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. *Solving problems in the presence of process crashes and lossy links*. Tech. rep. Cornell University, 1996.

[BE06] Stephen P Borgatti and Martin G Everett. 'A graph-theoretic perspective on centrality'. In: *Social networks* 28.4 (2006), pp. 466–484.

[Ber04] Marin Bertier. 'Service de détection de défaillances hiérarchique'. PhD thesis. Paris 6, 2004.

[BO99] Bhargav Bellur and Richard 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.

[Bor05] Stephen P Borgatti. 'Centrality and network flow'. In: *Social networks* 27.1 (2005), pp. 55–71.

[BP98] Sergey Brin and Lawrence Page. 'The anatomy of a large-scale hypertextual web search engine'. In: *Computer networks and ISDN systems* 30.1-7 (1998), pp. 107–117.

[Bra01] Ulrik Brandes. 'A faster algorithm for betweenness centrality'. In: *Journal of mathematical sociology* 25.2 (2001), pp. 163–177.

[BRS12] Martin Biely, Peter Robinson, and Ulrich Schmid. 'Agreement in directed dynamic networks'. In: *International Colloquium on Structural Information and Communication Complexity*. Springer. 2012, pp. 73–84.

[Bur80] James E Burns. 'A formal model for message-passing systems'. In: *Technical Report 91* (1980).

[Cal15] Carlos Gómez Calzado. 'Contributions on agreement in dynamic distributed systems'. PhD thesis. Universidad del País Vasco-Euskal Herriko Unibertsitatea, 2015.

[Cam20] Christian Robert Fernandez Campusano. 'Distributed eventual leader election in the crash-recovery and general omission failure models'. PhD thesis. Universidad del País Vasco-Euskal Herriko Unibertsitatea, 2020.

[Cas+11] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. 'Time-varying graphs and dynamic networks'. In: *Int. Conference on Ad-Hoc Networks and Wireless*. Springer. 2011, pp. 346–359.

[CBD02] Tracy Camp, Jeff Boleng, and Vanessa Davies. 'A survey of mobility models for ad hoc network research'. In: *Wireless communications and mobile computing* 2.5 (2002), pp. 483–502.

[CBL18] Rodolfo WL Coutinho, Azzedine Boukerche, and Antonio AF Loureiro. 'Design guidelines for information-centric connected and autonomous vehicles'. In: *IEEE Communications Magazine* 56.10 (2018), pp. 85–91.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[Cha82] Ernest J. H. Chang. 'Echo algorithms: Depth parallel operations on general graphs'. In: *IEEE Transactions on Software Engineering* 4 (1982), pp. 391–401.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. 'The weakest failure detector for solving consensus'. In: *Journal of the ACM (JACM)* 43.4 (1996), pp. 685–722.

[Coh+14] Edith Cohen, Daniel Delling, Thomas Pajor, and Renato F Werneck. 'Computing classic closeness centrality, at scale'. In: *Proceedings of the second ACM conference on Online social networks*. 2014, pp. 37–50.

[Com99] Standards Committee. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. IEEE Standard 802.11. 1999.

[Cor+09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[CR79] Ernest Chang and Rosemary Roberts. 'An improved algorithm for decentralized extrema-finding in circular configurations of processes'. In: *Communications of the ACM* 22.5 (1979), pp. 281–283.

[CT85] Francis Chin and HF Ting. 'An almost linear time and O (nlogn+ e) messages distributed algorithm for minimum-weight spanning trees'. In: *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. IEEE. 1985, pp. 257–266.

[CT96] Tushar Deepak Chandra and Sam Toueg. 'Unreliable failure detectors for reliable distributed systems'. In: *Journal of the ACM (JACM)* 43.2 (1996), pp. 225–267.

[Dal77] Yogen Kantilal Dalal. *Broadcast protocols in packet switched computer networks.* Stanford University, 1977.

[Dem+87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 'Epidemic algorithms for replicated database maintenance'. In: *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. 1987, pp. 1–12.

[DKR82] Danny Dolev, Maria Klawe, and Michael Rodeh. 'An O (n log n) unidirectional distributed algorithm for extrema finding in a circle'. In: *Journal of algorithms* 3.3 (1982), pp. 245–260.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 'Consensus in the presence of partial synchrony'. In: *Journal of the ACM (JACM)* 35.2 (1988), pp. 288–323.

[Du19] Donglei Du. 'Social network analysis: Centrality measures'. In: *University of New Brunswick* (2019).

[Dub11] Swan Dubois. 'Tolerating Transient, Permanent, and Intermittent Failures'. PhD thesis. Université Pierre et Marie Curie-Paris VI, 2011.

[Ein56] Albert Einstein. *Investigations on the Theory of the Brownian Movement*. Courier Corporation, 1956.

[Fav+19] Arnaud Favier, Nicolas Guittonneau, Luciana Arantes, Anne Fladenmuller, and Pierre Sens. 'Un algorithme d'élection de leader cross-layer pour réseaux mobiles ad hoc (résumé)'. In: *COMPAS 2019 - Conférence d'informatique en Parallélisme, Architecture et Système*. Anglet, France, June 2019.

[Fav+20a] Arnaud Favier, Nicolas Guittonneau, Luciana Arantes, Anne Fladenmuller, Jonathan Lejeune, and Pierre Sens. 'Topology Aware Leader Election Algorithm for Dynamic Networks'. In: *25th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2020, Perth, Australia, December 1-4, 2020*. IEEE, 2020, pp. 1–10.

[Fav+20b] Arnaud Favier, Nicolas Guittonneau, Jonathan Lejeune, Anne Fladenmuller, Luciana Arantes, and Pierre Sens. 'Topology Aware Leader Election Algorithm for MANET'. In: *COMPAS 2020 - Conférence francophone d'informatique en Parallélisme, Architecture et Système*. Lyon, France, June 2020.

[Fav+21] Arnaud Favier, Luciana Arantes, Jonathan Lejeune, and Pierre Sens. 'Centrality-Based Eventual Leader Election in Dynamic Networks'. In: *20th IEEE International Symposium on Network Computing and Applications, NCA 2021, Boston, MA, USA, November 23-26, 2021*. Ed. by Mauro Andreolini, Mirco Marchetti, and Dimiter R. Avresky. IEEE, 2021, pp. 1–8.

[Fer+17] Christian Fernández-Campusano, Mikel Larrea, Roberto Cortiñas, and Michel Raynal. 'A distributed leader election algorithm in crash-recovery and omissive systems'. In: *Information Processing Letters* 118 (2017), pp. 100–104.

[Fer04] Afonso Ferreira. 'Building a reference combinatorial model for MANETs'. In: *IEEE network* 18.5 (2004), pp. 24–29.

[FJR06] Antonio Fernández, Ernesto Jiménez, and Michel Raynal. 'Eventual Leader Election with Weak Assumptions on Initial Knowledge, Communication Reliability, and Synchrony'. In: (2006), pp. 166–178.

[FL84] Greg N Frederickson and Nancy A Lynch. 'The impact of synchronous communication on the problem of electing a leader in a ring'. In: *The 16th annual ACM symposium on Theory of computing*. 1984, pp. 493–503.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 'Impossibility of Distributed Consensus with One Faulty Process'. In: *J. ACM* 32.2 (Apr. 1985), pp. 374–382.

[FMS09] Paola Flocchini, Bernard Mans, and Nicola Santoro. 'Exploration of periodically varying graphs'. In: *International Symposium on Algorithms and Computation*. Springer. 2009, pp. 534–543.

[Fra82] Randolph Franklin. 'On an improved algorithm for decentralized extrema finding in circular configurations of processors'. In: *Communications of the ACM* 25.5 (1982), pp. 336–337.

[Fre77] Linton C Freeman. 'A set of measures of centrality based on betweenness'. In: *Sociometry* (1977), pp. 35–41.

[Fre78] Linton C Freeman. 'Centrality in social networks conceptual clarification'. In: *Social networks* 1.3 (1978), pp. 215–239.

[Gaf85] Eli Gafni. 'Improvements in the time complexity of two message-optimal election algorithms'. In: *The 4th annual ACM symposium on Principles of distributed computing*. 1985, pp. 175–185.

[Gal77] Robert G Gallager. 'Finding a leader in a network with O (e)+ O (n log n) messages'. In: *Unpublished Note* (1977).

[Gar82] Hector Garcia-Molina. 'Elections in a distributed computing system'. In: *IEEE transactions on Computers* 31.01 (1982), pp. 48–59.

[Gha18] Marwan Ghanem. 'Temporal centralities: a study of the importance of nodes in dynamic graphs'. PhD thesis. Sorbonne Universites, UPMC University of Paris 6, 2018.

[GHS83] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. 'A distributed algorithm for minimum-weight spanning trees'. In: *ACM Transactions on Programming Languages and systems (TOPLAS)* 5.1 (1983), pp. 66–77.

[GL08] Rachid Guerraoui and N Lynch. 'A general characterization of indulgence'. In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 3.4 (2008), pp. 1–19.

[Góm+13] Carlos Gómez-Calzado, Alberto Lafuente, Mikel Larrea, and Michel Raynal. 'Fault-tolerant leader election in mobile dynamic distributed systems'. In: *19th Pacific Rim Int. Symposium on Dependable Computing*. IEEE. 2013, pp. 78–87.

[GR04] Rachid Guerraoui and Michel Raynal. 'The information structure of indulgent consensus'. In: *IEEE Transactions on Computers* 53.4 (2004), pp. 453–466.

[GR06] Rachid Guerraoui and Luis Rodrigues. *Introduction to reliable distributed programming*. Springer Science & Business Media, 2006.

[Hal15] Saurav Haloi. *Apache zookeeper essentials*. Packt Publishing Ltd, 2015.

[Hat+99] Kostas P Hatzis, George P Pentaris, Paul G Spirakis, Vasilis T Tampakas, and Richard B Tan. 'Fundamental control algorithms in mobile networks'. In: *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*. 1999, pp. 251–260.

[HHL02] Zygmunt J Haas, Joseph Y Halpern, and Li Li. 'Gossip-based ad hoc routing'. In: *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*. Vol. 3. IEEE. 2002, pp. 1707–1716.

[HP93] Lisa Higham and Teresa Przytycka. 'A simple, efficient algorithm for maximum finding on rings'. In: *Int. Workshop on Distributed Algorithms*. Springer. 1993, pp. 249–263.

[HS80] Daniel S. Hirschberg and James B Sinclair. 'Decentralized extrema-finding in circular configurations of processors'. In: *Communications of the ACM* 23.11 (1980), pp. 627–628.

[HT94] Vassos Hadzilacos and Sam Toueg. *A modular approach to fault-tolerant broadcasts and related problems*. Tech. rep. Cornell University, 1994.

[Hum83] Pierre Humblet. 'A distributed algorithm for minimum weight directed spanning trees'. In: *IEEE Transactions on Communications* 31.6 (1983), pp. 756–762.

[Hut+08] Martin Hutle, Dahlia Malkhi, Ulrich Schmid, and Lidong Zhou. 'Chasing the weakest system model for implementing Ω and consensus'. In: *IEEE Transactions on Dependable and Secure Computing* 6.4 (2008), pp. 269–281.

[Ing+09] Rebecca Ingram, Patrick Shields, Jennifer E Walter, and Jennifer L Welch. 'An asynchronous leader election algorithm for dynamic networks'. In: *Int. Symposium on Parallel & Distributed Processing*. IEEE. 2009, pp. 1–12.

[Ing+13] Rebecca Ingram, Tsvetomira Radeva, Patrick Shields, Saira Viqar, Jennifer E Walter, and Jennifer L Welch. 'A leader election algorithm for dynamic networks with causal clocks'. In: *Distributed computing* 26.2 (2013), pp. 75–97.

[IR81] Alon Itai and Michael Rodeh. 'Symmetry breaking in distributive networks'. In: *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*. IEEE. 1981, pp. 150–158.

[JAF06] Ernesto Jiménez, Sergio Arévalo, and Antonio Fernández. 'Implementing unreliable failure detectors with unknown membership'. In: *Information Processing Letters* 100.2 (2006), pp. 60–63.

[Jel+07] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. 'Gossip-based peer sampling'. In: *ACM Transactions on Computer Systems (TOCS)* 25.3 (2007), 8–es.

[Kan78] P Kanellakis. 'An election problem in a network'. In: *Term paper, MIT* (May 1977-1978).

[KKK02] David Kempe, Jon Kleinberg, and Amit Kumar. 'Connectivity and inference problems for temporal networks'. In: *Journal of Computer and System Sciences* 64.4 (2002), pp. 820–842.

[KLO10] Fabian Kuhn, Nancy Lynch, and Rotem Oshman. 'Distributed computation in dynamic networks'. In: *Proceedings of the forty-second ACM symposium on Theory of computing*. 2010, pp. 513–522.

[KMG03] A-M Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. 'Probabilistic reliable dissemination in large-scale systems'. In: *IEEE Transactions on Parallel and Distributed systems* 14.3 (2003), pp. 248–258.

[KMZ84] Ephraim Korach, Shlomo Moran, and Shmuel Zaks. 'Tight lower and upper bounds for some distributed algorithms for a complete network of processors'. In: *Proceedings of the third annual ACM symposium on Principles of distributed computing*. 1984, pp. 199–207.

[Kos09] Vassilis Kostakos. 'Temporal graphs'. In: *Physica A: Statistical Mechanics and its Applications* 388.6 (2009), pp. 1007–1023.

[KW13] ChongGun Kim and Mary Wu. 'Leader election on tree-based centrality in ad hoc networks'. In: *Telecommunication Systems* 52.2 (2013), pp. 661–670.

[Lam77] Leslie Lamport. 'Proving the correctness of multiprocess programs'. In: *IEEE transactions on software engineering* 2 (1977), pp. 125–143.

[Lam78] Leslie Lamport. 'Time, Clocks, and the Ordering of Events in a Distributed System'. In: *Commun. ACM* 21.7 (1978), pp. 558–565.

[Lam87] Leslie Lamport. *distribution*. 1987. URL: https://lamport.azurewebsites.net/pubs/distributed-system.txt. (accessed: 10.09.2021).

[Lam98] Leslie Lamport. 'The part-time parliament'. In: *ACM Transactions on Computer Systems (TOCS)* 16.2 (1998), pp. 133–169.

[Lar+12] Mikel Larrea, Michel Raynal, Iratxe Soraluze, and Roberto Cortiñas. 'Specifying and implementing an eventual leader service for dynamic systems'. In: *International Journal of Web and Grid Services* 8.3 (2012), pp. 204–224.

[Le 77] Gérard Le Lann. 'Distributed Systems-Towards a Formal Approach.' In: *IFIP congress*. Vol. 7. 1977, pp. 155–160.

[LFA00] Mikel Larrea, Antonio Fernández, and Sergio 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.

[LK01] Hyojun Lim and Chongkwon Kim. 'Flooding in wireless ad hoc networks'. In: *Computer Communications* 24.3-4 (2001), pp. 353–363.

[LKF07] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. 'Graph evolution: Densification and shrinking diameters'. In: *ACM Trans. Knowl. Discov. Data* 1.1 (2007), p. 2.

[LM10] Avinash Lakshman and Prashant Malik. 'Cassandra: a decentralized structured storage system'. In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[LMS11] Mikel Larrea, Cristian Martín, and Iratxe Soraluze. 'Communication-efficient leader election in crash-recovery systems'. In: *Journal of Systems and Software* 84.12 (2011), pp. 2186–2195.

[LSP19] Leslie Lamport, Robert Shostak, and Marshall Pease. 'The Byzantine generals problem'. In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 203–226.

[LT87] Jan van Leeuwen and Richard B. Tan. 'An improved upperbound for distributed election in bidirectional rings of processors'. In: *Distributed Computing* 2.3 (1987), pp. 149–160.

[Lul+15] Alessandro Lulli, Laura Ricci, Emanuele Carlini, and Patrizio Dazzi. 'Distributed current flow betweenness centrality'. In: *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE. 2015, pp. 71–80.

[Lyn96] Nancy A Lynch. 'Distributed algorithms'. In: Elsevier, 1996. Chap. 4.1, pp. 52–56.

[MA+06] Salahuddin Mohammad Masum, Amin Ahsan Ali, et al. 'Asynchronous leader election in mobile ad hoc networks'. In: *20th International Conference on Advanced Information Networking and Applications-Volume 1 (AINA'06)*. Vol. 2. IEEE. 2006, 5–pp.

[MB12] Leila Melit and Nadjib Badache. 'An Ω-Based Leader Election Algorithm for Mobile Ad Hoc Networks'. In: *Networked Digital Technologies - 4th International Conference, NDT 2012, Dubai, UAE, April 24-26, 2012. Proceedings, Part I*. Ed. by Rachid Benlamri. Vol. 293. Communications in Computer and Information Science. Springer, 2012, pp. 483–490.

[MJ09] Alberto Montresor and Márk Jelasity. 'PeerSim: A Scalable P2P Simulator'. In: *The 9th Int. Conference on Peer-to-Peer (P2P'09)*. 2009, pp. 99–100.

[MK99] Jeff Magee and Jeff Kramer. *State models and java programs*. Vol. 10. 1999.

[MLJ09] Cristian Martín, Mikel Larrea, and Ernesto Jiménez. 'Implementing the omega failure detector in the crash-recovery failure model'. In: *Journal of Computer and System Sciences* 75.3 (2009), pp. 178–189.

[MMR03] Achour Mostefaoui, Eric Mourgaya, and Michel Raynal. 'Asynchronous implementation of failure detectors'. In: *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.* Citeseer. 2003, pp. 351–351.

[Mos+05] Achour Mostefaoui, Michel Raynal, Corentin Travers, Stacy Patterson, Divyakant Agrawal, and AE Abbadi. 'From static distributed systems to dynamic systems'. In: *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. IEEE. 2005, pp. 109–118.

[MOZ05] Dahlia Malkhi, Florin Oprea, and Lidong Zhou. 'Ω meets paxos: Leader election and stability without eventual timely links'. In: *International Symposium on Distributed Computing*. Springer. 2005, pp. 199–213.

[MRT04] Achour Mostefaoui, Michel Raynal, and Corentin Travers. 'Crash-resilient time-free eventual leadership'. In: *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.* IEEE. 2004, pp. 208–217.

[MRT06] Achour Mostéfaoui, Michel Raynal, and Corentin Travers. 'Time-Free and Timer-Based Assumptions Can Be Combined to Obtain Eventual Leadership'. In: *IEEE Trans. Parallel Distrib. Syst.* 17.7 (2006), pp. 656–666.

[MVK19] Levente Mészáros, Andras Varga, and Michael Kirsche. 'Inet framework'. In: *Recent Advances in Network Simulation*. Springer, 2019, pp. 55–106.

[MWV00] Navneet Malpani, Jennifer L Welch, and Nitin 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.

[Nak08] Satoshi Nakamoto. 'Bitcoin: A Peer-to-Peer Electronic Cash System'. In: (Oct. 2008).

[NT09] Mikhail Nesterenko and Sébastien Tixeuil. 'Discovering network topology in the presence of byzantine faults'. In: *Transactions on Parallel and Distributed Systems* 20.12 (2009), pp. 1777–1789.

[OO14] Diego Ongaro and John Ousterhout. 'In search of an understandable consensus algorithm'. In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 305–319.

[Pap+16] Michela Papandrea, Karim Keramat Jahromi, Matteo Zignani, Sabrina Gaito, Silvia Giordano, and Gian Paolo Rossi. 'On the properties of human mobility'. In: *Computer Communications* 87 (2016), pp. 19–36.

[PC97] Vincent Douglas Park and M Scott Corson. 'A highly adaptive distributed routing algorithm for mobile wireless networks'. In: *Proceedings of INFOCOM'97*. Vol. 3. IEEE. 1997, pp. 1405–1413.

[Pel90] David Peleg. 'Time-optimal leader election in general networks'. In: *Journal of parallel and distributed computing* 8.1 (1990), pp. 96–99.

[Pet82] Gary L Peterson. 'An O (n log n) unidirectional algorithm for the circular extrema problem'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.4 (1982), pp. 758–762.

[PKR82] Jan K Pachl, Ephraim Korach, and Doron Rotem. 'A technique for proving lower bounds for distributed maximum-finding algorithms (Preliminary Version)'. In: *Proceedings of the fourteenth annual ACM symposium on Theory of computing*. 1982, pp. 378–382.

[RAC08] Muhammad Rahman, M Abdullah-Al-Wadud, and Oksam Chae. 'Performance analysis of leader election algorithms in mobile ad hoc networks'. In: *Int. J. of Computer Science and Network Security* 8.2 (2008), pp. 257–263.

[Ray07] Michel Raynal. 'Eventual leader service in unreliable asynchronous systems: Why? how?' In: *Sixth IEEE International Symposium on Network Computing and Applications (NCA 2007)*. IEEE. 2007, pp. 11–24.

[Ray13] Michel Raynal. *Distributed algorithms for message-passing systems*. Vol. 500. Springer, 2013.

[Rhe+11] Injong Rhee, Minsu Shin, Seongik Hong, Kyunghan Lee, Seong Joon Kim, and Song Chong. 'On the levy-walk nature of human mobility'. In: *IEEE/ACM transactions on networking* 19.3 (2011), pp. 630–643.

[Sab66] Gert Sabidussi. 'The centrality index of a graph'. In: *Psychometrika* 31.4 (1966), pp. 581–603.

[SGK11] Giovanna Di Marzo Serugendo, Marie-Pierre Gleizes, and Anthony Karageorgos. *Self-organising Software*. Springer, 2011.

[SM01] Miguel Sánchez and Pietro Manzoni. 'ANEJOS: a Java based simulator for ad hoc networks'. In: *Future generation computer systems* 17.5 (2001), pp. 573–583.

[Spi77] P Spira. 'Communication complexity of distributed minimum spanning tree algorithms'. In: *The 2nd Berkeley conference on distributed data management and computer networks*. 1977.

[Tan+10] John Tang, Mirco Musolesi, Cecilia Mascolo, and Vito Latora. 'Characterising temporal distance and reachability in mobile and online social networks'. In: *ACM SIGCOMM Computer Communication Review* 40.1 (2010), pp. 118–124.

[Tap15] Tapiocozzo-Wikipedia. *Centrality - Wikipedia*. 2015. URL: https://en.wikipedia.org/wiki/Centrality. (accessed: 06.12.2021).

[TB10] Sara Tucci-Piergiovanni and Roberto Baldoni. 'Eventual leader election in infinite arrival message-passing system model with bounded concurrency'. In: *2010 European Dependable Computing Conference*. IEEE. 2010, pp. 127–134.

[Tel00] Gerard Tel. *Introduction to distributed algorithms*. Cambridge university press, 2000.

[TW+96] Andrew S Tanenbaum, David J Wetherall, et al. *Computer networks*. 1996.

[Var10] Andras Varga. 'OMNeT++'. In: *Modeling and tools for network simulation*. Springer, 2010, pp. 35–59.

[VH08] András Varga and Rudolf Hornig. 'An overview of the OMNeT++ simulation environment'. In: *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*. 2008, pp. 1–10.

[VKT04] Sudarshan Vasudevan, Jim Kurose, and Don 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.

[VT17] Maarten Van Steen and Andrew S Tanenbaum. *Distributed Systems*. 3. Maarten van Steen Leiden, The Netherlands, 2017.

[Woo+14] Gavin Wood et al. 'Ethereum: A secure decentralised generalised transaction ledger'. In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[WT15] Wei Wang and Choon Yik Tang. 'Distributed estimation of closeness centrality'. In: *2015 54th IEEE Conference on Decision and Control (CDC)*. IEEE. 2015, pp. 4860–4865.

Glossary

C
CEL Centrality-based Eventual Leader. 3–5, 67–69, 74–77, 79–88, 90, 91

D
DAG Directed Acyclic Graph. 29, 30, 32

F
FIFO First-In First-Out. 11, 31, 32

G
GST Global Stabilization Time. 7, 35, 42

M
MANET Mobile Ad Hoc Network. 1–4, 36, 41, 42, 55, 67, 74, 86
MIT Massachusetts Institute of Technology. 18
MST Minimum Spanning Tree. 26, 28

P
P2P Peer-to-Peer. 1, 17

S
$\mathcal{SatP}$ Stable Termination Property. 39
$\mathcal{SRP}$ Stabilized Responsiveness Property. 39

T
TBTT Target Beacon Transmission Time. 75
TVG Time-Varying Graph. 14, 38, 39

W
WWW World Wide Web. 1